Welcome to Eclipse

Lab User

Table of Contents

Welcome to Eclipse **1**
Who needs a platform? 1
End users 1
Software developers 1
The holy grail 1
What is Eclipse? 2
Open architecture 2
Platform structure 3
Out of the box 4

Platform architecture **5**
Platform SDK roadmap 5
Runtime core 6
Resource management 6
Workbench UI 6
Team support 7
Debug support 7
Help System 7
Java Development Tools (JDT) 7
Plug−in Development Environment (PDE) 7

Team support **8**
Repository providers 9
Extension point 9
Implementing a RepositoryProvider 9
Configuring a project 10
Finding a provider 11
Repository Providers and Capabilities 11
Resource modification hooks 12
Resource properties 13

Team Repository Provider **14**
Identifier: 14
Since: 14

Configuration Wizards **17**
Identifier: 17
Description: 17
Resource modification hooks 19
Resource move/delete hooks 19
File modification validators 19
General team hook 19

Move/Delete Hook **21**
Identifier: 21
Since: 21

Table of Contents

Team Hook **23**
Identifier: 23
Since: 23
Scheduling rules 24
Making your own rules 26
Rule hierarchies 27
Batching resource changes 28
Repository resource management 30
Ignored files 30
File Types 31
Team and linked resources 32
Team private resources 34
Project sets 34

Ignore **37**
Identifier: 37
Since: 37

File Types **39**
Identifier: 39
Since: 39
Linked resources 40
Path variables 40
Broken links 41
Compatibility with installed plug−ins 41
Linked resources in code 42
Project natures 43
Defining a nature 43
Associating the nature with a project 44
Nature descriptors 45

Project Natures **46**
Identifier: 46
Description: 46

Project Sets **52**
Identifier: 52
Since: 52
Synchronization Support 53
Terminology 53
The Basics − SyncInfo 54
Managing the synchronization state 55
Displaying the synchronizations state in the UI 56

Synchronize Participant Creation Wizards **58**
Identifier: 58
Since: 58
Local History Example 60

Table of Contents

Synchronize Participant Creation Wizards
Defining the variants for local history....61
Creating the Subscriber....63
Adding a Local History Synchronize Participant....64
Conclusion....66
Beyond the Basics....66
Implementing a Subscriber From Scratch....66
ThreeWaySubscriber....66
ThreeWaySynchronizer....67
ThreeWayRemoteTree....67
CachedResourceVariant....67
Building on Top of Existing Workspace Synchronization....67

Program debug and launch support....69

Plugging in help....70

Table of Contents (TOC)....71
Identifier:....71
Description:....71

Plug it in: Hello World meets the workbench....77
A minimal plug−in....77
Hello world view....78
Creating the plug−in project....78
Creating your plug−in project....78
The Hello World view....79
The Hello World manifests....80
Running the plug−in....81
Launching the workbench....81
Running Hello World....81

Views....84
Identifier:....84
Description:....84
Beyond the basics....88
Basic workbench extension points....88

Installing the examples....90
Installing examples via the Update Manager....90
Installing examples manually....90
Example − Readme Tool....90
Introduction....90
Running the example....91
Details....92
Workbench menu contributions....93

Table of Contents

Advanced workbench concepts **95**

Plugging into the workbench **96**
Quick tour of the workbench 96
Views 97
Editors 97

The JFace UI framework **98**
JFace and the workbench 98
JFace and SWT 98

The Standard Widget Toolkit **99**
Portability and platform integration 99

Resources overview **100**
Resources and the workspace 100
A sample resource tree 101
Resources and the local file system 101
Our sample tree on disk 102
Our sample tree in code 102
Mapping resources to disk locations 104
Resource API and the file system 104

Auto−refresh providers **106**
Identifier: 106
Since: 106
Refresh providers 107
Project−scoped preferences 108
Specifying the scope 108
Project−scoped preference nodes 108
Runtime preferences 109
Preference scopes 109
Using scopes and nodes 110
Extending the scopes 111
Products and features 111
Products extension point 112

Products **114**
Identifier: 114
Since: 114

Applications **119**
Identifier: 119
Description: 119
Customizing a product 121
About dialogs 121
Window images 122
Welcome page 122

Table of Contents

Applications
Preferences defaults....122
Splash screens....123

Intro Part....124
Identifier:....124
Since:....124
Intro support....126
Features....127

Feature Archives....129

Eclipse platform plug−in manifest....130

Eclipse platform feature manifest....136
Primary feature....143

Preferences....145
Identifier:....145
Since:....145
File encoding and content types....147
Setting a character set....147
Querying the character set....148
Content types for files in the workspace....148
Content types....148
Using content types....148
Finding out about content types....149
Detecting the content type for a data stream....149
Describing a data stream....149
Providing content−sensitive features....150
Resource markers....150
Marker operations....150
Extending the platform with new marker types....152

Resource Markers....155
Identifier:....155
Description:....155
Modifying the workspace....157
Tracking resource changes....157
Threading issues....161
Native event dispatching....161
SWT UI thread....162
Executing code from a non−UI thread....162
The workbench and threads....163
Concurrency and the workspace....164
Incremental project builders....165
Invoking a build....165
Defining an incremental project builder....166

Table of Contents

Resource Markers
Associating an incremental project builder with a project....168

Incremental Project Builders....169
Identifier:....169
Description:....169
Derived resources....172
Team private resources....172
Workspace save participation....173
Implementing a save participant....173
Using previously saved state....175
Workbench wizard extension points....176

Platform Extension Points....178
Platform runtime....178
Workspace....178
Platform text....178
Workbench....178
Team....179
Debug....180
Console....180
Help....180
Other....180

Adapters....182
Identifier:....182
Since:....182

Content Types....185
Identifier:....185
Since:....185

File Modification Validator....192
Identifier:....192
Description:....192

Annotation Model Creation....194
Identifier:....194
Since:....194

Document Creation....196
Identifier:....196
Since:....196

Document Setup....198
Identifier:....198
Since:....198

Table of Contents

Annotation Types **200**
Identifier: 200
Since: 200

Document Providers **203**
Identifier: 203
Since: 203

Marker Annotation Specification **206**
Identifier: 206
Since: 206

Marker Updaters **209**
Identifier: 209
Since: 209

Editor Template **212**
Identifier: 212
Since: 212

Reference Provider **217**
Identifier: 217
Since: 217

Spelling Engine **220**
Identifier: 220
Since: 220

Accelerator Configurations **223**
Identifier: 223
Since: 223

Commands **226**
Identifier: 226
Since: 226

Action Sets **234**
Identifier: 234
Description: 234

Bindings **250**
Identifier: 250
Since: 250

Contexts **255**
Identifier: 255
Since: 255

Table of Contents

Accelerator Scopes **257**
Identifier: 257
Since: 257

Accelerator Sets **260**
Identifier: 260
Since: 260

Action Definitions **264**
Identifier: 264
Since: 264

Action Set Part Associations **267**
Identifier: 267
Description: 267

Activities **269**
Identifier: 269
Since: 269

Activity Support **275**
Identifier: 275
Since: 275

Browser Support **281**
Identifier: 281
Since: 281

Cheat Sheet Content **283**
Identifier: 283
Since: 283

Cheat Sheet Content File XML Format **287**
cheatsheet 287
intro 287
description 287
item 288
subitem 289
conditional−subitem 289
repeated−subitem 290
action 291
perform−when 291
Example 292

Cheat Sheet Item Extension **294**
Identifier: 294
Since: 294

Table of Contents

Decorators **297**
Identifier: 297
Since: 297

Drop Actions **305**
Identifier: 305
Description: 305

Editor Menus, Toolbars and Actions **308**
Identifier: 308
Description: 308

Internal and External Editors **321**
Identifier: 321
Description: 321

Element Factories **326**
Identifier: 326
Description: 326

Encodings **328**
Identifier: 328
Since: 328

Export Wizards **330**
Identifier: 330
Description: 330

Font Definitions **333**
Identifier: 333
Since: 333

Handlers **336**
Identifier: 336
Since: 336

HelpSupport **345**
Identifier: 345
Since: 345

Marker Help **347**
Identifier: 347
Since: 347

Marker Image Providers **350**
Identifier: 350
Since: 350

Table of Contents

Marker Resolutions **353**
Identifier: 353
Since: 353

Project Nature Images **356**
Identifier: 356
Since: 356

Resource Filters **358**
Identifier: 358
Since: 358

Import Wizards **360**
Identifier: 360
Description: 360

Intro Part Configuration **363**
Identifier: 363
Since: 363

Intro Part Configuration Extension **378**
Identifier: 378
Since: 378

Intro Content File XML Format **382**
introContent 382
page 382
group 384
link 384
html 386
title 387
text 388
include 388
head 389
img 389
extensionContent 390
anchor 391
contentProvider 391

Keywords **393**
Identifier: 393
Since: 393

Creation Wizards **394**
Identifier: 394
Description: 394

Table of Contents

Perspective Extensions **399**
Identifier: 399
Description: 399

Perspectives **404**
Identifier: 404
Description: 404

Pop−up Menus **407**
Identifier: 407
Description: 407

Preference Pages **427**
Identifier: 427
Description: 427

Preference Transfer **430**
Identifier: 430
Since: 430

Presentation Factories **436**
Identifier: 436
Since: 436

Property Pages **438**
Identifier: 438
Description: 438

Startup **442**
Identifier: 442
Since: 442

System Summary Sections **444**
Identifier: 444
Since: 444

Themes **446**
Identifier: 446
Since: 446

View Menus,Toolbars and Actions **458**
Identifier: 458
Description: 458

Working Sets **471**
Identifier: 471
Since: 471

Table of Contents

Synchronize Participants **474**
Identifier: 474
Since: 474

Breakpoints **477**
Identifier: 477
Description: 477

Launch Configuration Comparators **479**
Identifier: 479
Description: 479

Launch Configuration Types **481**
Identifier: 481
Description: 481

Launch Delegates **484**
Identifier: 484
Since: 484

Launcher (Obsolete) **487**
Identifier: 487
Description: 487

Launch Modes **490**
Identifier: 490
Since: 490

Logical Structure Types **492**
Identifier: 492
Since: 492

Process Factories **495**
Identifier: 495
Since: 495

Source Container Types **497**
Identifier: 497
Since: 497

Source Locators **499**
Identifier: 499
Description: 499

Source Path Computers **501**
Identifier: 501
Since: 501

Table of Contents

Status Handlers **503**
Identifier: 503
Description: 503

watchExpressionDelegates **505**
Identifier: 505
Since: 505

Breakpoint Organizers **507**
Identifier: 507
Since: 507

Console Color Providers **510**
Identifier: 510
Since: 510

Console Line Trackers **512**
Identifier: 512
Since: 512

Context View Bindings **514**
Identifier: 514
Since: 514

Debug Model Context Bindings **517**
Identifier: 517
Since: 517

Debug Model Presentation **519**
Identifier: 519
Description: 519

Launch Configuration Tab Groups **521**
Identifier: 521
Description: 521

Launch Configuration Type Images **524**
Identifier: 524
Description: 524

Launch Groups **526**
Identifier: 526
Since: 526

Launch Shortcuts **529**
Identifier: 529
Description: 529

Table of Contents

Memory Renderings **538**
Identifier: 538
Since: 538

Source Container Presentations **546**
Identifier: 546
Since: 546

String Variable Presentations **549**
Identifier: 549
Since: 549

Variable Value Editors **551**
Identifier: 551
Since: 551

Console Factories **553**
Identifier: 553
Since: 553

Console Page Participants **555**
Identifier: 555
Since: 555

Console Pattern Match Listeners **562**
Identifier: 562
Since: 562

Help Content Producer **569**
Identifier: 569
Since: 569

Contexts **572**
Identifier: 572
Description: 572

Browser **575**
Identifier: 575
Since: 575

Lucene Analyzer **578**
Identifier: 578
Since: 578

Ant Properties **581**
Identifier: 581
Since: 581

Table of Contents

Ant Tasks **584**
Identifier: 584
Description: 584

Ant Types **586**
Identifier: 586
Description: 586

Extra Ant Classpath Entries **588**
Identifier: 588
Description: 588

ContentMerge Viewers **590**
Identifier: 590
Description: 590

Content Viewers **592**
Identifier: 592
Description: 592

Stream Merger **594**
Identifier: 594
Since: 594

Structure Creators **597**
Identifier: 597
Description: 597

StructureMerge Viewers **599**
Identifier: 599
Description: 599

Property Testers **601**
Identifier: 601
Since: 601

Dynamic String Substitution Variables **604**
Identifier: 604
Since: 604

Value Variables **606**
Identifier: 606
Since: 606

Copy Participants **609**
Identifier: 609
Since: 609

Table of Contents

Create Participants **616**
Identifier: 616
Since: 616

Delete Participants **623**
Identifier: 623
Since: 623

Move Participants **630**
Identifier: 630
Since: 630

Rename Participants **637**
Identifier: 637
Since: 637

Refactoring Change Preview Viewers **644**
Identifier: 644
Since: 644

Refactoring Status Context Viewers **651**
Identifier: 651
Since: 651

Search Pages **658**
Identifier: 658
Description: 658

Result Sorters **662**
Identifier: 662
Description: 662

Search Result View Pages **665**
Identifier: 665
Since: 665

Configuration Duplication Maps **667**
Identifier: 667
Since: 667

Feature Type Factory **669**
Identifier: 669
Description: 669

Global Install Handlers **671**
Identifier: 671
Description: 671

Table of Contents

Site Type Factory **673**
Identifier: 673
Description: 673

Runtime overview **675**
The runtime plug−in model 675
Plug−ins and bundles 675
Extension points and the registry 677
Contributing content types 678
Providing a new content type 678
Detecting and describing content 678
Extending an existing content type 679

OSGi Bundle Manifest Headers **681**
Eclipse Bundle Manifest Headers 681
Additional Export−Package Directives 681
The Eclipse−AutoStart Header 682
The Eclipse−PlatformFilter Header 682
The Eclipse−BuddyPolicy Header 683
The Eclipse−RegisterBuddy Header 683
The Eclipse−ExtensibleAPI Header 683
The Plugin−Class Header 683
Concurrency infrastructure 684
Jobs 684
Common job operations 685
Job states 685
Job change listeners 686
The job manager 686
Job families 686
Reporting progress 687
Progress monitors and the UI 687
Progress groups 688
Workbench concurrency support 689
Progress service 690
Showing that a part is busy 691
Progress Properties for Jobs 691
Workbench jobs 692
Long−running operations 692
Runnables and progress 692
Modal operations 693
Job scheduling 693
Locks 694
Workbench under the covers 696
Workbench 696
Page 697
Perspectives 697
Views and editors 697
org.eclipse.ui.views 698

Table of Contents

OSGi Bundle Manifest Headers
Viewers....700
Standard viewers....701
Viewer architecture....702
Viewers and the workbench....703
org.eclipse.ui.viewActions....704
org.eclipse.ui.editors....705
Contributing new retargetable actions....708
Content outliners....708
Text editors and platform text....710
org.eclipse.ui.editorActions....711
org.eclipse.ui.popupMenus....712
org.eclipse.ui.actionSets....714
Application dialogs....717
The plug−in class....719
Plug−in definition....719
AbstractUIPlugin....719
Preference pages....720
Contributing a preference page....720
Implementing a preference page....721
Field editors....725

Dialogs and wizards....726
Standard dialogs....726
Dialog settings....726
Wizards....727
Wizard dialog....727
Wizard....728
Wizard page....728
org.eclipse.ui.newWizards....728
org.eclipse.ui.importWizards....733
org.eclipse.ui.exportWizards....734
Wizard dialogs....735
Multi−page wizards....736
Validation and page control....736
Actions and contributions....737
Actions....737
Contribution items....738
Contribution managers....738
User interface resources....738
Image descriptors and the registry....739
Plug−in patterns for using images....740
ResourceManager....741
Font registry....741
JFaceResources....742
Widgets....742
Widget application structure....742
SWT standalone example − Hello World....744

Table of Contents

Dialogs and wizards
Hello World 1....744
Running the example....745
SWT standalone examples setup....745
Adding SWT to your workspace....745
Importing example source....745
Running the Example....746
Examples Overview....746
SWT standalone example – Address Book....746
Running the example....746
SWT Example Launcher....747
Running the Example Launcher....747
SWT example – Browser....747
Running the example....747
SWT standalone example – Clipboard....747
Running the example....747
SWT example – Controls....748
Running the example....748
SWT example – Custom Controls....748
Running the example....748
SWT standalone example – Drag and Drop....748
Running the example....748
SWT standalone example – File Viewer....749
Running the example....749
SWT standalone example – Hover Help....749
Running the example....749
SWT standalone example – Image Analyzer....749
Running the example....750
SWT standalone example – Java Syntax Viewer....750
Running the example....750
SWT example – Layouts....750
Running the example....750
SWT example – Paint Tool....751
Running the example....751
SWT standalone example – Text Editor....751
Running the example....751
Controls....751
Events....753
Custom widgets....755
Layouts....756
Custom layouts....758
Error handling....758
IllegalArgumentException....758
SWTException....758
SWTError....759
Graphics....759
Graphics context....759
Fonts....759

Table of Contents

Dialogs and wizards
Colors....760
Images....760
Graphics object lifecycle....760

UI Forms....762

UI Forms controls....763

Form control....764

Hyperlink control....765

Expandable composite and Section controls....766

FormText control....767
Rendering normal text (label mode)....767
Automatic conversion of URLs into hyperlinks....767

Parsing formatting markup....768

UI Forms Layouts....769

TableWrapLayout....770

ColumnLayout....771

Color and font management....772

Managed forms....773

Master/Details block....774

Multi−page form editors....775
Recommended practices for Eclipse Forms multi−page editors....775
Menu and toolbar paths....776
Menu paths....776
Tool bar paths....779
Using paths from another plug−in....781
Action set part associations....781
Boolean expressions and action filters....782
Boolean enablement expressions....782
Using objectState with content types....784
Retargetable actions....784
Setting a global action handler....786
Undoable operations....791
Writing an undoable operation....791
Operation history....792

Table of Contents

Multi−page form editors
Undo and redo action handlers....795
Application undo models....795
Undo and the IDE Workbench....796
Perspectives....797
Workbench part layout....797
Linking views and editors with "show−in"....798
org.eclipse.ui.perspectives....799
org.eclipse.ui.perspectiveExtensions....799
Decorators....800
Workbench key bindings....803
Commands....804
Key bindings....806
Schemes....807
Contexts and key bindings....808
Contexts....808
Element factories....810
Accessible user interfaces....812
Assistive technology....812
Accessibility resources....812
SWT and accessibility....812

Tips for making user interfaces accessible....813
Honoring single click support....814
Single click in JFace viewers....815
Single click in SWT controls....815
Activating editors on open....815
Working sets....815
Adding new working set types....816
Contributing resource filters....817
Filtering large user interfaces....818
Activities....819
Activities vs. perspectives....819
Guiding the user through tasks....823
Cheat sheets....823
Defining an intro part....827
Contributing a HelloWorld Intro Part....828
Using the CustomizableIntroPart....830
Defining an intro config....830
Defining intro content....831
Using XHTML as intro content....832
Displaying static HTML content in a CustomizableIntroPart....833
Extending an intro config....834
Extending the content of an intro config....834
Contributing a standby content part....835
Defining a custom IntroURL action....836
Workbench resource support....836
Contributing a property page....837

Table of Contents

Tips for making user interfaces accessible
Implementing a property page....838
Marker help and resolution....839
Text file encoding....842

Editors....844
Workbench editors....845
Editor parts and their inputs....845
Resetting the editor input....845
Navigating the editor input....845
Documents and partitions....845
Document providers and documents....846
Syntax coloring....848
Damage, repair, and reconciling....848
Configuring a source viewer....850
Source viewers and annotations....851
Annotations and rulers....851
Text and ruler hover....855
Text hover....855
Ruler hover....856
Content assist....857
Content assist processors....857
Content assist configuration....859
Registering editor actions....859
Editor menu bar actions....859
Editor context menus....861
Menu ids....862
Other text editor responsibilities....863
Preference handling....863
Key bindings....863

Building a help plug−in....865

Table of contents (toc) files....866
toc_Concepts.xml....866
toc_Tasks.xml....866

Help server and file locations....868
National language and translated documentation....868
Cross plug−in referencing....869

Completing the plug−in manifest....870

Building nested documentation structures....872
Top−down nesting....872
Bottom−up composition....872

Table of Contents

Dynamic topics **874**
Content producer example 874
Constraints 874

Context−sensitive help **875**

Declaring a context id **876**

Describing and packaging context−sensitive help content **877**
Context−sensitive help from multiple plug−ins 877

Dynamic creation of context help **879**

Infopops **880**

Information Search **881**

Federated search engine types **882**

Active help **884**

Writing the help action **885**

Invoking the action from HTML **886**

Tips for debugging active help **887**
Test your action ahead of time 887
Ensure the JavaScript is running 887

Search support **889**
Contributing a search page 889
Implementing the search page 890
Contributing a search result page 890

Compare support **891**
Compare viewers 891
Implementing a content viewer 891
Simple content viewers 891
Content merge viewers 892
Implementing a structure viewer 893
Tree−like structure viewers 893
Other hierarchical structure viewers 893
Merging multiple streams 895
Advanced compare techniques 895
Writing compare operations 895
Compare functionality outside of compare editors 896
Rich Team Integration 896
Getting started 898

Table of Contents

Compare support
Enhancing resource views....898
Handling user editing and changes to resources....898
Streamlining repository−related tasks....899
Enhancing platform integration....899
Adding team actions....899
Team decorators....901
Team and linked resources....902
Project sets....903
File types....905
Adding preferences and properties....907
Launching a program....909
Adding launchers to the platform....909
Handling errors from a launched program....914
Launch configuration dialog....915
Launch configuration type images....916
Launch shortcuts....917
Debugging a program....918
Platform debug model....918
Breakpoints....919
Expressions....920
Debug model presentation....921
Debug UI utility classes....921

Platform Ant support....923
Running Ant buildfiles programmatically....923
Special care for native libraries if build occurs within the same JRE as the workspace....923
Ant tasks provided by the platform....923
eclipse.refreshLocal....924
eclipse.incrementalBuild....924
eclipse.convertPath....924
Contributing tasks and types....924
Progress Monitors....925
Important rules when contributing tasks and types....925
Why a separate JAR for tasks and types?....925
Developing Ant tasks and types within Eclipse....926
Expanding the Ant classpath....926

Packaging and delivering Eclipse based products....927
Customizing a primary feature....927
About dialogs....928
Window images....928
Welcome page....928
Splash screens....929
Preferences defaults....929

Table of Contents

About.ini File Format....930
Locale specific files....930
Platform core mechanism....931
Defining NL fragments....932
Plug−ins and fragments....933

Plug−in Archives....934
Product installation guidelines....934
Multi−user issues....935
Uninstall issues....936
Reinstalling the product....936

How to write an Eclipse installer....937
Product installer creation script....937
Uninstaller behavior....941
Installer behavior when product already installed....941
Associating a JRE installed elsewhere....942
Extension installer creation script....942
Uninstaller behavior....944
Installer behavior when extension already installed....945
Product extensions....945
Installing and uninstalling extensions....946
Updating a product or extension....946
Feature and plug−in packaging....947
Update server layout....947
Update servers and policies....947

Update server site map....949
Site Map....949
Default Site Layout....951
Mirrors File....952
Controlling Access....952

Eclipse Update Policy Control....953
2. Update policy to the rescue....953
2.1 Support for creating local (proxy) update sites....953
2.2 Common update policy control....953
2.3 Automatic discovery of updates....955
3. Summary....955
Deploying eclipse based application with Java Web Start....955
Step 1, creating a wrappering feature....955
Step 2, exporting the wrappering feature and the startup.jar....956
Step 3, creating the main jnlp file....956
Plug−ins based application....957
Known limitations....958

Table of Contents

Building a Rich Client Platform application **959**

Eclipse Platform Map of Platform Plug−ins **960**
The browser example 964
Defining a rich client application 966
Customizing the workbench 967
The workbench life−cycle 967
Defining the actions 968
Making UI contributions 969
Adding the perspective 970
Adding views 971

Other Reference Information **972**
Basic platform information 972
User interface information 972
Help information 972
Product install and configuration information 972
Eclipse.org articles index 972

The Eclipse runtime options **973**
Command line arguments 973
Obsolete command line arguments 975
Others 975
System properties 975
Locations 980
More detail 980
Read−only Locations 981
Launcher ini file 981
File format 981

Starting Eclipse from Java **983**

Eclipse platform API rules of engagement **984**
What it means to be API 984
How to tell API from non−API 984
General rules 985
Calling public API methods 985
Instantiating platform API classes 986
Subclassing platform API classes 986
Calling protected API methods 986
Overriding API methods 986
Implementing platform API interfaces 987
Implementing public API methods 987
Accessing fields in API classes and interfaces 987
Casting objects of a known API type 987
Not following the rules 988

Table of Contents

Eclipse Platform Naming Conventions **989**
Java Packages 989
Classes and Interfaces 990
Methods 991
Variables 991
Constants 992
Plug−ins and Extension Points 992

Glossary of terms **993**

The project description file **995**

Message Bundles In Eclipse 3.1 ..... **998**
Description 998
Summary of the new approach: 998
When creating a new message: 998
Example Files: 998
Client Code 998
Messages.java 999
messages.properties 999

Eclipse multi−user installs **1000**
Basic concepts 1000
Locations 1000
Configuration initialization 1000
Scenario #1 − private install 1001
Scenario #2 − shared install 1001
Scenario #3 − shared configuration 1001
Setting the private configuration area location 1002
Updating 1002

Pre−built documentation index **1003**
Building a documentation index for a plug−in 1003
Building an index for a product 1003
Packaging and Installation of the product's pre−built index 1003

Installing the stand−alone help system **1005**
Installation/packaging 1005
How to call the help classes from Java 1005
How to call the help from command line 1006
[Optional] Installing a minimal set of plug−ins 1006

Help System Preferences **1008**
org.eclipse.help plug−in: 1008
org.eclipse.help.base plug−in: 1008
org.eclipse.help.appserver plug−in: 1010
org.eclipse.tomcat plug−in: 1011

Table of Contents

Installing the help system as an infocenter....1012
Installation/packaging....1012
How to start or stop infocenter from command line....1013
Using the infocenter....1013
How to start or stop infocenter from Java....1013
Making infocenter available on the web....1013
Running multiple instance of infocenter....1014
[Optional] Installing a minimal set of plug−ins....1014

Updating a running infocenter from command line....1016

The platform.xml file....1018
Since:....1018
Description:....1018
Configuration Markup:....1018

Running update manager from command line....1021

Bidirectional Support in the Eclipse Workbench....1024
Enabling Bidirectional Support in the SDK....1024
Enabling your plug−in for looking up alternate icons....1025
How to choose icons to override....1025

JFace Preference Stores....1026

Third party libraries and classloading....1027

Eclipse 3.1 Plug−in Migration Guide....1028

Eclipse 3.1 Plug−in Migration FAQ....1029
IPreferenceStore has more explicit API....1029
IWorkbenchWindow#getShell() has more explict API....1029

Incompatibilities between Eclipse 3.0 and 3.1....1030
1. Plug−in Preferences....1030
2. Changes to IPath constraints....1031
3. Extension registry....1032
4. Code formatter options....1032
5. API contract changes to AntCorePreferences....1033
6. API contract changes to Policy class in JFace....1033
7. API contract changes to allow a null default perspective....1033
8. API contract changes to IViewLayout....1033
9. API contract changes to IVMInstall....1034
10. SelectionEnabler.SelectionClass made package−visible....1034
11. ContributionItem.getParent() can return null....1034
12. Changes to isPropertySet(boolean) in IPropertySource and IPropertySource2....1035
13. handlerSubmission element deleted from the org.eclipse.ui.commands extension point....1035
14. Static non−final API field GLOBAL_IGNORES_CHANGED in TeamUI made final....1035

Table of Contents

Incompatibilities between Eclipse 3.0 and 3.1
15. ClassCastException using FillLayout....1036
16. IllegalArgumentException thrown creating a widget with a disposed parent....1036
Changes required when adopting 3.1 mechanisms and APIs....1036
Platform undo/redo support....1036
Platform undo/redo action handlers....1037
Help Enhancements....1038

Eclipse 3.0 Plug−in Migration Guide....1039

Eclipse 3.0 Plug−in Migration FAQ....1040

Incompatibilities between Eclipse 2.1 and 3.0....1042
1. Plug−in manifest version....1042
2. Restructuring of Platform UI plug−ins....1043
3. Restructuring of Platform Core Runtime plug−ins....1044
4. Removal of Xerces plug−in....1045
5. Eclipse 3.0 is more concurrent....1045
6. Opening editors on IFiles....1046
7. Editor goto marker....1047
8. Editor launcher....1047
9. Editor registry....1048
10. Workbench marker help registry....1048
11. Text editor document providers....1049
12. Text editors....1049
13. Headless annotation support....1051
14. Console view....1051
15. Java breakpoint listeners....1051
16. Clipboard access in UI thread....1052
17. Key down events....1052
18. Tab traversal of custom controls....1052
19. Selection event order in SWT table and tree widgets....1052
20. New severity level in status objects....1053
21. Build−related resource change notifications....1053
22. Intermediate notifications during workspace operations....1054
23. URL stream handler extensions....1054
24. Class load order....1055
25. Class loader protection domain not set....1055
26. PluginModel object casting....1056
27. ILibrary implementation incomplete....1056
28 Invalid assumptions regarding form of URLs....1056
29. BootLoader methods moved/deleted....1057
30. Plug−in export does not include the plug−in's JARs automatically....1057
31. Re−exporting runtime API....1057
32. Plug−in parsing methods on Platform....1058
33. Plug−in libraries supplied by fragments....1058
34. Changes to build scripts....1058
35. Changes to PDE build Ant task....1059

Table of Contents

Incompatibilities between Eclipse 2.1 and 3.0
36. Changes to eclipse.build Ant task....1059
37. Changes to eclipse.fetch Ant task....1059
38. Replacement of install.ini....1060
Changes required when adopting 3.0 mechanisms and APIs....1060
Getting off of org.eclipse.core.runtime.compatibility....1060
NL fragment structure....1063
API changes overview....1063
SWT Example Launcher....1078
Running the Example Launcher....1078
SWT example – Browser....1078
Running the example....1078
SWT example – Controls....1079
Running the example....1079
SWT example – Custom Controls....1079
Running the example....1079
SWT example – Layouts....1079
Running the example....1079
SWT example – OLE Web Browser....1080
Running the example....1080
SWT example – Paint Tool....1080
Running the example....1080
Example – Java Editor....1080
Introduction....1080
Features demonstrated in the example editor....1081
Features not demonstrated....1081
Running the example Java editor....1081
Principles for creating custom text editors....1082
Code organization of the example....1083
Example – Template Editor....1083
Introduction....1083
Features demonstrated in the template editor....1083
Features not demonstrated....1083
Running the example Template editor....1083
Code organization of the template editor example....1084
Example – Multi–page Editor....1084
Introduction....1084
Running the example....1084
Details....1084
Example – Property Sheet....1085
Introduction....1085
Running the example....1085
Details....1085
Example – Undo....1085
Introduction....1085
Features demonstrated in the example....1085
Features not demonstrated....1086
Running the example....1086

Table of Contents

Incompatibilities between Eclipse 2.1 and 3.0
Details....1086
Help....1087
Introduction....1087
Running the example....1087
Details....1087
Team – File System Repository Provider Example....1087
Introduction....1087
Running the example....1088
Team – Local History Synchronize Participant Example....1088
Introduction....1088
Running the example....1088
Compare Example – Structural Compare for Key/Value Pairs....1089
Introduction....1089
Running the example....1089
Code organization of the example....1089

Eclipse Platform XML Compare....1090
Installing the plugin....1090
Using the plugin....1090
ID Mapping Schemes....1090
Ordered entries....1090
Defining ID Mapping Schemes and Ordered entries....1091

Extension Points....1092

Tutorial and Examples....1093
General Matching vs. ID Mapping Schemes:How to create an ID Mapping Scheme to improve compare results....1093
Adding Ordered entries....1095

idMapping....1098

Platform questions index....1102
Getting started....1102
Core runtime....1102
Resources....1102
New file types in the UI....1102
Workbench UI....1102
Installation and upgrade....1103
Notices....1103
About This Content....1103
License....1103

Welcome to Eclipse

Welcome to the Eclipse platform!

The following sections discuss the issues and problems with building integrated tool suites, and how the Eclipse tooling platform can help solve these problems.

Who needs a platform?

On any given day, you can probably find an announcement about a strategic alliance, an open architecture, or a commercial API that promises to integrate all your tools, seamlessly move your data among applications, and simplify your programming life.

Down in the trenches, you're trying to apply enough import/export duct tape to let marketing say "suite" with a straight face.

Where is all this integration pressure coming from? Why is everyone trying to integrate their products into suites or build platforms to support open integration? Who needs these platforms?

End users

Let's face it. End users do not call the support line to say, "What I really need is an open tools platform."

But they do ask why your product doesn't integrate with their other tools. They ask for features outside of the scope of your application because they can't get their data to a tool that would do the job better. They run into problems importing and exporting between different programs. They wonder why their programs have completely different user interfaces for doing similar tasks. Doesn't it seem obvious that their web site design tool should be integrated with their scripting program?

Your users want the freedom to pick the best tool for the task. They don't want to be constrained because your software only integrates with a few other programs. They have a job to do, and it's not managing the flow of files and data between their tools. They're busy solving their own problems. It's your job to make the tools work, and even better if you can make them work together.

Software developers

Meanwhile, you are slaving on your tool implementing the next round of critical features, fixing bugs, and shipping releases. The last thing you need is another emergency import feature added to your list.

Wouldn't it be nice if you could just publish enough hooks to make integrating with your tool everyone else's problem? Unfortunately, unless you work for one of the giants, you just don't have enough clout to get away with that.

The holy grail

What we all want is a level of integration that magically blends separately developed tools into a well designed suite. And it should be simple enough that existing tools can be moved to the platform without using a shoehorn or a crowbar.

The platform should be open, so that users can select tools from the best source and know that their supplier has a voice in the development of the underlying platform.

It should be simple to understand, yet robust enough to support integration without a lot of extra glue.

It should provide tools that help automate mundane tasks. It should be stable enough so that industrial strength tools can build on top of it. And it should be useful enough that the platform developers can use it to build itself.

These are all goals of Eclipse. The remainder of this programming guide will help you determine how close Eclipse has come to delivering on these ideals.

What is Eclipse?

Eclipse is a platform that has been designed from the ground up for building integrated web and application development tooling. By design, the platform does not provide a great deal of end user functionality by itself. The value of the platform is what it encourages: rapid development of integrated features based on a **plug−in** model.

Eclipse provides a common user interface (UI) model for working with tools. It is designed to run on multiple operating systems while providing robust integration with each underlying OS. Plug−ins can program to the Eclipse portable APIs and run unchanged on any of the supported operating systems.

At the core of Eclipse is an architecture for dynamic discovery, loading, and running of plug−ins. The platform handles the logistics of finding and running the right code. The platform UI provides a standard user navigation model. Each plug−in can then focus on doing a small number of tasks well. What kinds of tasks? Defining, testing, animating, publishing, compiling, debugging, diagramming...the only limit is your imagination.

Open architecture

The Eclipse platform defines an open architecture so that each plug−in development team can focus on their area of expertise. Let the repository experts build the back ends and the usability experts build the end user tools. If the platform is designed well, significant new features and levels of integration can be added without impact to other tools.

The Eclipse platform uses the model of a common workbench to integrate the tools from the end user's point of view. Tools that you develop can plug into the workbench using well defined hooks called **extension points**.

The platform itself is built in layers of plug−ins, each one defining extensions to the extension points of lower−level plug−ins, and in turn defining their own extension points for further customization. This extension model allows plug−in developers to add a variety of function to the basic tooling platform. The artifacts for each tool, such as files and other data, are coordinated by a common platform resource model.

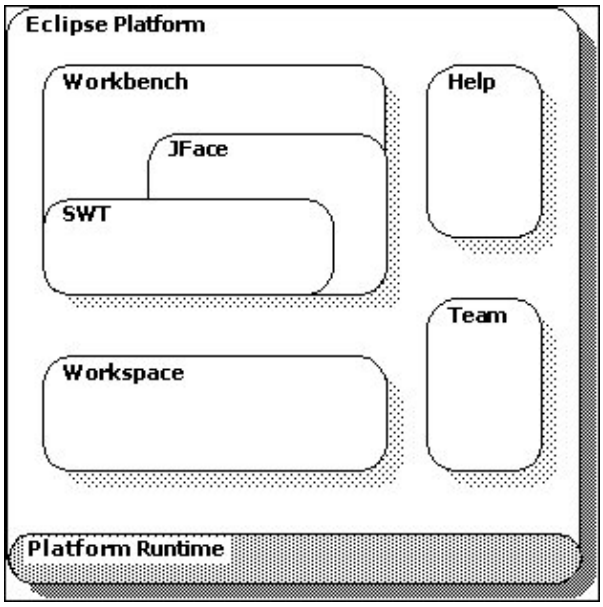
The platform gives the users a common way to work with the tools, and provides integrated management o f the resources they create with plug−ins.

Plug−in developers also gain from this architecture. The platform manages the complexity of different runtime environments, such as different operating systems or workgroup server environments. Plug−in

developers can focus on their specific task instead of worrying about these integration issues.

Platform structure

The Eclipse platform itself is structured as subsystems which are implemented in one or more plug−ins. The subsystems are built on top of a small runtime engine. The figure below depicts a simplified view.



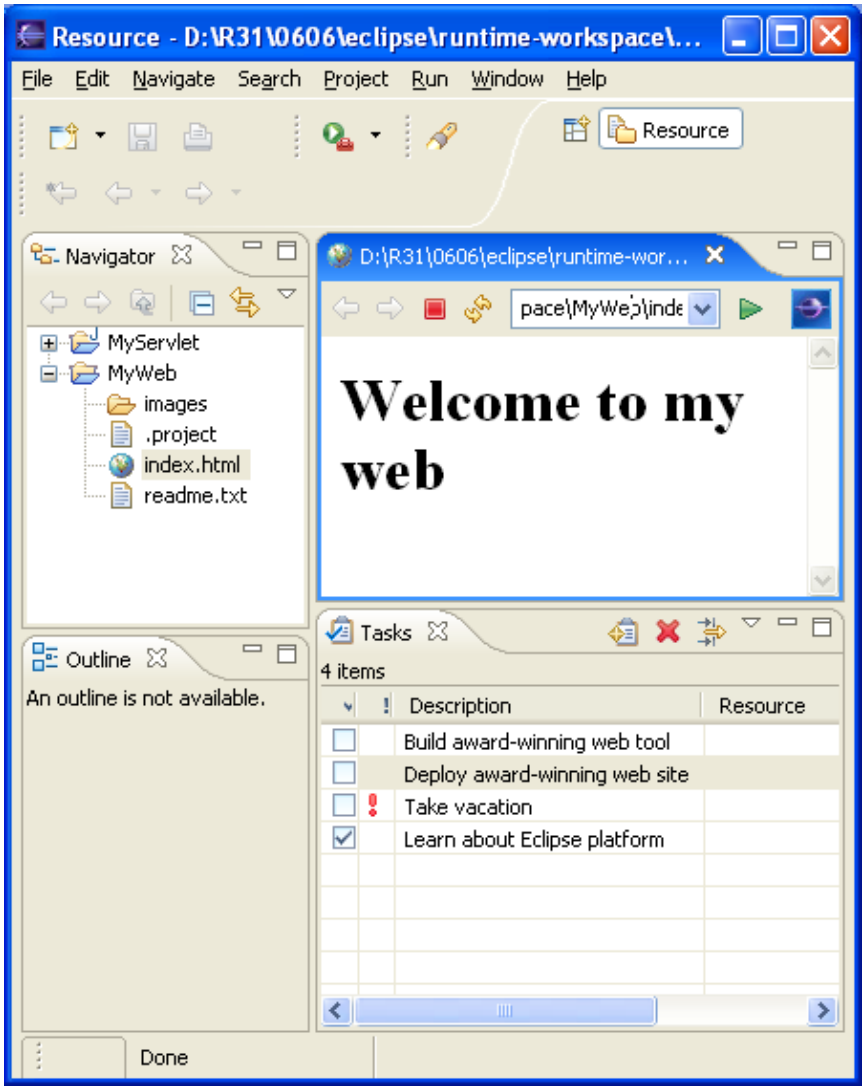
The plug−ins that make up a subsystem define extension points for adding behavior to the platform. The following table describes the major runtime components of the platform that are implemented as one or more plug−ins.

Platform runtime	Defines the extension point and plug−in model. It dynamically discovers plug−ins and maintains information about the plug−ins and their extension points in a platform registry. Plug−ins are started up when required according to user operation of the platform. The runtime is implemented using the OSGi framework.
Resource management (workspace)	Defines API for creating and managing resources (projects, files, and folders) that are produced by tools and kept in the file system.
Workbench UI	Implements the user cockpit for navigating the platform. It defines extension points for adding UI components such as views or menu actions. It supplies additional toolkits (JFace and SWT) for building user interfaces. The UI services are structured so that a subset of the UI plug−ins can be used to build rich client applications that are independent of the resource management and workspace model. IDE−centric plug−ins define additional function for navigating and manipulating resources.
Help system	Defines extension points for plug−ins to provide help or other documentation as browsable books.
	Defines a team programming model for managing and versioning resources.

Team support	
Debug support	Defines a language independent debug model and UI classes for building debuggers and launchers.
Other utilities	Other utility plug–ins supply function such as searching and comparing resources, performing builds using XML configuration files, and dynamically updating the platform from a server.

Out of the box

Out of the box – or off the web – the basic platform is an integrated development environment (IDE) for anything (and nothing in particular).



It's the plug–ins that determine the ultimate functionality of the platform. That's why the Eclipse SDK ships with additional plug–ins to enhance the functionality of the SDK.

Your plug–ins can provide support for editing and manipulating additional types of resources such as Java files, C programs, Word documents, HTML pages, and JSP files.

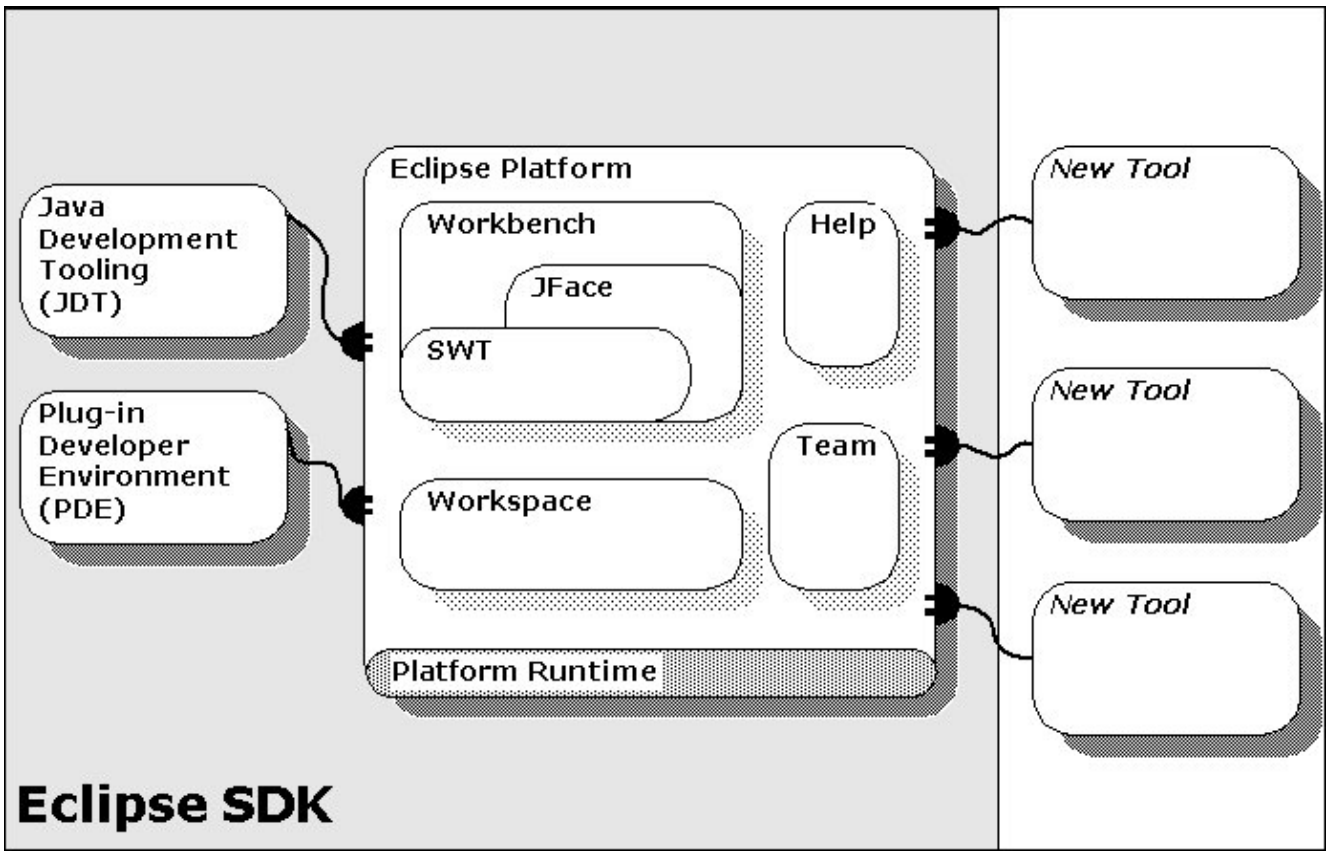
Platform architecture

The Eclipse platform is structured around the concept of **plug−ins**. Plug−ins are structured bundles of code and/or data that contribute function to the system. Function can be contributed in the form of code libraries (Java classes with public API), platform **extensions**, or even documentation. Plug−ins can define **extension points**, well−defined places where other plug−ins can add functionality.

Each subsystem in the platform is itself structured as a set of plug−ins that implement some key function. Some plug−ins add visible features to the platform using the extension model. Others supply class libraries that can be used to implement system extensions.

The Eclipse SDK includes the basic platform plus two major tools that are useful for plug−in development. The Java development tools (JDT) implement a full featured Java development environment. The Plug−in Developer Environment (PDE) adds specialized tools that streamline the development of plug−ins and extensions.

These tools not only serve a useful purpose, but also provide a great example of how new tools can be added to the platform by building plug−ins that extend the system.



Platform SDK roadmap

Runtime core

The platform runtime core implements the runtime engine that starts the platform base and dynamically discovers and runs plug−ins. A **plug−in** is a structured component that describes itself to the system using an OSGi manifest (**MANIFEST.MF**) file and a plug−in manifest (**plugin.xml**) file. The platform maintains a registry of installed plug−ins and the function they provide.

A general goal of the runtime is that the end user should not pay a memory or performance penalty for plug−ins that are installed, but not used. A plug−in can be installed and added to the registry, but the plug−in will not be activated unless a function provided by the plug−in has been requested according to the user's activity.

The platform runtime is implemented using the OSGi services model. While implementation details of the runtime may not be important to many application developers, those already familiar with OSGi will recognize that an Eclipse plug−in is, in effect, an OSGi bundle.

The best way to get a feel for the runtime system is to build a plug−in. See Plug it in: Hello World meets the workbench to get started building a plug−in. To understand the nuts and bolts of the runtime system, see Runtime overview.

Resource management

The resource management plug−in defines a common resource model for managing the artifacts of tool plug−ins. Plug−ins can create and modify **projects**, **folders**, and **files** for organizing and storing development artifacts on disk.

Resources overview provides an overview of the resource management system.

Workbench UI

The workbench UI plug−in implements the workbench UI and defines a number of extension points that allow other plug−ins to contribute menu and toolbar actions, drag and drop operations, dialogs, wizards, and custom views and editors.

Plugging into the workbench introduces the workbench UI extension points and API.

Additional UI plug−ins define frameworks that are generally useful for user interface development. These frameworks were used to develop the workbench itself. Using the frameworks not only eases the development of a plug−in's user interface, but ensures that plug−ins have a common look and feel and a consistent level of workbench integration.

The Standard Widget Toolkit (SWT) is a low−level, operating system independent toolkit that supports platform integration and portable API. It is described in Standard Widget Toolkit.

The JFace UI framework provides higher−level application constructs for supporting dialogs, wizards, actions, user preferences, and widget management. The functionality in JFace is described in Dialogs and wizards and JFace: UI framework for plug−ins.

Team support

The Team plug−ins allow other plug−ins to define and register implementations for team programming, repository access, and versioning. The Eclipse SDK includes a CVS plug−in that uses the team support to provide CVS client support in the SDK.

Team support is described in Team support.

Debug support

The Debug plug−ins allow other plug−ins to implement language specific program launchers and debuggers.

Debug support is described in Program debug and launching support.

Help System

The Help plug−in implements a platform optimized help web server and document integration facility. It defines extension points that plug−ins can use to contribute help or other plug−in documentation as browsable books. The documentation web server includes special facilities to allow plug−ins to reference files by using logical, plug−in based URLs instead of file system URLs.

Additional features are provided for integrating help topics in product level documentation configurations.

The help facility is described in Plugging in help.

Java Development Tools (JDT)

The Java development tools (JDT) plug−ins extend the platform workbench by providing specialized features for editing, viewing, compiling, debugging, and running Java code.

The JDT is installed as a set of plug−ins that are included in the SDK. The Java Development User Guide describes how to use the Java tools. The JDT Plug−in Developer Guide describes the structure and API of the JDT.

Plug−in Development Environment (PDE)

The Plug−in Development Environment (PDE) supplies tools that automate the creation, manipulation, debugging, and deploying of plug−ins.

The PDE is installed as a set of plug−ins that are included in the SDK. The PDE Guide describes how to use the environment.

Team support

The Eclipse Team support defines API that allow plug−ins to integrate the function of a versioning and configuration management repository. The function provided by a repository fundamentally affects the user workflow, since there are additional steps for retrieving files, comparing their content with local content, versioning them, and returning updated files to the repository. The goal of the team plug−in API is to be passive enough to allow repository plug−in providers to define their own workflow so that users familiar with their product can use the platform in a similar fashion and provide support for worflows that we have found are useful for team plug−ins.

This goal is accomplished by supplying several building blocks:

- Repository Providers

  A repository provider allows synchronization of workspace resources with a remote location. At a minimum it allows pushing resources in the workspace to a remote location and pulling resources from a remote location into the workspace. A repository provider is associated with a project and controls the resources in the project by optionally providing a IFileModificationValidator and IMoveDeleteHook. There is only one repository provider associated with each project. A user associates a repository provider with a project by providing a IConfigurationWizard. Repository providers can also participate in exporting and importing of projects into the workspace via the team project set feature. To support this a repository provider should implement a ProjectSetCapability.
- Resource Management

  Allows other plug−ins to indicate special handling of resources with respect to team operations. The repository provider can mark resources as team−private which essentially hides the resource from other plug−ins. This is done via the IResource#setTeamPrivateMember method and is commonly done to hide repository provider specific meta−files from the user. Also, builders will often mark build output as derived which is a hint to a repository provider that the resource is transient and could be ignored by the repository provider. A provider can check this flag on a resource via the IResource#isDerived method.

  In addition, other plug−ins can add provide hints to the repository provider about file type information via the org.eclipse.team.core.fileTypes extension and about common files that should be ignored by the repository via the org.eclipse.team.core.ignore extension.

- Synchronization Support [**new in 3.0**]

  Synchronize support provides classes and interfaces for managing dynamic collections of synchronization information (SyncInfo, SyncInfoSet). This support helps you manage information about variants of the resources in the workspace. For example, with FTP you could store timestamps for the latest remote file and the base for the currently loaded resource. Synchronization support provides APIs to help manage and persist resource variants and display synchronization state to the user.

The UI support is also structured passively. Placeholders for team provider actions, preferences, and properties are defined by the team UI plug−in, but it's up to the team plug−in provider to define these UI elements. The team UI plug−in also includes a simple, extendable configuration wizard that lets users associate projects with repositories. Plug−ins can supply content for this wizard that let the user specify repository specific information.

Multiple repository providers can coexist peacefully within the platform. In fact, it's even possible to have different client implementations for the same repository installed. For example, one could install a CVS client designed for experts and a different one for novice users.

Repository providers

A repository provider (**RepositoryProvider**) is the central class in the implementation of your repository. This class is responsible for configuring a project for repository management and providing the necessary hooks for resource modification. Providers are mapped to a project using project persistent properties. The mechanism for mapping providers to a project is not central to the team API, but you'll need to be aware of it when filtering out resources in your UI. For the most part, you'll be using team API to work with projects and associate them to your provider.

To implement a provider, you must define a repository using **org.eclipse.team.core.repository** and supply a class derived from **RepositoryProvider**. We'll use the CVS client as an example to see how this works.

Extension point

The **org.eclipse.team.core.repository** extension point is used to add a repository definition. Here is the markup for the CVS client.

```
<extension
    point="org.eclipse.team.core.repository">
      <repository
            class="org.eclipse.team.internal.ccvs.core.CVSTeamProvider"
            id="org.eclipse.team.cvs.core.cvsprovider">
      </repository>
</extension>
```

This registers your team provider with the team support plug−in and assigns an id that should be used when your provider is associated with a project. The specified **class** for the repository must extend **RepositoryProvider**.

Implementing a RepositoryProvider

The class identified in the extension must be a subclass of **RepositoryProvider**. Its primary responsibilities are to configure and deconfigure a project for repository support, and supply any necessary resource modification hooks. The CVS client serves as a good example. Its repository provider is **CVSTeamProvider**.

```
public class CVSTeamProvider extends RepositoryProvider {
...
```

RepositoryProvider defines two abstract methods, **configureProject** and **deconfigure**. All providers must

implement these methods.

A project is configured when it is first associated with a particular repository provider. This typically happens when the user selects a project and uses the team wizards to associate a project with your repository. Regardless of how the operation is triggered, this is the appropriate time to compute or cache any data about the project that you'll need to provide your repository function. (Assume that mapping the project to your provider has already happened. You'll be taking care of this in your configuration wizard.)

The CVS provider simply broadcasts the fact that a project has been configured:

```
public void configureProject() throws CoreException {
        CVSProviderPlugin.broadcastProjectConfigured(getProject());
}
```

We won't follow the implementation of the plug−in broadcast mechanism. Suffice to say that any parties that need to compute or initialize project specific data can do so at this time.

A project is deconfigured when the user no longer wants to associate a team provider with a project. It is up to your plug−in to implement the user action that causes this to happen (and unmapping the project from your team provider will happen there). The **deconfigure** method is the appropriate time to delete any project related caches or remove any references to the project in the UI. The CVS provider flushes project related caches kept in its views and broadcasts the fact that the project is deconfigured.

```
public void deconfigure() throws CoreException {
        ...
        try {
                EclipseSynchronizer.getInstance().flush(getProject(), true, true /*flush deep*/,
        } catch(CVSException e) {
                throw new CoreException(e.getStatus());
        } finally {
                CVSProviderPlugin.broadcastProjectDeconfigured(getProject());
        }
}
```

Configuring a project

Typically, the first step in building a team UI is implementing a wizard page that allows users to configure a project for your plug−in's team support. This is where your team provider's id will be added to the project's properties. You participate in project configuration by contributing to the **org.eclipse.team.ui.configurationWizards** extension point. This wizard is shown when the user chooses **Team−>Share Project...**

We'll look at this in the context of the CVS client implementation. Here is the CVS UI markup for its configuration wizard:

```
<extension
      point="org.eclipse.team.ui.configurationWizards">
   <wizard
        name="%SharingWizard.name"
        icon="icons/full/wizards/newconnect_wiz.png"
        class="org.eclipse.team.internal.ccvs.ui.wizards.SharingWizard"
        id="org.eclipse.team.ccvs.ui.SharingWizard">
   </wizard>
</extension>
```

As usual, plug−ins supply a **class** that implements the extension and a unique **id** to identify their extension. The **name** and **icon** are shown in the first page of the project configuration wizard if there are multiple providers to choose from.

Once the user has selected a provider, the next page shows the specific configuration information for your provider. (If your provider is the only team provider plug−in installed, then the wizard skips directly to your page.) Your wizard must implement **IConfigurationWizard**, which initializes the wizard for a specified workbench and project. The rest of the implementation depends on the design of your wizard. You must gather up any information needed to associate the project with your team support.

When the wizard is completed, you must map your team provider to the project using **RepositoryProvider.map(IProject, String)**. Mapping handles the assignment of the correct project persistent property to your project.

The CVS client does this work in its provider's **setSharing** method, which is called when its wizard is finished:

```
public void setSharing(IProject project, FolderSyncInfo info, IProgressMonitor monitor) throws Te

        // Ensure provided info matches that of the project
        ...
        // Ensure that the provided location is managed
        ...
        // Register the project with Team
        RepositoryProvider.map(project, CVSProviderPlugin.getTypeId());
}
```

Finding a provider

Static methods in **RepositoryProvider** make it easy for clients to map projects to providers and to find the providers associated with a given project.

- **map(IProject, String) −** instantiates a provider of the specified provider id and maps the specified project to it. This call sets the proper project persistent property on the project.
- **unmap(IProject, String) −** removes the association of the specified provider id from the specified project. Leaves the project unassociated with any team provider.
- **getProvider(IProject) −** answers the provider for a given project. Can be used to find any team provider for a project.
- **getProvider(IProject, String) −** answers the provider for a given project with the specified provider id. Can be used to check whether a particular team provider type is associated with a given project. It is commonly used by providers to quickly check whether a given project is under their care. This call is safer for clients since it does not return a provider that does not match the client's id.

Repository Providers and Capabilities

If a product chooses to add a Repository plug−in to a capability, it should bind the capability to the repository id. Here are the two steps to take to enable a RepositoryProvider as a capability:

1. Bind the capability to the repository provider id. This allows the Team plug−in to activate/disable based on repository provider ids.

   ```
   <activityPatternBinding
   ```

```
    activityId="org.eclipse.team.cvs"
    pattern="org\.eclipse\.team\.cvs\.core/.*cvsnature">
</activityPatternBinding>
```

2. Next bind the capability to all UI packages for the provider:

```
<activityPatternBinding
     activityId="org.eclipse.team.cvs"
     pattern="org\.eclipse\.team\.cvs\.ui/.*">
</activityPatternBinding>
```

There are two capability triggers points defined by the Team plug−ins. The first is the Team > Share Project... wizard which allows filtering of repository providers based on the enabled/disabled state of workbench capabilities, and the other is the Team plug−in auto−enablement trigger.

Resource modification hooks

Most of the interesting function associated with a repository provider occurs as the user works with resources in the project that is configured for the provider. In order to be aware of changes the user makes to a resource, the provider can implement resource modification hooks. The resources plug−in provides these hooks as extension points. The documentation for **IMoveDeleteHook**, **IFileModificationValidator** and ***ResourceRuleFactory*** describe the details for implementing these hooks.

The team plug−in optimizes and simplifies the association of the hook with appropriate resources by registering generic hooks with the resources plug−in. These generic hooks simply look up the repository provider for a given resource and obtain its hook. This has the advantage of calling only one provider hook rather than having each provider implementation register a hook that must first check whether the resource is managed by the provider.

What this means to your plug−in is that you provide any necessary hooks by overriding methods in **RepositoryProvider**. The default implementation of these methods answers null, indicating that no hook is necessary (except for the resource rule factory, as described below):

- **getMoveDeleteHook** − answers an **IMoveDeleteHook** appropriate for the provider. This hook allows providers to control how moves and deletes occur and includes the ability to prevent them from happening. Implementors can provide alternate implementations for moving or deleting files, folders, and projects. The CVS client uses this hook to monitor folder deletions and ensure that any files contained in deleted folders are remembered so that they can later be deleted from the repository if desired.
- **getFileModificationValidator** − answers an **IFileModificationValidator** appropriate for the provider. This hook allows providers to pre−check any modifications or saves to files. This hook is typically needed when a repository provider wants to implement **pessimistic versioning**. In pessimistic versioning, a file must be checked out before modifying it, and only one client can check out a file at any given time. Pessimistic versioning could be implemented by checking out a file (if not already checked out) whenever a file is edited, and checking the file back in when it is saved. Since CVS uses an optimistic versioning scheme, it does not implement this hook.
- ***getRuleFactory*** − answers a resource rule factory appropriate for the provider. Providers should always override this method as the default factory locks the workspace for all operations for backwards compatibility reasons. Provides should subclass ***ResourceRuleFactory*** and override those rules required to ensure that the proper rules are obtained for operations that invoke the move/delete hook and file modification validator. The rule methods of particular interest to repository providers are:
    - *deleteRule* − move/delete hook

- *moveRule* –move/delete hook
- *validateEditRule* – file modification validator validateEdit
- *modifyRule* – file modification validator validateSave

Resource properties

Resources have properties that can be used to store meta–information about the resource. Your plug–in can use these properties to hold information about a resource that is specific to your purpose. Resource properties are declared, accessed, and maintained by various plug–ins, and are not interpreted by the platform. When a resource is deleted from the workspace, its properties are also deleted.

There are two kinds of resource properties:

- **Session properties** allow your plug–ins to easily cache information about a resource in key–value pairs. The values are arbitrary objects. These properties are maintained in memory and lost when a resource is deleted from the workspace, or when the project or workspace is closed.
- **Persistent properties** are used to store resource–specific information on disk. The value of a persistent property is an arbitrary string. Your plug–in decides how to interpret the string. The strings are intended to be short (under 2KB). Persistent properties are stored on disk with the platform metadata and maintained across platform shutdown and restart.

If you follow the convention of qualifying property key names with the unique id of your plug–in, you won't have to worry about your property names colliding with those of other plug–ins.

If your plug–in needs to store persistent information about a project that is much larger than 2 KB, then these properties should be exposed as resources in their own right, rather than using the persistent properties API.

See **IResource** for a description of the API for getting and setting the different kinds of resource properties.

Team Repository Provider

Identifier:

org.eclipse.team.core.repository

Since:

2.0

Description:

The Team plugin contains the notion of Repositories. The job of a repository is to provide support for sharing resources between Team members. Repositories are configured on a per−project basis. Only one repository can be mapped to a project at a time.

Repositories that extend this extension point can provide implementations for common repository specific rules for resource modifications, moving and deleting. See the following interfaces for more details `IFileModificationValidator` and `MoveDeleteHook`.

A Repository type can also be specified in order to provide non−project specific funtionality such as a `org.eclipse.team.core.ProjectSetCapability`.

Optionally, a repository provider type can designate that it can import projects from second provider, in the case where the second provider's plugin is not available in the current install. This is provided as a means to support the migration from one provider implementation to another where the resuse of the same id for the two providers was not possible.

A repository provider type can also specify one or more meta−file paths (delimited by comas) that are relative to a parent container. If an unsghared project or folder contains files that match all the meta−file paths associated with a repository definition, the method `RepositoryProviderType#metaFilesDetected` will be invoked with the parent container as an argument. This is done to give the repository type a chance to mark the files team−private and potentially share the project as well. see the javadoc of the above mentioned method for more details.

Configuration Markup:

<!ELEMENT extension (repository)>

<!ATTLIST extension

point CDATA #REQUIRED>

<!ELEMENT repository EMPTY>

<!ATTLIST repository

id CDATA #IMPLIED

class CDATA #REQUIRED

typeClass CDATA #IMPLIED

canImportId CDATA #IMPLIED

metaFilePaths CDATA #IMPLIED>

- **id** – an optional identifier of the extension instance
- **class** – the fully–qualified name of a subclass of `org.eclipse.team.core.RepositoryProvider`.
- **typeClass** – the fully–qualified name of a subclass of `org.eclipse.team.core.RepositoryProviderType`.
- **canImportId** –
- **metaFilePaths** –

Examples:

<extension point=

"org.eclipse.team.core.repository"

>

<repository class=

"org.eclipse.myprovider.MyRepositoryProvider"

typeClass=

"org.eclipse.myprovider.MyRepositoryProvider"

id=

"org.eclipse.myprovider.myProviderID"

canImportId=

"org.eclipse.myprovider.myOldProviderID"

metaFilePaths=

".meta/files,.meta/version"

>

</repository>

Since:

</extension>

API Information:

The value of the class attribute must represent a subclass of `org.eclipse.team.core.RepositoryProvider` and the value of the typeClass attribute must represent a subclass of `org.eclipse.team.core.RepositoryProviderType`

Supplied Implementation:

The provided implementation of RepositoryProvider provides helper methods and common code for mapping and unmapping providers to projects. The optional RepositoryProviderType provides project set import and export through a ProjectSetCapability.

Configuration Wizards

Identifier:

org.eclipse.team.ui.configurationWizards

Description:

This extension point is used to register a method for configuration of a project. Configuration involves the association of a project with a team provider, including all information necessary to initialize that team provider, including such things as username, password, and any relevant information necessary to locate the provider.

Providers may provide an extension for this extension point, and an implementation of `org.eclipse.team.ui.IConfigurationWizard` which gathers the necessary information and configures the projects.

Configuration Markup:

<!ELEMENT extension (wizard?)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

<!ELEMENT wizard EMPTY>

<!ATTLIST wizard

name CDATA #REQUIRED

icon CDATA #REQUIRED

class CDATA #REQUIRED

id CDATA #REQUIRED>

- **name –** The name of the configuration type as it should appear in the configuration wizard. Examples are "CVS", "WebDAV".
- **icon –** the icon to present in the configuration wizard next to the name.
- **class –** a fully qualified name of the Java class implementing `org.eclipse.team.ui.IConfigurationWizard`.
- **id –** a unique identifier for this extension.

Examples:

Following is an example of a configuration wizard extension:

<extension point=

"org.eclipse.team.ui.configurationWizards"

>

<wizard name=

"WebDAV"

icon=

"webdav.gif"

class=

"com.xyz.DAVDecorator"

id=

"com.xyz.dav"

>

</wizard>

</extension>

API Information:

The value of the `class` attribute must represent a class that implements `org.eclipse.team.ui.IConfigurationWizard`. This interface supports configuration of a wizard given a workbench and a project.

Supplied Implementation:

The plug−in org.eclipse.team.provider.examples.ui contains sample implementations of IConfigurationWizard for the WebDAV and filesystem provider types.

Resource modification hooks

So far, we've assumed that resource API is being used to modify resources that are located in the user's file system. This is indeed the fundamental structure of the workspace, but it's also possible that a plug−in adds capabilities for manipulation of resources that are managed somewhere else. For example, the platform Team support plug−ins add the ability to work with resources that are under the management of a versioning repository.

The resource API includes capabilities that have been added specifically to enable the team support plug−ins and plug−ins that implement repository providers using the team support. The following discussion covers the generic mechanism for registering resource hooks. See Implementing a repository provider for a discussion of how team uses these hooks.

Resource move/delete hooks

This hook allows the team plug−in and its providers to control how resource moves and deletes are implemented. The hook includes the ability to prevent these operations from happening. Implementors can provide alternate implementations for moving or deleting files, folders, and projects.

The team plug−in uses the **org.eclipse.core.resources.moveDeleteHook** extension point to register its hook:

```
<extension point="org.eclipse.core.resources.moveDeleteHook" id="MoveDeleteHook">
        <moveDeleteHook class="org.eclipse.team.internal.core.MoveDeleteManager"/>
</extension>
```

The supplied class must implement **IMoveDeleteHook**, which is called by the platform whenever a resource is moved or deleted. The team plug−in installs a move delete hook manager that can determine which team provider is managing a resource and invoke its specific hook.

File modification validators

It's also possible that team repository providers will need to prevent or intervene in the editing or saving of a file. The team plug−in accomplishes this by using the extension point **org.eclipse.core.resources.fileModificationValidator** to register a validator that is called whenever a resource is to be modified.

```
<extension point="org.eclipse.core.resources.fileModificationValidator" id="FileValidator">
        <fileModificationValidator class="org.eclipse.team.internal.core.FileModificationValidato
</extension>
```

The supplied class must implement **IFileModificationValidator**, which is called by the platform whenever a resource is saved or opened. The team plug−in installs a file modification manager that can determine which team provider is managing a resource and invoke its specific validator.

General team hook

Repository providers sometimes need to hook into additional workspace operations in order to impose extra restrictions or customize workspace behavior. The **org.eclipse.core.resources.teamHook** extension point provides some other special functions for team providers. In particular, this hook allows a team provider to decide whether linked folders and files should be allowed for a given project. Some repository systems have

strict rules about the physical layout of projects on disk, and are not able to handle resources linked to arbitrary locations.

The team hook also allows a repository provider to supply a scheduling rule factory that will be used by all workspace operations. Each time an API method is called that modifies the workspace in some way, a scheduling rule is obtained by the workspace. This scheduling rule prevents other threads from modifying those resources during the invocation of the API method. If a repository provider is performing additional work inside a file modification validator or move/delete hook, the provider must also tell the workspace what additional scheduling rules it will need. See the section on resource batching for more details on how the workspace uses scheduling rules.

The supplied class for the team hook must implement **TeamHook**. The team plug−in installs the single team hook that can determine which team provider is managing a resource and invoke its specific hook.

> *Note: All three of these team hooks are designed specifically for use by the team core plug−in. They are not intended for general use. Team providers should not install hooks using these extension points, but instead implement their hooks in their Repository Provider class. See Team resource modification hooks for more information about using these hooks.*

Move/Delete Hook

Identifier:

org.eclipse.core.resources.moveDeleteHook

Since:

2.0

Description:

For providing an implementation of an IMoveDeleteHook to be used in the IResource.move and IResource.delete mechanism. This extension point tolerates at most one extension.

Configuration Markup:

<!ELEMENT extension (moveDeleteHook?)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT moveDeleteHook EMPTY>

<!ATTLIST moveDeleteHook

class CDATA #REQUIRED>

- **class –** the fully−qualified name of a class which implements `org.eclipse.core.resources.team.IMoveDeleteHook`

Examples:

The following is an example of using the moveDeleteHook extension point. (in file plugin.xml)

<extension point=

"org.eclipse.core.resources.moveDeleteHook"

>

<moveDeleteHook class=

"org.eclipse.team.internal.MoveDeleteHook"

/>

</extension>

API Information:

The value of the class attribute must represent an implementation of `org.eclipse.core.resources.team.IMoveDeleteHook`.

Supplied Implementation:

The Team component will generally provide the implementation of the move/delete hook. The extension point should not be used by any other clients.

Team Hook

Identifier:

org.eclipse.core.resources.teamHook

Since:

2.1

Description:

For providing an implementation of a TeamHook that is used for mechanisms available only to team providers. This extension point tolerates at most one extension.

Configuration Markup:

<!ELEMENT extension (teamHook)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT teamHook EMPTY>

<!ATTLIST teamHook

class CDATA #REQUIRED>

- **class –** the fully–qualified name of a class which subclasses `org.eclipse.core.resources.team.TeamHook`

Examples:

The following is an example of using the teamHook extension point. (in file plugin.xml)

<extension point=

"org.eclipse.core.resources.teamHook"

>

<teamHook class=

"org.eclipse.team.internal.TeamHook"

/>

</extension>

API Information:

The value of the class attribute must represent a subclass of `org.eclipse.core.resources.team.TeamHook`.

Supplied Implementation:

The Team component will generally provide the implementation of the team hook. The extension point should not be used by any other clients.

Scheduling rules

Job *scheduling rules* can be used to control when your jobs run in relation to other jobs. In particular, scheduling rules allow you to prevent multiple jobs from running concurrently in situations where concurrency can lead to inconsistent results. They also allow you to guarantee the execution order of a series of jobs. The power of scheduling rules is best illustrated by an example. Let's start by defining two jobs that are used to turn a light switch on and off concurrently:

```
public class LightSwitch {
   private boolean isOn = false;
   public boolean isOn() {
      return isOn;
   }
   public void on() {
      new LightOn().schedule();
   }
   public void off() {
      new LightOff().schedule();
   }
   class LightOn extends Job {
      public LightOn() {
         super("Turning on the light");
      }
      public IStatus run(IProgressMonitor monitor) {
         System.out.println("Turning the light on");
```

```
            isOn = true;
            return Status.OK_STATUS;
         }
      }
      class LightOff extends Job {
         public LightOff() {
            super("Turning off the light");
         }
         public IStatus run(IProgressMonitor monitor) {
            System.out.println("Turning the light off");
            isOn = false;
            return Status.OK_STATUS;
         }
      }
   }
```

Now we create a simple program that creates a light switch and turns it on and off again:

```
LightSwitch light = new LightSwitch();
light.on();
light.off();
System.out.println("The light is on? " + switch.isOn());
```

If we run this little program enough times, we will eventually obtain the following output:

```
Turning the light off
Turning the light on
The light is on? true
```

How can that be? We told the light to turn on and then off, so its final state should be off! The problem is that there was nothing preventing the `LightOff` job from running at the same time as the `LightOn` job. So, even though the "on" job was scheduled first, their concurrent execution means that there is no way to predict the exact execution order of the two concurrent jobs. If the `LightOff` job ends up running before the `LightOn` job, we get this invalid result. What we need is a way to prevent the two jobs from running concurrently, and that's where scheduling rules come in.

We can fix this example by creating a simple scheduling rule that acts as a *mutex* (also known as a *binary semaphore*):

```
class Mutex implements ISchedulingRule {
   public boolean isConflicting(ISchedulingRule rule) {
      return rule == this;
   }
   public boolean contains(ISchedulingRule rule) {
      return rule == this;
   }
}
```

This rule is then added to the two light switch jobs from our previous example:

```
public class LightSwitch {
   final MutextRule rule = new MutexRule();
   ...
   class LightOn extends Job {
      public LightOn() {
         super("Turning on the light");
         setRule(rule);
      }
```

```
        ...
    }
    class LightOff extends Job {
        public LightOff() {
            super("Turning off the light");
            setRule(rule);
        }
        ...
    }
}
```

Now, when the two light switch jobs are scheduled, the job infrastructure will call the `isConflicting` method to compare the scheduling rules of the two jobs. It will notice that the two jobs have conflicting scheduling rules, and will make sure that they run in the correct order. It will also make sure they never run at the same time. Now, if you run the example program a million times, you will always get the same result:

```
Turning the light on
Turning the light off
The light is on? false
```

Rules can also be used independently from jobs as a general locking mechanism. The following example acquires a rule within a try/finally block, preventing other threads and jobs from running with that rule for the duration between invocations of `beginRule` and `endRule`.

```
IJobManager manager = Platform.getJobManager();
try {
    manager.beginRule(rule, monitor);
    ... do some work ...
} finally {
    manager.endRule(rule);
}
```

You should exercise extreme caution when acquiring and releasing scheduling rules using such a coding pattern. If you fail to end a rule for which you have called `beginRule`, you will have locked that rule forever.

Making your own rules

Although the job API defines the contract of scheduling rules, it does not actually provide any scheduling rule implementations. Essentially, the generic infrastructure has no way of knowing what sets of jobs are ok to run concurrently. By default, jobs have no scheduling rules, and a scheduled job is executed as fast as a thread can be created to run it.

When a job does have a scheduling rule, the `isConflicting` method is used to determine if the rule conflicts with the rules of any jobs that are currently running. Thus, your implementation of `isConflicting` can define exactly when it is safe to execute your job. In our light switch example, the `isConflicting` implementation simply uses an identity comparison with the provided rule. If another job has the identical rule, they will not be run concurrently. When writing your own scheduling rules, be sure to read and follow the API contract for `isConflicting` carefully.

If your job has several unrelated constraints, you can compose multiple scheduling rules together using a MultiRule. For example, if your job needs to turn on a light switch, and also write information to a network socket, it can have a rule for the light switch and a rule for write access to the socket, combined into a single rule using the factory method `MultiRule.combine`.

Rule hierarchies

We have discussed the `isConflicting` method on ISchedulingRule, but thus far have not mentioned the `contains` method. This method is used for a fairly specialized application of scheduling rules that many clients will not require. Scheduling rules can be logically composed into hierarchies for controlling access to naturally hierarchical resources. The simplest example to illustrate this concept is a tree−based file system. If an application wants to acquire an exclusive lock on a directory, it typically implies that it also wants exclusive access to the files and sub−directories within that directory. The `contains` method is used to specify the hierarchical relationship among locks. If you do not need to create hierarchies of locks, you can implement the `contains` method to simply call `isConflicting`.

Here is an example of a hierarchical lock for controlling write access to `java.io.File` handles.

```
public class FileLock implements ISchedulingRule {
   private String path;
   public FileLock(java.io.File file) {
      this.path = file.getAbsolutePath();
   }
   public boolean contains(ISchedulingRule rule) {
      if (this == rule)
         return true;
      if (rule instanceof FileLock)
         return path.startsWith(((FileLock) rule).path);
      if (rule instanceof MultiRule) {
         MultiRule multi = (MultiRule) rule;
         ISchedulingRule[] children = multi.getChildren();
         for (int i = 0; i < children.length; i++)
            if (!contains(children[i]))
               return false;
            return true;
         }
      return false;
   }
   public boolean isConflicting(ISchedulingRule rule) {
      if (!(rule instanceof FileLock))
         return false;
      String otherPath = ((FileLock)rule).path;
      return path.startsWith(otherPath) || otherPath.startsWith(path);
   }
}
```

The `contains` method comes into play if a thread tries to acquire a second rule when it already owns a rule. To avoid the possibility of deadlock, any given thread can only own *one* scheduling rule at any given time. If a thread calls `beginRule` when it already owns a rule, either through a previous call to `beginRule` or by executing a job with a scheduling rule, the `contains` method is consulted to see if the two rules are equivalent. If the `contains` method for the rule that is already owned returns `true`, the `beginRule` invocation will succeed. If the `contains` method returns `false` an error will occur.

To put this in more concrete terms, say a thread owns our example `FileLock` rule on the directory at "c:\temp". While it owns this rule, it is only allowed to modify files within that directory subtree. If it tries to modify files in other directories that are not under "c:\temp", it should fail. Thus a scheduling rule is a concrete specification for what a job or thread is allowed or not allowed to do. Violating that specification will result in a runtime exception. In concurrency literature, this technique is known as *two−phase locking*. In a two−phase locking scheme, a process much specify in advance all locks it will need for a particular task, and is then not allowed to acquire further locks during the operation. Two−phase locking eliminates the

hold−and−wait condition that is a prerequisite for circular wait deadlock. Therefore, it is impossible for a system using only scheduling rules as a locking primitive to enter a deadlock.

Batching resource changes

When you need to modify resources in the workspace, it is important to keep in mind that other plug−ins might be working with the same resources. The resources API provides robust mechanisms for keeping plug−ins informed about changes in the workspace, and for making sure that multiple plug−ins do not modify the same resource at the same time. Where possible, your plug−in's modifications to the workspace should be batched in units of work inside a **workspace runnable**. These runnables help to reduce the amount of change notifications generated by changes. They also allow you to declare which part of the workspace is to be modified, so that other plug−ins can be locked out of changing the same part of the workspace.

The protocol for **IWorkspaceRunnable** is fairly simple. A workspace runnable looks just like a long−running operation or platform job. The actual work is done inside a **run** method, with progress reported to the supplied **IProgressMonitor**. Code that manipulates the workspace is performed inside the **run** method.

```
IWorkspaceRunnable myRunnable =
        new IWorkspaceRunnable() {
public void run(IProgressMonitor monitor) throws CoreException {
                        //do the actual work in here
                        ...
                }
}
```

When it is time to the run the code, your plug−in tells the workspace to run the code on its behalf. This way, the workspace can generate any necessary change events and ensure that no two plug−ins are modifying the same resource at the same time. (Even if your plug−in is not using background jobs and the concurrency framework to modify the workspace, other plug−ins may be doing so.)

Scheduling rules and locking

IWorkspace protocol is used to run a workspace runnable. The preferred technique is using the long form of the **run** method which supplies a scheduling rule and specifies how resource change events are broadcast.

Specifying a scheduling rule when running a workspace runnable allows the workspace to determine whether the resource changes will conflict with workspace changes happening in other threads. (See Scheduling rules for an overview of scheduling rules and **ISchedulingRule** protocol.) Fortunately, **IResource** protocol includes the protocol for **ISchedulingRule**, which means that a resource can often be used as a scheduling rule for itself.

Confused? Code can help to clarify this point. Suppose your plug−in is getting ready to modify a bunch of resources in a particular project. It can use the project itself as the scheduling rule for making the changes. The following snippet runs the workspace runnable that we created earlier:

```
IWorkspace workspace = ResourcesPlugin.getWorkspace();
workspace.run(myRunnable, myProject, IWorkspace.AVOID_UPDATE, null);
```

The runnable is passed to the workspace, followed by the project that the code is manipulating. This tells the workspace that all of the changes in the runnable are confined to `myProject`. Any requests by other threads to change `myProject` will be blocked until this runnable completes. Likewise, this call will block if some other thread is already modifying `myProject`. By specifying which part of the resource tree will be

modified by the runnable, you are allowing other threads to continue modifying other portions of the workspace. It is important to be sure that your resource rule matches the work being done inside the runnable. Any attempt to access a resource outside the scope of the scheduling rule will trigger an exception.

The third parameter to the **run** method specifies whether any periodic resource change events should be broadcast during the scope of this call. Using `IWorkspace.AVOID_UPDATE` tells the platform to suppress any resource change events while the runnable is running and to broadcast one event at the end of the changes. During this call, any other runnables created in the runnable will be considered part of the parent batch operation. Resource changes made in those runnables will appear in the parent's resource change notification.

Resource rule factory

In the example above, we assumed that the code inside our runnable only modified resources in a particular project. This made it very easy to specify a scheduling rule for the runnable. In practice, it can be more difficult to compute what parts of the workspace are affected by a particular change. For example, moving a resource from one project to another affects both projects. **IResourceRuleFactory** can be used to help compute an appropriate resource rule for certain kinds of resource changes. You can get a resource rule factory from the workspace itself.

```
IWorkspace workspace = ResourcesPlugin.getWorkspace();
IResourceRuleFactory ruleFactory = workspace.getRuleFactory();
```

The factory can supply rules appropriate for many kinds of operations. If your runnable is moving a resource from one location to another, it can obtain a rule appropriate for this operation:

```
ISchedulingRule movingRule = ruleFactory.moveResource(sourceResource, destinationResource);
workspace.run(myRunnable, movingRule, IWorkspace.AVOID_UPDATE, null);
```

See the javadoc for **IResourceRuleFactory** for the list of available rules. The resources plug−in uses these rules itself to implement most resource operations. Browsing the code that references these rule methods will help demonstrate how they are used in practice.

Multiple rules can be combined using **MultiRule**.

```
ISchedulingRule movingRule = ruleFactory.moveResource(sourceResource, destinationResource);
ISchedulingRule modifyRule = ruleFactory.modifyResource(destinationResource);
workspace.run(myRunnable, MultiRule.combine(movingRule, modifyRule), IWorkspace.AVOID_UPDATE, nul
```

Ignoring the rules

The short form of the **run** method in **IWorkspace** is also available. It is retained for backward compatibility. The short form does not include a rule or an update flag.

```
workspace.run(myRunnable, null);
```

is effectively the same as calling

```
workspace.run(myRunnable, workspace.getRoot(), IWorkspace.AVOID_UPDATE, null);
```

Specifying the workspace root as the scheduling rule will put a lock on the entire workspace until the runnable is finished. This is the most conservative way to perform a workspace update, but it is not very friendly to other concurrency−minded plug−ins.

Repository resource management

Once you have created a **RepositoryProvider**, there are other resource management mechanism that should be understood:

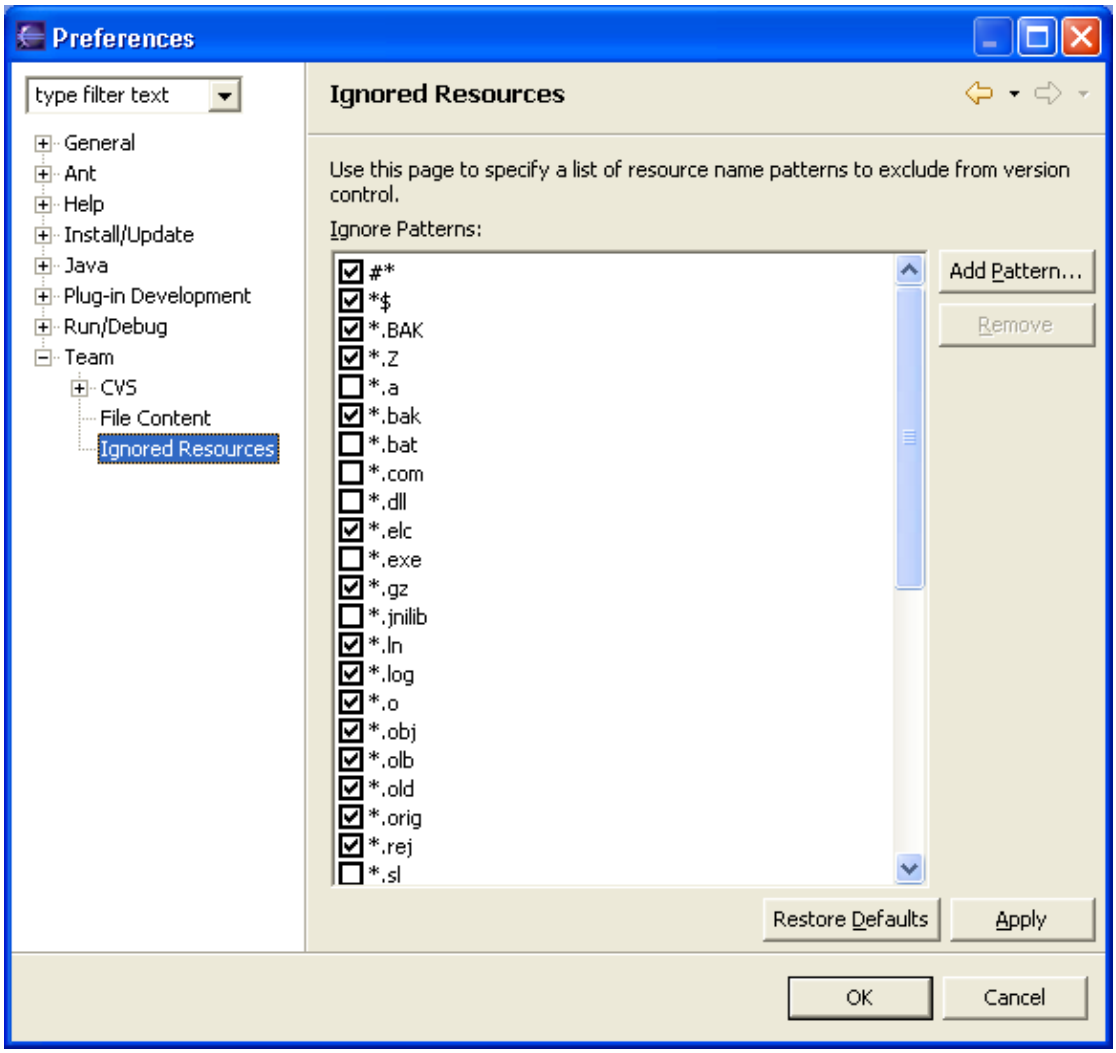
- In order to allow other plug−ins to indicate special handling for their projects and files the team plug−in defines extension points that other providers and other plug−ins can use to register file types and to declare files that should be ignored by a repository provider.
- Team providers can also register a class that can be used to persist a set a projects so that references to the project can be shared across a team, with the actual contents coming from the repository.
- Repository providers should consider how they will handle linked resources.
- Finally, team providers can mark resources that should be hidden from the user as team private.

Ignored files

In several cases, it may be unnecessary to keep certain files under repository control. For example, resources that are derived from existing resources can often be omitted from the repository. For example, compiled source files, (such as Java ".class" files), can be omitted since their corresponding source (".java") file is in the repository. It also may be inappropriate to version control metadata files that are generated by repository providers. The **org.eclipse.team.core.ignore** extension point allows providers to declare file types that should be ignored for repository provider operations. For example, the CVS client declares the following:

```
<extension point="org.eclipse.team.core.ignore">
        <ignore pattern = ".#*" selected = "true"/>
</extension>
```

The markup simply declares a file name **pattern** that should be ignored and a **selected** attribute which declares the default selection value of the file type in the preferences dialog. It is ultimately up to the user to decide which files should be ignored. The user may select, deselect, add or delete file types from the default list of ignored files.



File Types

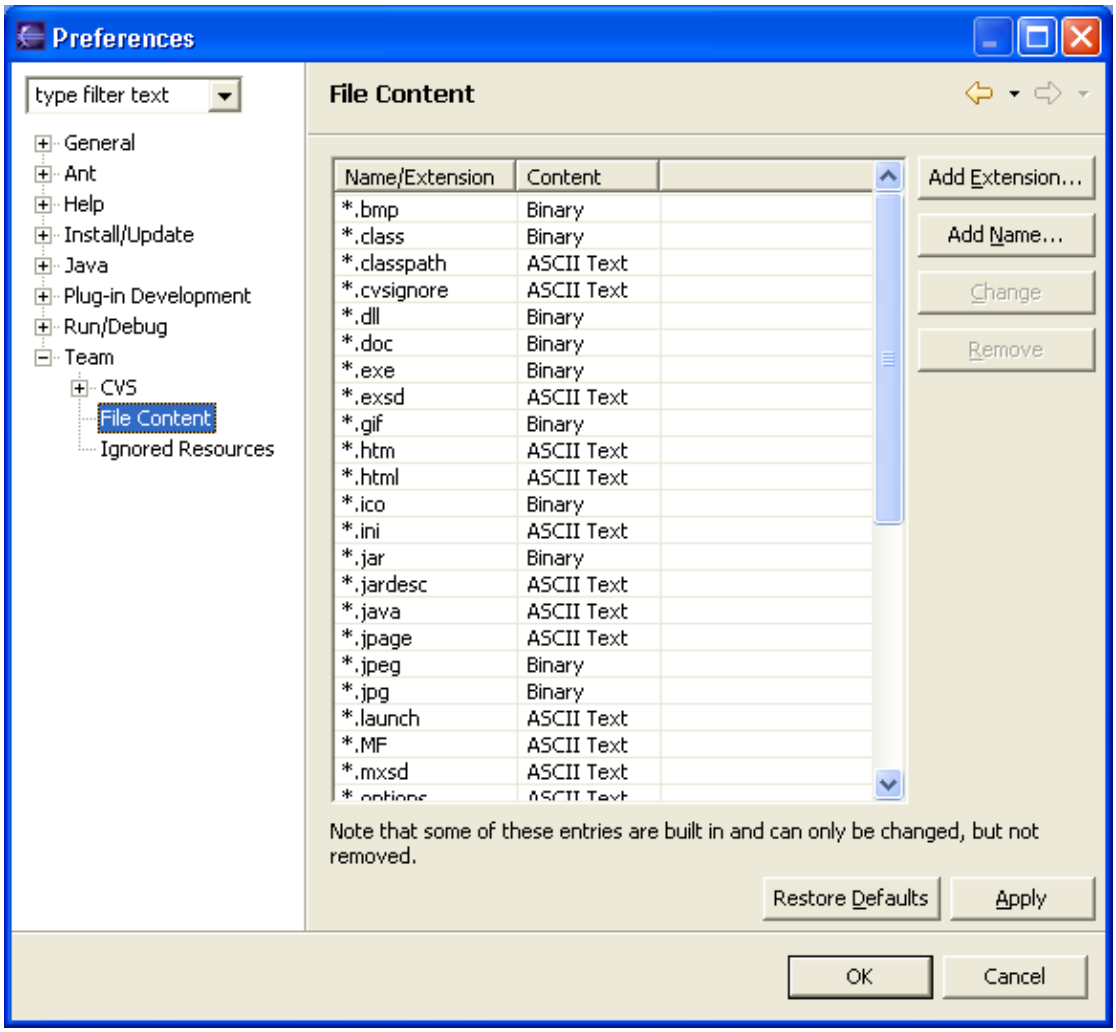
Some repositories implement different handling for text vs. binary files. The **org.eclipse.team.core.fileTypes** extension allows plug–ins to declare file types as text or binary files. For example, the Java tooling declares the following:

```
<extension point="org.eclipse.team.core.fileTypes">
  <fileTypes extension="java" type="text"/>

  <fileTypes extension="classpath" type="text"/>
  <fileTypes extension="properties" type="text"/>
  <fileTypes extension="class" type="binary"/>

  <fileTypes extension="jar" type="binary"/>
  <fileTypes extension="zip" type="binary"/>
</extension>
```

The markup lets plug–ins define a file type by **extension** and assign a **type** of text or binary. As with ignored files, it is ultimately up to the user to manage the list of text and binary file types.



Team and linked resources

A project may contain resources that are not located within the project's directory in the local file system. These resources are referred to as linked resources.

Consequences for Repository Providers

Linked resources can pose particular challenges for repository providers which operate directly against the file system. This is a consequence of the fact that linked resources by design do not exist in the immediate project directory tree in the file system.

Providers which exhibit the following characteristics may be affected by linked resources:

1. Those which call out to an external program that then operates directly against the file system.
2. Those which are implemented in terms of IResource but assume that all the files/folders in a project exist as direct descendents of that single rooted directory tree.

In the first case, lets assume the user picks a linked resource and tries to perform a provider operation on it. Since the provider calls a command line client, we can assume that the provider does something equivalent to first calling **IResource**.getLocation().toOSString(), feeding the resulting file system location as an argument

to the command line program. If the resource in question is a linked resource, this will yield a file/folder outside of the project directory tree. Not all command line clients may expect and be able to handle this case. In short, if your provider ever gets the file system location of a resource, it will likely require extra work to handle linked resources.

The second case is quite similar in that there is an implicit assumption that the structure of the project resources is 1:1 with that of the file system files/folders. In general, a provider could be in trouble if they mix IResource and java.io.File operations. For example, for links, the parent of **IFile** is not the same as the java.io.File's parent and code which assumes these to be the same will fail.

Backwards Compatibility

It was important that the introduction of linked resources did not inadvertently break existing providers. Specifically, the concern was for providers that reasonably assumed that the local file system structure mirrored the project structure. Consequently, by default linked resources can not be added to projects that are mapped to such a provider. Additionally, projects that contain linked resources can not by default be shared with that provider.

Strategies for Handling Linked Resources

In order to be "link friendly", a provider should allow projects with linked resources to be version controlled, but can disallow the version controlling of linked resources themselves.

A considerably more complex solution would be to allow the versioning of the actual linked resources, but this should be discouraged since it brings with it complex scenarios (e.g. the file may already be version controlled under a different project tree by another provider). Our recommendation therefore is to support version controlled projects which contain non−version controlled linked resources.

Technical Details for Being "Link Friendly"

Repository provider implementations can be upgraded to support linked resources by overriding the **RepositoryProvider.canHandleLinkedResources()** method to return ***true***. Once this is done, linked resources will be allowed to exist in projects shared with that repository provider. However, the repository provider must take steps to ensure that linked resources are handled properly. As mentioned above, it is strongly suggested that repository providers ignore all linked resources. This means that linked resources (and their children) should be excluded from the actions supported by the repository provider. Furthermore, the repository provider should use the default move and delete behavior for linked resources if the repository provider implementation overrides the default **IMoveDeleteHook**.

Team providers can use **IResource.isLinked()** to determine if a resource is a link. However, this method only returns true for the root of a link. The following code segment can be used to determine if a resource is the child of a link.

```
String linkedParentName = resource.getProjectRelativePath().segment(0);
IFolder linkedParent = resource.getProject().getFolder(linkedParentName);
boolean isLinked = linkedParent.isLinked();
```

Repository providers should ignore any resource for which the above code evaluates to ***true***.

Team private resources

It is common for repository implementations to use extra files and folders to store information specific about the repository implementation. Although these files may be needed in the workspace, they are of no interest to other plug−ins or to the end user.

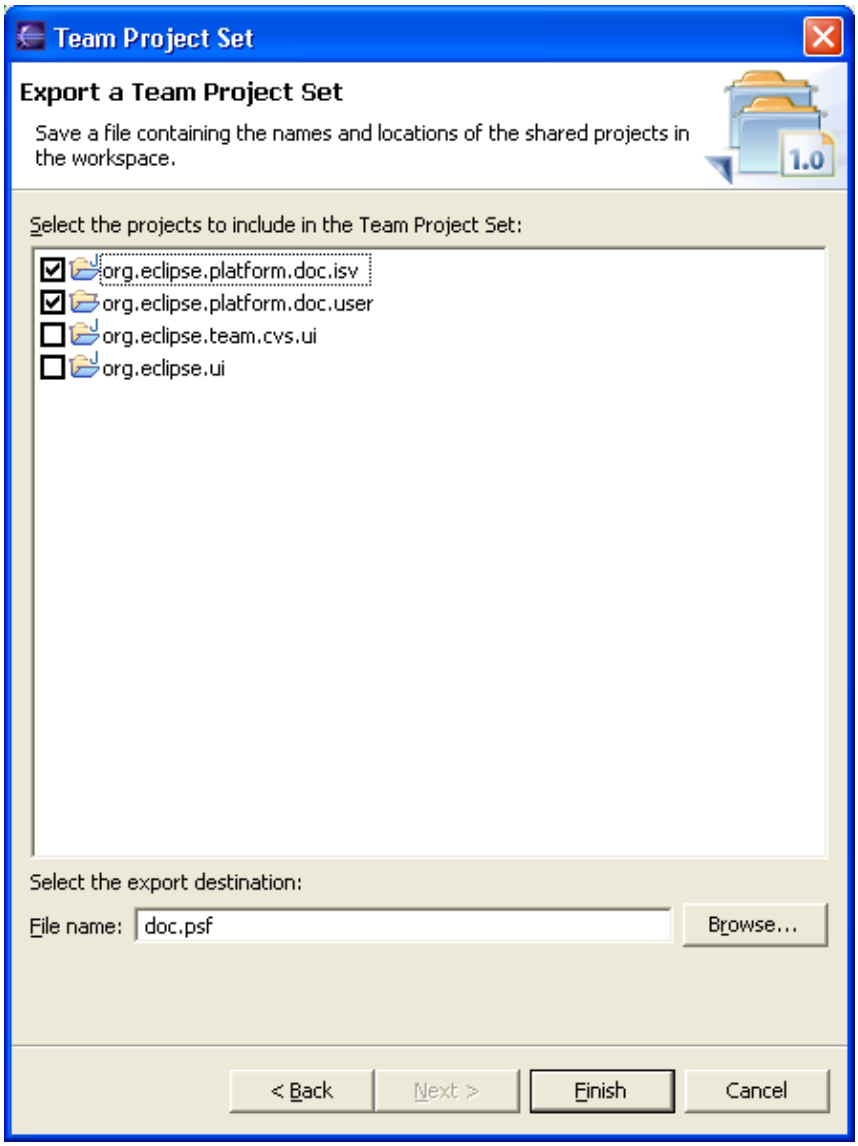
Team providers may use **IResource.setTeamPrivateMember(boolean)** to indicate that a resource is private to the implementation of a team provider. Newly created resources are not private members by default, so this method must be used to explicitly mark the resource as team private. A common use is to mark a subfolder of the project as team private when the project is configured for team and the subfolder is created.

Other resource API that enumerates resources (such as resource delta trees) will exclude team private members unless explicitly requested to include them. This means that most clients will not "see" the team private resources and they will not be shown to the user. The resource navigator does not show team private members by default, but users can indicate via Preferences that they would like to see team private resources.

Attempts to mark projects or the workspace root as team private will be ignored.

Project sets

Since the resources inside a project under version control are kept in the repository, it is possible to share projects with team members by sharing a reference to the repository specific information needed to reconstruct a project in the workspace. This is done using a special type of file export for **team project sets**.



In 3.0, API was added to ProjectSetCapability to allow repository providers to declare a class that implements project saving for projects under their control. When the user chooses to export project sets, only the projects configured with repositories that define project sets are shown as candidates for export. This API replaces the old project set serialization API (see below).

The project set capability class for a repository provider is obtained from the RepositoryProviderType class which is registered in the same extension as the repository provider. For example:

```
<extension point="org.eclipse.team.core.repository">
    <repository
          typeClass="org.eclipse.team.internal.ccvs.core.CVSTeamProviderType"

          class="org.eclipse.team.internal.ccvs.core.CVSTeamProvider"
          id="org.eclipse.team.cvs.core.cvsnature">
    </repository>
</extension>
```

Prior to 3.0, The **org.eclipse.team.core.projectSets** extension point allowed repository providers to declare a class that implements project saving for projects under their control. When the user chooses to export project sets, only the projects configured with repositories that define project sets are shown as candidates for export.

For example, the CVS client declares the following:

```
<extension point="org.eclipse.team.core.projectSets">
        <projectSets id="org.eclipse.team.cvs.core.cvsnature" class="org.eclipse.team.internal.cc
</extension>
```

The specified class must implement **IProjectSetSerializer**. Use of this interface is still supported in 3.0 but has been deprecated.

Ignore

Identifier:

org.eclipse.team.core.ignore

Since:

2.0

Description:

This extension point is used to register information about whether particular resources should be ignored; that is, excluded from version configuration management operations. Providers may provide an extension for this extension point. No code beyond the XML extension declaration is required.

Configuration Markup:

<!ELEMENT extension (ignore*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED>

<!ELEMENT ignore EMPTY>

<!ATTLIST ignore

pattern CDATA #REQUIRED

enabled (true | false) >

- **pattern –** the pattern against which resources will be compared.
- **enabled –** one of "true" or "false", determines whether this ignore pattern is enabled.

Examples:

Following is an example of an ignore extension:

<extension point=

"org.eclipse.team.core.ignore"

>

<ignore pattern=

"*.class"

enabled=

"true"

/>

</extension>

File Types

Identifier:

org.eclipse.team.core.fileTypes

Since:

2.0

Description:

This extension point is used to register information about whether particular file types should be considered to contain text or binary data. This information is important to some repository providers as it affects how the data is stored, compared and transmitted.

Providers may provide an extension for this extension point. No code beyond the XML extension declaration is required.

Configuration Markup:

<!ELEMENT extension (fileTypes*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED>

<!ELEMENT fileTypes EMPTY>

<!ATTLIST fileTypes

extension CDATA #REQUIRED

type CDATA #REQUIRED>

- **extension –** the file extension being identified by this contribution.
- **type –** one of either "text" or "binary", identifying the contents of files matching the value of extension.

Examples:

Following is an example of a fileTypes extension:

<extension point=

"org.eclipse.team.core.fileTypes"

>

<fileTypes extension=

"txt"

type=

"text"

/>

</extension>

Linked resources

Earlier discussions of resources and the file system (Mapping resources to disk locations) assumed that all resources in a project are located in the same place in the file system. This is generally true. However, the concept of **linked resources** in the workbench is provided so that files and folders inside a project can be stored in the file system outside of the project's location.

Linked resources must have a project as their parent resource. They can be located virtually anywhere in the file system. They can reside outside the project location, or even within another project. There are only a few restrictions on linked resource locations. The method **IWorkspace.validateLinkLocation** can be used to ensure that a given location is valid for creating a linked resource.

Linked resources are created using the method **IFolder.createLink** or **IFile.createLink**. To determine programmatically whether a given resource is a linked resource, you can use the method **IResource.isLinked**. Note that this method will only return `true` for linked resources, not for children of linked resources.

Apart from these special methods for creating linked resources and finding out if a resource is linked, you can use normal workspace API when manipulating linked resources. In most respects, linked resources act exactly like any other resource in the workspace. However, some restrictions apply when moving, copying, or deleting linked resources. See **IResource** and its sub−classes for information on individual operatiosn and their limitations.

Path variables

Path variables can be used when specifying the location of linked resources. A path variable is a simple (String −> **IPath**) mapping that defines a shortcut for a location in the file system. Variables can ease the

management of linked resources by reducing the number of places where hard−coded, absolute file system paths are used.

Path variables streamline the management of linked resources for users in several ways:

- Allows a single reference to the absolute path when defining several linked resources under a common root
- Allows the location of several resources to be redefined by changing a single variable
- Allows users to share projects containing linked resources without updating the paths of each resource (since the absolute path can vary from one machine to another.)

The last item in this list deserves a bit of explanation. When a user creates a linked resource in a project, a description of the linked resource is added to the project description file (".project") in the project's location. By using a path variable, users can share a project (by copying the project's content or by using a repository), and redefine the variable to suit each individual workstation. For example, one user might store external resources under c:\temp on one system, while another user using Unix might store the same resources in /home/username/tmp. Defining a path variable on each workspace (`TEMP=c:\temp` and `TEMP=/home/userb/tmp`) allows users to work around this difference and share the projects with linked resources as is.

IPathVariableManager defines the API for creating, manipulating, and resolving path variables. It also provides methods for validating variable names and values before creating them, and for installing a listener to be notified when path variable definitions change. You can obtain an instance of this class using **IWorkspace.getPathVariableManager**. See the code examples at the end of this section for more detail.

The method **IResource.getRawLocation** can be used to find out the unresolved location of a linked resource. That is, to get the actual path variable name instead of resolving it to its value. If a resource location is not defined with a path variable, the **getRawLocation** method acts exactly like the **getLocation** method.

Broken links

Clients that manipulate resources programmatically need to be aware of the possibility of broken links. Broken links occur when a linked resource's location does not exist, or is specified relative to an undefined path variable. The following special cases apply when using **IResource** protocol:

- The **copy** and **move** methods will fail when called on broken links.
- Calling **refreshLocal** on a broken link will not cause the resource to be removed from the workspace, as it does for normal resources that are missing from the file system.
- The method **getLocation** will return `null` for linked resources whose locations are relative to undefined path variables.
- You can still use **delete** to remove broken links from the workspace.

Compatibility with installed plug−ins

Some plug−ins may not be able to handle linked resources, so there are a number of mechanisms available for disabling them. If you are writing a plug−in that absolutely needs all of a project's contents to be located in the project's default location, you can use these mechanisms to prevent users from creating linked resources where you don't want them to appear.

The first mechanism is called the *project nature veto*. If you define your own project nature, you can specify in the nature definition that the nature is not compatible with linked resources. Here is an example of a nature definition that employs the nature veto:

```
<extension
        id="myNature"
        name="My Nature"
        point="org.eclipse.core.resources.natures">
        <runtime>
                <run class="com.xyz.MyNature"/>
        </runtime>
        <allowLinking="false"/>
</extension>
```

The second mechanism for preventing linked resource creation is the *team hook*. If you define your own repository implementation, you can make use of the **org.eclipse.core.resources.teamHook** extension point to prevent the creation of linked resources in projects that are shared with your repository type. By default, repository providers do *not* allow linked resources in projects connected to the repository.

If the repository support is provided by an older plug−in that is not aware of linked resources, you will not be able to create linked resources in those projects.

Finally, there is a preference setting that can be used to disable linked resources for the entire workspace. While the previous two veto mechanisms work on a per−project basis, this preference affects all projects in the workspace. To set this preference programatically, use the preference `ResourcesPlugin.PREF_DISABLE_LINKING`. Note that even when set, users or plug−ins can override this by turning the preference off.

Linked resources in code

Let's go into some examples of using linked resources in code. We'll start by defining a path variable:

```
IWorkspace workspace = ResourcesPlugin.getWorkspace();
IPathVariableManager pathMan = workspace.getPathVariableManager();
String name = "TEMP";
IPath value = new Path("c:\\temp");
if (pathMan.validateName(name).isOK() && pathMan.validateValue(value).isOK()) {
   pathMan.setValue(name, value);
} else {
   //invalid name or value, throw an exception or warn user
}
```

Now we can create a linked resource relative to the defined path variable:

```
IProject project = workspace.getProject("Project");//assume this exists
IFolder link = project.getFolder("Link");
IPath location = new Path("TEMP/folder");
if (workspace.validateLinkLocation(location).isOK()) {
   link.createLink(location, IResource.NONE, null);
} else {
   //invalid location, throw an exception or warn user
}
```

That's it! You now have a linked folder in your workspace called "Link" that is located at "c:\temp\folder".

Let's end with some code snippets on this linked resource to illustrate the behavior other methods related to linked resources:

```
link.getFullPath() ==> "/Project/Link"
link.getLocation() ==> "c:\temp\folder"
link.getRawLocation() ==> "TEMP/folder"
link.isLinked() ==> "true"

IFile child = link.getFile("abc.txt");
child.create(...);
child.getFullPath() ==> "/Project/Link/abc.txt"
child.getLocation() ==> "c:\temp\folder\abc.txt"
child.getRawLocation() ==> "c:\temp\folder\abc.txt"
child.isLinked() ==> "false"
```

Project natures

Project natures allow a plug−in to tag a project as a specific kind of project. For example, the Java development tools (JDT) use a "Java nature" to add Java−specific behavior to projects. Project natures are defined by plug−ins, and are typically added or removed per−project when the user performs some action defined by the plug−in.

A project can have more than one nature. However, when you define a project nature, you can define special constraints for the nature:

- **one−of−nature** − specifies that the nature is one of a named set. Natures in a set are mutually exclusive; that is, only one nature belonging to the set can exist for a project.
- **requires−nature** − specifies that the nature depends on another nature and can only be added to a project that already has the required nature.

To implement your own nature, you need to define an extension and supply a class which implements **IProjectNature**.

Defining a nature

The **org.eclipse.core.resources.natures** extension point is used to add a project nature definition. The following markup adds a nature for the hypothetical **com.example.natures** plug−in.

```
<extension
   point="org.eclipse.core.resources.natures"
   id="mynature"
   name="My Nature">
   <runtime>
      <run class="com.example.natures.MyNature">
      </run>
   </runtime>
</extension>
```

The class identified in the extension must implement the platform interface **IProjectNature**. This class implements plug−in specific behavior for associating nature−specific information with a project when the nature is configured.

```
public class MyNature implements IProjectNature {
```

```
   private IProject project;

   public void configure() throws CoreException {
      // Add nature-specific information
      // for the project, such as adding a builder
      // to a project's build spec.
   }
   public void deconfigure() throws CoreException {
      // Remove the nature-specific information here.
   }
   public IProject getProject() {
      return project;
   }
   public void setProject(IProject value) {
      project = value;
   }
}
```

The **configure()** and **deconfigure()** methods are sent by the platform when natures are added and removed from a project. You can implement the **configure()** method to add a builder to a project as discussed in Builders.

Associating the nature with a project

Defining the nature is not enough to associate it with a project. You must assign a nature to a project by updating the project's description to include your nature. This usually happens when the user creates a new project with a specialized new project wizard that assigns the nature. The following snippet shows how to assign our new nature to a given project.

```
try {
   IProjectDescription description = project.getDescription();
   String[] natures = description.getNatureIds();
   String[] newNatures = new String[natures.length + 1];
   System.arraycopy(natures, 0, newNatures, 0, natures.length);
   newNatures[natures.length] = "com.example.natures.mynature";
   description.setNatureIds(newNatures);
   project.setDescription(description, null);
} catch (CoreException e) {
   // Something went wrong
}
```

NOTE: The nature id is the fully qualified id of the nature extension. The fully qualified id of an extension is created by combining the plug−in id with the simple extension id in the plugin.xml file. For example, a nature with simple extension id "mynature" in the plug−in "com.example.natures" would have the name "com.example.natures.mynature"

The natures are not actually assigned to (and configured) for the project until you set the project description into the project. Also note that the identifier used for the nature is the fully qualified name (plug−in id + extension id) of the nature extension.

If the nature has been defined with constraints, then workspace API can be used to validate the new nature. For example, suppose a nature is defined with a prerequisite:

```
<extension
   point="org.eclipse.core.resources.natures"
   id="myOtherNature"
```

```
   name="My Other Nature">
   <runtime>
      <run class="com.example.natures.MyOtherNature">
      </run>
   </runtime>
<requires-nature id="com.example.natures.mynature"/>
</extension>
```

The new nature is not valid unless the first nature exists for the project. Depending on the design of your plug−in, you may want to check whether the prerequisite nature has been installed, or you may want to add the prerequisite nature yourself. Either way, you can check on the validity of proposed combinations of project natures using workspace API.

```
try {
   IProjectDescription description = project.getDescription();
   String[] natures = description.getNatureIds();
   String[] newNatures = new String[natures.length + 1];
   System.arraycopy(natures, 0, newNatures, 0, natures.length);
   newNatures[natures.length] = "com.example.natures.myOtherNature";
   IStatus status = workspace.validateNatureSet(natures);

   // check the status and decide what to do
   if (status.getCode() == IStatus.OK) {
     description.setNatureIds(newNatures);
     project.setDescription(description, null);
   } else {
     // raise a user error
     ...
   }
} catch (CoreException e) {
   // Something went wrong
}
```

Nature descriptors

In addition to working with natures by their id, you can obtain the descriptor (**IProjectNatureDescriptor**) which describes a nature, its constraints, and its label. You can query a particular nature for its descriptor, or get descriptors from the workspace. The following snippet gets the project nature descriptor for our new nature:

```
IProjectNatureDescriptor descriptor = workspace.getNatureDescriptor("com.example.natures.my
```

You can also get an array of descriptors for all installed natures:

```
IProjectNatureDescriptor[] descriptors = workspace.getNatureDescriptors();
```

Project Natures

Identifier:

org.eclipse.core.resources.natures

Description:

The workspace supports the notion of project natures (or "natures" for short"). A nature associates lifecycle behaviour with a project. Natures are installed on a per−project basis using the setDescription method defined on `org.eclipes.core.resources.IProject`. They are configured automatically when a project is opened and deconfigured when a project is closed. For example, the Java nature might install a Java builder and do other project configuration when added to a project

The natures extension−point allows nature writers to register their nature implementation under a symbolic name that is then used from within the workspace to find and configure natures. The symbolic name is id of the nature extension. When defining a nature extension, users are encouraged to include a human−readable value for the "name" attribute which identifies their meaning and potentially may be presented to users.

Natures can specify relationship constraints with other natures. The "one−of−nature" constraint specifies that at most one nature belong to a given set can exist on a project at any given time. This enforces mutual exclusion between natures that are not compatible with each other. The "requires−nature" constraint specifies a dependency on another nature. When a nature is added to a project, all required natures must also be added. The natures are guaranteed to be configured and deconfigured in such a way that their required natures will always be configured before them and deconfigured after them. For this reason, cyclic dependencies between natures are not permitted.

Natures cannot be added to or removed from a project if that change would violate any constraints that were previously satisfied. If a nature is configured on a project, but later finds that its constraints are not satisfied, that nature and all natures that require it are marked as *disabled*, but remain on the project. This can happen, for example, when a required nature goes missing from the install. Natures that are missing from the install, and natures involved in dependency cycles are also marked as disabled.

Natures can also specify which incremental project builders, if any, are configured by them. With this information, the workspace will ensure that builders will only run when their corresponding nature is present and enabled on the project being built. If a nature is removed from a project, but the nature's deconfigure method fails to remove its corresponding builders, the workspace will remove those builders from the spec automatically. It is not permitted for two natures to specify the same incremental project builder in their markup.

Natures also have the ability to disallow the creation of linked resources on projects they are associated with. By setting the `allowLinking` attribute to "false", a nature can declare that linked resources should never be created. This feature is new in release 2.1.

Starting with release 3.1, natures can declare affinity with arbitrary content types, affecting the way content type determination happens for files in the workspace. In case of conflicts (two or more content types deemed equally suitable for a given file), the content type having affinity with any of the natures configured for the corresponding project will be chosen.

Configuration Markup:

<!ELEMENT extension (runtime , (one−of−nature | requires−nature | builder | content−type)* , options?)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #REQUIRED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT runtime (run)>

<!ELEMENT run (parameter*)>

<!ATTLIST run

class CDATA #REQUIRED>

- **class –** the fully−qualified name of a class which implements `org.eclipse.core.resources.IProjectNature`

<!ELEMENT parameter EMPTY>

<!ATTLIST parameter

name CDATA #REQUIRED

value CDATA #REQUIRED>

- **name –** the name of this parameter made available to instances of the specified nature class
- **value –** an arbitrary value associated with the given name and made available to instances of the specificed nature class

<!ELEMENT one−of−nature EMPTY>

<!ATTLIST one−of−nature

id CDATA #REQUIRED>

- **id** – the name of an exclusive project nature category.

<!ELEMENT requires–nature EMPTY>

<!ATTLIST requires–nature

id CDATA #REQUIRED>

- **id** – the fully–qualified id of another nature extension that this nature extension requires.

<!ELEMENT builder EMPTY>

<!ATTLIST builder

id CDATA #REQUIRED>

- **id** – the fully–qualified id of an incremental project builder extension that this nature controls.

<!ELEMENT options EMPTY>

<!ATTLIST options

allowLinking (true | false) >

- **allowLinking** – an option to specify whether this nature allows the creation of linked resources. By default, linking is allowed.

<!ELEMENT content–type EMPTY>

<!ATTLIST content–type

id CDATA #REQUIRED>

- **id** – the fully–qualified id of a content type associated to this nature.

Examples:

Following is an example of three nature configurations. The waterNature and fireNature belong to the same exclusive set, so they cannot co–exist on the same project. The snowNature requires waterNature, so

snowNature will be disabled on a project that is missing waterNature. It naturally follows that snowNature cannot be enabled on a project with fireNature. The fireNature also doesn't allow the creation of linked resources.

<extension id=

"fireNature"

name=

"Fire Nature"

point=

"org.eclipse.core.resources.natures"

>

<runtime>

<run class=

"com.xyz.natures.Fire"

/>

</runtime>

<one−of−nature id=

"com.xyz.stateSet"

/>

<options allowLinking=

"false"

/>

</extension>

<extension id=

"waterNature"

name=

"Water Nature"

point=

"org.eclipse.core.resources.natures"

>

<runtime>

<run class=

"com.xyz.natures.Water"

/>

</runtime>

<one−of−nature id=

"com.xyz.stateSet"

/>

</extension>

<extension id=

"snowNature"

name=

"Snow Nature"

point=

"org.eclipse.core.resources.natures"

>

<runtime>

<run class=

"com.xyz.natures.Snow"

>

<parameter name=

"installBuilder"

value=

"true"

/>

</run>

</runtime>

<requires−nature id=

"com.xyz.coolplugin.waterNature"

/>

<builder id=

"com.xyz.snowMaker"

/>

</extension>

If these extensions were defined in a plug−in with id "com.xyz.coolplugin", the fully qualified name of these natures would be "com.xyz.coolplugin.fireNature", "com.xyz.coolplugin.waterNature" and "com.xyz.coolplugin.snowNature".

API Information:

The value of the class attribute must represent an implementor of `org.eclipse.core.resources.IProjectNature`. Nature definitions can be examined using the `org.eclipse.core.resources.IProjectNatureDescriptor` interface. The descriptor objects can be obtained using the methods `getNatureDescriptor(String)` and `getNatureDescriptors()` on `org.eclipse.core.resources.IWorkspace`.

Supplied Implementation:

The platform itself does not have any predefined natures. Particular product installs may include natures as required.

Project Sets

Identifier:

org.eclipse.team.core.projectSets

Since:

2.0

Description:

This extension point is used to register a handler for creating and reading project sets. Project sets are lightweight, portable method of sharing a particular lineup of team−shared projects in a workspace. A project set file may be used to provide team memebers with a simple way of creating a workspace with a particular lineup of projects form one or more team providers. Providers may provide an extension for this extension point.

deprecated: see RepositoryProvider#getProjectSetCapability.

Configuration Markup:

<!ELEMENT extension (projectSets*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED>

<!ELEMENT projectSets EMPTY>

<!ATTLIST projectSets

id CDATA #REQUIRED

class CDATA #REQUIRED>

- **id** – the nature id of the provider for which this handler creates and reads project sets.
- **class** – the fully qualified name of a class implementing `org.eclipse.team.core.IProjectSerializer`.

Examples:

Following is an example of a projectSets extension:

```
<extension point=

"org.eclipse.team.core.projectSets"

>

<projectSets id=

"org.eclipse.team.cvs.core.cvsnature"

class=

"org.eclipse.team.cvs.core.CVSProjectSetSerializer"

>

</projectSets>

</extension>
```


Synchronization Support

New in Eclipse 3.0 are APIs for managing and displaying synchronization state between workspace resources and resources in another location. We refer to a resource outside of the workspace as a variant. Synchronizing is the act of displaying the changes between resources in different locations and optionally allowing the user to affect the synchronization state by performing an action. The synchronize APIs are orthogonal to the RepositoryProvider APIs and can be used without a repository provider. The purpose of the synchronization API is to ease the task of implementing different ways of presenting the synchronization state of resources. As such, the API requires a means to query the synchronization state of resources but does not require a means to affect the state. The means of affecting the state is left to the implementer (although the UI does provide hooks for adding provider specific menu items to menus).

Terminology

Before the synchronization API is described, it is helpful to present some of the terminology and concepts that apply when discussing workspace synchronization.

> *Resource Variant*: A local resource that is mapped to a resource that exists at another location can be referred to as a variant of that resource. That is, the resources are usually very similar but may differ slightly (either due to modifications to the local resource or changes made the remote copy by other users). We take a local workspace centric view of this, referring to the local copy as the resource and any remote copy as resource variants.

Synchronize: We refer to synchronize as the action of displaying to the user the differences between resource variants. Synchronizing doesn't affect the state of the variants, but instead provides a view to help the user understand the differences between different sets of variants. It is common however to allow users to affect the states of the variants (e.g. allowing to check−in, or revert) while synchronizing.

Two−way vs. Three−way Synchronization: There are two basic types of synchronization state determination: two−way and three−way. A two−way comparison only considers the local resource and a single resource variant, referred to as the remote resource variant. This type of comparison can only show the differences between the two resources but cannot offer hints as to how the changes interrelate. Most code repository systems support a three−way comparison for synchronization state determination. This type of comparison involves the local resource, a remote resource variant and a base resource variant. The base resource variant represents a common ancestor for the local and remote resources. This allows for more sophisticated synchronization states that indicate the direction of the change.

Table 1: The synchronization states

Two−Way	Three−Way
Changed Deleted Added	Outgoing Change Incoming Change Outgoing Deletion Incoming Deletion Outgoing Addition Incoming Addition Conflicting Change Conflicting Deletion Conflicting Addition

The Basics – SyncInfo

The classes in the org.eclipse.team.core.synchronize are used to describe the synchronization state. The most important class is SyncInfo because it is the class that actually defines the synchronization state. It can be used as follows:

```
SyncInfo info = getSyncInfo(resource); // this is a simulated method of obtaining the sync info f
int changekind = info.getKind();
if(info.getResourceComparator().isThreeWay()) {
  if((changeKind & SyncInfo.DIRECTION_MASK) == SyncInfo.INCOMING) {
    // do something
  }
} else if(changeKind == SyncInfo.CHANGE) {
    // do something else
}
```

The SyncInfo class provides both the two−way and three−way comparison algorithms, a client must provide the resources and a class that can compare the resources (IResourceVariantComparator). Here is an example variant comparator:

```
public class TimestampVariantComparator implements IResourceVariantComparator {
  protected boolean compare(IResourceVariant e1, IResourceVariant e2) {
    if(e1.isContainer()) {
```

```
      if(e2.isContainer()) {
        return true;
      }
      return false;
    }
    if(e1 instanceof MyResourceVariant && e2 instanceof MyResourceVariant) {
      MyResourceVariant myE1 = (MyResourceVariant)e1;
      MyResourceVariant myE2 = (MyResourceVariant)e2;
      return myE1.getTimestamp().equals(myE2.getTimestamp());
    }
    return false;
  }
  protected boolean compare(IResource e1, IResourceVariant e2) {

  }
  public boolean isThreeWay() {
    return true;
  }
}

SyncInfo info = new SyncInfo(resource, variant1, variant2, new TimestampComparator());
info.init(); // calculate the sync info
```

This package also contains collections specifically designed to contain SyncInfo and filters that can be applied to SyncInfo instances.

Managing the synchronization state

As we have seen in the examples above, SyncInfo and IResourceVariantComparator classes provide access to the synchronization state of resources. But what we haven't seen yet is how the state is managed. A Subscriber provides access to the synchronization state between the resources in the local workspace and a set of resource variants for these resources using either a two−way or three−way comparison, depending on the nature of the subscriber. A subscriber provides the following capabilities:

- local workspace traversal: a subscriber supports the traversal of the local workspace resources that are *supervised* by the subscriber. As such, the subscriber has a set of *root* resources that define the workspace subtrees under the subscriber's control, as well as a *members* method that returns the supervised members of a workspace resource. This traversal differs from the usual workspace resource traversal in that the resources being traversed may include resources that do not exist locally, either because they have been deleted by the user locally or created by a 3rd party.
- resource synchronization state determination: For supervised resources, the subscriber provides access to the synchronization state of the resource, including access to the variants of the resource. For each supervised resource, the subscriber provides a *SyncInfo* object that contains the synchronization state and the *variants* used to determine the state.The subscriber also provides an *IResourceVariantComparator* which determines whether two−way or three−way comparison is to be used and provides the logic used by the SyncInfo to comparing resource variants when determining the synchronization state.
- refresh of synchronization state and change notification: Clients can react to changes that happen to local resources by listening to the Core resource deltas. When a local resource is changed, the synchronization state of the resource can then be re−obtained from the subscriber. However, clients must explicitly query the server to know if there are changes to the resource variants. For subscribers, this process is broken up into two parts. A client can explicitly *refresh* a subscriber. In response the subscriber will obtain the latest state of the resource variants from the remote location and fire *synchronization state change* events for any resource variants that have changed. The change

notification is separate from the refresh since there may be other operations that contact the remote location and obtain the latest remote state.

The APIs do not not define how a Subscriber is created, this is left to the specific implementations. For example the CVS plug−in creates a Subscriber when a merge is performed, another for a comparison, and another when synchronizing the local workspace with the current branch.

So let's revisit our first example of using SyncInfo and see how a Subscriber could be used to access SyncInfo.

```
// Create a file system subscriber and specify that the
// subscriber will synchronize with the provided file system location
Subscriber subscriber = new FileSystemSubscriber("c:\temp\repo");

// Allow the subscriber to refresh its state
subscriber.refresh(subscriber.roots(), IResource.DEPTH_INFINITE, monitor);

// Collect all the synchronization states and print
IResource[] children = subscriber.roots();
for(int i=0; i < children.length; i++) {
  printSyncState(children[i]);
}

...

void printSyncState(Subscriber subscriber, IResource resource) {
  System.out.println(subscriber.getSyncInfo(resource).toString());
  IResource[] children = subscriber.members(resource);
  for(int i=0; i < children.length; i++) {
    IResource child = children[i];
    if(! child.exists()) {
      System.out.println(resource.getFullPath() + " doesn't exist in the workspace");
    }
    printSyncState(subscriber, children[i]);
  }
}
```

The important point to remember is that the Subscriber knows about resources that do not exist in the workspace and non−existing resources can be returned from the Subscriber#members() and SyncInfo#getLocal().

Displaying the synchronizations state in the UI

We could spend more time explaining how to manage synchronization state but instead let's see how to actually get the state shown to the user. A ISynchronizeParticipant is the user interface component that displays synchronization state and allows the user to affect its state. The Synchronize View displays synchronize participants, but it is also possible to show these in dialogs and wizards. In order to provide support for users to show any type of synchronization state to the user, even those not based on SyncInfo and Subscribers, a participant is a very generic component.

There is also an extension point called org.eclipse.team.ui.synchronizeWizards to add a synchronization creation wizard. This will put your wizard in the global synchronize action and in the Synchronize View, so that users can easily create a synchronization of your type.

However, if you have implemented a Subscriber you can benefit from a concrete participant called SubscriberParticipant which will provide the following functionality:

- Collects SyncInfo from a Subscriber in the background.
- Listens to changes in the workspace and those found when a Subscriber is refreshed and keeps the synchronization state updated dynamically.
- Provides the user interface that support modes for filtering the changes, and layouts.
- Support scheduling a refresh with the Subscriber so that the synchronization states are kept up−to−date.
- Supports refreshing a Subscriber in the background.
- Supports navigation of the changes and showing the differences between the files.
- Supports configuration of the actions, toolbars, and decorators by subclasses.

The best way to explain these concepts are to see them used in the context of a simple example. Go to the local history synchronization example to see how all of these pieces can be used together. Or if you want pointers on how to use the more advanced APIs, go to Beyond The Basics.

Synchronize Participant Creation Wizards

Identifier:

org.eclipse.team.ui.synchronizeWizards

Since:

3.0

Description:

This extension point is used to register a synchronize participant creation wizard. These wizards are used to create synchronize participants that will appear in the Synchronize View. A provider will typically create a creation wizard to allow the user to perform and manage a particular type of synchronize participant. Providers may provide an extension for this extension point, and an implementation of `org.eclipse.jface.wizard.IWizard`.

Configuration Markup:

<!ELEMENT extension (wizard?)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

<!ELEMENT wizard EMPTY>

<!ATTLIST wizard

name CDATA #REQUIRED

description CDATA #REQUIRED

icon CDATA #REQUIRED

class CDATA #REQUIRED

id CDATA #REQUIRED>

- **name –** The name of the synchronize participant creation type. Examples are "CVS", "CVS Merge", "WebDAV".
- **description –** The description for the creation wizard.
- **icon –** The icon to be shown when this wizard type is shown to the user.

- **class** – A fully qualified name of the Java class implementing `org.eclipse.jface.wizard.IWizard`.
- **id** – A unique identifier for this extension.

Examples:

Following is an example of a synchronize participant creation wizard extension:

<extension point=

"org.eclipse.team.ui.synchronizeWizards"

>

<wizard name=

"WebDAV"

description=

"Create a WebDAV participant to view changes between workspace resources and their remote WebDAV location"

icon=

"webdav.gif"

class=

"com.xyz.DAVWizard"

id=

"com.xyz.dav.synchronizeWizard"

>

</wizard>

</extension>

API Information:

The value of the `class` attribute must represent a class that implements `org.eclipse.jface.wizard.IWizard`.

Since:

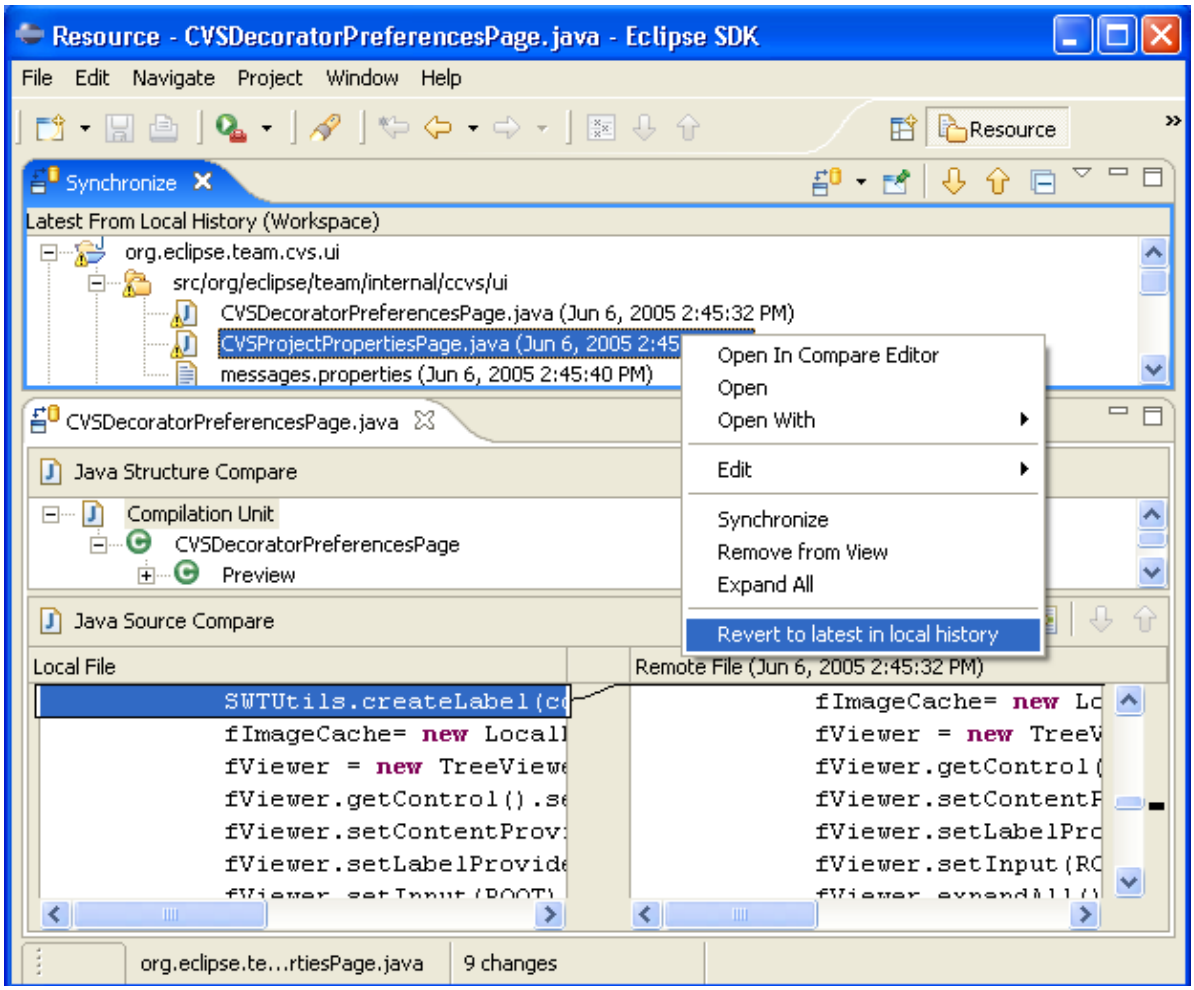
Supplied Implementation:

The plug−in org.eclipse.team.cvs.ui contains example definitions of synchronizeWizards extension point.

Local History Example

The best way to understand the Synchronize APIs is to create a simple example that actually works. In this example we will be creating a page in the Synchronize View that will display the latest local history state for all files in the workspace. The local history synchronization will update automatically when changes are made to the workspace, and a compare editor can open to browse, merge, then changes. We will also add a custom decorator to show the last timestamp of the local history element and an action to revert the workspace files to their latest saved local history state. This is an excellent example because we already have a store of resource variants available and we don't have to manage it.

For the remainder of this example we will make use of a running example. Much, but not all, of the source code will be included on this page. The full source code can be found in the local history package of the org.eclipse.team.examples.filesystem plug−in. You can check the project out from the CVS repository and use it as a reference while you are reading this tutorial. *Disclaimer*: The source code in the example plug−ins may change over time. To get a copy that matches what is used in this example, you can check out the project using the 3.0 version tag (most likely R3_0) or a date tag of June 28, 2004.



This screen shot shows the local history synchronization in the Synchronize View. With it you can browse the changes between the local resource and the latest state in history. It has a custom decorator for displaying the timestamp associated with the local history entry and a custom action to revert your file to the contents in the local history. Notice also that the standard Synchronize View presentation is used which provide problem annotations, compressed folder layout, and navigation buttons.

Defining the variants for local history

The first step is to define a variant to represent the elements from local history. This will allow the synchronize APIs to access the contents from the local history so it can be compared with the current contents and displayed to the user.

```
public class LocalHistoryVariant implements IResourceVariant {
  private final IFileState state;

  public LocalHistoryVariant(IFileState state) {
    this.state = state;
  }

  public String getName() {
    return state.getName();
  }

  public boolean isContainer() {
```

```
    return false;
  }

  public IStorage getStorage(IProgressMonitor monitor) throws TeamException {
    return state;
  }

  public String getContentIdentifier() {
    return Long.toString(state.getModificationTime());
  }

  public byte[] asBytes() {
    return null;
  }
}
```

Since the IFileState interface already provides access to the contents of the file from local history (i.e. implements the IStorage interface), this was easy. Generally, when creating a variant you have to provide a way of accessing the content, a content identifier that will be displayed to the user to identify this variant, and a name. The asBytes() method is only required if persisting the variant between sessions.

Next, let's create a variant comparator that allows the SyncInfo calculation to compare local resources with their variants. Again, this is easy because the existence of a local history state implies that the content of the local history state differs from the current contents of the file. This is because the specification for local history says that it won't create a local history state if the file hasn't changed.

```
public class LocalHistoryVariantComparator implements IResourceVariantComparator {
  public boolean compare(IResource local, IResourceVariant remote) {
    return false;
  }

  public boolean compare(IResourceVariant base, IResourceVariant remote) {
    return false;
  }

  public boolean isThreeWay() {
    return false;
  }
}
```

Because we know that the existence of the local history state implies that it is different from the local, we can simply return ***false*** when comparing the file to it's local history state. Also, synchronization with the local history is only two−way because we don't have access to a base resource so the method for comparing two resource variants is not used.

Note that the synchronize calculation won't call the compare method of the comparator if the variant doesn't exist (i.e. is null). It is only called if both elements exist. In our example, this would occur both for files that don't have a local history and for all folders (which never have a local history). To deal with this, we need to define our own subclass of SyncInfo in order to modify the calculated synchronization state for these cases.

```
public class LocalHistorySyncInfo extends SyncInfo {
  public LocalHistorySyncInfo(IResource local, IResourceVariant remote, IResourceVariantComparato
    super(local, null, remote, comparator);
  }

  protected int calculateKind() throws TeamException {
    if (getRemote() == null)
```

```
      return IN_SYNC;
    else
      return super.calculateKind();
  }
}
```

We have overridden the constructor to always provide a base that is *null* (since we are only using two−way comparison) and we have modified the synchronization kind calculation to return *IN_SYNC* if there is no remote (since we only care about the cases where there is a local file and a file state in the local history.

Creating the Subscriber

Now we will create a Subscriber that will provide access to the resource variants in the local history. Since local history can be saved for any file in the workspace, the local history Subscriber will supervise every resource and the set of roots will be all projects in the workspace. Also, there is no need to provide the ability to refresh the subscriber since the local history changes only when the contents of a local file changes. Therefore, we can update our state whenever a resource delta occurs. That leaves only two interesting method on our local history subscriber: obtaining a SyncInfo and traversing the workspace.

```
public SyncInfo getSyncInfo(IResource resource) throws TeamException {
  try {
    IResourceVariant variant = null;
    if(resource.getType() == IResource.FILE) {
      IFile file = (IFile)resource;
      IFileState[] states = file.getHistory(null);
      if(states.length > 0) {
        // last state only
        variant = new LocalHistoryVariant(states[0]);
      }
    }
    SyncInfo info = new LocalHistorySyncInfo(resource, variant, comparator);
    info.init();
    return info;
  } catch (CoreException e) {
    throw TeamException.asTeamException(e);
  }
}
```

The Subscriber will return a new SyncInfo instance that will contain the latest state of the file in local history. The SyncInfo is created with a local history variant for the remote element. For projects, folders and files with no local history, no remote resource variant is provided, which will result in the resource being considered in−sync due to the *calculateKind* method in our LocalHistorySyncInfo.

The remaining code in the local history subscriber is the implementation of the *members* method:

```
public IResource[] members(IResource resource) throws TeamException {
  try {
    if(resource.getType() == IResource.FILE)
      return new IResource[0];
    IContainer container = (IContainer)resource;
    List existingChildren = new ArrayList(Arrays.asList(container.members()));
    existingChildren.addAll(
      Arrays.asList(container.findDeletedMembersWithHistory(IResource.DEPTH_INFINITE, null)));
    return (IResource[]) existingChildren.toArray(new IResource[existingChildren.size()]);
  } catch (CoreException e) {
    throw TeamException.asTeamException(e);
  }
```

```
}
```

The interesting detail of this method is that it will return non−existing children if a deleted resource has local history. This will allow our Subscriber to return SyncInfo for elements that only exist in local history and are no longer in the workspace.

Adding a Local History Synchronize Participant

So far we have created the classes which provide access to SyncInfo for elements in local history. Next, we will create the UI elements that will allow us to have a page in the Synchronize View to display the last history state for every element in local history. Since we have a Subscriber, adding this to the Synchronize View is easy. Let's start by adding an synchronize participant extension point:

```
<extension
       point="org.eclipse.team.ui.synchronizeParticipants">
    <participant
        persistent="false"
        icon="synced.png"
        class="org.eclipse.team.synchronize.example.LocalHistoryParticipant"
        name="Latest From Local History"
        id="org.eclipse.team.synchronize.example"/>
 </extension>
```

Next we have to implement the LocalHistoryParticipant. It will subclass SubscriberParticipant which will provide all the default behavior for collecting SyncInfo from the subscriber and updating sync states when workspace changes occur. In addition, we will add an action to revert the workspace resources to the latest in local history.

First, we will look at how a custom action is added to the participant.

```
public static final String CONTEXT_MENU_CONTRIBUTION_GROUP = "context_group_1"; //$NON-NLS-1$

private class LocalHistoryActionContribution extends SynchronizePageActionGroup {
  public void initialize(ISynchronizePageConfiguration configuration) {
    super.initialize(configuration);
    appendToGroup(
      ISynchronizePageConfiguration.P_CONTEXT_MENU, CONTEXT_MENU_CONTRIBUTION_GROUP,
      new SynchronizeModelAction("Revert to latest in local history", configuration) { //$NON-NLS
        protected SynchronizeModelOperation getSubscriberOperation(ISynchronizePageConfiguration
          return new RevertAllOperation(configuration, elements);
        }
      });
  }
}
```

Here we are adding a specific SynchronizeMoidelAction and operation. The behavior we get for free here is the ability to run in the background and show busy status for the nodes that are being worked on. The action reverts all resources in the workspace to their latest state in local history. The action is added by adding an action contribution to the participants configuration. The configuration is used to describe the properties used to build the participant page that will display the actual synchronize UI.

The participant will initialize the configuration as follows in order to add the local history action group to the context menu:

```
protected void initializeConfiguration(ISynchronizePageConfiguration configuration) {
```

```
  super.initializeConfiguration(configuration);
  configuration.addMenuGroup(
      ISynchronizePageConfiguration.P_CONTEXT_MENU,
      CONTEXT_MENU_CONTRIBUTION_GROUP);
  configuration.addActionContribution(new LocalHistoryActionContribution());
  configuration.addLabelDecorator(new LocalHistoryDecorator());
}
```

Now lets look at how we can provide a custom decoration. The last line of the above method registers the following decorator with the page's configuration.

```
public class LocalHistoryDecorator extends LabelProvider implements ILabelDecorator {
  public String decorateText(String text, Object element) {
    if(element instanceof ISynchronizeModelElement) {
      ISynchronizeModelElement node = (ISynchronizeModelElement)element;
      if(node instanceof IAdaptable) {
        SyncInfo info = (SyncInfo)((IAdaptable)node).getAdapter(SyncInfo.class);
        if(info != null) {
          LocalHistoryVariant state = (LocalHistoryVariant)info.getRemote();
          return text+ " ("+ state.getContentIdentifier() + ")";
        }
      }
    }
    return text;
  }

  public Image decorateImage(Image image, Object element) {
    return null;
  }
}
```

The decorator extracts the resource from the model element that appears in the synchronize view and appends the content identifier of the local history resource variant to the text label that appears in the view.

The last and final piece is to provide a wizard that will create the local history participant. The Team Synchronizing perspective defines a global synchronize action that allows users to quickly create a synchronization. In addition, the ability to create synchronizations in available from the Synchronize view toolbar. To start, create a synchronizeWizards extension point:

```
<extension
         point="org.eclipse.team.ui.synchronizeWizards">
    <wizard
        class="org.eclipse.team.synchronize.example.LocalHistorySynchronizeWizard"
        icon="synced.png"
        description="Creates a synchronization against the latest local history state of all reso
        name="Latest From Local History Synchronize"
        id="ExampleSynchronizeSupport.wizard1"/>
</extension>
```

This will add our wizard to the list and in the wizards finish() method we will simply create our participant and add it to the synchronize manager.

```
LocalHistoryPartipant participant = new LocalHistoryPartipant();
ISynchronizeManager manager = TeamUI.getSynchronizeManager();
manager.addSynchronizeParticipants(new ISynchronizeParticipant[] {participant});
ISynchronizeView view = manager.showSynchronizeViewInActivePage();
view.display(participant);
```

Conclusion

This is a simple example of using the synchronize APIs and we have glossed over some of the details in order to make the example easier to understand. Writing responsive and accurate synchronization support is non−trivial, the hardest part being the management of synchronization information and the notification of synchronization state changes. The user interface, if the one associated with SubscriberParticipants is adequate, is the easy part once the Subscriber implementation is complete. For more examples please refer to the org.eclipse.team.example.filesystem plug−in and browse the subclasses in the workspace of Subscriber and ISynchronizeParticipant.

The next section describes some class and interfaces that can help you write a Subscriber from scratch including how to cache synchronization states between workbench sessions.

Beyond the Basics

If you plan on providing synchronization support and don't have an existing mechanism for managing synchronization state, this section explains how to implementing a Subscriber from scratch. This means that there is no existing synchronization infrastructure and illustrates how to use some API that is provided to maintain the synchronization state.

For the remainder of this example we will make use of a running example. The source code can be found in the file system provider package of the org.eclipse.team.examples.filesystem plug−in. You should check the project out from the CVS repository and use as a reference while you are reading this tutorial.

Implementing a Subscriber From Scratch

This first example assumes that there is no existing infrastructure for maintaining the synchronization state of the local workspace. When implementing a subscriber from scratch, you can make use of some additional API provided in the *org.eclipse.team.core* plug−in. The org.eclipse.team.core.variants package contains two subclasses of Subscriber which can be used to simplify implementation. The first is ResourceVariantTreeSubscriber which will be discussed in the second example below. The second is a subclass of the first: ThreeWaySubscriber. This subscriber implementation provides several helpful classes for managing the synchronization state of a local workspace. If you do not have any existing infrastructure, this is a good place to start.

Implementing a subscriber from scratch will be illustrated using the file system example available in the org.eclipse.team.examples.filesystem plug−in. The code in the following description is kept to a minimum since it is available from the Eclipse CVS repository. Although not technically a three−way subscriber, the file system example can still make good use of this infrastructure. The FTP and WebDAV plug−ins also are built using this infrastructure.

ThreeWaySubscriber

For the file system example, we already had an implementation of a RepositoryProvider that associated a local project with a file system location where the local contents were mirrored. FileSystemSubscriber was created as a subclass of ThreeWaySubscriber in order to make use of a ThreeWaySynchronizer to manage workspace synchronization state. Subclasses of this class must do the following:

- create a ThreeWaySynchronizer instance to manage the local workspace synchronization state.

- create an instance of a ThreeWayRemoteTree subclass to provide remote tree refresh.
    - The class FileSystemRemoteTree was defined for this purpose
- implement a method to create the resource variant handles used to calculate the synchronization state.
    - The class FileSystemResourceVariant (a subclass of CachedResourceVariant) was defined for this
- implement the roots method.
    - The roots for the subscriber are all the projects mapped to the FileSystemProvider. Callbacks were added to FileSystemProvider in order to allow the FileSystemSubscriber to generate change events when projects are mapped and unmapped.

In addition to the subscriber implementation, the get and put operations for the file system provider were modified to update the synchronization state in the ThreeWaySynchronizer. The operations are implemented in the class org.eclipse.team.examples.filesystem.FileSystemOperations.

ThreeWaySynchronizer

ThreeWaySynchronizer manages the synchronization state between the local workspace and a remote location. It caches and persists the local, base and remote timestamps in order to support the efficient calculation of the synchronization state of a resource. It also fires change notifications to any registered listeners. The ThreeWaySubscriber translates these change events into the proper format to be sent to listeners registered with the subscriber.

The ThreeWaySynchronizer makes use of Core scheduling rules and locks to ensure thread safety and provide change notification batching.

ThreeWayRemoteTree

A ThreeWayRemoteTree is a subclass of ResourceVariantTree that is tailored to the ThreeWaySubscriber. It must be overridden by clients to provide the mechanism for fetching the remote state from the server. ResourceVariantTree is discussed in more detail in the next example.

CachedResourceVariant

A CachedResourceVariant is a partial implementation of IResourceVariant that caches any fetched contents for a period of time (currently 1 hour). This is helpful since the contents may be accessed several times in a short period of time (for example, to determine the synchronization state and display the contents in a compare editor). Subclasses must still provide the unique content identifier along with the byte array that can be persisted in order to recreate the resource variant handle.

Building on Top of Existing Workspace Synchronization

Many repository providers may already have a mechanism for managing their synchronization state (e.g. if they have existing plug−ins). The ResourceVariantTreeSubscriber and its related classes provide the ability to build on top of an existing synchronization infrastructure. For example, this is the superclass of all of the CVS subscribers.

ResourceVariantTreeSubscriber

As was mentioned in the previous example, the ThreeWaySubscriber is a subclass of ResourceVariantTreeSubscriber that provides local workspace synchronization using a

ThreeWaySynchronizer. Subclasses of ResourceVariantTreeSubscriber must provide:

- Subclasses of ResourceVariantTree (or AbstractResourceVariantTree) that provide the behavior for traversing and refreshing the remote resource variants and, for subscribers that support three−way comparisons, the base resource variants.
- An implementation of IResourceVariantComparator that performs the two−way or three−way comparison for a local resource and its base and remote resource variants.It is common to also provide a subclass of SyncInfo in order to customize the synchronization state determination algorithm.
- An implementation of the roots method for providing the roots of the subscriber and an implementation of the isSupervised method for determining what resources are supervised by the subscriber.

The other capabilities of the subscriber are implemented using these facilities.

ResourceVariantTree

ResourceVariantTree is a concrete implementation of IResourceVariantTree that provides the following:

- traversal of the resource variant tree
- logic to merge the previous resource variant tree state with the current fetched state.
- caching of the resource variant tree using a ResourceVariantByteStore.

The following must be implemented by subclasses:

- creation of resource variant handles from the cached bytes that represent a resource variant
- fetching of the current remote state from the server
- creation of the byte store instance used to cache the bytes that uniquely identify a resource variant

Concrete implementations of ResourceVariantByteStore are provided that persist bytes across workbench invocations (PersistantResourceVariantByteStore) or cached the bytes only for the current session (SessionResourceVariantByteStore). However, building a subscriber on top of an existing workspace synchronization infrastructure will typically require the implementation of ResourceVariantByteStore subclasses that interface with the underlying synchronizer. For example the ThreeWayRemoteTree makes use of a byte store implementation that stores the remote bytes in the ThreeWaySynchronizer.

The creation of resource variant handles for this example does not differ from the previous example except that the handles are requested from a resource variant tree instance instead of the subscriber.

Program debug and launch support

The resources plug−in in the Eclipse platform allows you to manage a set of source files for a program and compile them using an incremental project builder. Plug−ins can define new builders that handle special resource types, such as source files for a particular programming language. Once an executable program is built with your plug−in's builder, how can you make sure that it gets invoked correctly?

The **org.eclipse.debug.core** plug−in provides the API that allows a program to define a configuration for launching a program. The program can be launched in different modes. For example, it could be launched for regular execution, for debugging, for profiling, or any other mode defined by your plug−in. The Eclipse Java development tooling (JDT) uses the platform debug support to launch Java VM's and the Java debugger.

The **org.eclipse.debug.ui** plug−in includes support for user configuration of launch parameters and utility classes that ease the implementation of powerful debuggers.

There are some shared concepts in launching and debugging programs that are implemented in the platform debug support. However, the best way to understand how to use the platform debug support is to study a robust concrete implementation of launching and debugging, such as the JDT launching and debug tools. We'll review the major concepts of the platform debug support in the context of the JDT concrete implementation.

Plugging in help

The Eclipse platform's help facilities provide you with the raw building blocks to structure and contribute documentation to the platform. It does not dictate structure or granularity of documentation. You can choose the tools and structure for your documentation that suits your needs. The help plug−in allows you to describe your documentation structure to the platform using a table of contents (toc) file.

Your plug−in's online help is contributed using the **org.eclipse.help.toc** extension point. You can either contribute the online help as part of your code plug−in or provide it separately in its own documentation plug−in.

Separating the documentation into a separate plug−in is beneficial in those situations where the code and documentation teams are different groups or where you want to reduce the coupling between the documentation and code.

Advanced features of the help system include context−sensitive help with reference links and the ability to invoke platform code from the documentation. A help browser lets you view, print, and search your documentation. Since 3.1, the new Help view adds dynamic help and federated information search capability.

The best way to demonstrate the contribution of help is to create a documentation plug−in.

Table of Contents (TOC)

Identifier:

org.eclipse.help.toc

Description:

For registering an online help contribution for an individual plug−in.

Each plug−in that contributes help files should in general do the following:

- author the html files, zip html files into doc.zip, and store the zip file in the plug−in directory.
- create TOC files that describe Table of Contents for the help and the necessary topic interleaving. See the syntax below.
- the plugin.xml file should extend the `org.eclipse.help.toc` extension point and specify TOC file(s).

Optionally, a search index can be prebuilt and registered using `index` element in order to performance of the first search attempt. Only one index per plug−in can be registered − multiple `index` elements will result in undefined behaviour.

Configuration Markup:

<!ELEMENT extension (toc* , index?)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

<!ELEMENT toc EMPTY>

<!ATTLIST toc

file CDATA #REQUIRED

primary (true | false) "false"

extradir CDATA #IMPLIED>

- **file** − the name of the TOC file which contains the table of contents or section for this plug−in's online help.

  Configuration Markup for toc file:

```
<!ELEMENT toc (topic | anchor | link)* >
<!ATTLIST toc link_to CDATA #IMPLIED >
<!ATTLIST toc label CDATA #REQUIRED >
<!ATTLIST toc topic CDATA #IMPLIED >

<!ELEMENT topic (topic | anchor | link )* >
<!ATTLIST topic label CDATA #REQUIRED >
<!ATTLIST topic href CDATA #IMPLIED >

<!ELEMENT anchor EMPTY >
<!ATTLIST anchor id ID  #REQUIRED >

<!ELEMENT link EMPTY >
<!ATTLIST link toc CDATA #REQUIRED >
```

In general, a plug−in that needs to provide online help will define its own TOC files. In the end, the help system is configured to be launched as some actions, and the path of the TOC file can be used to do so.

The topic element

All help topic element are contributed as part of the toc container element. They can have a hierarchical structure, or can be listed as a flat list.

The topic element is the workhorse of structure of Table of Contents. There are two typical uses for the topic element:

1. To provide a link to a documentation file − usually an HTML file.
2. To act as a container for other toc, either in the same manifest or another.

1. Topics as links
The simplest use of a topic is as a link to a documentation file.

```
<topic label="Some concept file" href="concepts/some_file.html" />
```

The href attribute is relative to the plug−in that the manifest file belongs to. If you need to access a file in another plug−in, you can use the syntax

```
<topic label="topic in another plug-in"
href="../other.plugin.id/concepts/some_other_file.html" />
```

2. Topics as containers
The next most common use of a topic is to use it as a container for other toc. The container topic itself can always refer to a particular file as well.

```
<topic label="Integrated Development Environment"
href="concepts/ciover.htm" >
  <topic label="Starting the IDE" href="concepts/blah.htm" />
  ...
</topic>
```

The link element

The link element allows to link Table of Contents defined in another toc file. All the topics from the toc file specified in the toc attribute will appear in the table of contents as if they were defined directly in place of the link element. To include toc from api.xml file you could write

```
<topic label="References" >
  ...
  <link toc="api.xml" />
  ...
</topic>
```

The anchor element

The anchor element defines a point that will allow linking other toc files to this navigation, and extending it, without using the link element and referencing other toc files from here. To allow inserting Table of Contents with more topics after the "ZZZ" document you would define an anchor as follows:

```
...
<topic label="zzz" href="zzz.html" />
<anchor id="moreapi" />
...
```

The toc element

The toc element is a Table of Contents that groups topics and other elements defined in this file. The label identifies the table of contents to the user, when it is displayed to the user. The optional topic attribute is the path to a topic file describing the TOC. The optional link_to attribute allows for linking toc from this file into another toc file being higher in the navigation hierarchy. The value of the link_to attribute must specify an anchor in another toc file. To link toc from myapi.xml to api.xml file, specified in another plugin you would use the syntax

```
<toc link_to="../anotherPlugin/api.xml#moreapi" label="My Tool
API"/>
...
<toc />
```

where # character separates toc file name from the anchor identifier.

- **primary** – specifies whether the TOC file is a primary table of contents and is meant to be the master table of contents, or not primary and intended to be integrated into another table of contents.
- **extradir** – specifies relative directory name of containing additional documents that are associated with the table of contents. All help documents in this directory, and all subdirectories, will be indexed, and accessible through the documentation search, even if `topic` elements in the TOC file do not refer to these documents.

<!ELEMENT index EMPTY>

Description:

<!ATTLIST index

path CDATA #REQUIRED>

(**since 3.1**) an optional element that allows declaration of prebuilt search index created from documents contributed by this plug−in.

- **path** − a plug−in−relative path of the prebuilt search index. The index referenced by the path must exist. Missing index will be flagged in the log file. Note that each locale must have a different index. If a plug−in contributes index directories for multiple locales, it should append the locale using standard Eclipse NLS lookup. (e.g. `index/`, `nl/ja/JP/index/`, `nl/en/US/index/` etc.).

Examples:

The following is an example of using the `toc` extension point.

(in file `plugin.xml`)

<extension point=

"org.eclipse.help.toc"

>

<toc file=

"maindocs.html"

primary=

"true"

/>

<toc file=

"task.xml"

/>

<toc file=

"sample.xml"

extradir=

"samples"

/>

<index path=

"index/"

/>

</extension>

(in file `maindocs.xml`)

```
<toc label="Help System Example">
 <topic label="Introduction" href="intro.html"/>
 <topic label="Tasks">
  <topic label="Creating a Project" href="tasks/task1.html">
   <topic label="Creating a Web Project"
href="tasks/task11.html"/>
   <topic label="Creating a Java Project"
href="tasks/task12.html"/>
  </topic>
  <link toc="task.xml" />
  <topic label="Testing a Project" href="tasks/taskn.html"/>
 </topic>
 <topic label="Samples">
  <topic label="Creating Java Project"
href="samples/sample1.html">
   <topic label="Launch a Wizard"
href="samples/sample11.html"/>
   <topic label="Set Options" href="samples/sample12.html"/>
   <topic label="Finish Creating Project"
href="samples/sample13.html"/>
  </topic>
  <anchor id="samples" />
 </topic>
</toc>
```

(in file `tasks.xml`)

```
<toc label="Building a Project">
 <topic label="Building a Project" href="build/building.html">
  <topic label="Building a Web Project"
href="build/web.html"/>
  <topic label="Building a Java Project"
href="build/java.html"/>
 </topic>
</toc>
```

(in file samples.xml)

```
<toc link_to="maindocs.xml#samples" label="Using The Compile
Tool">
 <topic label="The Compile Tool Sample"
href="compilesample/example.html">
  <topic label="Step 1" href="compilesample/step1.html"/>
  <topic label="Step 2" href="compilesample/step2.html"/>
  <topic label="Step 3" href="compilesample/step3.html"/>
  <topic label="Step 4" href="compilesample/step4.html"/>
 </topic>
</toc>
```

Assuming more documents exists with the path starting with "samples", they will not be displayed in the navigation tree, but be accessible using search. It is due to the presence of "extradir" attribute in the element `<toc file="sample.xml" extradir="samples" />` inside `plugin.xml` file. For example searching for "Creating Java Project" could return a document "Other Ways of Creating Java Project", which path is `samples/sample2.html`.

Internationalization The TOC XML files can be translated and the resulting copy (with translated labels) should be placed in nl/<language>/<country> or nl/<language> directory. The <language> and <country> stand for two letter language and country codes as used in locale codes. For example, Traditional Chinese translations should be placed in the nl/zh/TW directory. The nl/<language>/<country> directory has a higher priority than nl/<language>. Only if no file is found in the nl/<language>/<country>, the file residing in nl/<language> will be used. The the root directory of a plugin will be searched last.

The documentation contained in doc.zip can be localized by creating a doc.zip file with translated version of documents, and placing doc.zip in
nl/<language>/<country> or nl/<language> directory. The help system will look for the files under this directories before defaulting to plugin directory.

API Information:

No code is required to use this extension point. All that is needed is to supply the appropriate manifest files mentioned in the plugin.xml file.

Supplied Implementation:

The default implementation of the help system UI supplied with the Eclipse platform fully supports the `toc` extension point.

Plug it in: Hello World meets the workbench

The Eclipse platform is structured as a core runtime engine and a set of additional features that are installed as platform **plug−ins**. Plug−ins contribute functionality to the platform by contributing to pre−defined **extension points**. The workbench UI is contributed by one such plug−in. When you start up the workbench, you are not starting up a single Java program. You are activating a platform runtime which can dynamically discover registered plug−ins and start them as needed.

When you want to provide code that extends the platform, you do this by defining system extensions in your plug−in. The platform has a well−defined set of extension points − places where you can hook into the platform and contribute system behavior. From the platform's perspective, your plug−in is no different than basic plug−ins like the resource management system or the workbench itself.

So how does your code become a plug−in?

- Decide how your plug−in will be integrated with the platform.
- Identify the extension points that you need to contribute in order to integrate your plug−in.
- Implement these extensions according to the specification for the extension points.
- Provide a manifest file (manifest.mf) that describes the packaging and prerequisites for your code, and a plug−in manifest (plugin.xml) that describes the extensions you are defining.

The process for creating a plug−in is best demonstrated by implementing an old classic, "Hello World," as a plug−in. The intention of this example is to give you a flavor of how plug−in development is different from Java application development. We'll gloss over a lot of details in order to get the plug−in built and running. Then we'll look at extension points in more detail, see where they are defined, and learn how plug−ins describe their implementation of an extension.

A minimal plug−in

We all know what "Hello World" looks like in plain old Java without using any user interface frameworks or other specialized libraries.

```
public class HelloWorld {
   public static void main(String[] args) {
      System.out.println("Hello World");
   }
}
```

What happens to this old standard in the context of the Eclipse platform? Instead of thinking of Hello World as a self−contained program, we recast it as an extension of the platform. Since we want to say hello to the world, we need to figure out how to extend the workbench to include our greeting.

When we get deeper into the platform user interface components, we'll do an exhaustive review of the ways that you can extend and customize the workbench UI. For now, let's start with one of the simplest workbench extensions − a view.

You can think of the workbench window as a frame that presents various visual parts. These parts fall into two major categories: views and editors. We will look at editors later. **Views** provide information about some object that the user is working with in the workbench. Views often change their content as the user selects different objects in the workbench.

Hello world view

For our hello world plug−in, we will implement our own view to greet the user with "Hello World."

The plug−in **org.eclipse.ui.workbench** defines most of the public interfaces that make up the workbench API. These interfaces can be found in the package **org.eclipse.ui** and its sub packages. Many of these interfaces have default implementation classes that you can extend to provide simple modifications to the system. In our hello world example, we will extend a workbench view to provide a label that says hello.

The interface of interest is **IViewPart**, which defines the methods that must be implemented to contribute a view to the workbench. The class **ViewPart** provides a default implementation of this interface. In a nutshell, a view part is responsible for creating the widgets needed to show the view.

The standard views in the workbench often display some information about an object that the user has selected or is navigating. Views update their contents based on actions that occur in the workbench. In our case, we are just saying hello, so our view implementation will be quite simple.

Before jumping into the code, we need to make sure our environment is set up for plug−in development...

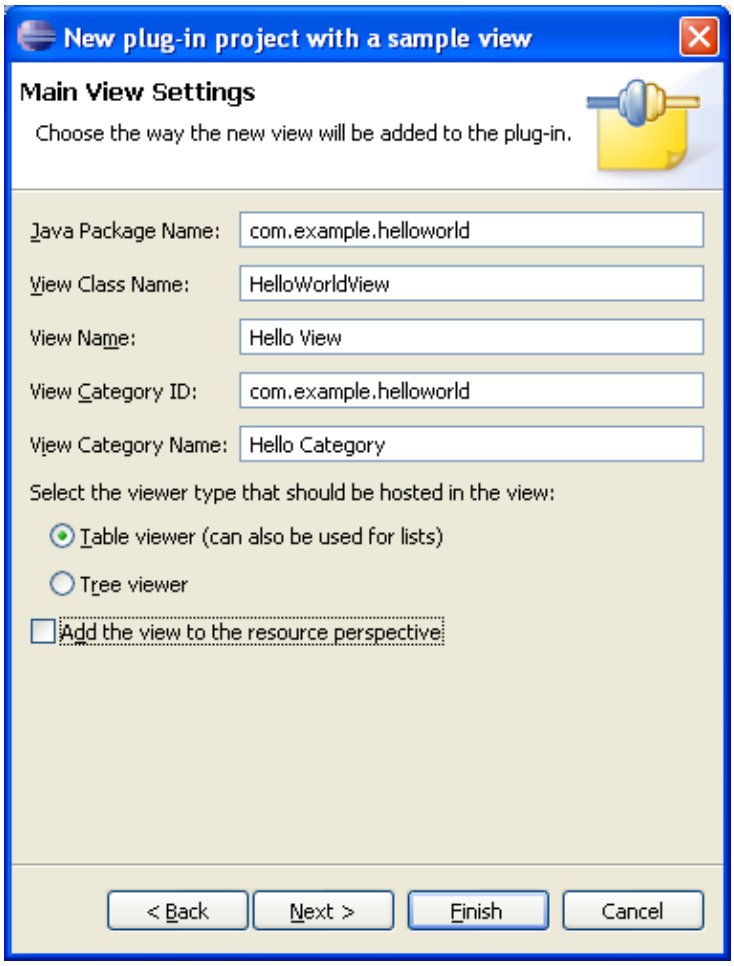
Creating the plug−in project

You can use any Java IDE you wish to build Eclipse plug−ins, but of course the Eclipse SDK provides tooling specific for plug−in development. We'll walk through the steps for building our plug−in with the Eclipse SDK, since this is the typical case. If you are not already familiar with the Eclipse workbench and the Java IDE, consult the Java Development User Guide or PDE Guide for further explanations of the steps we are taking. For now we are focusing on the code, not the tool; however, there are some IDE logistics for getting started.

Creating your plug−in project

You will need to create a project that contains your work. We'll take advantage of some of the code−generation facilities of the Plug−in Development Environment (PDE) to give us a template to start from. This will set up the project for writing Java code and generate the default plug−in manifest files (explained in a moment) and a class to hold our view.

1. Open the **New Project...** wizard (***File > New > Project...***) and choose **Plug−in Project** from the **Plug−in Development** category and click ***Next***.
2. On the ***Plug−in Project*** page, use **com.example.helloworld** as the name for your project and check the box for ***Create a Java project*** (this should be the default). Leave the other settings on the page with their default settings and then click ***Next*** to accept the default plug−in project structure.
3. On the ***Plug−in Content*** page, look at the default settings. The wizard sets ***com.example.helloworld*** as the id of the plug−in. The wizard will also generate a plug−in class for your plug−in and allow you to supply additional information about contributing to the UI. These defaults are acceptable, so click ***Next***.
4. On the ***Templates*** page, check the box for ***Create a plug−in using one of the templates***. Then select the ***Plug−in with a view*** template. Click ***Next***.
5. We want to create a minimal plug−in, so at this point we need to change the default settings to keep things as simple as possible. On the ***Main View Settings*** page, change the suggested defaults as



follows:

- Change the ***Java Package Name*** from ***com.example.helloworld.views*** to ***com.example.helloworld*** (we don't need a separate package for our view).
- Change the ***View Class Name*** to ***HelloWorldView***.
- Change the ***View Name*** to ***Hello View***.
- Leave the default ***View Category Id*** as ***com.example.helloworld***.
- Change the ***View Category Name*** to ***Hello Category***.
- Leave the default viewer type as ***Table viewer*** (we will change this in the code to make it even simpler).
- Uncheck the box for ***Add the view to the resource perspective***.
- Click ***Next*** to proceed to the next page.

6. On the ***View Features*** page, uncheck all of the boxes so that no extra features are generated for the plug−in. Click ***Finish*** to create the project and the plug−in skeleton.
7. When asked if you would like to switch to the Plug−in Development perspective, answer ***Yes***.
8. Navigate to your new project and examine its contents.

The skeleton project structure includes several folders, files, and a Java package. The important files at this stage are the ***plugin.xml*** and ***MANIFEST.MF*** (manifest) files and the Java source code for your plug−in. We'll start by looking at the implementation for a view and then examine the manifest files.

The Hello World view

Now that we've created a project, package, and view class for our plug−in, we're ready to study some code. Here is everything you need in your **HelloWorldView**. Copy the contents below into the class you created,

replacing the auto−generated content.

```
package com.example.helloworld;

import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.SWT;
import org.eclipse.ui.part.ViewPart;

public class HelloWorldView extends ViewPart {
   Label label;
   public HelloWorldView() {
   }
   public void createPartControl(Composite parent) {
      label = new Label(parent, SWT.WRAP);
      label.setText("Hello World");
   }
   public void setFocus() {
      // set focus to my widget.  For a label, this doesn't
      // make much sense, but for more complex sets of widgets
      // you would decide which one gets the focus.
   }
}
```

The view part creates the widgets that will represent it in the **createPartControl** method. In this example, we create an SWT label and set the "Hello World" text into it. This is about the simplest view that can be created.

The Hello World manifests

Before we run the new view, let's take a look at the manifest files that were generated for us. First, double−click the plugin.xml file to open the plug−in editor and select the ***plugin.xml*** tab.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin>
        <extension point="org.eclipse.ui.views">
                <category
                        name="Hello Category"
                        id="com.example.helloworld">
                </category>
                <view
                        name="Hello View"
                        icon="icons/sample.gif"
                        category="com.example.helloworld"
                        class="com.example.helloworld.HelloWorldView"
                        id="com.example.helloworld.HelloWorldView">
                </view>
        </extension>
</plugin>
```

The information about the view that we provided when we created the plug−in project was used to generate an entry in the plugin.xml file that defines our view extension. In the extension definition, we define a **category** for the view, including its **name** and **id**. We then define the view itself, including its **name** and **id**, and we associate it with the **category** using the id we defined for our category. We also specify the **class** that implements our view, **HelloWorldView**.

As you can see, the plug−in manifest file wraps up all the information about our extension and how to run it into a nice, neat package.

The other manifest file that is generated by the PDE is the OSGi manifest, MANIFEST.MF. This file is created in the META−INF directory of the plug−in project, but is most easily viewed by clicking on the ***MANIFEST.MF*** tab of the plug−in editor. The OSGi manifest describes lower−level information about the packaging of the plug−in, using the OSGi bundle terminology. It contains information such as the name of the plug−in (bundle) and the bundles that it requires.

Running the plug−in

We have all the pieces needed to run our new plug−in. Now we need to build the plug−in. If your Eclipse workbench is set up to build automatically, then your new view class should have compiled as soon as you saved the new content. If not, then select your new project and choose ***Project > Build Project***. The class should compile without error.

There are two ways to run a plug−in once it has been built.

1. The plug−in's manifest files and jar file can be installed in the ***eclipse/plugins*** directory. When the workbench is restarted, it will find the new plug−in.
2. The PDE tool can be used to run another workbench from within your current workbench. This ***runtime workbench*** is handy for testing new plug−ins immediately as you develop them from your workbench. (For more information about how a runtime workbench works, check the PDE guide.)

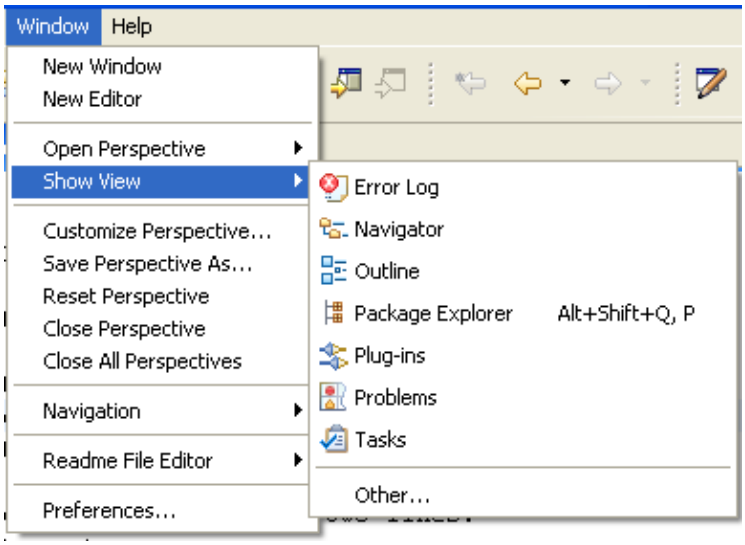
For simplicity, we'll run the new plug−in from within the Eclipse workbench.

Launching the workbench

To launch a runtime workbench, choose ***Run > Run...***. This dialog will show you all the different kinds of ways you can launch a program. Choose ***Eclipse Application***, click ***New*** and accept all of the default settings. This will cause another instance of the Eclipse workbench, the runtime workbench, to start.

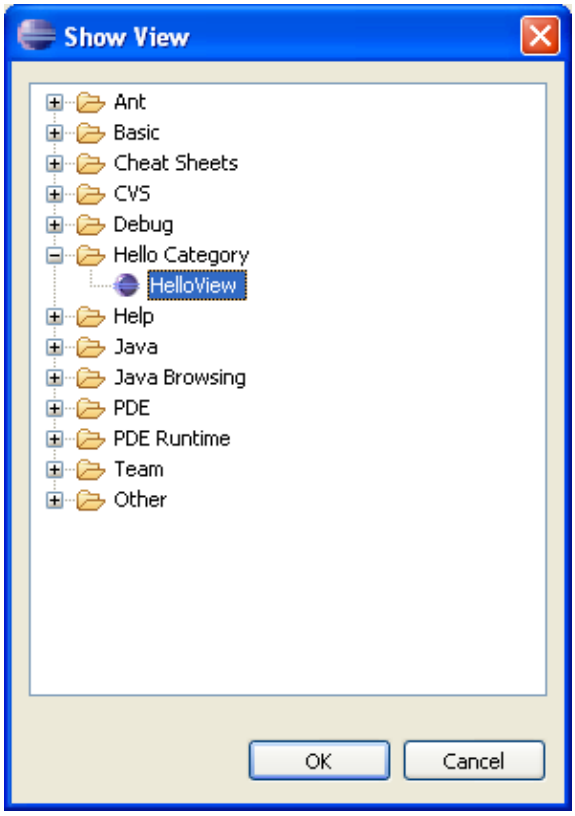
Running Hello World

So where is our new view? We can see all of the views that have been contributed by plug−ins using the **Window > Show View** menu.



This menu shows us what views are available for the current perspective. You can see all of the views that are contributed to the platform (regardless of perspective) by selecting **Other...**. This will display a list of view categories and the views available under each category.

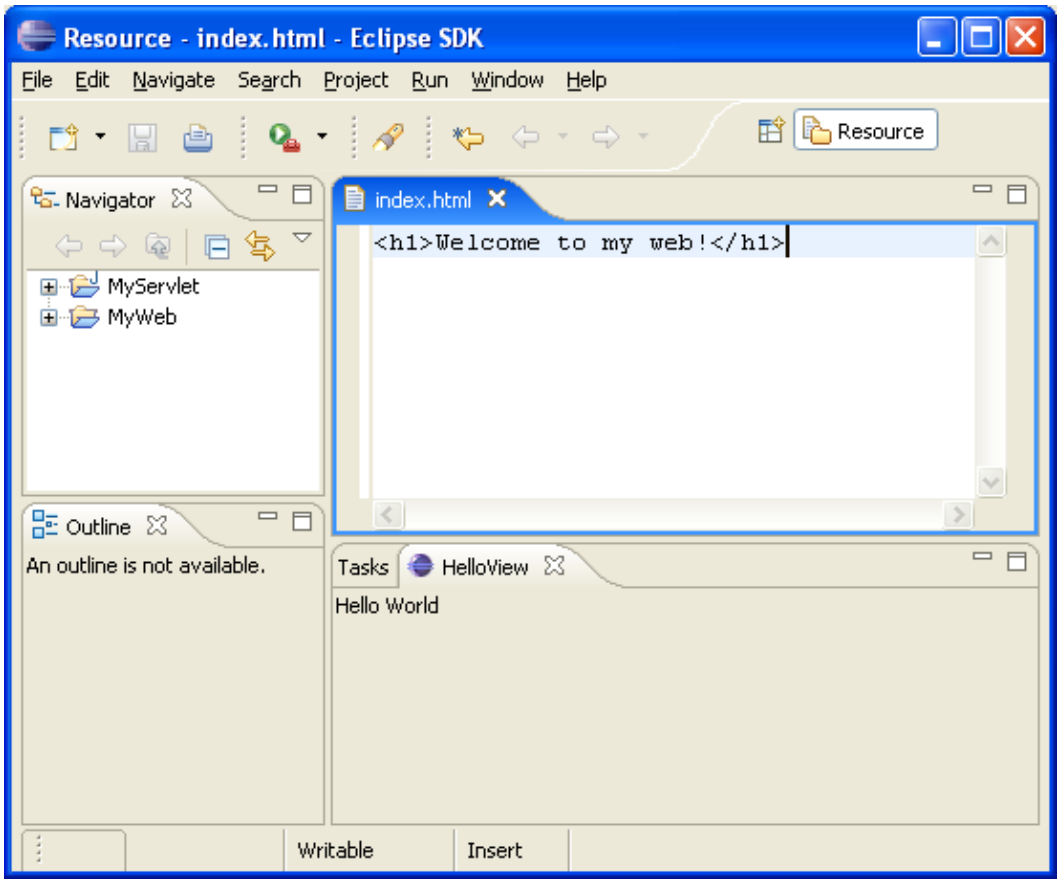
The workbench creates the full list of views by using the extension registry to find all the plug−ins that have provided extensions for the **org.eclipse.ui.views** extension point.



There we are! The view called "Hello View" has been added to the **Show View** window underneath our category "Hello Category." The labels for our category and view were obtained from the extension point configuration markup in the **plugin.xml**.

Up to this point, we still have not run our plug–in code! The declarations we made in the **plugin.xml** (which can be seen by other plug–ins using the extension registry) are enough for the workbench to find out that there is a view called "Hello View" available in the "Hello" category. It even knows what class implements the view. But none of our code will be run until we decide to show the view.

If we choose the "Hello View" view from the **Show View** list, the workbench will activate our plug–in, instantiate and initialize our view class, and show the new view in the workbench along with all of the other views. Now our code is running.



There it is, our first plug–in! We'll cover more specifics about UI classes and extension points later on.

Views

Identifier:

org.eclipse.ui.views

Description:

This extension point is used to define additional views for the workbench. A view is a visual component within a workbench page. It is typically used to navigate a hierarchy of information (like the workspace), open an editor, or display properties for the active editor. The user can make a view visible from the Window > Show View menu or close it from the view local title bar.

In order to reduce the visual clutter in the Show View Dialog, views should be grouped using categories.

Configuration Markup:

<!ELEMENT extension (category | view | stickyView)*>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT category EMPTY>

<!ATTLIST category

id CDATA #REQUIRED

name CDATA #REQUIRED

parentCategory CDATA #IMPLIED>

- **id –** a unique name that will be used to identify this category
- **name –** a translatable name that will be used in the UI for this category
- **parentCategory –** an optional path composed of category IDs separated by '/'. This allows the creation of a hierarchy of categories.

<!ELEMENT view (description?)>

<!ATTLIST view

id CDATA #REQUIRED

name CDATA #REQUIRED

category CDATA #IMPLIED

class CDATA #REQUIRED

icon CDATA #IMPLIED

fastViewWidthRatio CDATA #IMPLIED

allowMultiple (true | false) >

- **id –** a unique name that will be used to identify this view
- **name –** a translatable name that will be used in the UI for this view
- **category –** an optional attribute that is composed of the category IDs separated by '/'. Each referenced category must be declared in a corresponding category element.
- **class –** a fully qualified name of the class that implements `org.eclipse.ui.IViewPart`. A common practice is to subclass `org.eclipse.ui.part.ViewPart` in order to inherit the default functionality.
- **icon –** a relative name of the icon that will be associated with the view.
- **fastViewWidthRatio –** the percentage of the width of the workbench that the view will take up as an active fast view. This must be defined as a floating point value and lie between 0.05 and 0.95. If no value is supplied, a default ratio will be used.
- **allowMultiple –** flag indicating whether this view allows multiple instances to be created using IWorkbenchPage.showView(String id, String secondaryId). The default is false.

<!ELEMENT description (#PCDATA)>

an optional subelement whose body should contain text providing a short description of the view.

<!ELEMENT stickyView EMPTY>

<!ATTLIST stickyView

id CDATA #REQUIRED

location (RIGHT|LEFT|TOP|BOTTOM)

closeable (true | false)

moveable (true | false) >

A sticky view is a view that will appear by default across all perspectives in a window once it is opened. Its initial placement is governemed by the location attribute, but nothing prevents it from being moved or closed by the user. Use of this element will only cause a placeholder for the view to be created, it will not show the view. Please note that usage of this element should be done with great care and should only be applied to views that truely have a need to live across perspectives. Since 3.0

- **id** – the id of the view to be made sticky.
- **location** – optional attribute that specifies the location of the sticky view relative to the editor area. If absent, the view will be docked to the right of the editor area.
- **closeable** – optional attribute that specifies wether the view should be closeable. If absent it will be closeable.
- **moveable** – optional attribute that specifies wether the view should be moveable. If absent it will be moveable.

Examples:

The following is an example of the extension point:

<extension point=

"org.eclipse.ui.views"

>

<category id=

"com.xyz.views.XYZviews"

name=

"XYZ"

/>

<view id=

"com.xyz.views.XYZView"

name=

"XYZ View"

category=

"com.xyz.views.XYZviews"

class=

"com.xyz.views.XYZView"

icon=

"icons/XYZ.gif"

/>

</extension>

The following is an example of a sticky view declaration:

<extension point=

"org.eclipse.ui.views"

>

<stickyView id=

"com.xyz.views.XYZView"

/>

</extension>

API Information:

The value of the `class` attribute must be a fully qualified name of the class that implements `org.eclipse.ui.IViewPart`. It is common practice to subclass `org.eclipse.ui.part.ViewPart` when developing a new view.

Supplied Implementation:

The Eclipse Platform provides a number of standard views including Navigator, Properties, Outline and Tasks. From the user point of view, these views are no different from any other view provided by the plug−ins. All the views can be shown from the "Show View" submenu of the "Window" menu. The position of a view is persistent: it is saved when the view is closed and restored when the view is reopened in a single session. The position is also persisted between workbench sessions.

Beyond the basics

Hopefully you've gotten a flavor of how you can contribute code in the form of an extension, and package that functionality into a plug−in. From here, you can start diving into more detail:

- Basic workbench extension points
- Workbench menu contributions
- Advanced workbench concepts
- Workbench wizard extension points

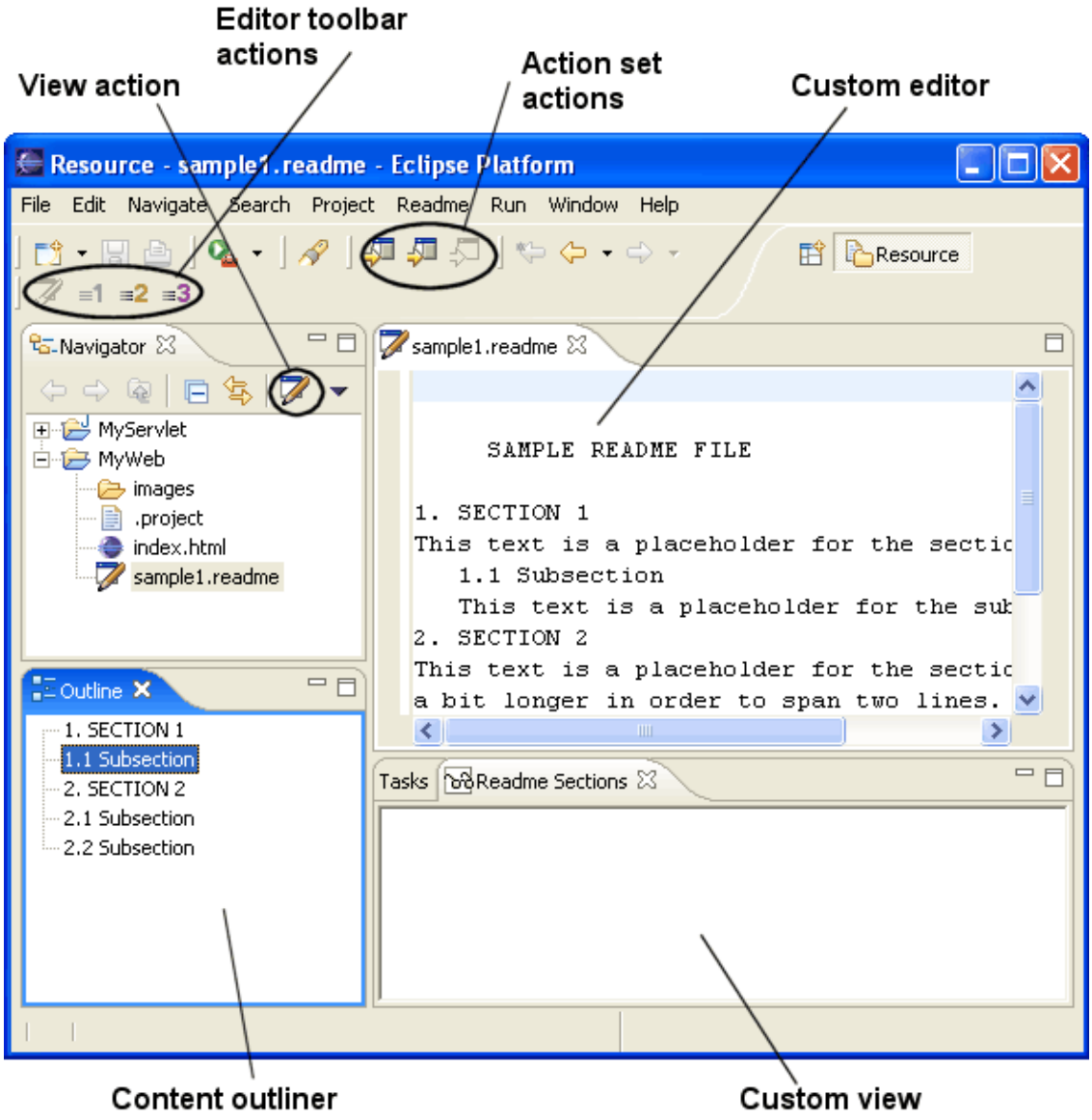
A complete list of extension points can be found in the Platform Extension Point Reference.

Basic workbench extension points

The workbench defines extension points that allow plug−ins to contribute behaviors to existing views and editors or to provide implementations for new views and editors. We are going to take a look at the contributions to these extension points from one of the workbench sample applications, the readme tool.

The readme tool is a plug−in that provides custom editing and navigation for a specific resource , a **.readme** file. The example shows many typical (but simplified) ways that extensions can be used to provide specialized tools.

The readme tool contributes to the menus of the navigator view, adds editor related actions to the workbench menus and tool bar, defines a custom view and content outliner, and defines markers and marker resolutions. The figure below shows some of the customized features added to the workbench by the readme tool.



The readme tool also contributes preference and properties pages to the workbench. Later we'll also look at some wizard contributions in Dialogs and wizards.

The readme tool is located in the **org.eclipse.ui.examples.readmetool** package. The **readmetool.jar** and **plugin.xml** can be found in the **org.eclipse.ui.examples.readmetool** directory underneath the **plugins** directory. To follow along, you will need to make sure that you have installed the platform examples. (See the Examples Guide for more information.)

The readme tool implements many different workbench extensions. We will start with one of the simplest workbench extension points, a view. We'll continue by looking at additional readme tool extensions.

Installing the examples

Installing examples via the Update Manager

The Eclipse SDK examples are found on the Eclipse project update site at http://update.eclipse.org/updates. To locate and install them into a product:

1. Open the main update manager by clicking **Help > Software Updates > Find and Install**. This opens the Install Wizard.
2. Select "Search for new features" and click **Next**.
3. In the sites to search page, add an update site by clicking on **Add Update Site** button: In the Add Update Site dialog that opens, give the site the name "Eclipse.org" and URL "http://update.eclipse.org/updates". Note: this step is not needed if the site is already there.
4. Click **Next** and wait for the search results to return all the features found on that site.
5. Select the SDK Examples feature and click **Next**.
6. Accept the license and click **Next**.
7. Select the directory into which the example feature is to be installed and hit **Next**.
8. Click **Install** to allow the downloading and installing to proceed.
9. Click **Yes** when asked to exit and restart the workbench for the changes to take effect. The examples are now installed in the workbench. Note: you can also click on **Apply Now** to dynamically install the examples into the current configuration.

Installing examples manually

To install the examples without the Update Manager, download the appropriate Eclipse SDK Example zip file from the Eclipse project web site at http://www.eclipse.org/downloads.

The workbench should not be running while the examples are being installed. Extract the contents of the zip file to the root directory of your Eclipse installation.

For example, if you installed the Eclipse Project SDK on **d:\eclipse−sdk** then extract the contents of the examples zip file to **d:\eclipse−sdk** (the subdirectories named **eclipse** should line up).

Start the workbench. The workbench will report that updates have been detected; accept them. The examples are now installed in the workbench.

You can verify that examples have been installed by looking for **File > New > Example...** in the workbench menu bar.

Example – Readme Tool

Introduction

The Readme editor shows how to define your own extension points for use by other plugins. It also shows how to create extensions for resource popup menu entries, new resource wizards, file editors on an extension (.readme), a custom view and property pages.

Running the example

To start using this example create a file with the .readme extension using the file creation wizard or create one using the example creation wizard. The additional view provided by this example can be seen by selecting Window > Show View > Other and expanding the Readme section. The view action can be seen by clicking on the readme icon on the Navigator View.

Creating a new readme file

Create a new file with file extension .readme. From the File menu, select New and then select Other... from the sub menu. Click on Simple in the wizard, then select File in the list on the left. Click on Next to supply the file name (make sure the extension is .readme) and the folder in which the file should be contained.

Example creation wizards

From the File menu, select New and from the sub menu select Example... Now select Example Creation Wizards. Select Readme File. Click Next. Select the folder in which you want the file to be created. Type the name of the file with a .readme extension. Click Finish.

Readme view extension action

In the Navigator View, select a file with extension .readme. If there isn't one create a file with that extension. On the local toolbar of the Navigator View, you will see a button whose hover help says Run Readme View Extension. Click on this button. A dialog will popup saying View Action executed.

Popup menus

In the Navigator View, select a file with extension .readme. If there isn't one create a file with that extension. Select the file to bring up the popup menu. Notice there is a menu item that says Show Readme Action in the popup menu. Choose this menu item to open a dialog that says Popup Menu Action Executed.

Preference page

From the Window menu, select Preferences. Click on the page called Readme Example. This shows an example of a preference page.

Property page

In the Navigator View, select a file with extension .readme. If there isn't one create a file with that extension. Select the file to bring up the popup menu, then select the Properties menu item. Click on the page called Readme Tool to see an example of a property page.

Readme file editor

The Readme File Editor is the default editor for files of type *.readme. Create a file with extension .readme and open the file by double clicking on it, or by bringing up the popup menu, selecting Open With, then selecting Readme File Editor from the sub menu. Notice the editor has an icon with a pencil. This is the editor that the readme tool uses by default for files of type *.readme.

Readme Editor Actions

This demonstrates an example of actions that are applicable only to a particular editor. When a readme file editor has focus, notice 4 additional tool bar buttons – Run Readme Editor Extension, Readme Editor Action 1, Readme Editor Action 2, Readme Editor Action 3.

A pull down menu named Readme appears when a readme file editor has focus. It contains the actions previously described: Readme Editor Action 1, Readme Editor Action 2, Readme Editor Action 3.

Readme sections view

To see this Readme Sections view, from the Window menu select Show View, then select Other... from the sub menu. Expand the Readme item and then select Readme Sections. This will show a list of the sections in the current *.readme file when a .readme file is selected in the Navigator View. You can also see the structure of a *.readme file in the Outline view.

A file with extension .readme can be broken down into sections when each section begins with a number. For example, if the following text were entered into the readme file editor, the readme tool would detect 2 sections. To see how sections are detected in the readme tool, type some text in the readme file editor, save the file by either typing CTRL–S or selecting File–>Save. Open the Readme Sections view and select the .readme file in the Navigator View.

Example text:

99.1 This is my first section
This is some text in my first section.

99.1.1 This is a sub section
This is some text in my sub–section.

Drag and Drop

The Drag and Drop functionality can be seen by selecting a section in the Outline View and dragging the selection over top of a text file. The contents of the selection will be appended to the file.

Help contribution

The readme tool example also demonstrates how to use and implement context help on all of the extensions it supplies – a view, various actions and a wizard page. To get context help on an action, hover over the menu item, but do no select it, then hit the F1 key. You can also get context sensitive (F1) help on the Readme Sections view and the Example Creation Wizards page (in the New wizard).

Details

The Readme Tool example declares one extension point and supplies a number of extensions. The extensions supplied are quite comprehensive in understanding how the Workbench functions, as it utilizes a number of the more interesting extension points declared by the workbench. Supplied extensions included in this example are views and view actions, preference pages, property pages, wizards, editors and editor actions, popup menus, action sets, help contributions, help contexts, and drop actions.

This example also supplies an extension point declared in the plug−in.. The class IReadmeFileParser is required for any plug−in that uses the org.eclipse.ui.examples.readmetool.sectionParser extension that this example defines. The class DefaultSectionParser is an example implementation of IReadmeFileParser.

The class ReadmeEditor implements IEditorPart and is defined as an editor on files with the extension .readme in the plugin.xml using the org.eclipse.ui.editors extension point. The class ReadmeSectionsView implements IViewPart and is defined as a view using the org.eclipse.ui.views extension point. This extension point also defines a category for the view for use in view selection.

Two types of preference settings are defined in this example, workbench preferences and resource properties. The workbench preference is defined in class ReadmePreferencePage which implements IWorkbenchPreferencePage so that it will be added to the Window−>Preferences dialog. The class is defined in the extension point org.eclipse.ui.preferencePages in the plugin.xml. The two resource properties pages are ReadmeFilePropertyPage and ReadmeFilePropertyPage2 both of which implement IWorkbenchPropertyPage. They are both defined to be invoked on the IFile type by the objectClass tag in the plugin.xml in the org.eclipse.ui.propertyPages extension point.

The class ReadmeCreationWizard implements INewWizard and is defined in the org.eclipse.ui.newWizards extension point in the plugin.xml. This extension point also defines the category that the wizard that is shown when the user selects File−>New−>Example....

Several action stubs are added to this example. The action set declares a menu labeled Readme File Editor to be included in the workbench window menu bar using the extension point org.eclipse.ui.actionSets. It also defines an action for the workbench toolbar and menu bar using the tags toolbarPath and menubarPath. It uses the class WindowActionDelegate which implements IWorkbenchWindowActionDelegate to implement the action. The action for the popup menu is defined as an objectContribution by the class PopupMenuActionDelegate in the extension point org.eclipse.ui.popupMenus. PopupMenuActionDelegate implements IObjectActionDelegate and uses the IWorkbenchPart provided to open a message dialog. The view action ViewActionDelegate is defined in the extension point org.eclipse.ui.viewActions and implements IViewActionDelegate. The View it appears in is defined by the tag targetID which in this example is org.eclipse.ui.views.ResourceNavigator. The editor action is defined by the class EditorActionDelegate which implements IEditorActionDelegate and is added using the org.eclipse.ui.editorActions extension point. The editor that it is applied to is defined by the tag targetID which in this example is defined on org.eclipse.ui.examples.readmetool.ReadmeEditor.

The class ReadmeDropActionDelegate implements IDropDelegate. IDropDelegates are informed every time there is a drop action performed in the workbench . The extension point for this action is org.eclipse.ui.dropActions.

Workbench menu contributions

We've seen several different extension points that contribute to various menus and toolbars in the workbench. How do you know which one to use? The following table summarizes the various menu contributions and their use.

Extension point name	Location of Actions	Details
viewActions	Actions appear in a specific view's local toolbar and local pulldown menu.	Contribute an action class that implements **IViewActionDelegate**. Specify the id of the contribution and the id of the target view that should show the action. The label and

		image dictate the appearance of the action in the UI. The path specifies the location relative to the view's menu and toolbar items.
editorActions	Actions are associated with an editor and appear in the workbench menu and/or tool bar.	Contribute an action class that implements **IEditorActionDelegate**. Specify the id of the contribution and the id of the target editor that causes the action to be shown. The label and image specify the appearance of the action in the UI. Separate menu and toolbar paths specify the existence and location of the contribution in the workbench menu and toolbar.
popupMenus	Actions appear in the popup menu of an editor or view. Actions associated with an object type show up in all popups of views and editors that show the object type. Actions associated with a specific popup menu appear only in that popup menu.	**Object contributions** specify the type of object for which the action should appear in a popup menu. The action will be shown in all view and editor popups that contain the object type. Provide an action class that implements **IObjectActionDelegate**. **Viewer contributions** specify the id of the target popup menu in which the menu item should appear. Provide an action class that implements **IEditorActionDelegate** or **IViewActionDelegate**.
actionSets	Actions appear in the workbench main menus and toolbar. Actions are grouped into action sets. All actions in an action set will show up in the workbench menus and toolbars according to the user's selection of action sets and the current perspective shown in the workbench. May be influenced by actionSetPartAssociations (below).	Contribute an action class that implements **IWorkbenchWindowActionDelegate** or **IWorkbenchWindowPulldownDelegate**. Specify the **name** and **id** of the action set. Enumerate all of the actions that are defined for that action set. For each action, separate menu and toolbar paths specify the existence and location of the contribution in the workbench menu and toolbar.
actionSetPartAssociations	Actions sets are shown only when the specified views or editors are active. This is ignored if the user has customized the current perspective.	Specify an action set by **id** and followed by one or more parts (by **id**) that must be active in the current perspective in order to show the action set.

Advanced workbench concepts

Plugging into the workbench looks at the basic workbench extension points in the context of the readme tool example. However, there are many more extension points available for contributing to the workbench.

The next topics cover additional workbench extensions and concepts that you will likely encounter once you've implemented your plug−in and have begun to refine its function. In order to understand the next few topics, you should already be familiar with

- Plugging into the Workbench
- JFace UI framework
- SWT
- Resources overview

Since the readme tool does not contribute to all of these extension points, we will look at example extensions that are implemented by the platform workbench, the platform help system, and Java tooling (JDT).

When using any of these workbench features, it is a good idea to keep the rules of engagement for threading in mind. See The workbench and threads for more information.

Plugging into the workbench

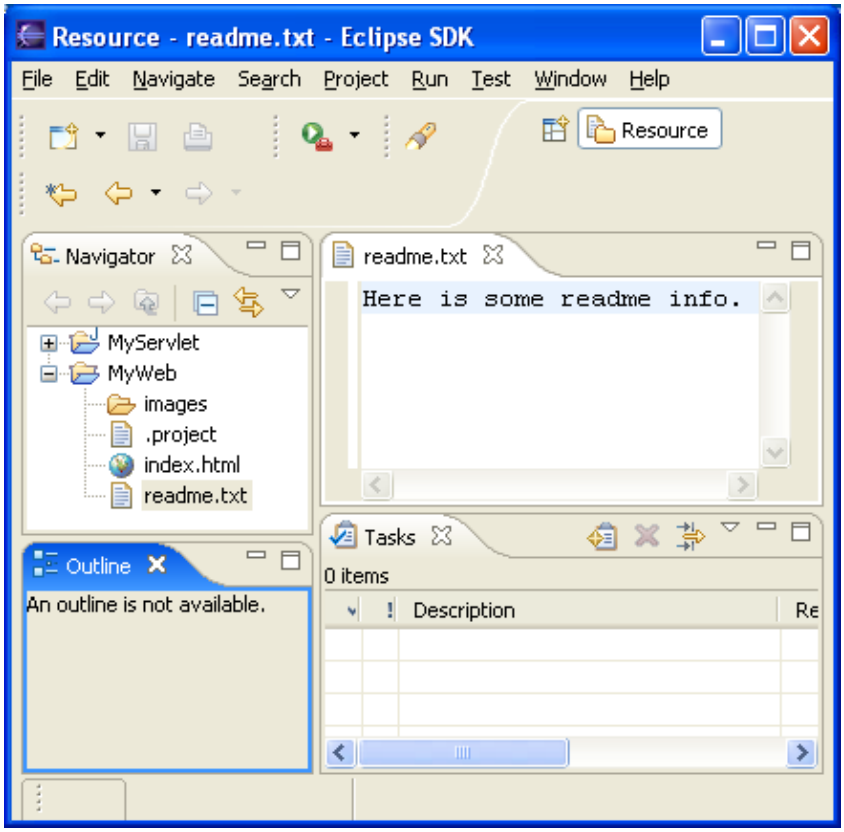
By now, you should be quite familiar with the operation of the workbench and how it uses views and editors to display information. If not, read the quick tour of the workbench below.

The sections following the quick tour will look at the workbench user interface from an API perspective. We will show how a plug–in can contribute to the workbench UI.

Quick tour of the workbench

The workbench is the cockpit for navigating all of the function provided by plug–ins. By using the workbench, we can navigate resources and we can view and edit the content and properties of these resources.

When you open your workbench on a set of projects, it looks something like this.

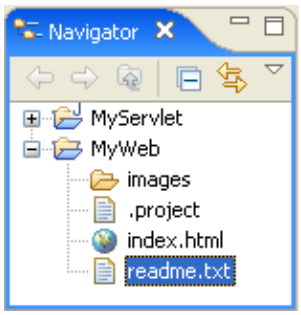


The workbench is just a frame that can present various visual parts. These parts fall into two major categories: **views** and **editors**.

- **Editors** allow the user to edit something in the workbench. Editors are "document–centric," much like a file system editor. Like file system editors, they follow an open–save–close lifecycle. Unlike file system editors, they are tightly integrated into the workbench.
- **Views** provide information about some object that the user is working with in the workbench. Views often change their content as the user selects different objects in the workbench. Views often support editors by providing information about the content in the active editor.

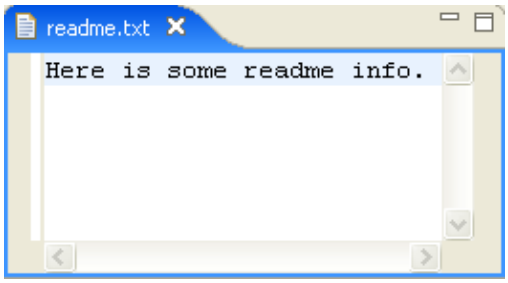
Views

The workbench provides several standard views that allow the user to navigate or view something of interest. For example, the resource navigator lets the user navigate the workspace and select resources.



Editors

Editors allow the user to open, edit, and save objects. The workbench provides a standard editor for text resources.



Additional editors, such as Java code editors or HTML editors, can be supplied by plug−ins

The JFace UI framework

We've seen that the workbench defines extension points for plug−ins to contribute UI function to the platform. Many of these extension points, particularly wizard extensions, are implemented using classes in the **org.eclipse.jface.*** packages. What's the distinction?

JFace is a UI toolkit that provides helper classes for developing UI features that can be tedious to implement. JFace operates above the level of a raw widget system. It provides classes for handling common UI programming tasks:

- **Viewers** handle the drudgery of populating, sorting, filtering, and updating widgets.
- **Actions** and **contributions** introduce semantics for defining user actions and specifying where to make them available.
- **Image** and **font registries** provide common patterns for handling UI resources.
- **Dialogs** and **wizards** define a framework for building complex interactions with the user.

JFace frees you up to focus on the implementation of your specific plug−in's function, rather than focusing on the underlying widget system or solving problems that are common in almost any UI application.

JFace and the workbench

Where does JFace end and the workbench begin? Sometimes the lines aren't so obvious. In general, the JFace APIs (from the packages **org.eclipse.jface.***) are independent of the workbench extension points and APIs. Conceivably, a JFace program could be written without using any workbench code at all.

The workbench makes use of JFace but attempts to reduce dependencies where possible. For example, the workbench part model (**IWorkbenchPart**) is designed to be independent of JFace. We saw earlier that views and editors can be implemented using SWT widgets directly without using any JFace classes. The workbench attempts to remain "JFace neutral" wherever possible, allowing programmers to use the parts of JFace they find useful. In practice, the workbench uses JFace for much of its implementation and there are references to JFace types in API definitions. (For example, the JFace interfaces for **IMenuManager**, **IToolBarManager**, and **IStatusLineManager** show up as types in the workbench **IActionBar** methods.)

When using JFace API, it's a good idea to keep in mind the rules of engagement for using background threads. See The workbench and threads for more information.

JFace and SWT

The lines between SWT and JFace are much cleaner. SWT does not depend on any JFace or platform code at all. Many of the SWT examples show how you can build a standalone application.

JFace is designed to provide common application UI function on top of the SWT library. JFace does not try to "hide" SWT or replace its function. It provides classes and interfaces that handle many of the common tasks associated with programming a dynamic UI using SWT.

The relationship between JFace and SWT is most clearly demonstrated by looking at viewers and their relationship to SWT widgets.

The Standard Widget Toolkit

The Standard Widget Toolkit (SWT) is a widget toolkit for Java developers that provides a portable API and tight integration with the underlying native OS GUI platform.

Many low level UI programming tasks are handled in higher layers of the Eclipse platform. For example, JFace viewers and actions provide implementations for the common interactions between applications and widgets. However, knowledge of SWT is important for understanding how the rest of the platform works.

Portability and platform integration

SWT defines a common portable API that is provided on all supported platforms, and implements the API on each platform using native widgets wherever possible. This allows the toolkit to immediately reflect any changes in the underlying OS GUI look and feel while maintaining a consistent programming model on all platforms.

Resources overview

An essential plug−in for Eclipse IDE applications is the resources plug−in (named **org.eclipse.core.resources**). The resources plug−in provides services for accessing the projects, folders, and files that a user is working with.

For more information on workspace resources, check out some of the following documents:

- Resources and the workspace
- Resources and the local file system
- Resource properties
- Project−scoped preferences
- File encoding and content types
- Linked resources
- Resource markers
- Modifying the workspace
    - Batching resource changes
    - Tracking resource changes
    - Concurrency and the workspace
- Incremental project builders
- Derived resources
- Workspace save participation
- Project natures
- Resource modification hooks
- Refresh providers

Resources and the workspace

The central hub for your user's data files is called a **workspace**. You can think of the platform workbench as a tool that allows the user to navigate and manipulate the workspace. The resources plug−in provides APIs for creating, navigating, and manipulating resources in a workspace. The workbench uses these APIs to provide this functionality to the user. Your plug−in can also use these APIs.

From the standpoint of a resource−based plug−in, there is exactly one workspace, and it is always open for business as long as the plug−in is running. The workspace gets opened automatically when the resources plug−in is activated, and closed when the platform is shut down. If your plug−in requires the resources plug−in, then the resources plug−in will be started before your plug−in, and the workspace will be available to you.

The workspace contains a collection of resources. From the user's perspective, there are three different types of resources: **projects**, **folders**, and **files**. A project is a collection of any number of files and folders. It is a container for organizing other resources that relate to a specific area. Files and folders are just like files and directories in the file system. A folder contains other folders or files. A file contains an arbitrary sequence of bytes. Its content is not interpreted by the platform.

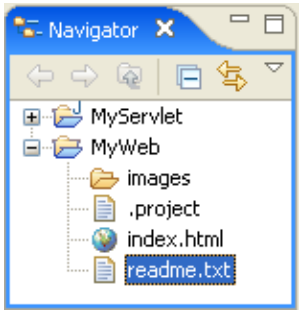
A workspace's resources are organized into a tree structure, with projects at the top, and folders and files underneath. A special resource, the workspace root resource, serves as the root of the resource tree. The workspace root is created internally when a workspace is created and exists as long as the workspace exists.

A workspace can have any number of projects, each of which can be stored in a different location on disk.

The workspace resource namespace is always case−sensitive and case−preserving. Thus the workspace allows multiple sibling resources to exist with names that differ only in case. The workspace also doesn't put any restrictions on valid characters in resource names, the length of resource names, or the size of resources on disk. Of course, if you store resources on a disk that is not case−sensitive, or that does have restrictions on resource names, then those restrictions will show through when you actually try to create and modify resources.

A sample resource tree

The tree below (represented in the workbench navigator view) illustrates a typical hierarchy of resources in a workspace. The (implied) root of the tree is the workspace root. The projects are immediate children of the workspace root. Each node (other than the root) is one of the three kinds of resources, and each has a name that is different from its siblings.



Resource names are arbitrary strings (almost −− they must be legal file names). The platform itself does not dictate resource names, nor does it specify any names with special significance. (One exception is that you cannot name a project "**.metadata**" since this name is used internally.)

Projects contain files and folders, but not other projects. Projects and folders are like directories in a file system. When you delete a project, you will be asked whether you want to delete all of the files and folders that it contains. Deleting a folder from a project will delete the folder and all of its children. Deleting a file is analogous to deleting a file in the file system.

Resources and the local file system

When the platform is running and the resources plug−in is active, the workspace is represented by an instance of **IWorkspace**, which provides protocol for accessing the resources it contains. An **IWorkspace** instance represents an associated collection of files and directories in the local file system. You can access the workspace from the resources plug−in class (defined in **org.eclipse.core.resources**).

```
IWorkspace workspace = ResourcesPlugin.getWorkspace();
```

When the resources plug−in is not running, the workspace exists solely in the local file system and is viewed or manipulated by the user via standard file−based tools. Let's look at what a workspace looks like on disk as we explain the resources plug−in API.

Our sample tree on disk

When you installed the platform SDK, you unzipped the files into a directory of your choosing. We will call this directory the platform root directory. This is the directory that contains the **plugins** directory, among others. Inside the platform root directory, there is a **workspace** directory which is used to hold the resources that are created and manipulated by the platform. If you look in your **workspace** directory, you'll see separate subdirectories for each project that exists in the workspace. Within these subdirectories are the folders and files that each project contains.

If the SDK in our example is installed in **c:\MySDK**, then inside the **c:\MySDK\workspace** directory we find subdirectories named after the workspace's projects, **MyWeb** and **MyServlet**. These are called the projects' content directories. Content directories are created by the platform when the user creates a project.

Inside each directory, we find the files and folders within the project, laid out exactly the same as they are in the workspace's resource tree. All file names are the same, and the files' contents are the same whether accessed from the file system or from the workspace. The only surprise is the **.project** file, explained in a moment.

```
C:\MySDK\workspace  (workspace root)
   .metadata\ (platform metadata directory
   MyWeb\ (project content directory for MyWeb)
      .project
      index.html
      images\
         logo.png
   MyServlet\ (project content directory for MyServlet)
      .project
      src\
         main.java
      bin\
         main.class
```

The platform has a special **.metadata** directory for holding platform internal information. The **.metadata** directory of a workspace is considered to be a "black box." Important information about the workspace structure, such as a project's references or a resource's properties, is stored in the metadata portion of the workspace and should only be accessed by tools through the platform API. These files should never be edited or manipulated using generic file system API.

In addition, each project has its own **.project** file, where metadata about the project is kept. This file is basically an on−disk equivalent of the information found in a project's **IProjectDescription**.

Apart from the **.metadata** directory and the **.project** files, the folders and files in the workspace directory are fair game for other tools. The files and folders can be manipulated by non−integrated tools, such as text editors and file system utilities. The only issue is that the user must be careful when editing these files both in the workbench and externally. (This is no different than when a user edits a file using two independent stand−alone tools.) The workbench provides refresh operations to reconcile the workspace view of resources with the actual state in the file system and periodically refreshes the workspace based on the state of the file system.

Our sample tree in code

The resource API allows us to manipulate this resource tree in code. Here we will look at some code snippets for a quick taste of the resource API. The resource API is defined in a series of interfaces in

org.eclipse.core.resources. There are interfaces for all of the resource types, such as **IProject**, **IFolder**, and **IFile**. Extensive common protocol is defined in **IResource**. We also make use of the **org.eclipse.core.runtime** interface **IPath**, which represents segmented paths such as resource or file system paths.

Manipulating resources is very similar to manipulating files using **java.io.File**. The API is based on **handles**. When you use API like **getProject** or **getFolder**, you are returned a handle to the resource. There is no guarantee or requirement that the resource itself exists until you try to do something with the handle. If you expect a resource to exist, you can use **exists** method to ensure this is the case.

To navigate the workspace from a plug−in, we must first obtain the **IWorkspaceRoot**, which represents the top of the resource hierarchy in the workspace.

```
IWorkspaceRoot myWorkspaceRoot = ResourcesPlugin.getWorkspace().getRoot();
```

Once we have a workspace root, we can access the projects in the workspace.

```
IProject myWebProject = myWorkspaceRoot.getProject("MyWeb");
// open if necessary
if (myWebProject.exists() && !myWebProject.isOpen())
   myWebProject.open(null);
```

Before we can manipulate a project, we must open it. Opening the project reads the project's structure from disk and creates the in−memory object representation of the project's resource tree. Opening a project is an explicit operation since each open project consumes memory to represent the resource tree internally and open projects participate in various resource lifecycle events (such as building) which can be lengthy. In general, closed projects cannot be accessed and will appear empty even though the resources are still present in the file system.

You'll notice that many of these resource examples pass a **null** parameter when manipulating resources. Many resource operations are potentially heavyweight enough to warrant progress reporting and user cancellation. If your code has a user interface, you will typically pass an **IProgressMonitor**, which allows the resources plug−in to report progress as the resource is manipulated and allows the user to cancel the operation if desired. For now, we simply pass **null**, indicating no progress monitor.

Once we have an open project, we can access its folders and files, as well as create additional ones. In the following example we create a file resource from the contents of a file located outside of our workspace.

```
IFolder imagesFolder = myWebProject.getFolder("images");
if (imagesFolder.exists()) {
   // create a new file
   IFile newLogo = imagesFolder.getFile("newLogo.png");
   FileInputStream fileStream = new FileInputStream(
      "c:/MyOtherData/newLogo.png");
   newLogo.create(fileStream, false, null);
   // create closes the file stream, so no worries.
}
```

In the example above, the first line obtains a handle to the images folder. We must check that the folder **exists** before we can do anything interesting with it. Likewise, when we get the file **newLogo**, the handle does not represent a real file until we create the file in the last line. In this example, we create the file by populating it with the contents of **logo.png**.

The next snippet is similar to the previous one, except that it copies the **newLogo** file from the original logo rather than create a new one from its contents.

```
IFile logo = imagesFolder.getFile("logo.png");
if (logo.exists()) {
   IPath newLogoPath = new Path("newLogo.png");
   logo.copy(newLogoPath, false, null);
   IFile newLogo = imagesFolder.getFile("newLogo.png");
   ...
}
```

Finally, we'll create another images folder and move the newly created file to it. We rename the file as a side effect of moving it.

```
...
IFolder newImagesFolder = myWebProject.getFolder("newimages");
newImagesFolder.create(false, true, null);
IPath renamedPath = newImagesFolder.getFullPath().append("renamedLogo.png");
newLogo.move(renamedPath, false, null);
IFile renamedLogo = newImagesFolder.getFile("renamedLogo.png");
```

Many of the resource API methods include a **force** boolean flag which specifies whether resources that are out of synch with the corresponding files in the local file system will be updated anyway. See **IResource** for more information. You can also use **IResource.isSynchronized** to determine whether a particular resource is in synch with the file system.

Mapping resources to disk locations

In the sample resource tree, we've assumed that all of the project content directories are in the **workspace** directory underneath the platform root directory (**C:\MySDK\workspace**). This is the default configuration for projects. However, a project's content directory can be remapped to any arbitrary directory in the file system, perhaps on a different disk drive.

The ability to map the location of a project independent of other projects allows the user to store the contents of a project in a place that makes sense for the project and the project team. A project's content directory should be considered "out in the open." This means that users can create, modify, and delete resources by using the workbench and plug–ins, or by directly using file system based tools and editors.

Resource path names are not complete file system paths. Resource paths are always based on the project's location (usually the **workspace** directory). To obtain the full file system path to a resource, you must query its location using **IResource.getLocation.** Howevever, you cannot use **IProjectDescription.setLocation** to change its location, because that method is just a simple setter for a data structure.

Conversely, if you want to get the corresponding resource object given a file system path, you can use **IWorkspaceRoot.getFileForLocation** or **IWorkspaceRoot.getContainerForLocation**.

Resource API and the file system

When we use the resources API to modify our workspace's resource tree, the files are changed in the file system in addition to updating our resource objects. What about changes to resource files that happen outside of the platform's API?

External changes to resources will not be reflected in the workspace and resource objects until detected by the resources plug−in. The resources plug−in also uses a mechanism appropriate for each particular native operating system for discovering external changes made in the file system. In addition, clients can use resource API to reconcile workspace and resource objects with the local file system quietly and without user intervention. The user can also explicitly force a refresh in the resource navigator view of the workbench.

Many of the methods in the resource APIs include a force parameter which specifies how resources that are out of sync with the file system should be handled. The API Reference for each method provides specific information about this parameter. Additional methods in the API allow programmatic control of file system refresh, such as **IResource.refreshLocal(int depth, IProgressMonitor monitor)**. See **IResource** for information on correct usage and costs.

Plug−ins that wish to supply their own mechanism for periodically refreshing the workspace based on the state of the external file system may do so using the **org.eclipse.core.resources.refreshProviders** extension point. See Refresh providers for more information.

Auto−refresh providers

Identifier:

org.eclipse.core.resources.refreshProviders

Since:

3.0

Description:

The workspace supports a mode where changes that occur in the file system are automatically detected and reconciled with the workspace in memory. By default, this is accomplished by creating a monitor that polls the file system and periodically searching for changes. The monitor factories extension point allows clients to create more efficient monitors, typically by hooking into some native file system facility for change callbacks.

Configuration Markup:

<!ELEMENT extension (refreshProvider)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

<!ELEMENT refreshProvider EMPTY>

<!ATTLIST refreshProvider

name CDATA #REQUIRED

class CDATA #REQUIRED>

- **name** − A human−readable name for the monitor factory
- **class** − The fully qualified name of a class implementing `org.eclipse.core.resources.refresh.RefreshProvider`.

Examples:

Following is an example of an adapter declaration. This example declares that this plug−in will provide an adapter factory that will adapt objects of type IFile to objects of type MyFile.

```
<extension id=

"coolProvider"

point=

"org.eclipse.core.resources.refreshProviders"

>

<refreshProvider name=

"Cool Refresh Provider"

class=

"com.xyz.CoolRefreshProvider"

>

</refreshProvider>

</extension>
```

API Information:

Refresh provider implementations must subclass the abstract type `RefreshProvider` in the `org.eclipse.core.resources.refresh` package. Refresh requests and failures should be forward to the provide `IRefreshResult`. Clients must also provide an implementation of `IRefreshMonitor` through which the workspace can request that refresh monitors be uninstalled.

Supplied Implementation:

The `org.eclipse.core.resources.win32` fragment provides a native refresh monitor that uses win32 file system notification callbacks. The workspace also supplies a default naive polling−based monitor that can be used for file systems that do not have native refresh callbacks available.

Refresh providers

The **org.eclipse.core.resources.refreshProviders** extension point allows plug−ins to register and implement their own mechanisms for monitoring the external file system and refreshing the workspace appropriately. This extension point is intended for plug−ins that implement specialized, often native, schemes for monitoring file system changes.

Since:

The plug−in fragment **org.eclipse.core.resources.win32** implements a native refresh monitor based on file system callbacks. A more naive refresh monitor based on polling is defined for other platforms.

The following snippet shows the definition for this extension in the **org.eclipse.core.resources.win32** fragment.

```
<extension
      id="win32"
      point="org.eclipse.core.resources.refreshProviders">
   <refreshProvider
         name="%win32MonitorFactoryName"
         class="org.eclipse.core.internal.resources.refresh.win32.Win32RefreshProvider">
   </refreshProvider>
</extension>
```

The **class** attribute must be a class that extends **RefreshProvider**. This class is responsible for installing a monitor on a specific resource and its resource subtree if it is a project or folder. The monitor must implement **IRefreshMonitor**.

Project−scoped preferences

In Runtime preferences, we looked at the infrastructure for defining and storing preferences with different scopes. We also saw that the **org.eclipse.core.runtime.preferences** extension can be used to define additional scopes for preferences. The platform resources plug−in defines its own preference scope, called "Project," in order to define project−scoped preferences. Project−scoped preferences are stored in a file located inside the project. This makes it easy to store a set of preferences and exchange them with other users using resource−oriented mechanisms such as a version control system.

Specifying the scope

The definition for new scopes is pretty simple. The plug−in defines the name of the scope, as well as the class that implements it. The resources plug−in defines the project scope as follows:

```
<extension id="preferences" point="org.eclipse.core.runtime.preferences" name="preference
        <scope name="project" class="org.eclipse.core.internal.resources.ProjectPreferenc
</extension>
```

The specified class must implement the **IScope** interface, which means it must be capable of creating preference nodes for the scope.

Project−scoped preference nodes

Since the project scope for preferences is not one of the standard runtime scopes, the node representing a project−level preference must be obtained specifically. From the root preference node, you must navigate to the project−scoped preference. This can be achieved using the **ProjectScope**:

```
IScopeContext projectScope = new ProjectScope(MyProject);
```

Once the project scope for a particular is project is found, the preference values can be obtained using the same mechanisms seen earlier. Preferences are named using the string name of the preference. The names are qualified with another string (often a plug−in id) that qualifies the namespace of the preference. The following snippet gets a preference node from the project scope. You'll notice that once the correct scope is obtained,

working with the nodes is no different than with nodes from other scopes.

```
...
Preferences projectNode = projectScope.node("com.example.myplugin");
if (projectNode != null) {
        value = node.getBoolean("MyPreference", "true");
        //do something with the value.
}
...
```

To save the value to a file in the project, the node is flushed. The resources plug−in handles the logistics for managing the project−level preferences file.

```
projectNode.flush();
```

Runtime preferences

The **org.eclipse.core.runtime.preferences** package provides infrastructure for storing a plug−in's preferences. Preferences typically map to settings controlled by the user on the **Preferences** page, although this is not required by the underlying infrastructure. Plug−in preferences are key/value pairs, where the key describes the name of the preference, and the value is one of several different types (boolean, double, float, int, long, or string). Preferences can be stored and retrieved by the platform from the file system. The exact location of the saved preferences depends upon the **scope** of the preference.

Preference scopes

The **scope** of a preference is closely related to where the preference is stored. Plug−in developers can decide which of the standard scopes apply for their preferences, or can define new scopes that make sense for their plug−in. Let's look first at the scopes defined by the platform runtime:

- **Instance** scoped preferences are stored per workspace, or per running instance of the platform.
- **Configuration** scoped preferences are stored per installation of the platform. They are shared between workspaces. For example, if a user has a single installation of the platform, but runs several different workspaces, preferences scoped at the configuration level will be shared between the workspaces.
- **Default** scoped preferences represent the default values for preferences. These are not changed or stored by the platform. However, the values originate from files stored with the plug−in's product or primary feature. (See What is a product? for an explanation of products and primary features, and their related files.)

You can think of the overall preference store as a hierarchy of nodes, where each main branch of the hierarchy represents a particular scope. The children of any particular node depend upon how that scope is defined. For the instance and configuration scopes, the child nodes are the preferences for a particular plug−in as specified by a preference **qualifier**, usually the plug−in's **id**.

If all of this sounds confusing, don't worry. If you don't care about scopes and nodes, you need not worry about any particular scope or about which node of the tree actually contains your preference value. The preferences API will automatically traverse the nodes in the proper order (instance, configuration, default) when you query a preference value and use the supplied qualifier and preference name to find the node that actually contains the value.

Preferences are accessed using **IPreferencesService** protocol. The platform's default preference service can be accessed using the **Platform** class.

```
...
IPreferencesService service = Platform.getPreferencesService();
...
```

Once the preference service is obtained, preference values can be queried by name using any of **get...** methods provided in **IPreferencesService**. For example, the following snippet queries the value of the "MyPreference" preference in the plug−in "com.example.myplugin".

```
...
IPreferencesService service = Platform.getPreferencesService();
boolean value = service.getBoolean("com.example.myplugin", "MyPreference", null);
//do something with the value.
...
```

The last parameter in the query method is an array of scope contexts to use when searching for the preference node. If the array is **null**, then the platform assumes that the default scope search order should be used and guesses the appropriate preference node. If an array of scopes contexts is passed, then this determines the scope lookup order that should be used to find the preference node. The default scope lookup order is always consulted if no node can be found using the specified scopes.

Using scopes and nodes

If a plug−in needs finer control over the scope search order, classes that represent the scopes can be used to access the actual node that represents the preference at a particular scope. In this way, an array of nodes can be created that specifies the particular search order required. The following snippet queries the preferences service for the same preference used above, but searches the configuration scope for the plug−in, followed by the instance scope for the plug−in. When nodes are specified for the search order, the default scoping is not considered. That is, the platform will only search the exact nodes that have been provided.

```
...
IPreferencesService service = Platform.getPreferencesService();
Preferences configurationNode = new ConfigurationScope().getNode("com.example.myplugin");
Preferences instanceNode = new InstanceScope().getNode("com.example.myplugin");
Preferences[] nodes = new Preferences[] {configurationNode, instanceNode};
stringValue = service.get("MyPreference", "true", nodes);
//do something with the value.
...
```

A plug−in may also implement its own traversal through the preference tree nodes. The root node of the preference tree can be obtained from the preferences service. The scope classes can be used to further traverse the tree. The following snippet traverses to a specific node and retrieves the preference value from the node itself.

```
...
IPreferencesService service = Platform.getPreferencesService();
Preferences root = service.getRootNode();
Preferences myInstanceNode = root.node(InstanceScope.SCOPE).node("com.example.myplugin");
if (myInstanceNode != null) {
        value = node.getBoolean("MyPreference", "true");
        //do something with the value.
}
...
```

Extending the scopes

Plug−ins may define their own specialized scopes using the **org.eclipse.core.runtime.preferences** extension. In this extension, the plug−in defines the name of the new scope, as well a class that can create preference nodes for the new scope. Optionally, it can specify the name of a class that initializes the default preference values at that scope. When a plug−in defines a new scope, it is up to that plug−in to implement the traversal order for any new scope relative to the platform traversal order. We'll look at this capability in more detail using the specific example of Project−scoped preferences.

Products and features

An Eclipse based **product** is a stand−alone program built with the Eclipse platform. A product may optionally be packaged and delivered as one or more **features**, which are simply groupings of plug−ins that are managed as a single entity by the Eclipse update mechanisms.

Products include all the code and plug−ins needed to run them. This includes a Java runtime environment (JRE) and the Eclipse platform code. The plug−in code, JRE, and Eclipse platform are typically installed with a product−specific installation program. Product providers are free to use any installation tool or program that is appropriate for their needs.

Once installed, the user launches the product and is presented with an Eclipse workbench configured specifically for the purpose supported by the product, such as web development, C++ program development, or any other purpose. The platform makes it easy to configure labels, about dialogs, graphics, and splash screens, so that the user has no concept of a platform workbench, but instead views the workbench as the main window of the product itself.

The top level directory structure of such a product looks something like this for a hypothetical product called "AcmeWeb" that has been installed on a Windows platform:

```
acmeweb/
        acmeweb.exe    (product executable)
        eclipse/       (directory for installed Eclipse files)
                .eclipseproduct    (marker file)
                eclipse.exe
                startup.jar
                configuration/
                        config.ini
                jre/
             (installed features if any)
                        com.example.acme.acmefeature_1.0.0/
                                feature.xml
                        ...
                plugins/
                        com.example.acme.acmefeature_1.0.0/
                                plugin.xml
                                about.ini
                                about.html
                                about.mappings
                                about.properties
                                acme.png
                                splash.jpg
                        com.example.acme.acmewebsupport_1.0.0/
                        ...
                links/
                        ...
```

There are actually two ways of defining a product in Eclipse. The preferred mechanism is to use the products extension point (new to Eclipse 3.0). This extension point allows you to define your product and customize branding such as splash screens, window icons, and the like. The older mechanism (used in Eclipse 2.1) uses features, and in particular assumes the existence of a primary feature. Under the covers, Eclipse 3.0 uses the products extension point mechanism, but provides compatibility functions that integrate the legacy definitions into the new model if necessary.

We'll look at both mechanisms and how they are used to customize a product. Even when the products extension point mechanism is used, features can still be used as a way to group functionality that is delivered by the update manager. In the next few topics, we'll assume that feature groupings are present in our plug−in, in addition to product definitions.

Products extension point

The preferred mechanism for defining a product based on the Eclipse platform is to contribute to the **org.eclipse.core.runtime.products** extension point. To do this, a plug−in simply declares the name and id of its product, as well as the id of the application extension that should be run when the product is invoked. This is the technique used by the Eclipse platform itself in defining the Eclipse product. Here is the extension definition found in **org.eclipse.platform**:

```
<extension id="ide" point="org.eclipse.core.runtime.products">
 product<name="%productName" application="org.eclipse.ui.ide.workbench" description="%productBlur
                <property name="windowImages" value="eclipse.png,eclipse32.png"/>
                <property name="aboutImage" value="eclipse_lg.png"/>
                <property name="aboutText" value="%productBlurb"/>
                <property name="appName" value="Eclipse"/>
                <property name="preferenceCustomization" value="plugin_customization.ini"/>
        </product>
</extension>
```

A product extension is defined whose **application** id is "org.eclipse.ui.ide.workbench". This is the application id defined by the plug−in **org.eclipse.ui.ide** in its contribution to the **org.eclipse.core.runtime.applications** extension point.

```
<extension
id="workbench"
        point="org.eclipse.core.runtime.applications">
        <application>
                <run
                        class="org.eclipse.ui.internal.ide.IDEApplication">
                </run>
        </application>
</extension>
```

This extension is defined with the same id that is referenced in the **application** property of the product extension. (The fully qualified name, with plug−in prefix, is used when referring to the application id from the other plug−in.) Using this mechanism, a separate plug−in can define all of the product−specific branding, and then refer to an existing plug−in's application as the application that is actually run when the product is started.

In addition to the application, the **org.eclipse.core.runtime.products** extension describes product customization properties that are used to configure the product's branding information. This information is described as named properties. Let's look again at that portion of the markup for the platform plug−in.

```
<property name="windowImages" value="eclipse.png,eclipse32.png"/>
<property name="aboutImage" value="eclipse_lg.png"/>
<property name="aboutText" value="%productBlurb"/>
<property name="appName" value="Eclipse"/>
<property name="preferenceCustomization" value="plugin_customization.ini"/>
```

The possible property names that are honored by the platform for product customization are defined in **IProductConstants**. See the javadoc for a complete description of these properties and their values. We'll look at these further in Customizing a product.

Products

Identifier:

org.eclipse.core.runtime.products

Since:

3.0

Description:

Products are the Eclipse unit of branding. Product extensions are supplied by plug−ins wishing to define one or more products. There must be one product per extension as the extension id is used in processing and identifying the product.

There are two possible forms of product extension, static and dynamic. Static product extensions directly contain all relevant information about the product. Dynamic product extensions identify a class (an `IProductProvider`) which is capable of defining one or more products when queried.

Configuration Markup:

<!ELEMENT extension ((product | provider))>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

<!ELEMENT product (property*)>

<!ATTLIST product

application CDATA #REQUIRED

name CDATA #REQUIRED

description CDATA #IMPLIED>

- **application** − the default application to run when running this product
- **name** − the human−readable name of this product
- **description** − the human−readable description of this product

<!ELEMENT property EMPTY>

<!ATTLIST property

name CDATA #REQUIRED

value CDATA #REQUIRED>

- **name** – the key under which this property is stored
- **value** – the value of this property

<!ELEMENT provider (run)>

details of a product provider

<!ELEMENT run EMPTY>

<!ATTLIST run

class CDATA #REQUIRED>

- **class** – the fully–qualified name of a class which implements `org.eclipse.core.runtime.IProductProvider`.

Examples:

Following is an example of static product declaration:

<extension id=

"coolProduct"

point=

"org.eclipse.core.runtime.products"

>

<product name=

"%coolName"

application=

"coolApplication"

description=

"%coolDescription"

>

<property name=

"windowImage"

value=

"window.gif"

/>

<property name=

"aboutImage"

value=

"image.gif"

/>

<property name=

"aboutText"

value=

"%aboutText"

/>

<property name=

"appName"

value=

"CoolApp"

/>

<property name=

"welcomePage"

value=

"nl/welcome.xml"

/>

<property name=

"preferenceCustomization"

value=

"plugin_customization.ini"

/>

</product>

</extension>

The following is an example of a dynamic product (product provider) declaration: Following is an example of an application declaration:

<extension id=

"coolProvider"

point=

"org.eclipse.core.runtime.products"

>

<provider>

<run class=

"com.example.productProvider"

/>

</provider>

</extension>

API Information:

Static product extensions provided here are represented at runtime by instances of `IProduct`. Dynamic product extensions must identify an implementor of `IProductProvider`. See

Since:

`org.eclipse.ui.branding.IProductConstants` for details of the branding related product properties defined by the Eclipse UI.

Supplied Implementation:

No implementations of `IProductProvider` are supplied.

Applications

Identifier:

org.eclipse.core.runtime.applications

Description:

Platform runtime supports plug−ins which would like to declare main entry points. That is, programs which would like to run using the platform runtime but yet control all aspects of execution can declare themselves as an application. Declared applications can be run directly from the main platform launcher by specifying the *−application* argument where the parameter is the id of an extension supplied to the applications extension point described here. This application is instantiated and run by the platform. Platform clients can also use the platform to lookup and run multiple applications.

Configuration Markup:

<!ELEMENT extension (application)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #REQUIRED

name CDATA #IMPLIED>

<!ELEMENT application (run?)>

<!ELEMENT run (parameter*)>

<!ATTLIST run

class CDATA #REQUIRED>

- **class** − the fully−qualified name of a class which implements `org.eclipse.core.runtime.IPlatformRunnable`.

<!ELEMENT parameter EMPTY>

<!ATTLIST parameter

name CDATA #REQUIRED

value CDATA #REQUIRED>

- **name** – the name of this parameter made available to instances of the specified application class
- **value** – the value of this parameter made available to instances of the specified application class

Examples:

Following is an example of an application declaration:

<extension id=

"coolApplication"

point=

"org.eclipse.core.runtime.applications"

>

<application>

<run class=

"com.xyz.applications.Cool"

>

<parameter name=

"optimize"

value=

"true"

/>

</run>

</application>

</extension>

API Information:

The value of the class attribute must represent an implementor of `org.eclipse.core.runtime.IPlatformRunnable`.

Supplied Implementation:

The platform supplies a number of applications including the platform workbench itself.

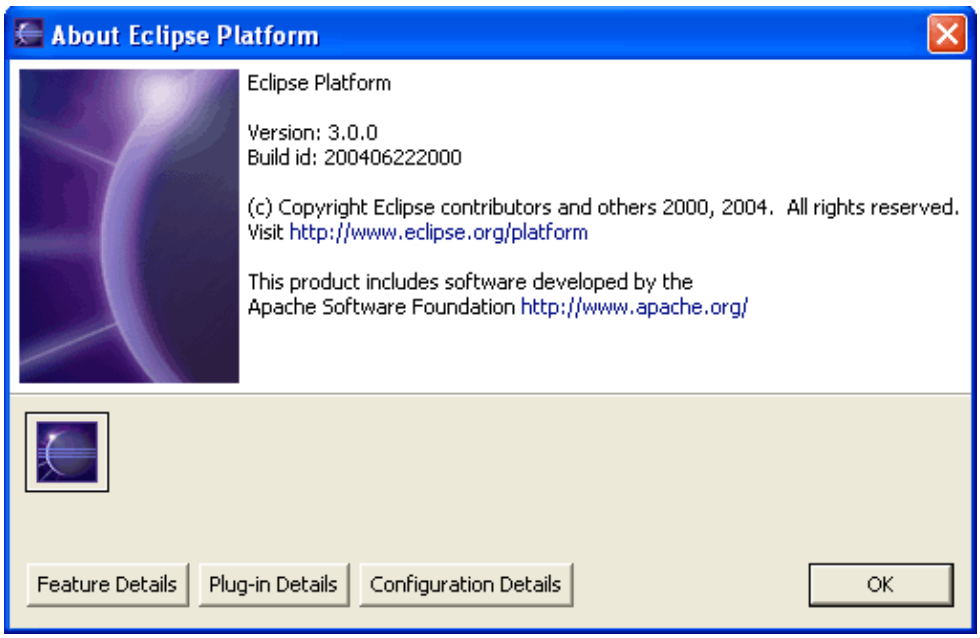
Customizing a product

There are many customizable aspects of a product, such as its splash screen, about dialog text, window icons, etc. Most of these customizations are defined in the contribution to the **org.eclipse.core.runtime.products** extension point.

Let's look at how some of the more common customizable elements are defined.

About dialogs

The platform "about" dialog is shown whenever the user selects **Help > About** in the workbench menu.



The upper part of the about dialog shows product level information while the lower part details the features (if any) and plug-ins installed. The feature details (branding if you will) are supplied using **about.ini**, **about.properties** and **about.html** files in the plug-in associated with the feature. This information is displayed when the user requests feature details and selects a particular feature.

The product branding (top half of the dialog) is specify by extension properties that describe the text and images that are shown in this dialog. For example, the following extract from the Eclipse Platform product declaration.

Description:

```
<property
        name="aboutText"
        value="%aboutText"/>
<property
        name="aboutImage"
        value="icons/eclipse_lg.png"/>
</product>
```

- **aboutText** specifies the text to show in the about dialog
- **aboutImage** specifies an image that should be used in the about dialog. Large images will be shown by themselves, and smaller images will be shown along with the about text.

See **IProductConstants** for more information about these properties.

Window images

A 16x16 pixel color image can be used to brand windows created by the product. It will appear in the upper left hand corner of product windows. It is specified in the **windowImage** attribute in the products extension definition. Alternatively, the **windowImages** attribute can be used to describe an array of images of different sizes that should be used in the upper left hand corner.

```
<property
        name="windowImages"
        value="icons/eclipse.png,icons/eclipse32.png"/>
```

The **windowImages** property supercedes the **windowImage** property if both are specified.

Welcome page

Products using the Eclipse 2.1 welcome page mechanism can specify the name and location of their welcome page file in the **welcomePage** property.

```
<property
        name="welcomePage"
        value="$nl$/welcome.xml"/>
```

Use of this property is discouraged in Eclipse 3.0 in favor of the **org.eclipse.ui.intro** extension point. See Intro support for more detail about the new welcome/intro mechanisms.

Preferences defaults

The **preferenceCustomization** property can be used to specify a file containing default preference values for the product.

```
<property
        name="preferenceCustomization"
        value="plugin_customization.ini"/>
```

This file is a **java.io.Properties** format file. Typically this file is used to set the values for preferences that are published as part of a plug−in's public API. That is, you are taking a risk if you refer to preferences that are used by plug−ins but not defined formally in the API.

Splash screens

The product splash screen is specified in the `config.ini` which is located underneath the product's `configuration` directory. The **osgi.splashPath** property in this file describes places to search for a file called **splash.bmp**. The ***osgi.splashLocation*** property identifes a complete and exact path to the splash screen file to use. Specifying a splash path allows for locale specific splash screens to be used as the given search path can indicate several plug−ins or fragments to search as well as `$nl$` style paths. See the platform SDK's `config.ini` file for a complete description of properties that can be configured in this file. The image should be supplied in 24−bit color BMP format (RGB format) and should be approximately 500x330 pixels in size.

Intro Part

Identifier:

org.eclipse.ui.intro

Since:

3.0

Description:

This extension point is used to register implementations of special workbench parts, called intro parts, that are responsible for introducing a product to new users. An intro part is typically shown the first time a product is started up. Rules for associating an intro part implementation with particular products are also contributed via this extension point.

The life cycle is as follows:

- The intro area is created on workbench start up. As with editor and view areas, this area is managed by an intro site (implementing `org.eclipse.ui.intro.IIntroSite`).
- The id of the current product (Platform.getProduct()) is used to choose the relevant intro part implementation.
- The intro part class (implementing `org.eclipse.ui.intro.IIntroPart`) is created and initialized with the intro site.
- While the intro part is showing to the user, it can transition back and forth between full and standby mode (either programmatically or explicitly by the user).
- Eventually the intro part is closed (either programmatically or explicitly by the user). The current perspective takes over the entire workbench window area.

Configuration Markup:

<!ELEMENT extension (intro* , introProductBinding*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT intro EMPTY>

<!ATTLIST intro

id CDATA #REQUIRED

icon CDATA #IMPLIED

class CDATA #REQUIRED>

Specifies an introduction. An introduction is a product−specific presentation shown to first−time users on product start up.

- **id –** a unique identifier for this introduction
- **icon –** a plug−in−relative file name of the icon that will be associated with this introduction
- **class –** a fully qualified name of the class implementing the `org.eclipse.ui.intro.IIntroPart` interface. A common practice is to subclass `org.eclipse.ui.part.intro.IntroPart` in order to inherit the default functionality. This class implements the introduction.

<!ELEMENT introProductBinding EMPTY>

<!ATTLIST introProductBinding

productId CDATA #REQUIRED

introId CDATA #REQUIRED>

Specifies a binding between a product and an introduction. These bindings determine which introduction is appropriate for the current product (as defined by `org.eclipse.core.runtime.Platform.getProduct()`).

- **productId –** unique id of a product
- **introId –** unique id of an introduction

Examples:

The following is an example of an intro part extension that contributes an particular introduction and associates it with a particular product:

<extension point=

"org.eclipse.ui.intro"

>

```
<intro id=

"com.example.xyz.intro.custom"

class=

"com.example.xyz.intro.IntroPart"

/>

<introProductBinding productId=

"com.example.xyz.Product"

introId=

"com.example.xyz.intro.custom"

/>

</extension>
```

API Information:

The value of the `class` attribute must be the fully qualified name of a class that implements the`org.eclipse.ui.intro.IIntroPart` interface by subclassing `org.eclipse.ui.part.intro.IntroPart`.

Supplied Implementation:

There are no default implementations of the initial user experience. Each Eclipse−based product is responsible for providing one that is closely matched to its branding and function.

Intro support

Intro support is a set of extension points and workbench parts that allow plug−ins to define specialized pages that introduce a platform product to new users. These extension points can be used to create the "Initial User Experience" of the product in the form of welcome pages. Welcome pages are intended to guide users into discovering a product's functionality. They are typically shown the first time a product is started. They can be anything from a single page displaying a Macromedia Flash demo, to multiple pages that are extensible and follow the Eclipse platform's pluggable architecture.

Intro support is typically configured at the product level, although individual plug−ins can contribute intro information to known product intro configurations.

Since:

From a workbench point of view, the root of the intro support is in the **intro part**. This part is specified in an extension definition. When the workbench initializes, it creates an intro site that reserves space for the intro page. The intro part implementation for the site is determined using product configuration information. Once an intro part is shown, it can move between two modes:

- in **full mode**, the intro part takes over the main workbench area.
- in **standby mode**, the intro part moves to the side, allowing the current perspective to remain visible.

Once an intro part is established, it must be configured with intro information. This is done using an **intro config** which is also contributed using an extension. Individual plug−ins can add to the basic product intro config using their own extensions.

We'll look at the platform SDK intro page as an example in order to better understand these concepts.

Features

A **feature** is a way of grouping and describing different functionality that makes up a product. Grouping plug−ins into features allows the product to be installed and updated using the Eclipse update server and related support. The platform itself is partitioned into three major features:

- Platform
- JDT (Java Development Tooling)
- PDE (Plug−in Developer Environment)

There are other minor features, such as examples and OS−dependent portions of the platform.

> *Note: The platform installation and update framework allows you to build your own custom implementations of the concepts discussed here. That is, you can define your own types of* ***features****, (their packaging formats, install procedures, etc.), as well as your own types of server sites for updating your features. The remainder of this discussion is focused on the platform default implementations for features and update sites.*

Features do not contain any code. They merely describe a set of plug−ins that provide the function for the feature and information about how to update it. Features are packaged in a feature archive file and described using a feature manifest file, feature.xml. The following is the first part of the manifest for the platform feature:

```
<?xml version="1.0" encoding="UTF-8"?>
<feature
      id="org.eclipse.platform"
      label="%featureName"
      version="3.0.0"
      provider-name="%providerName"
      plugin=""
      image="eclipse_update_120.jpg"
      primary="true"
      application="org.eclipse.ui.ide.workbench">

   <description>
      %description
   </description>
```

```
   <license url="%licenseURL">
      %license
   </license>

   <url>
      <update label="%updateSiteName" url="http://update.eclipse.org/updates/3.0"/>
      <discovery label="%updateSiteName" url="http://update.eclipse.org/updates/3.0"/>
   </url>

   <plugin
         id="org.apache.ant"
         download-size="0"
         install-size="0"
         version="1.6.1"/>

   <plugin
         id="org.apache.lucene"
         download-size="0"
         install-size="0"
         version="1.3.0"/>
        ...
</feature>
```

Feature Archives

The feature packaging information is placed into a separate Java .jar. Standard Java jar facilities are used for constructing feature archives. Feature archives reference separately packaged plug−in archives (see next section) and non−plug−in files.

Features are identified using a structured identifier based on the provider internet domain name. For example, organization eclipse.org produces feature org.eclipse.jdt. The character set used for feature identifiers is as specified for plug−in identifiers (see Plug−in Manifest).

The recommended convention for naming the feature archives is
`<id>_<version>.jar`

Where `<id>` is the feature identifier and `<version>` is the full version identifier contained in the respective feature.xml. Note that this is a recommended convention that minimizes chance of collisions, but is not required by the Eclipse architecture. For example, the following are valid feature archive names

```
org.eclipse.jdt_2.0.0.jar
org.eclipse.pde_2.0.jar
my_feature.jar
```

Internally, each feature archive is packaged relative to its feature directory (but not including the directory path element). The archive has the following structure

```
feature.xml
feature<_locale>.properties (see "Translated Feature Information")
other feature files and subdirectories (TBD)
META-INF/
    Java jar manifest and security files
```

Note that feature archives do not contain their constituent plug−ins and fragments.

Eclipse platform plug–in manifest

Version 3.0 – Last revised June 24, 2004

The manifest markup definitions below make use of various naming tokens and identifiers. To eliminate ambiguity, here are some production rules for these [are referenced in text below]. In general, all identifiers are case–sensitive.

```
SimpleToken := sequence of characters from ('a-z','A-Z','0-9','_')
ComposedToken := SimpleToken | (SimpleToken '.' ComposedToken)
JavaClassName := ComposedToken
PlugInId := ComposedToken
PlugInPrereq := PlugInId | 'export' PlugInId
ExtensionId := SimpleToken
ExtensionPointId := SimpleToken
ExtensionPointReference := ExtensionPointID | (PlugInId '.' ExtensionPointId)
```

The remainder of this section describes the plugin.xml file structure as a series of DTD fragments. File plugin.dtd presents the DTD definition in its entirety.

```
<?xml encoding="US-ASCII"?>
<!ELEMENT plugin (requires?, runtime?, extension-point*, extension*)>
<!ATTLIST plugin
  name                CDATA #REQUIRED
  id                  CDATA #REQUIRED
  version             CDATA #REQUIRED
  provider-name       CDATA #IMPLIED
  class               CDATA #IMPLIED
>
```

The <plugin> element defines the body of the manifest. It optionally contains definitions for the plug–in runtime, definitions of other plug–ins required by this one, declarations of any new extension points being introduced by the plug–in, as well as configuration of functional extensions (configured into extension points defined by other plug–ins, or introduced by this plug–in). <plugin> attributes are as follows:

- **name** – user displayable (translatable) name for the plug–in
- **id** – unique identifier for the plug–in.
  - To minimize potential for naming collisions, the identifier should be derived from the internet domain id of the supplying provider (reversing the domain name tokens and appending additional name tokens separated by dot [.]). For example, provider ibm.com could define plug–in identifier com.ibm.db2
  - [production rule: PlugInId]
- **version** – plug–in version number. See org.eclipse.core.runtime.PluginVersionIdentifier for details. Plug–in version format is **major.minor.service.qualifier**. Change in the major component is interpreted as an incompatible version change. Change in the minor component is interpreted as a compatible version change. Change in the service component is interpreted as *cumulative* service applied to the minor version. Change in the qualifier component is interpreted as different source code control version of the same component.
- **provider–name** – user–displayable name of the provider supplying the plug–in.
- **class** – name of the plug–in class for this plug–in. The class must be a subclass of org.eclipse.core.runtime.Plugin.

The XML DTD construction rule *element** means zero or more occurrences of the element; *element?* means zero or one occurrence of the element; and *element+* (used below) means one or more occurrences of the element. Based on the <plugin> definition above, this means, for example, that a plug−in containing only a run−time definition and no extension point declarations or extension configurations is valid (for example, common libraries that other plug−ins depend on). Similarly, a plug−in containing only extension configurations and no runtime or extension points of its own is also valid (for example, configuring classes delivered in other plug−ins into extension points declared in other plug−ins).

The <requires> section of the manifest declares any dependencies on other plug−ins.

```
<!ELEMENT requires (import+)>
<!ELEMENT import EMPTY>
<!ATTLIST import
 plugin               CDATA #REQUIRED
 version              CDATA #IMPLIED
 match                (perfect | equivalent | compatible | greaterOrEqual) "compatible"
 export               (true | false) "false"
 optional             (true | false) "false"
>
```

Each dependency is specified using an <import> element. It contains the following attributes:

- **plugin** − identifier of the required plug−in
- **version** − optional version specification
- **match** − version matching rule. Ignored if version attribute is not specified. Determines whether the dependency is satisfied only with a plug−in that has this exact specified version, with a plug−in that has a service or qualifier more recent than this one, with any compatible version (including a more recent minor version of the plug−in) or with any more recent version of this plug−in
- **export** − specifies whether the dependent plug−in classes are made visible (are (re)exported) to users of this plug−in. By default, dependent classes are not exported (are not made visible)
- **optional** − specifies whether or not this dependency will be strictly enforced. If set to <true> and this dependency cannot be satisfied, the dependency will be ignored

The <runtime> section of the manifest contains a definition of one or more libraries that make up the plug−in runtime. The referenced libraries are used by the platform execution mechanisms (the plug−in class loader) to load and execute the correct code required by the plug−in.

```
<!ELEMENT runtime (library+)>
<!ELEMENT library (export*, packages?)>
<!ATTLIST library
  name               CDATA #REQUIRED
  type               (code | resource) "code"
>
<!ELEMENT export EMPTY>
<!ATTLIST export
  name               CDATA #REQUIRED
>
<!ELEMENT packages EMPTY>
<!ATTLIST packages
  prefixes           CDATA #REQUIRED
>
```

The <runtime> element has no attributes.

The <library> elements collectively define the plug−in runtime. At least one <library> must be specified. Each <library> element has the following attributes:

- **name** − string reference to a library file or directory containing classes (relative to the plug−in install directory). Directory references must contain trailing file separator.
- **type** − specifies whether this library contains executable code (<code>) or just resources. If the library is of type <code> accessing anything in this library will cause activation of the plug−in. Accessing a <resource> will not cause plug−in activation (a potential for significant performance improvement). It should be noted that specifying a library of type <code> allows it to contain both code and resources. But specifying a library of type <resource> assumes it will only be used for resources.

Each <library> element can specify which portion of the library should be exported. The export rules are specified as a set of export masks. By default (no export rules specified), the library is considered to be private. Each export mask is specified using the **name** attribute, which can have the following values:

- ***** − indicates all contents of library are exported (public)
- **package.name.*** − indicates all classes in the specified package are exported. The matching rules are the same as in the Java import statement.
- **package.name.ClassName** − fully qualified java class name

Eclipse 2.1 plug−ins only: Each library can also specify the package prefixes. These are used to enhance the classloading performance for the plug−in and/or fragment. If the <packages> element is not specified, then by default the classloading enhancements are not used. The <packages> element has the following attribute:

- **prefixes** − a comma−separated list of package prefixes for the runtime library

The platform's architecture is based on the notion of configurable extension points. The platform itself predefines a set of extension points that cover the task of extending the platform and desktop (for example, adding menu actions, contributing embedded editor). In addition to the predefined extension points, each supplied plug−in can declare additional extension points. By declaring an extension point the plug−in is essentially advertising the ability to configure the plug−in function with externally supplied extensions. For example, the Page Builder plug−in may declare an extension point for adding new Design Time Controls (DTCs) into its builder palette. This means that the Page Builder has defined an architecture for what it means to be a DTC and has implemented the code that looks for DTC extensions that have been configured into the extension points.

```
<!ELEMENT extension-point EMPTY>
<!ATTLIST extension-point
  name               CDATA #REQUIRED
  id                 CDATA #REQUIRED
  schema             CDATA #IMPLIED
>
```

The <extension−point> element has the following attributes:

- **name** − user−displayable (translatable) name for the extension point
- **id** − simple id token, unique within this plug−in. The token cannot contain dot (.) or whitespace.
  - ♦ [production rule: ExtensionPointId]
- **schema** − schema specification for this extension point. The exact details are being defined as part of the Plug−In Development Environment (PDE). The schema is currently not used at runtime. The reference is a file name relative to the plug−in installation location.

Actual extensions are configured into extension points (predefined, or newly declared in this plug−in) in the <extension> section. The configuration information is specified as well−formed XML contained between the <extension> and </extension> tags. The platform does not specify the actual form of the configuration markup (other than requiring it to be well−formed XML). The markup is defined by the supplier of the plug−in that declared the extension point. The platform does not actually interpret the configuration markup. It simply passes the configuration information to the plug−in as part of the extension point processing (at the time the extension point logic queries all of its configured extensions).

```
<!ELEMENT extension ANY>
<!ATTLIST extension
  point              CDATA #REQUIRED
  id                 CDATA #IMPLIED
  name               CDATA #IMPLIED
>
```

The <extension> element has the following attributes:

- **point** − reference to an extension point being configured. The extension point can be one defined in this plug−in or another plug−in
    - ♦ [production rule: ExtensionPointReference]
- **id** − optional identifier for this extension point configuration instance. This is used by extension points that need to uniquely identify (rather than just enumerate) the specific configured extensions. The identifier is specified as a simple token unique within the definition of the declaring plug−in. When used globally, the extension identifier is qualified by the plug−in identifier
    - ♦ [production rule: ExtensionId]
- **name** − user−displayable (translatable) name for the extension

Important: The content of the <extension> element is declared using the `ANY` rule. This means that any well−formed XML can be specified within the extension configuration section (between <extension> and </extension> tags).

Fragments are used to increase the scope of a plug−in. An example would be to incorporate data such as messages or labels in another language.

```
<?xml encoding="US-ASCII"?>
<!ELEMENT fragment (requires?, runtime?, extension-point*, extension*)>
<!ATTLIST fragment
  name                CDATA #REQUIRED
  id                  CDATA #REQUIRED
  version             CDATA #REQUIRED
  provider-name       CDATA #IMPLIED
  plugin-id           CDATA #REQUIRED
  plugin-version      CDATA #REQUIRED
  match               (perfect | equivalent | compatible | greaterOrEqual) "compatible"
>
```

Each fragment must be associated with a specific plug−in. The associated plug−in is identified with <plugin−id>, <plugin−version> and optionally, <match>. Note that if this specification matches more than one plug−in, the matching plug−in with the highest version number will be used.

The <requires>, <runtime>, <extension−point>, and <extension> components of a fragment will be logically added to the matching plug−in.

<fragment> attributes are as follows:

- **name** – user displayable (translatable) name for the fragment
- **id** – unique identifier for the fragment.
  - To minimize potential for naming collisions, the identifier should be derived from the id of the associated plug–in in addition to something which identifies the scope of this fragment. For example, org.eclipse.core.runtime.nl1 could define a natural language fragment for the org.eclipse.core.runtime plug–in
  - [production rule: PlugInId]
- **version** – fragment version number. See org.eclipse.core.runtime.PluginVersionIdentifier for details. Fragment version format is the same as plug–in version format.
- **provider–name** – user–displayable name of the provider supplying the fragment.
- **plugin–id** – matches the id of the associated plug–in
- **plugin–version** – matches the version of the associated plug–in
- **match** – the matching rule used to find an associated plug–in using <plugin–id> and <plugin–version>. See the definition of <match> in the <requires> clause for complete details.

```
<?xml encoding="US-ASCII"?>

<!ELEMENT plugin (requires?, runtime?, extension-point*, extension*)>
<!ATTLIST plugin
 name                CDATA #REQUIRED
 id                  CDATA #REQUIRED
 version             CDATA #REQUIRED
 provider-name       CDATA #IMPLIED
 class               CDATA #IMPLIED
>

<!ELEMENT fragment (requires?, runtime?, extension-point*, extension*)>
<!ATTLIST fragment
 name                CDATA #REQUIRED
 id                  CDATA #REQUIRED
 version             CDATA #REQUIRED
 provider-name       CDATA #IMPLIED
 plugin-id           CDATA #REQUIRED
 plugin-version      CDATA #REQUIRED
 match               (perfect | equivalent | compatible | greaterOrEqual) "compatible"
>

<!ELEMENT requires (import+)>
<!ELEMENT import EMPTY>
<!ATTLIST import
 plugin              CDATA #REQUIRED
 version             CDATA #IMPLIED
 match               (perfect | equivalent | compatible | greaterOrEqual) "compatible"
 export              (true | false) "false"
 optional            (true | false) "false"
>

<!ELEMENT runtime library+)>

<!ELEMENT library (export*, packages?)>
<!ATTLIST library
 name                CDATA #REQUIRED
 type                (code | resource) "code"
>

<!ELEMENT export EMPTY>
<!ATTLIST export
 name                CDATA #REQUIRED
>
```

```
<!ELEMENT packages EMPTY>
<!ATTLIST packages
 prefixes            CDATA #REQUIRED
>

<!ELEMENT extension-point EMPTY>
<!ATTLIST extension-point
 name                CDATA #REQUIRED
 id                  CDATA #REQUIRED
 schema              CDATA #IMPLIED
>

<!ELEMENT extension ANY>
<!ATTLIST extension
 point               CDATA #REQUIRED
 name                CDATA #IMPLIED
 id                  CDATA #IMPLIED
>
```

Eclipse platform feature manifest

Version 3.0 – Last revised June 22, 2004

The feature manifest format is defined by the following dtd:

```
<?xml encoding="ISO-8859-1"?>

<!ELEMENT feature (install-handler? | description? | copyright? |
license? | url? | includes* | requires? | plugin* | data*)>
<!ATTLIST feature
    id            CDATA #REQUIRED
    version       CDATA #REQUIRED
    label         CDATA #IMPLIED
    provider-name CDATA #IMPLIED
    image         CDATA #IMPLIED
    os            CDATA #IMPLIED
    arch          CDATA #IMPLIED
    ws            CDATA #IMPLIED
    nl            CDATA #IMPLIED
    colocation-affinity
                  CDATA #IMPLIED
    primary       (true | false) "false"
    exclusive     (true | false) "false"
    plugin        CDATA #IMPLIED
    application   CDATA #IMPLIED
>

<!ELEMENT install-handler EMPTY>
<!ATTLIST install-handler
    library       CDATA #IMPLIED
    handler       CDATA #IMPLIED
>

<!ELEMENT description (#PCDATA)>
<!ATTLIST description
    url           CDATA #IMPLIED
>

<!ELEMENT copyright (#PCDATA)>
<!ATTLIST copyright
    url           CDATA #IMPLIED
>

<!ELEMENT license (#PCDATA)>
<!ATTLIST license
    url           CDATA #IMPLIED
>

<!ELEMENT url (update?, discovery*)>
```

```
<!ELEMENT update EMPTY>
<!ATTLIST update
    url           CDATA #REQUIRED
    label         CDATA #IMPLIED
>

<!ELEMENT discovery EMPTY>
<!ATTLIST discovery
    type          (web | update) "update"
    url           CDATA #REQUIRED
    label         CDATA #IMPLIED
>

<!ELEMENT includes EMPTY>
<!ATTLIST includes
    id                CDATA #REQUIRED
    version           CDATA #REQUIRED
    name              CDATA #IMPLIED
    optional          (true | false) "false"
    search-location   (root | self | both) "root"
    os                CDATA #IMPLIED
    arch              CDATA #IMPLIED
    ws                CDATA #IMPLIED
    nl                CDATA #IMPLIED
>

<!ELEMENT requires (import+)>

<!ELEMENT import EMPTY>
<!ATTLIST import
    plugin        CDATA #IMPLIED
    feature       CDATA #IMPLIED
    version       CDATA #IMPLIED
    match         (perfect | equivalent | compatible | greaterOrEqual)
"compatible"
    patch         (true | false) "false"
>

<!ELEMENT plugin EMPTY>
<!ATTLIST plugin
    id            CDATA #REQUIRED
    version       CDATA #REQUIRED
    fragment      (true | false) "false"
    os            CDATA #IMPLIED
    arch          CDATA #IMPLIED
    ws            CDATA #IMPLIED
    nl            CDATA #IMPLIED
    download-size CDATA #IMPLIED
    install-size  CDATA #IMPLIED
    unpack        (true | false) "true"
>
```

```
<!ELEMENT data EMPTY>
<!ATTLIST data
    id            CDATA #REQUIRED
    os            CDATA #IMPLIED
    arch          CDATA #IMPLIED
    ws            CDATA #IMPLIED
    nl            CDATA #IMPLIED
    download-size CDATA #IMPLIED
    install-size  CDATA #IMPLIED
>
```

The element and attribute definitions are as follows:

- <feature> – defines the feature
  - id – required feature identifier (eg. com.xyz.myfeature)
  - version – required component version (eg. 1.0.3)
  - label – optional displayable label (name). Intended to be translated.
  - provider–name – optional display label identifying the organization providing this component. Intended to be translated.
  - image – optional image to use when displaying information about the feature. Specified relative to the feature.xml.
  - os – optional operating system specification. A comma–separated list of operating system designators defined by Eclipse (see Javadoc for `org.eclipse.core.runtime.Platform`). Indicates this feature should only be installed on one of the specified OS systems. If this attribute is not specified, the feature can be installed on all systems (portable implementation). This information is used as a hint by the installation and update support (user can force installation of feature regardless of this setting).
  - arch – optional machine architecture specification. A comma–separated list of architecture designators defined by Eclipse (see Javadoc for `org.eclipse.core.runtime.Platform`). Indicates this feature should only be installed on one of the specified systems. If this attribute is not specified, the feature can be installed on all systems (portable implementation). This information is used as a hint by the installation and update support (user can force installation of feature regardless of this setting).
  - ws – optional windowing system specification. A comma–separated list of window system designators defined by Eclipse (see Javadoc for `org.eclipse.core.runtime.Platform`). Indicates this feature should only be installed on one of the specified WS's. If this attribute is not specified, the feature can be installed on all systems (portable implementation). This information is used as a hint by the installation and update support (user can force installation of feature regardless of this setting).
  - nl – optional locale specification. A comma–separated list of locale designators defined by Java. Indicates this feature should only be installed on a system running with a compatible locale (using Java locale–matching rules). If this attribute is not specified, the feature can be installed on all systems (language–neutral implementation). This information is used as a hint by the installation and update support (user can force installation of feature regardless of this setting).
  - colocation–affinity – optional reference to another feature identifier used to select the default installation location for this feature. When this feature is being installed as a new feature (no other versions of it are installed), an attempt is made to install this feature in the same

installation location as the referenced feature.
    - primary – optional indication specifying whether this feature can be used as a primary feature. Default if *false* (not a primary feature).
    - application – optional identifier of the Eclipse application that is to be used during startup when the declaring feature is the primary feature. The application identifier must represent a valid application registered in the `org.eclipse.core.runtime.applications` extension point. Default is `org.eclipse.ui.ide.workbench`.
    - plugin – optional identifier that represents the id of the plug–in listed in the feature that is used to carry branding information for the feature (images, translations, splash screens in case of primary feature etc.). If not specified, the assumption will be made the attribution plug–in has the same id as the feature.
    - exclusive – optional flag that, if "true", indicates that the feature cannot be installed in a group with other features.
- <install–handler>
    - library – optional .jar library containing the install handler classes. If specified, the referenced .jar must be contained in the feature archive. It is specified as a path within the feature archive, relative to the feature.xml entry. If not specified, the feature archive itself is used to load the install handler classes. This attribute is only interpreted if *class* attribute is also specified
    - handler – optional identifier of the install handler. The value is interpreted depending on the value of the *library* attribute. If *library* is specified, the value is interpreted as a fully qualified name of a class contained in the specified *library*. If *library* is not specified, the value is is interpreted as an extension identifier of an extension registered in the *org.eclipse.update.installHandlers* extension point. In either case, the resulting class must implement the *IInstallHandler* interface. The class is dynamically loaded and called at specific points during feature processing. When the handler is specified as a class, it has visibility to the API classes from the org.eclipse.update.core plug–in, and Eclipse plug–ins required by this plug–in; otherwise, when is specified as extension, it has access to all the classes as the plug–in defining the extension.
- <description> – brief component description as simple text. Intended to be translated.
    - url – optional URL for the full description as HTML. The URL can be specified as absolute of relative. If relative, it is assumed to be relative to (and packaged in) the feature archive. Note, that for NL handling the URL value should be separated to allow alternate URLs to be specified for each national language.
- <copyright> – feature copyright as simple text. Intended to be translated.
    - url – optional URL for the full description as HTML. The URL can be specified as absolute of relative. If relative, it is assumed to be relative to (and packaged in) the feature archive. Note, that for NL handling the URL value should be separated to allow alternate URLs to be specified for each national language.
- <license> – feature "click–through" license as simple text. Intended to be translated. It is displayed in a standard dialog with [Accept] [Reject] actions during the download/ installation process. Note, that click–through license must be specified for any feature that will be selected for installation or update using the Eclipse update manager. When using nested features, only the nesting parent (i.e. the feature selected for installation or update) must have click–through license text defined. The license text is required even if the optional *url* attribute is specified.
    - url – optional URL for the full description as HTML. The URL can be specified as absolute of relative. If relative, it is assumed to be relative to (and packaged in) the feature archive. Note, that for NL handling the URL value should be separated to allow alternate URLs to be specified for each national language. Note, that the "content" of this URL is **not** what is presented as the click–through license during installation processing. The click–through license is the actual value of the <*license*> element (eg. `<license>click through`

`text</license>`)

- <url> – optional URL specifying site(s) contain feature updates, or new features
  - <update> – URL to go to for updates to this feature
    - url – actual URL
    - label – displayable label (name) for the referenced site
  - <discovery> – URL to go to for new features. In general, a provider can use this element to reference its own site(s), or site(s) of partners that offer complementary features. Eclipse uses this element simply as a way to distribute new site URLs to the clients. Sites that belong to root features (at the top of the hierarchy) typically appear in "Sites to Visit" in Update Manager.
    - url – actual URL
    - label – displayable label (name) for the referenced site
    - type (new in 2.1) – by default, discovery sites are assumed to be update sites ("update"). By setting the value of this attribute to "web", it is possible to indicate to Eclipse that the URL should be treated as a regular Web hyperlink that can be directly displayed in a suitable browser.
- <includes> – optional reference to a nested feature that is considered to be part of this feature. Nested features must be located on the same update site as this feature
  - id – required nested feature identifier. If the feature is a patch (see the <requires> section below), this must be the id of another patch.
  - version – required nested feature version
  - optional – it is possible to include a feature as optional when this attribute is "true". Users are allowed to not install optional features, disable them if they are installed, and install them later. A missing optional feature is not treated as an error.
  - name – if an optional feature is missing, Eclipse cannot render its name properly. This attribute can be used as a 'place holder' to allow Eclipse to render the name of the optional feature when it is not installed.
  - search–location – an included feature can be updated by patches. By default, search location is "root" which means that URL specified in the "update" element within the "url" element of the parent will be considered. If an included feature has its own "update" element defined, it will be ignored by default. If the parent feature wants to allow the child to be updated from its own location, it can set this attribute to "both" or "self".
  - os – optional operating system specification. A comma–separated list of operating system designators defined by Eclipse (see Javadoc for `org.eclipse.core.runtime.Platform`). Indicates this entry should only be installed on one of the specified OS systems. If this attribute is not specified, the entry can be installed on all systems (portable implementation). This information is used as a hint by the installation and update support (user can force installation of entry regardless of this setting).
  - arch – optional machine architecture specification. A comma–separated list of architecture designators defined by Eclipse (see Javadoc for `org.eclipse.core.runtime.Platform`). Indicates this feature should only be installed on one of the specified systems. If this attribute is not specified, the feature can be installed on all systems (portable implementation). This information is used as a hint by the installation and update support (user can force installation of feature regardless of this setting).
  - ws – optional windowing system specification. A comma–separated list of window system designators defined by Eclipse (see Javadoc for `org.eclipse.core.runtime.Platform`). Indicates this entry should only be installed on one of the specified WS's. If this attribute is not specified, the entry can be installed on all systems (portable implementation). This information is used as a hint by the installation and update support (user can force installation of entry regardless of this setting).

- nl – optional locale specification. A comma–separated list of locale designators defined by Java. Indicates this entry should only be installed on a system running with a compatible locale (using Java locale–matching rules). If this attribute is not specified, the entry can be installed on all systems (language–neutral implementation). This information is used as a hint by the installation and update support (user can force installation of entry regardless of this setting).

- <requires> – optional feature dependency information. Is expressed in terms of plug–in dependencies. If specified, is enforced by the installation and update support at the time of installation
  - <import> – dependency entry. Specification and processing is a subset of the <import> specification in plugin.xml
    - plugin – identifier of dependent plug–in, if plug–in is used to express dependency
    - feature (new in 2.1) – identifier of dependent feature, if feature is used to express dependency. **Either plugin or feature attribute must be set, but not both.** If "patch" is "true", feature attribute must be used.
    - version – optional plug–in version specification. If "patch" is "true", version must be set.
    - match – optional matching rule. Valid values and processing are as follows:
      - if version attribute is not specifies, the match attribute (if specified) is ignored.
      - ***perfect*** – dependent plug–in version must match exactly the specified version. If "patch" is "true", "perfect" is assumed and other values cannot be set.
      - ***equivalent*** – dependent plug–in version must be at least at the version specified, or at a higher service level (major and minor version levels must equal the specified version).
      - ***compatible*** – dependent plug–in version must be at least at the version specified, or at a higher service level or minor level (major version level must equal the specified version).
      - ***greaterOrEqual*** – dependent plug–in version must be at least at the version specified, or at a higher service, minor or major level.
    - patch – if "true", this constraint declares the enclosing feature to be a patch for the referenced feature. Certain rules must be followed when this attribute is set:
      - feature attribute must be used to identifier of feature being patched
      - version attribute must be set
      - match attribute should not be set and "perfect" value will be assumed.
      - if other features are <include>'ed, they must also be patches.

      A patch is a special feature that carries newer versions of plug–ins for the feature it is patching. It does not replace the feature. A patch can also carry other patches by inclusion.

- <plugin> – identifies referenced plug–in
  - id – required plug–in identifier (from plugin.xml)
  - version – required plug–in version (from plugin.xml)
  - fragment – optional specification indicating if this entry is a plug–in fragment. Default is "false"
  - os – optional operating system specification. A comma–separated list of operating system designators defined by Eclipse (see Javadoc for `org.eclipse.core.runtime.Platform`). Indicates this entry should only be installed on one of the specified os systems. If this attribute is not specified, the entry can be installed on all systems (portable implementation). This information is used as a hint by the installation and update support (user can force installation of entry regardless of this setting).

- arch – optional machine architecture specification. A comma–separated list of architecture designators defined by Eclipse (see Javadoc for `org.eclipse.core.runtime.Platform`). Indicates this feature should only be installed on one of the specified systems. If this attribute is not specified, the feature can be installed on all systems (portable implementation). This information is used as a hint by the installation and update support (user can force installation of feature regardless of this setting).
- ws – optional windowing system specification. A comma–separated list of window system designators defined by Eclipse (see Javadoc for `org.eclipse.core.runtime.Platform`). Indicates this entry should only be installed on one of the specified WS's. If this attribute is not specified, the entry can be installed on all systems (portable implementation). This information is used as a hint by the installation and update support (user can force installation of entry regardless of this setting).
- nl – optional locale specification. A comma–separated list of locale designators defined by Java. Indicates this entry should only be installed on a system running with a compatible locale (using Java locale–matching rules). If this attribute is not specified, the entry can be installed on all systems (language–neutral implementation). This information is used as a hint by the installation and update support (user can force installation of entry regardless of this setting).
- download–size – optional hint supplied by the feature packager, indicating the download size in KBytes of the referenced plug–in archive. If not specified, the download size is not known (**Implementation Note:** the implementation needs to distinguish between "not known" and 0 size)
- install–size – optional hint supplied by the feature packager, indicating the install size in KBytes of the referenced plug–in archive. If not specified, the install size is not known (**Implementation Note:** the implementation needs to distinguish between "not known" and 0 size)
- unpack (new in 3.0) – optional specification supplied by the feature packager, indicating that plugin is capable of running from a jar, and contents of plug–in jar should not be unpacked into a directory. Default is "true". (**Implementation Note:** partial plug–ins – delivered in a feature specifying `org.eclipse.update.core.DeltaInstallHandler` as an install handler should not set unpack to "false")

- <data> – identifies non–plugin data that is part of the feature
  - id – required data identifier in the form of a relative path.
  - os – optional operating system specification. A comma–separated list of operating system designators defined by Eclipse (see Javadoc for `org.eclipse.core.runtime.Platform`). Indicates this entry should only be installed on one of the specified OS's. If this attribute is not specified, the entry can be installed on all systems (portable implementation). This information is used as a hint by the installation and update support (user can force installation of entry regardless of this setting).
  - arch – optional machine architecture specification. A comma–separated list of architecture designators defined by Eclipse (see Javadoc for `org.eclipse.core.runtime.Platform`). Indicates this feature should only be installed on one of the specified systems. If this attribute is not specified, the feature can be installed on all systems (portable implementation). This information is used as a hint by the installation and update support (user can force installation of feature regardless of this setting).
  - ws – optional windowing system specification. A comma–separated list of window system designators defined by Eclipse (see Javadoc for `org.eclipse.core.runtime.Platform`). Indicates this entry should only be installed on one of the specified WS's If this attribute is not specified, the entry can be

installed on all systems (portable implementation). This information is used as a hint by the installation and update support (user can force installation of entry regardless of this setting).

- nl – optional locale specification. A comma–separated list of locale designators defined by Java. Indicates this entry should only be installed on a system running with a compatible locale (using Java locale–matching rules). If this attribute is not specified, the entry can be installed on all systems (language–neutral implementation). This information is used as a hint by the installation and update support (user can force installation of entry regardless of this setting).
- download–size – optional hint supplied by the feature packager, indicating the download size in KBytes of the referenced data archive. If not specified, the download size is not known (**Implementation Note:** the implementation needs to distinguish between "not known" and 0 size)
- install–size – optional hint supplied by the feature packager, indicating the install size in KBytes of the referenced data archive. If not specified, the install size is not known (**Implementation Note:** the implementation needs to distinguish between "not known" and 0 size)

When interacting with the update site, the feature implementation maps the `<plugin>` and `<data>` elements into path identifiers used by the site to determine the actual files to download and install. The default feature implementation supplied by Eclipse constructs the path identifiers as follows:

- `<plugin>` element results in a path entry in the form "`plugins/<pluginId>_<pluginVersion>.jar`" (for example, "`plugins/org.eclipse.core.boot_2.0.0.jar`")
- `<data>` element results in a path entry in the form "`features/<featureId>_<featureVersion>/<dataId>`" (for example, "`features/com.xyz.tools_1.0.3/examples.zip`")

Note, that in general the feature.xml manifest documents should specify UTF–8 encoding. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
```

Translatable text contained in the feature.xml can be separated into feature<_locale>.properties files using Java property bundle conventions. Note that the translated strings are used at installation time (i.e. do not employ the plug–in fragment runtime mechanism).

Primary feature

In Eclipse 2.1, the notion of a **primary feature** was used to customize the product–branding aspects of a product. This mechanism is still supported in Eclipse 3.0, but is effectively deprecated. Newly developed products should use the products extension point to define a product.

The remainder of this discussion describes the legacy product definition using primary features.

When the Eclipse platform is started, exactly one feature can control the overall "personality" or "branding" of the platform, including the splash screen, window images, about box, welcome page, and other customizable aspects of the platform. This feature is called the product's **primary feature**.

Let's look again at the description of the platform feature from Eclipse 2.1:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<feature
      id="org.eclipse.platform"
      label="%featureName"
      version="2.1.0"
      provider-name="%providerName"
      plugin=""
      image="eclipse_update_120.jpg"
      primary="true"
      application="org.eclipse.ui.ide.workbench">
      ...
```

The platform feature has been designated as a primary feature. Although it's possible to designate many features as primary features in their **feature.xml** files, only one primary feature gets control when the platform is launched. This is controlled by setting the **eclipse.product** property in the product's **config.ini** file underneath the **eclipse/configuration** directory. If there are multiple eligible primary features, the **–product** command line option for **eclipse.exe** overrides the choice made in **config.ini**.

Preferences

Identifier:

org.eclipse.core.runtime.preferences

Since:

3.0

Description:

The preferences extension point allows plug−ins to add new preference scopes to the Eclipse preference mechanism as well as specify the class to run to initialize default preference values at runtime.

Configuration Markup:

<!ELEMENT extension (scope* , initializer* , modifier*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

<!ELEMENT scope EMPTY>

<!ATTLIST scope

name CDATA #REQUIRED

class CDATA #REQUIRED>

Element describing a client's definiton of a new preference scope.

- **name** − The name of the scope.
- **class** − The name of the class.

<!ELEMENT initializer EMPTY>

<!ATTLIST initializer

class CDATA #REQUIRED>

Element which defines the class to use for runtime preference initialization.

- **class** – The name of the class.

<!ELEMENT modifier EMPTY>

<!ATTLIST modifier

class CDATA #REQUIRED>

Element which defines the class to use for preference modification listening.

- **class** – The name of the class.

Examples:

Following is an example of a preference scope declaration. This example declares that this plug–in will provide a preference implementation for the scope "foo". It also declares that when the default values are loaded for this plug–in, the class "MyPreferenceInitializer" contains code to be run to initialize preference default values at runtime.

<extension point=

"org.eclipse.core.runtime.preferences"

>

<scope name=

"foo"

class=

"com.example.FooPrefs"

/>

<initializer class=

"com.example.MyPreferenceInitializer"

/>

<modifier class=

"com.example.MyModifyListener"

/>

</extension>

API Information:

The preference service (obtained by calling `org.eclipse.core.runtime.Platform.getPreferencesService()`) is the hook into the Eclipse preference mechanism.

Supplied Implementation:

The org.eclipse.core.runtime plug−in provides preference implementations for the "configuration", "instance", and "default" scopes. The org.eclipse.core.resources plug−in provides an implementation for "project" preferences.

File encoding and content types

The platform runtime plug−in defines infrastructure for defining and discovering **content types** for data streams. (See Content types for an overview of the content framework.) An important part of the content type system is the ability to specify different encodings (character sets) for different kinds of content. The resources API further allows default character sets to be established for projects, folders, and files. These default character sets are consulted if the content of the file itself does not define a particular encoding inside its data stream.

Setting a character set

We've seen in Content types that default file encodings can be established for content types. More fine−grained control is provided by the resources API.

IContainer defines protocol for setting the default character set for a particular project or folder. This gives plug−ins (and ultimately the user) more freedom in determining an appropriate character set for a set of files when the default character sets from the content type may not be appropriate.

IFile defines API for setting the default character set for a particular file. If no encoding is specified inside the file contents, then this character set will be used. The file's default character set takes precedence over any default character set specified in the file's folder, project, or content type.

Since:

Both of these features are available to the end−user in the properties page for a resource.

Querying the character set

IFile also defines API for querying the character set of a file. A boolean flag specifies whether only the character set explicitly defined for the file should be returned, or whether an implied character set should be returned. For example:

```
String charset = myFile.getCharset(false);
```

returns null if no character set was set explicitly on myFile. However,

```
String charset = myFile.getCharset(true);
```

will first check for a character set that was set explicitly on the file. If none is found, then the content of the file will be checked for a description of the character set. If none is found, then the file's containing folders and projects will be checked for a default character set. If none is found, the default character set defined for the content type itself will be checked. And finally, the platform default character set will be returned if there is no other designation of a default character set. The convenience method **getCharset()** is the same as using **getCharset(true)**.

Content types for files in the workspace

For files in the workspace, ***IFile*** provides API for obtaining the file content description:

```
IFile file = ...;
IContentDescription description = file.getDescription();
```

This API should be used even when clients are only interested in determining the content type − the content type can be easily obtained from the content description. It is possible to detect the content type or describe files in the workspace by obtaining the contents and name and using the API described in Using content types, but that is not recommended. Content type determination using ***IFile.getContentDescription()*** takes into account project natures and project−specific settings. If you go directly to the content type manager, you are ignoring that. But more importantly, because reading the contents of files from disk is very expensive. The Resources plug−in maintains a cache of content descriptions for files in the workspace. This reduces the cost of content description to an acceptable level.

Content types

The **org.eclipse.core.runtime.content** package provides support for detecting the content type for data streams. Content types are used by several content−sensitive features of Eclipse, such as automatic encoding determination, editor selection, and menu contributions. A central content registry allows plug−ins to request content type detection and to find out what content types are available and how they related to each other. The content type registry is extensible so plug−ins can contribute new content type definitions.

Using content types

> *Note: For this discussion, we specifically avoid the use of the word **file** when talking about content. The runtime content engine does not assume that content is contained in a file in the file system. However, it does include protocol that allows content types to be associated with*

file−naming patterns. In practice, these file names represent files in the file system, but nothing in the implementation of the content system assumes that the content is located in the file system. File encoding and content types discusses the file−oriented content type capabilities contributed by the platform resources plug−in and is a must−read for developers interested in using the content type API in that context.

Finding out about content types

Content types are represented by **IContentType**. This interface represents a unique content type that knows how to read a data stream and interpret content type−specific information. Content types are hierarchical in nature. For example, a content type for XML data is considered a child of the text content type. This allows new content types to leverage the attributes or behavior of more general content types.

The **IContentTypeManager** is the entry point that gives access to most of the content type related API provided by the platform runtime. To obtain a reference to the platform **IContentTypeManager**, clients can use the **Platform** API:

```
IContentTypeManager contentTypeManager = Platform.getContentTypeManager();
```

Clients can use the platform **IContentTypeManager** to find out about the content types in the system.

- **getAllContentTypes** allows clients to get all of the content types defined in the platform.
- **getContentType** allows clients to obtain a content type by its unique identifier.

Detecting the content type for a data stream

Given a stream of bytes, it is possible to determine its content type by calling the **IContentTypeManager** API as follows:

```
InputStream stream = ...;
IContentType contentType = contentTypeManager.findContentTypeFor(stream, "file.xml");
stream.close();
```

This will return the most appropriate ***IContentType*** given the input provided, or `null` if none can be found. Multiple content types might be deemed appropriate for a given data stream. In that case, the platform uses some heuristics to determine which one should be selected. The file name is the first criterion by which content types are selected. It can be omitted, but this has two issues: the results might not be as correct because many unrelated content types might accept the same input; there is also a big performance hit, since all content types in the platform have to be given a chance of analysing the stream. So, unless it is not available, clients should always provide a file name along with the stream.

Describing a data stream

Another interesting feature of the content type support in the platform is the ability of *describing the contents* of a binary or character stream. The following code snippet shows how to do that:

```
InputStream stream = ...;
IContentDescription description = contentTypeManager.getDescriptionFor(stream, "file.xml");
stream.close();
```

The returned ***IContentDescription*** instance describes the content type and additional relevant information extracted from the contents provided. The content description stores content−specific properties in form of key/value pairs. The platform itself is able to describe properties such as the character set and the byte order of text−based streams, but others can be defined by content type providers.

Providing content−sensitive features

New content types are often defined as specialization of existing ones. This hierarchy establishes a "is a" relationship between a derived content type and its base type. Plug−in developers must honor this when implementing content sensitive features. If a given feature is applicable to a given content type, the feature must be applicable to any derived content types as well. The ***IContentType.isKindOf(IContentType superType)*** method allows determining whether two ***IContentType***s are related. The method ***IContentType.getBaseType()*** allows determining the base type of a given ***IContentType***.

Resource markers

We know that plug−ins can define specialized file extensions and contribute editors that provide specialized editing features for these file types. During the course of editing (or building) a resource, a plug−in may need to tag resources to communicate problems or other information to the user. The resource marker mechanism is used to manage this kind of information.

A **marker** is like a yellow sticky note stuck to a resource. On the marker you can record information about a problem (e.g., location, severity) or a task to be done. Or you can simply record a location for a marker as a bookmark.

Users can quickly jump to the marked location within a resource. The workbench UI supports presentation of bookmarks, breakpoints, tasks, and problems along the side of the editor. These markers can also be shown as items in views, such as the tasks or bookmarks view.

The platform resources API defines methods for creating markers, setting marker values, and extending the platform with new marker types. While the platform manages markers, it is the plug−ins that control their creation, removal and attribute values.

Markers are intended to be small, lightweight objects. There could be hundreds, even thousands of markers in a single project. For example, the Java compiler uses a marker to flag each problem it finds in source code.

The platform will throw away markers attached to resources that are deleted, but plug−ins are responsible for removing their stale markers when they no longer apply to a resource that still exists.

Marker operations

Manipulating a marker is similar to manipulating a resource. Markers are **handle** objects. You can obtain a marker handle from a resource, but you don't know if it actually exists until you use **exists()** protocol or otherwise try to manipulate it. Once you've established that a marker exists, you can query named attributes that may have been assigned to it.

Markers are owned and managed by the platform, which takes care of making markers persistent and notifying listeners as markers are added, deleted, or changed. Plug−ins are responsible for creating any necessary markers, changing their attributes, and removing them when they are no longer needed.

Marker creation

Markers are not directly created using a constructor. They are created using a factory method (**IResource.createMarker()**) on the associated resource.

```
IMarker marker = file.createMarker(IMarker.TASK);
```

To create a marker that has global scope (not associated with any specific resource), you can use the workspace root (**IWorkspace.getRoot()**) as the resource.

Marker deletion

The code for deleting a marker is straightforward.

```
try {
   marker.delete();
} catch (CoreException e) {
   // Something went wrong
}
```

When a marker is deleted, its marker object (handle) becomes "stale." Plug−ins should use the **IMarker.exists()** protocol to make sure a marker object is still valid.

Markers can be deleted in batch by asking a resource to delete its markers. This method is useful when removing many markers at once or if individual marker references or ids are not available.

```
int depth = IResource.DEPTH_INFINITE;
try {
   resource.deleteMarkers(IMarker.PROBLEM, true, depth);
} catch (CoreException e) {
   // something went wrong
}
```

When deleting a group of markers, you specify a marker **type** to delete, such as **IMarker.PROBLEM**, or **null** to delete all markers. The second argument indicates whether you want to delete subtype markers. (We'll look a subtypes in a moment when we define new marker types.) The **depth** argument controls the depth of deletion.

You can also delete markers using **IWorkspace.deleteMarkers(IMarker [])**.

Marker attributes

Given a marker, you can ask for its associated resource, its id (unique relative to that resource), and its type. You can also access additional information via generic attributes.

Each type of marker has a specific set of attributes that are defined by the creator of the marker type using naming conventions. The **IMarker** interface defines a set of constants containing the standard attribute names (and some of the expected values) for the platform marker types. The following method manipulates attributes using the platform constants.

```
IMarker marker = file.createMarker(IMarker.TASK);
if (marker.exists()) {
   try {
      marker.setAttribute(IMarker.MESSAGE, "A sample marker message");
```

```
        marker.setAttribute(IMarker.PRIORITY, IMarker.PRIORITY_HIGH);
    } catch (CoreException e) {
        // You need to handle the case where the marker no longer exists      }
}
```

Attributes are maintained generically as name/value pairs, where the names are strings and a value can be any one of the supported value types (boolean, integer, string). The limitation on value types allows the platform to persist the markers quickly and simply.

Querying markers

Resources can be queried for their markers and the markers of their children. For example, querying the workspace root with infinite depth considers all of the markers in the workspace.

```
IMarker[] problems = null;
int depth = IResource.DEPTH_INFINITE;
try {
   problems = resource.findMarkers(IMarker.PROBLEM, true, depth);
} catch (CoreException e) {
   // something went wrong
}
```

The result returned by **findMarkers** depends on the arguments passed. In the snippet above, we are looking for all markers of type **PROBLEM** that appear on the resource and all of its direct and indirect descendants.

If you pass **null** as the marker type, you will get all the marker types associated with the resource. The second argument specifies whether you want to look at the resource's children. The **depth** argument controls the depth of the search when you are looking at the resource's children. The depth can be **DEPTH_ZERO** (just the given resource), **DEPTH_ONE** (the resource and all of its direct children) or **DEPTH_INFINITE** (the resource and all of its direct and indirect descendants).

Marker persistence

The platform standard markers (task, problem, and bookmark) are **persistent**. This means that their state will be saved across workbench shutdown and startup. However, markers of a persistent type may be selectively made transient by setting the reserved attribute ***transient*** to true.

New marker types declared by plug−ins are not persistent unless they are declared as such.

Extending the platform with new marker types

Plug−ins can declare their own marker types using the **org.eclipse.core.resources.markers** extension point. The standard marker types for problems, tasks and bookmarks are declared by the platform in the resources plug−in's markup.

```
<extension
   id="problemmarker"
   point="org.eclipse.core.resources.markers"
   name="%problemName">
   <super type="org.eclipse.core.resources.marker"/>
   <persistent value="true"/>
   <attribute name="severity"/>
   <attribute name="message"/>
   <attribute name="location"/>
```

```
</extension>
<extension
   id="taskmarker"
   point="org.eclipse.core.resources.markers"
   name="%taskName">
   <super type="org.eclipse.core.resources.marker"/>
   <persistent value="true"/>
   <attribute name="priority"/>
   <attribute name="message"/>
   <attribute name="done"/>
   <attribute name="userEditable"/>
</extension>
<extension
   id="bookmark"
   point="org.eclipse.core.resources.markers"
   name="%bookmarkName">
   <super type="org.eclipse.core.resources.marker"/>
   <persistent value="true"/>
   <attribute name="message"/>
   <attribute name="location"/>
</extension>
```

New marker types are derived from existing ones using multiple inheritance. New marker types inherit all of the attributes from their super types and add any new attributes defined as part of the declaration. They also transitively inherit attributes from the super types of their super types. The following markup defines a new kind of marker in a hypothetical **com.example.markers** plug−in.

```
<extension
   id="mymarker"
   point="org.eclipse.core.resources.markers" />
<extension
   id="myproblem"
   point="org.eclipse.core.resources.markers">
   <super type="org.eclipse.core.resources.problemmarker"/>
   <super type="com.example.markers.mymarker"/>
   <attribute name="myAttribute" />
   <persistent value="true" />
</extension>
```

Note that the type **org.eclipse.core.resources.problemmarker** is actually one of the pre−defined types (aka **IMarker.PROBLEM**).

The only aspect of a marker super type that is not inherited is its **persistence** flag. The default value for persistence is false, so any marker type that should be persistent must specify **<persistent value=''true''/>**.

After declaring the new marker type in your plug−in manifest file, you can create instances of **com.example.markers.myproblem** marker type and freely set or get the **myAttribute** attribute.

Declaring new attributes allows you to associate data with markers that you plan to use elsewhere (in your views and editors). Markers of a particular type do not have to have values for all of the declared attributes. The attribute declarations are more for solving naming convention problems (so everyone uses "message" to talk about a marker's description) than for constraining content.

```
public IMarker createMyMarker(IResource resource) {
   try {
      IMarker marker = resource.createMarker("com.example.markers.myproblem");
      marker.setAttribute("myAttribute", "MYVALUE");
      return marker;
```

```
    } catch (CoreException e) {
       // You need to handle the cases where attribute value is rejected
    }
}
```

You can query your own marker types in the same way you query the platform marker types. The method below finds all **mymarkers** associated with the given target resource and all of its descendents. Note that this will also find all **myproblems** since true is passed for the **includeSubtypes** argument.

```
public IMarker[] findMyMarkers(IResource target) {
    String type = "com.example.markers.mymarker";
    IMarker[] markers = target.findMarkers(type, true, IResource.DEPTH_INFINITE);
}
```

Resource Markers

Identifier:

org.eclipse.core.resources.markers

Description:

The workspace supports the notion of markers on arbitrary resources. A marker is a kind of metadata (similar to properties) which can be used to tag resources with user information. Markers are optionally persisted by the workspace whenever a workspace save or snapshot is done.

Users can define and query for markers of a given type. Marker types are defined in a hierarchy that supports multiple−inheritance. Marker type definitions also specify a number attributes which must or may be present on a marker of that type as well as whether or not markers of that type should be persisted.

The markers extension−point allows marker writers to register their marker types under a symbolic name that is then used from within the workspace to create and query markers. The symbolic name is the id of the marker extension. When defining a marker extension, users are encouraged to include a human−readable value for the "name" attribute which indentifies their marker and potentially may be presented to users.

Configuration Markup:

<!ELEMENT extension (super* , persistent? , attribute*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point −** a fully qualified identifier of the target extension point
- **id −** an optional identifier of the extension instance
- **name −** an optional name of the extension instance

<!ELEMENT super EMPTY>

<!ATTLIST super

type CDATA #REQUIRED>

- **type −** the fully−qualified id of a marker super type (i.e., a marker type defined by another marker extension)

<!ELEMENT persistent EMPTY>

<!ATTLIST persistent

value (true | false) >

- **value –** "`true`" or "`false`" indicating whether or not markers of this type should be persisted by the workspace. If the persistent characteristic is not specified, the marker type is **not** persisted.

<!ELEMENT attribute EMPTY>

<!ATTLIST attribute

name CDATA #REQUIRED>

- **name –** the name of an attribute which may be present on markers of this type

Examples:

Following is an example of a marker configuration:

<extension id=

"com.xyz.coolMarker"

point=

"org.eclipse.core.resources.markers"

name=

"Cool Marker"

>

<persistent value=

"true"

/>

<super type=

"org.eclipse.core.resources.problemmarker"

/>

<super type=

"org.eclipse.core.resources.textmarker"

/>

<attribute name=

"owner"

/>

</extension>

API Information:

All markers, regardless of their type, are instances of `org.eclipse.core.resources.IMarker`.

Supplied Implementation:

The platform itself has a number of pre−defined marker types. Particular product installs may include additional markers as required.

Modifying the workspace

In the course of performing its function, your plug−in may need to make changes to resources in the workspace. The workspace is an important data model for many plug−ins in the system, many of which rely on keeping up with the current state of the workspace. Plug−ins may even be concurrently updating the workspace. It's important for your plug−in to act as a responsible workspace citizen when making changes to resources. What makes a plug−in a good workspace citizen?

- **Batching** of changes where possible to avoid flooding the system with unnecessary events or triggering unnecessary processing on an interim state.
- **Listening** to resource change events and updating models as the workspace changes.
- **Fine−grained locking** of the workspace when making modifications instead of locking the entire workspace.

The next few sections look at these concepts in more detail.

Tracking resource changes

We've just seen how to batch resource changes in a runnable (Batching resource changes). Let's look at the other side of the coin. What if you want to keep track of all of the changes to the workspace that happen while

your plug−in is running? You can register an **IResourceChangeListener** with the workspace. Your listener will be notified of the changes via an **IResourceChangeEvent** object, which describes the changes.

Registering a listener

First, you must register a resource change listener with the workspace.

```
IResourceChangeListener listener = new MyResourceChangeReporter();
ResourcesPlugin.getWorkspace().addResourceChangeListener(
   listener, IResourceChangeEvent.POST_CHANGE);
```

Your listener will be notified after modifications to the workspace resources have been made. Resource API methods that modify resources trigger these events as part of their documented behavior. The method comment for a resource API method explicitly states whether or not it triggers a resource change event. For example, the following is included in the **IFile.setContents()** comment:

```
This method changes resources; these changes will be reported in a subsequent
resource change event, including an indication that this file's content have
been changed.
```

Methods that create, delete, or change a resource typically trigger these events. Methods that read, but do not write, resources typically do not trigger these events.

Resource change events

The resource change event describes the specifics of the change (or set of changes) that have occurred in the workspace. The event contains a **resource delta** that describes the *net* effect of the changes. For example, if you add a resource and later delete it during one batch of changes, the resource will not appear in the delta.

The resource delta is structured as a tree rooted at the workspace root. The resource delta tree describes these types of changes:

- Resources that have been created, deleted, or changed. If you have deleted (or added) a folder, the resource delta will include the folder and all files contained in the folder.
- Resources that have been moved or renamed using the **IResource.move()** API.
- Markers that have been added, removed, or changed. Marker modification is considered to be a workspace modification operation.
- Files that have been modified. Changed files are identified in the resource delta, but you do not have access to the previous content of the file in the resource delta.

To traverse a resource delta tree, you may implement the **IResourceDeltaVisitor** interface or traverse the tree explicitly using **IResource.getAffectedChildren**. Resource delta visitors implement a **visit** method that is called by the resource delta as it enumerates each change in the tree.

> *Note: Changes made to resource session properties or resource persistent properties are not identified in the resource delta.*

Resource change events are sent whenever a change (or batched set of changes) is made to the workspace. In addition, resource change events are sent for certain specific workspace operations. The table below summarizes the types of resource change events and when they are reported.

Event type	Description

PRE_CLOSE	Notifies listeners that a project is about to be closed. This event can be used to extract and save necessary information from the in−memory representation (e.g., session properties) of a project before it is closed. (When a project is closed, the in−memory representation is disposed). The workspace is locked (no resources can be updated) during this event. The event contains the project that is being closed.
PRE_DELETE	Notifies listeners that a project is about to deleted. This event can be used to perform clean−up operations, such as removing any saved state that is related to the project from your plug−in's directory. The workspace is locked (no resources can be updated) during this event. The event contains the project that is being deleted.
PRE_AUTOBUILD	Notifies listeners before any auto−building occurs. This event is broadcast when the platform detects an auto−build needs to occur, regardless of whether auto−building is actually enabled. The workspace is **not** locked during this event (resources can be updated). The event contains a resource delta describing the changes that have occurred since the last **POST_CHANGE** event was reported.
POST_AUTOBUILD	Notifies listeners after any auto−building has occurred. This event is broadcast after the platform would have performed an auto−build, regardless of whether auto−building is actually enabled. The workspace is **not** locked during this event (resources can be updated). The event contains a resource delta describing the changes that have occurred since the last **POST_CHANGE** event was reported.
POST_CHANGE	Describes a set of changes that have occurred to the workspace since the last **POST_CHANGE** event was reported. Triggered after a resource change API is used individually or in a batched set of workspace changes. Also triggered after any **PRE_AUTOBUILD** or **POST_AUTOBUILD** notification is complete. The event contains a resource delta describing the net changes since the last **POST_CHANGE** event. The workspace is locked (no resources can be updated) during this event.

Implementing a resource change listener

The following example implements a console−based resource change listener. A resource change listener is registered for specific types of events and information about these events is printed to the console:

```
IResourceChangeListener listener = new MyResourceChangeReporter();
ResourcesPlugin.getWorkspace().addResourceChangeListener(listener,
   IResourceChangeEvent.PRE_CLOSE
   | IResourceChangeEvent.PRE_DELETE
   | IResourceChangeEvent.PRE_AUTO_BUILD
   | IResourceChangeEvent.POST_AUTO_BUILD
   | IResourceChangeEvent.POST_CHANGE);
```

The listener checks for each event type and reports information about the resource that was changed and the kinds of changes that occurred. Although this example is designed to show a general listener that handles all the types of resource events, a typical listener would register for just one type of event.

The implementation for **POST_CHANGE** uses another class that can be used to visit the changes in the resource delta.

```
import org.eclipse.resources.*;
import org.eclipse.runtime.*;

public class MyResourceChangeReporter implements IResourceChangeListener {
   public void resourceChanged(IResourceChangeEvent event) {
      IResource res = event.getResource();
      switch (event.getType()) {
```

```
            case IResourceChangeEvent.PRE_CLOSE:
               System.out.print("Project ");
               System.out.print(res.getFullPath());
               System.out.println(" is about to close.");
               break;
            case IResourceChangeEvent.PRE_DELETE:
               System.out.print("Project ");
               System.out.print(res.getFullPath());
               System.out.println(" is about to be deleted.");
               break;
            case IResourceChangeEvent.POST_CHANGE:
               System.out.println("Resources have changed.");
               event.getDelta().accept(new DeltaPrinter());
               break;
            case IResourceChangeEvent.PRE_AUTO_BUILD:
               System.out.println("Auto build about to run.");
               event.getDelta().accept(new DeltaPrinter());
               break;
            case IResourceChangeEvent.POST_AUTO_BUILD:
               System.out.println("Auto build complete.");
               event.getDelta().accept(new DeltaPrinter());
               break;
         }
      }
   }
```

The **DeltaPrinter** class implements the **IResourceDeltaVisitor** interface to interrogate the resource delta. The **visit()** method is called for each resource change in the resource delta. The visitor uses a return value to indicate whether deltas for child resources should be visited.

```
class DeltaPrinter implements IResourceDeltaVisitor {
   public boolean visit(IResourceDelta delta) {
      IResource res = delta.getResource();
      switch (delta.getKind()) {
         case IResourceDelta.ADDED:
            System.out.print("Resource ");
            System.out.print(res.getFullPath());
            System.out.println(" was added.");
            break;
         case IResourceDelta.REMOVED"
            System.out.print("Resource ");
            System.out.print(res.getFullPath());
            System.out.println(" was removed.");
            break;
         case IResourceDelta.CHANGED:
            System.out.print("Resource ");
            System.out.print(res.getFullPath());
            System.out.println(" has changed.");
            break;
      }
      return true; // visit the children
   }
}
```

Further information can be obtained from the supplied resource delta. The following snippet shows how the **IResourceDelta.CHANGED** case could be implemented to further describe the resource changes.

```
...
case IResourceDelta.CHANGED:
   System.out.print("Resource ");
```

```
      System.out.print(delta.getFullPath());
      System.out.println(" has changed.");
      int flags = delta.getFlags();
      if ((flags & IResourceDelta.CONTENT) != 0) {
            System.out.println("--> Content Change");
      }
      if ((flags & IResourceDelta.REPLACED) != 0) {
            System.out.println("--> Content Replaced");
      }
      if ((flags & IResourceDelta.MARKERS) != 0) {
            System.out.println("--> Marker Change");
            IMarkerDelta[] markers = delta.getMarkerDeltas();
            // if interested in markers, check these deltas
      }
      break;
...
```

For a complete description of resource deltas, visitors, and marker deltas, consult the API specification for **IResourceDelta**, **IResourceDeltaVisitor**, and **IMarkerDelta**.

> *Note: Resource change listeners are useful for tracking changes that occur to resources while your plug−in is activated. If your plug−in registers a resource change listener during its startup code, it's possible that many resource change events have been triggered before the activation of your plug−in. The resource delta contained in the first resource change event received by your plug−in will not contain all of the changes made since your plug−in was last activated. If you need to track changes made between activations of your plug−in, you should use the support provided for workspace saving. This is described in Workspace save participation.*
>
> *Note: Some resource change events are triggered during processing that occurs in a background thread. Resource change listeners should be thread−safe. See Threading issues for a discussion about thread safety with the UI.*

Threading issues

When working with a widget toolkit, it is important to understand the underlying thread model that is used for reading and dispatching platform GUI events. The implementation of the UI thread affects the rules that applications must follow when using Java threads in their code.

Native event dispatching

Underneath any GUI application, regardless of its language or UI toolkit, the OS platform detects GUI events and places them in application event queues. Although the mechanics are slightly different on different OS platforms, the basics are similar. As the user clicks the mouse, types characters, or surfaces windows, the OS generates application GUI events, such as mouse clicks, keystrokes, or window paint events. It determines which window and application should receive each event and places it in the application's event queue.

The underlying structure for any windowed GUI application is an event loop. Applications initialize and then start a loop which simply reads the GUI events from the queue and reacts accordingly. Any work that is done while handling one of these events must happen quickly in order to keep the GUI system responsive to the user.

Long operations triggered by UI events should be performed in a separate thread in order to allow the event loop thread to return quickly and fetch the next event from the application's queue. However, access to the widgets and platform API from other threads must be controlled with explicit locking and serialization. An application that fails to follow the rules can cause an OS call to fail, or worse, lock up the entire GUI system.

SWT UI thread

SWT follows the threading model supported directly by the platforms. The application program runs the event loop in its main thread and dispatches events directly from this thread. The UI thread is the thread in which the ***Display*** was created. All other widgets must be created in the UI thread.

Since all event code is triggered from the application's UI thread, application code that handles events can freely access the widgets and make graphics calls without any special techniques. However, the application is responsible for forking computational threads when performing long operations in response to an event.

> *Note: SWT will trigger an **SWTException** for any calls made from a non−UI thread that must be made from the UI thread.*

The main thread, including the event loop, for an SWT application has the following structure:

```
public static void main (String [] args) {
   Display display = new Display ();
   Shell shell = new Shell (display);
   shell.open ();
   // start the event loop. We stop when the user has done
   // something to dispose our window.
   while (!shell.isDisposed ()) {
      if (!display.readAndDispatch ())
         display.sleep ();
   }
   display.dispose ();
}
```

Once the widgets are created and the shell is opened, the application reads and dispatches events from the OS queue until the shell window is disposed. If there are no events available for us in the queue, we tell the display to sleep to give other applications a chance to run.

SWT provides special access methods for calling widget and graphics code from a background thread.

Executing code from a non−UI thread

Applications that wish to call UI code from a non−UI thread must provide a ***Runnable*** that calls the UI code. The methods ***syncExec(Runnable)*** and ***asyncExec(Runnable)*** in the ***Display*** class are used to execute these runnables in the UI thread during the event loop.

- ***syncExec(Runnable)*** should be used when the application code in the non−UI thread depends on the return value from the UI code or otherwise needs to ensure that the runnable is run to completion before returning to the thread. SWT will block the calling thread until the runnable has been run from the application's UI thread. For example, a background thread that is computing something based on a window's current size would want to synchronously run the code to get the window's size and then continue with its computations.
- ***asyncExec(Runnable)*** should be used when the application needs to perform some UI operations, but

is not dependent upon the operations being completed before continuing. For example, a background thread that updates a progress indicator or redraws a window could request the update asynchronously and continue with its processing. In this case, there is no guaranteed relationship between the timing of the background thread and the execution of the runnable.

The following code snippet demonstrates the pattern for using these methods:

```
// do time-intensive computations
...
// now update the UI. We don't depend on the result,
// so use async.
display.asyncExec (new Runnable () {
   public void run () {
      if (!myWindow.isDisposed())
         myWindow.redraw ();
   }
});
// now do more computations
...
```

It is good practice to check if your widget is disposed from within the runnable when using asyncExec. Since other things can happen in the UI thread between the call to `asyncExec` and the execution of your runnable, you can never be sure what state your widgets are in by the time your runnable executes.

The workbench and threads

The threading rules are very clear when you are implementing an SWT application from the ground up since you control the creation of the event loop and the decision to fork computational threads in your application.

Things get a bit more complicated when you are contributing plug−in code to the workbench. The following rules can be considered "rules of engagement" when using platform UI classes, although from release to release there may be exceptions to these rules:

- In general, any workbench UI extensions you add to the platform will be executing in the workbench's UI thread, unless they are specifically related to threads or background jobs (such as background job progress indication).
- If you receive an event from the workbench, it is not guaranteed that it is executing in the UI thread of the workbench. Consult the javadoc for the particular class that defines the listener or event. If there is no specific documentation discussing threading, and the class is clearly a UI−related class, you may expect that the event arrives in the UI thread of the workbench.
- Likewise, a platform UI library should not be considered thread−safe unless it is specifically documented as such. Note that most platform UI classes dispatch listeners from the calling thread that triggered the event. Workbench and JFace API calls do not check that the caller is executing in the UI thread.This means that your plug−in may introduce a problem if you call a method that triggers an event from a non−UI thread. SWT triggers an ***SWTException*** for all API calls made from a non−UI thread. In general, avoid calling platform UI code from another thread unless the javadoc specifically allows it.
- If your plug−in forks a computational thread or uses a workbench Job, it must use the ***Display asyncExec(Runnable)*** or ***syncExec(Runnable)*** methods when calling any API for the workbench, JFace, or SWT, unless the API specifically allows call−in from a background thread.
- If your plug−in uses the JFace ***IRunnableContext*** interface to invoke a progress monitor and run an operation, it supplies an argument to specify whether a computational thread is forked for running the operation.

Concurrency and the workspace

We've already seen that workspace code must be aware of concurrency even if it is not using the concurrency framework. Batching of workspace changes and use of scheduling rules helps in sharing the workspace with other plug−ins (and their threads) that are modifying the workspace. Once your plug−in is using batching and rules (see Batching resource changes), it is easy to perform the same work using the platform concurrency mechanisms.

Workspace jobs

A **Job** is a basic unit of asynchronous work running concurrently with other jobs. The resources plug−in defines **WorkspaceJob** as a convenient mechanism for defining asynchronous resource modifications. Code that would normally be batched in an **IWorkspaceRunnable** is instead put in the **runInWorkspace** method of a workspace job subtype. Instead of running the code using **IWorkspace** protocol, the job is scheduled just like any other job. The appropriate scheduling rules must be added on the job before it is scheduled.

Let's look at an example workspace runnable and what we should do to make it a job:

```
IWorkspaceRunnable myRunnable =
        new IWorkspaceRunnable() {
                public void run(IProgressMonitor monitor) throws CoreException {
                        //do the actual work in here
                        doSomeWork();
                        ...
                }
}
```

The work is moved to the appropriate method of our **WorkspaceJob** subtype.

```
class MyWorkspaceJob extends WorkspaceJob {
        public MyWorkspaceJob() {
                super("My Workspace Job");
        }
        public IStatus runInWorkspace(IProgressMonitor monitor) {
                //do the actual work in here
                doSomeWork();
                return Status.OK_STATUS;
        }
}
```

Our runnable had to be invoked specifically:

```
IWorkspace workspace = ResourcesPlugin.getWorkspace();
workspace.run(myRunnable, myProject, IWorkspace.AVOID_UPDATE, null);
```

Our job is scheduled like any other job. The platform job manager will run it according to its priority, other jobs in the queue, and the scheduling rules. Note that we must attach the scheduling rule to the job in order to prevent simultaneous modification of `myProject`.

```
MyWorkspaceJob job = new MyWorkspaceJob();
job.setRule(myProject);
job.schedule();
```

Now that the operation has been structured as a job, all of the scheduling mechanisms (priority, delay, rescheduling) can be used. Resource change events will be batched until the job is finished running.

Incremental project builders

An **incremental project builder** is an object that manipulates the resources in a project in a particular way. Incremental project builders are often used to apply a transformation on a resource to produce a resource or artifact of another kind.

Plug−ins contribute incremental project builders to the platform in order to implement specialized resource transformations. For example, the Java development tools (JDT)define an incremental project builder that compiles a Java source file into a class file any time a file is added or modified in a Java project. It also keeps track of dependent files and recompiles them when necessary.

From an API point of view, the platform defines two basic types of builds:

- A **full build** performs a build from scratch. It treats all resources in a project as if they have never been seen by the builder.
- An **incremental build** uses a "last build state," maintained internally by the builder, to do an optimized build based on the changes in the project since the last build.

Incremental builds are seeded with a resource change delta. The delta reflects the net effect of all resource changes since the builder last built the project. This delta is similar to the one used inside resource change events.

Projects can be periodically **cleaned** by the user in order to force a rebuild of a complete project the next time an incremental build is performed on that project. Cleaning a project removes build information such as problem markers and class files.

Builders are best understood by example. The JDT Java compiler is driven by a Java incremental project builder which recompiles the files in a project that are affected by changes. When a full build is triggered, (or an incremental build after a clean), all of the **.java** files in the project are compiled. Any compile problems encountered are added as problem markers on the affected **.java** files. When an incremental build is triggered, the builder selectively recompiles the added, changed, or otherwise affected **.java** files that are described in the resource delta and updates the problem markers as necessary. Any **.class** files or markers that are no longer appropriate are removed.

Incremental building has obvious performance benefits for projects with hundreds or thousands of resources, most of which are unchanging at any given point in time.

The technical challenge for incremental building is to determine exactly what needs to be rebuilt. For example, the internal state maintained by the Java builder includes things like a dependency graph and a list of compilation problems reported. This information is used during an incremental build to identify which classes need to be recompiled in response to a change in a Java resource.

Although the basic structure for building is defined in the platform, the real work is done in the builder code. Patterns for implementing complex incremental builders are beyond the scope of this discussion, since the implementation is dependent on the specific builder design.

Invoking a build

A builder can be invoked explicitly in one of the following ways:

- **IProject.build()** runs the build processing on the receiving project according to the build method's arguments.
- **IWorkspace.build()** runs the build processing on all open projects in the workspace.

In practice, the workbench user triggers a build by selecting corresponding commands in the resource navigator menu.

Incremental project builders are also invoked implicitly by the platform during an auto−build. If enabled, auto−builds run whenever the workspace is changed.

Defining an incremental project builder

The **org.eclipse.core.resources.builders** extension point is used to contribute an incremental project builder to the platform. The following markup shows how the hypothetical plug−in **com.example.builders** could contribute an incremental project builder.

```
<extension
   id="mybuilder" name="My Sample Builder" point="org.eclipse.core.resources.builders">
   <builder
      <run
         class="com.example.builders.BuilderExample">
         <parameter name="optimize" value="true" />
         <parameter name="comment" value="Builder comment" />
      </run>
   </builder>
</extension>
```

The **class** identified in the extension point must extend the platform class **IncrementalProjectBuilder**.

```
public class BuilderExample extends IncrementalProjectBuilder {
   IProject[] build(int kind, Map args, IProgressMonitor monitor)
         throws CoreException {
      // add your build logic here
      return null;
   }
   protected void startupOnInitialize() {
      // add builder init logic here
   }
   protected void clean(IProgressMonitor monitor) {
      // add builder clean logic here
   }
}
```

Build processing begins with the **build()** method, which includes information about the kind of build that has been requested. The build is one of the following values:

- **FULL_BUILD** indicates that all resources in the project should be built.
- **INCREMENTAL_BUILD** indicates that the build is incremental.
- **AUTO_BUILD** indicates that an incremental build is being triggered automatically because a resource has changed and the autobuild feature is on.

If an incremental build has been requested, a resource delta is provided to describe the changes in the resources since the last build. The following snippet further refines the **build()** method.

```
protected IProject[] build(int kind, Map args, IProgressMonitor monitor
```

```
      throws CoreException {
   if (kind == IncrementalProjectBuilder.FULL_BUILD) {
      fullBuild(monitor);
   } else {
      IResourceDelta delta = getDelta(getProject());
      if (delta == null) {
         fullBuild(monitor);
      } else {
         incrementalBuild(delta, monitor);
      }
   }
   return null;
}
```

It sometimes happens that when building project "X," a builder needs information about changes in some other project "Y." (For example, if a Java class in X implements an interface provided in Y.) While building X, a delta for Y is available by calling **getDelta(Y)**. To ensure that the platform can provide such deltas, X's builder must have declared the dependency between X and Y by returning an array containing Y from a previous **build()** call. If a builder has no dependencies, it can simply return null. See **IncrementalProjectBuilder** for further information.

Full build

The logic required to process a full build request is specific to the plug−in. It may involve visiting every resource in the project or even examining other projects if there are dependencies between projects. The following snippet suggests how a full build might be implemented.

```
protected void fullBuild(final IProgressMonitor monitor) throws CoreException {
   try {
      getProject().accept(new MyBuildVisitor());
   } catch (CoreException e) { }
}
```

The build visitor would perform the build for the specific resource (and answer true to continue visiting all child resources).

```
class MyBuildVisitor implements IResourceVisitor {
   public boolean visit(IResource res) {
      //build the specified resource.
      //return true to continue visiting children.
      return true;
   }
}
```

The visit process continues until the full resource tree has been traveled.

Incremental build

When performing an incremental build, the builder works with a resource change delta instead of a complete resource tree.

```
protected void incrementalBuild(IResourceDelta delta,
      IProgressMonitor monitor) throws CoreException {
   // the visitor does the work.
   delta.accept(new MyBuildDeltaVisitor());
}
```

The visit process continues until the complete resource delta tree has been traveled. The specific nature of changes is similar to that described in Implementing a resource change listener. One important difference is that with incremental project builders, you are working with a resource delta based on a particular project, not the entire workspace.

Cleaning before a build

The workbench allows users to **clean** a project or set of projects before initiating a build. This feature allows the user to force a rebuild from scratch on only certain projects. Builders should implement this method to clean up any problem markers and derived resources in the project.

Associating an incremental project builder with a project

To make a builder available for a given project, it must be included in the build spec for the project. A project's build spec is a list of commands to run, in sequence, when the project is built. Each command names a single incremental project builder.

NOTE: The builder name in a build command is the fully qualified id of the builder extension. The fully qualified id of an extension is created by combining the plug−in id with the simple extension id in the plugin.xml file. For example, a builder with simple extension id "mybuilder" in the plug−in "com.example.builders" would have the name "com.example.builders.mybuilder"

The following snippet adds a new builder as the first builder in the existing list of builders.

```
final String BUILDER_ID = "com.example.builders.mybuilder";
IProjectDescription desc = project.getDescription();
ICommand[] commands = desc.getBuildSpec();
boolean found = false;

for (int i = 0; i < commands.length; ++i) {
   if (commands[i].getBuilderName().equals(BUILDER_ID)) {
      found = true;
      break;
   }
}
if (!found) {
   //add builder to project
   ICommand command = desc.newCommand();
   command.setBuilderName(BUILDER_ID);
   ICommand[] newCommands = new ICommand[commands.length + 1];

   // Add it before other builders.
   System.arraycopy(commands, 0, newCommands, 1, commands.length);
   newCommands[0] = command;
   desc.setBuildSpec(newCommands);
   project.setDescription(desc, null);
}
```

Configuring a project's builder is done just once, usually as the project is being created.

Incremental Project Builders

Identifier:

org.eclipse.core.resources.builders

Description:

The workspace supports the notion of an incremental project builder (or "builder" for short"). The job of a builder is to process a set of resource changes (supplied as a resource delta). For example, a Java builder would recompile changed Java files and produce new class files.

Builders are confgured on a per−project basis and run automatically when resources within their project are changed. As such, builders should be fast and scale with respect to the amount of change rather than the number of resources in the project. This typically implies that builders are able to incrementally update their "built state".

The builders extension−point allows builder writers to register their builder implementation under a symbolic name that is then used from within the workspace to find and run builders. The symbolic name is the id of the builder extension. When defining a builder extension, users are encouraged to include a human−readable value for the "name" attribute which identifies their builder and potentially may be presented to users.

Configuration Markup:

<!ELEMENT extension (builder)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #REQUIRED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT builder (run?)>

<!ATTLIST builder

hasNature (true | false)

isConfigurable (true | false) >

- **hasNature –** "`true`" or "`false`" indicating whether the builder is owned by a project nature. If "`true`" and no corresponding nature is found, this builder will not run but will remain in the project's build spec. If the attribute is not specified, it is assumed to be "`false`".

- **isConfigurable** – "`true`" or "`false`" indicating whether the builder allows customization of what build triggers it will respond to. If "`true`", clients will be able to use the API `ICommand.setBuilding` to specify if this builder should be run for a particular build trigger. If the attribute is not specified, it is assumed to be "`false`".

<!ELEMENT run (parameter*)>

<!ATTLIST run

class CDATA #REQUIRED>

- **class** – the fully−qualified name of a subclass of `org.eclipse.core.resources.IncrementalProjectBuilder`.

<!ELEMENT parameter EMPTY>

<!ATTLIST parameter

name CDATA #REQUIRED

value CDATA #REQUIRED>

- **name** – the name of this parameter made available to instances of the specified builder class
- **value** – an arbitrary value associated with the given name and made available to instances of the specified builder class

Examples:

Following is an example of a builder configuration:

<extension id=

"coolbuilder"

name=

"Cool Builder"

point=

"org.eclipse.core.resources.builders"

>

<builder hasNature=

"false"

>

<run class=

"com.xyz.builders.Cool"

>

<parameter name=

"optimize"

value=

"true"

/>

<parameter name=

"comment"

value=

"Produced by the Cool Builder"

/>

</run>

</builder>

</extension>

If this extension was defined in a plug−in with id "com.xyz.coolplugin", the fully qualified name of this builder would be "com.xyz.coolplugin.coolbuilder".

API Information:

The value of the class attribute must represent a subclass of `org.eclipse.core.resources.IncrementalProjectBuilder`.

Supplied Implementation:

The platform itself does not have any predefined builders. Particular product installs may include builders as required.

Derived resources

Many resources get created in the course of translating, compiling, copying, or otherwise processing files that the user creates and edits. **Derived resources** are resources that are not original data, and can be recreated from their source files. It is common for derived files to be excluded from certain kinds of processing.

For example, derived resources are typically not kept in a team repository, since they clutter the repository, change regularly, and can be recreated from their source files. It is not practical for team providers to make decisions about which files are derived. The resource API provides a common mechanism for plug−ins to indicate the resources they create that are derived.

Plug−ins may use **IResource.setDerived(boolean)** to indicate that a resource is derived from other resources. Newly created resources are not derived by default, so this method must be used to explicitly mark the resource as derived. A common use is to mark a subfolder of the project as derived when an "output" folder (such as the "bin" folder in Java projects) is created by the plug−in.

Other plug−ins, usually team providers, can use **IResource.isDerived** to determine whether a particular resource should be managed by the repository. Attempts to mark projects or the workspace root as derived will be ignored.

> *Note: The concept of derived resources is provided for other (non−team) plug−ins to indicate which resources are inappropriate for repository management. Special files created by team implementations to manage their data should not be marked as derived resources. See Team private resources for a technique for marking team−related implementation resources hidden.*

Team private resources

It is common for repository implementations to use extra files and folders to store information specific about the repository implementation. Although these files may be needed in the workspace, they are of no interest to other plug−ins or to the end user.

Team providers may use **IResource.setTeamPrivateMember(boolean)** to indicate that a resource is private to the implementation of a team provider. Newly created resources are not private members by default, so this method must be used to explicitly mark the resource as team private. A common use is to mark a subfolder of the project as team private when the project is configured for team and the subfolder is created.

Other resource API that enumerates resources (such as resource delta trees) will exclude team private members unless explicitly requested to include them. This means that most clients will not "see" the team private resources and they will not be shown to the user. The resource navigator does not show team private members by default, but users can indicate via Preferences that they would like to see team private resources.

Attempts to mark projects or the workspace root as team private will be ignored.

Workspace save participation

Workspace save processing is triggered when the workbench is shut down by the user and at other times periodically by the platform. Plug−ins can participate in the workspace save process so that critical plug−in data is saved to disk whenever the rest of the workspace's persistent data is saved.

The workspace save process can also be used to track changes that occur between activations of your plug−in.

Implementing a save participant

To participate in workspace saving, you must add a save participant to the workspace. This is typically done during your plug−in's startup method. This is also where you read any state that you might have saved when your plug−in was last shut down.

Let's look at a simple plug−in which will demonstrate the save process.

```
package com.example.saveparticipant;

import org.eclipse.core.runtime.*;
import org.eclipse.core.resources.*;
import java.io.File;
import java.util.*;

public class MyPlugin extends Plugin {
   private static MyPlugin plugin;

   public MyPlugin(IPluginDescriptor descriptor) {
      super(descriptor);
      plugin = this;
   }

   public static MyPlugin getDefault() {
      return plugin;
   }

   protected void readStateFrom(File target) {
   }

   public void startup() throws CoreException {
      super.startup();
      ISaveParticipant saveParticipant = new MyWorkspaceSaveParticipant();
      ISavedState lastState =
         ResourcesPlugin.getWorkspace().addSaveParticipant(this, saveParticipant);
      if (lastState == null)
         return;
      IPath location = lastState.lookup(new Path("save"));
      if (location == null)
         return;
      // the plugin instance should read any important state from the file.
      File f = getStateLocation().append(location).toFile();
      readStateFrom(f);
   }

   protected void writeImportantState(File target) {
   }
```

```
}
```

ISaveParticipant defines the protocol for a workspace save participant. Implementors of this interface can provide behavior for different stages of the save process. Let's look at the stages and how our class **WorkspaceSaveParticipant** implements each of these steps.

- **prepareToSave** notifies the participant that the workspace is about to be saved and that it should suspend normal operation until further notice. Our save particpant does nothing here.

```
public void prepareToSave(ISaveContext context) throws CoreException {
}
```

- **saving** tells the participant to save its important state.

```
public void saving(ISaveContext context) throws CoreException {
   switch (context.getKind()) {
      case ISaveContext.FULL_SAVE:
         MyPlugin myPluginInstance = MyPlugin.getDefault();
         // save the plug-in state
         int saveNumber = context.getSaveNumber();
         String saveFileName = "save-" + Integer.toString(saveNumber);
         File f = myPluginInstance.getStateLocation().append(saveFileName).toFile();
         // if we fail to write, an exception is thrown and we do not update the path
         myPluginInstance.writeImportantState(f);
         context.map(new Path("save"), new Path(saveFileName));
         context.needSaveNumber();
         break;
      case ISaveContext.PROJECT_SAVE:
         // get the project related to this save operation
         IProject project = context.getProject();
         // save its information, if necessary
         break;
      case ISaveContext.SNAPSHOT:
         // This operation needs to be really fast because
         // snapshots can be requested frequently by the
         // workspace.
         break;
   }
}
```

The **ISaveContext** describes information about the save operation. There are three kinds of save operations: **FULL_SAVE**, **SNAPSHOT**, and **PROJECT_SAVE**. Save participants should be careful to perform the processing appropriate for the kind of save event they have received. For example, snapshot events may occur quite frequently and are intended to allow plug–ins to save their critical state. Taking a long time to save state which can be recomputed in the event of a crash will slow down the platform.

A save number is used to create data save files that are named using sequential numbers (**save–1**, **save–2**, etc.) Each save file is mapped to a logical file name (**save**) that is independent of the save number. Plug–in data is written to the corresponding file and can be retrieved later without knowing the specific save number of the last successful save operation. Recall that we saw this technique in our plug–in's startup code:

```
IPath location = lastState.lookup(new Path("save"));
```

After we have saved our data and mapped the file name, we call **needSaveNumber** to indicate that we have actively participated in a workspace save and want to assign a number to the save activity. The save numbers can be used to create data files as above.

- **doneSaving** notifies the participant that the workspace has been saved and the participant can continue normal operation.

```
public void doneSaving(ISaveContext context) {
   MyPlugin myPluginInstance = MyPlugin.getDefault();

   // delete the old saved state since it is not necessary anymore
   int previousSaveNumber = context.getPreviousSaveNumber();
   String oldFileName = "save-" + Integer.toString(previousSaveNumber);
   File f = myPluginInstance.getStateLocation().append(oldFileName).toFile();
   f.delete();
}
```

Here, we clean up the save information from the previous save operation. We use **getPreviousSaveNumber** to get the save number that was assigned in the previous save operation (not the one we just completed). We use this number to construct the name of the file that we need to delete. Note that we do not use the save state's logical file map since we've already mapped our current save file number.

- **rollback** tells the participant to rollback the important state because the save operation has failed.

```
public void rollback(ISaveContext context) {
   MyPlugin myPluginInstance = MyPlugin.getDefault();

   // since the save operation has failed, delete the saved state we have just written
   int saveNumber = context.getSaveNumber();
   String saveFileName = "save-" + Integer.toString(saveNumber);
   File f = myPluginInstance.getStateLocation().append(saveFileName).toFile();
   f.delete();
}
```

Here, we delete the state that we just saved. Note that we use the current save number to construct the file name of the file we just saved. We don't have to worry about the fact that we mapped this file name into the **ISaveContext.** The platform will discard the context when a save operation fails.

If your plug−in throws an exception at any time during the save lifecycle, it will be removed from the current save operation and will not get any of the remaining lifecycle methods. For example, if you fail during your **saving** method, you will not receive a **rollback** or **doneSaving** message.

Using previously saved state

When you add a save participant to the workspace, it will return an **ISavedState** object, which describes what your plug−in saved during its last save operation (or **null** if your plug−in has not previously saved any state). This object can be used to access information from the previous save file (using the save number and file map) or to process changes that have occurred between activations of a plug−in.

Accessing the save files

If a file map was used to save logically named files according to the save number, this same map can be used to retrieve the data from the last known save state.

```
ISaveParticipant saveParticipant = new MyWorkspaceSaveParticipant();
ISavedState lastState =
   ResourcesPlugin.getWorkspace().addSaveParticipant(myPluginInstance, saveParticipant);

if (lastState != null) {
   String saveFileName = lastState.lookup(new Path("save")).toString();
   File f = myPluginInstance.getStateLocation().append(saveFileName).toFile();
   // the plugin instance should read any important state from the file.
   myPluginInstance.readStateFrom(f);
}
```

Processing resource deltas between activations

Recall that any number of resource change events could occur in the workspace before your plug−in is ever activated. If you want to know what changes have occurred since your plug−in was deactivated, you can use the save mechanism to do so, even if you don't need to save any other data.

The save participant must request that the platform keep a resource delta on its behalf. This is done as part of the save operation.

```
public void saving(ISaveContext context) throws CoreException {
   // no state to be saved by the plug-in, but request a
   // resource delta to be used on next activation.
   context.needDelta();
}
```

During plug−in startup, the previous saved state can be accessed and change events will be created for all changes that have occurred since the last save.

```
ISaveParticipant saveParticipant = new MyWorkspaceSaveParticipant();
ISavedState lastState =
   ResourcesPlugin.getWorkspace().addSaveParticipant(myPluginInstance, saveParticipant);
if (lastState != null) {
   lastState.processResourceChangeEvents(new MyResourceChangeReporter());
}
```

The provided class must implement **IResourceChangeListener**, as described in Tracking resource changes. The changes since the last save are reported as part of the **POST_AUTO_BUILD** resource change event.

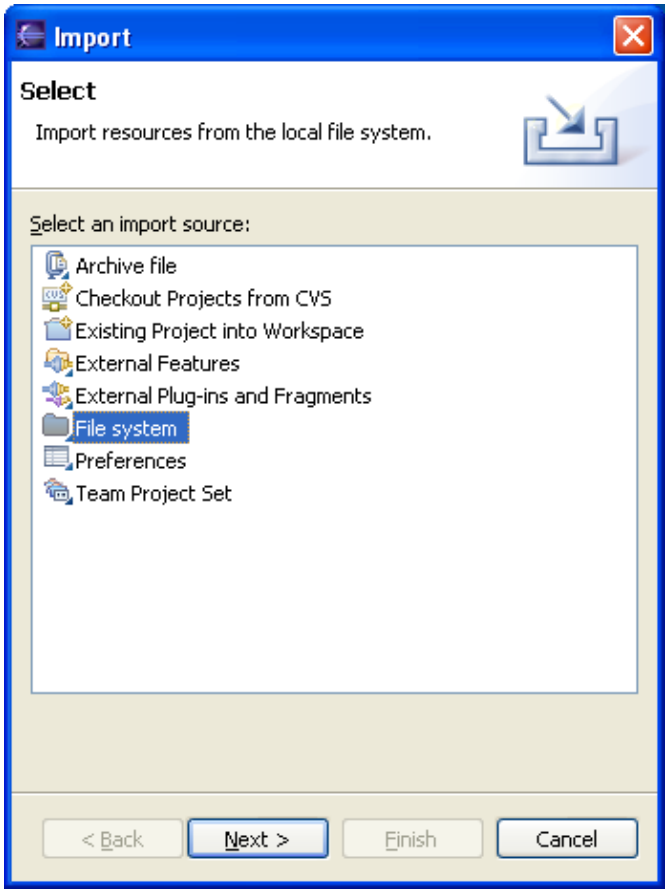
> *Note: Marker changes are not reported in the change events stored in an **ISavedState**. You must assume that any or all markers have changed since your last state was saved.*

Workbench wizard extension points

The workbench defines extension points for wizards that create new resources, import resources, or export resources.

When you make selections in the new, import, or export menu, the workbench uses various wizard selection dialogs to display all the wizards that have been contributed for that particular extension point. The import

wizard dialog is shown below.



Your wizard takes control once it is selected in the list and the **Next** button is pressed. This is when your first page becomes visible.

Platform Extension Points

The following extension points can be used to extend the capabilities of the platform infrastructure:

Platform runtime

- org.eclipse.core.runtime.adapters
- org.eclipse.core.runtime.applications
- org.eclipse.core.runtime.contentTypes
- org.eclipse.core.runtime.preferences
- org.eclipse.core.runtime.products

Workspace

- org.eclipse.core.resources.builders
- org.eclipse.core.resources.fileModificationValidator
- org.eclipse.core.resources.markers
- org.eclipse.core.resources.moveDeleteHook
- org.eclipse.core.resources.natures
- org.eclipse.core.resources.refreshProviders
- org.eclipse.core.resources.teamHook

Platform text

- org.eclipse.core.filebuffers.annotationModelCreation
- org.eclipse.core.filebuffers.documentCreation
- org.eclipse.core.filebuffers.documentSetup
- org.eclipse.ui.editors.annotationTypes
- org.eclipse.ui.editors.documentProviders
- org.eclipse.ui.editors.markerAnnotationSpecification
- org.eclipse.ui.editors.markerUpdaters
- org.eclipse.ui.editors.templates
- org.eclipse.ui.workbench.texteditor.quickDiffReferenceProvider
- org.eclipse.ui.workbench.texteditor.spellingEngine

Workbench

- org.eclipse.ui.acceleratorConfigurations
- org.eclipse.ui.acceleratorScopes
- org.eclipse.ui.acceleratorSets
- org.eclipse.ui.actionDefinitions
- org.eclipse.ui.actionSetPartAssociations
- org.eclipse.ui.actionSets
- org.eclipse.ui.activities
- org.eclipse.ui.activitySupport
- org.eclipse.ui.bindings
- org.eclipse.ui.browserSupport
- org.eclipse.ui.cheatsheets.cheatSheetContent
- org.eclipse.ui.cheatsheets.cheatSheetItemExtension

- org.eclipse.ui.commands
- org.eclipse.ui.contexts
- org.eclipse.ui.decorators
- org.eclipse.ui.dropActions
- org.eclipse.ui.editorActions
- org.eclipse.ui.editors
- org.eclipse.ui.elementFactories
- org.eclipse.ui.encodings
- org.eclipse.ui.exportWizards
- org.eclipse.ui.fontDefinitions
- org.eclipse.ui.handlers
- org.eclipse.ui.helpSupport
- org.eclipse.ui.ide.markerHelp
- org.eclipse.ui.ide.markerImageProviders
- org.eclipse.ui.ide.markerResolution
- org.eclipse.ui.ide.projectNatureImages
- org.eclipse.ui.ide.resourceFilters
- org.eclipse.ui.importWizards
- org.eclipse.ui.intro
- org.eclipse.ui.intro.config
- org.eclipse.ui.intro.configExtension
- org.eclipse.ui.keywords
- org.eclipse.ui.newWizards
- org.eclipse.ui.perspectiveExtensions
- org.eclipse.ui.perspectives
- org.eclipse.ui.popupMenus
- org.eclipse.ui.preferencePages
- org.eclipse.ui.preferenceTransfer
- org.eclipse.ui.presentationFactories
- org.eclipse.ui.propertyPages
- org.eclipse.ui.startup
- org.eclipse.ui.systemSummarySections
- org.eclipse.ui.themes
- org.eclipse.ui.viewActions
- org.eclipse.ui.views
- org.eclipse.ui.workingSets

Team

- org.eclipse.team.core.fileTypes
- org.eclipse.team.core.ignore
- org.eclipse.team.core.projectSets
- org.eclipse.team.core.repository
- org.eclipse.team.ui.configurationWizards
- org.eclipse.team.ui.synchronizeParticipants
- org.eclipse.team.ui.synchronizeWizards

Debug

- org.eclipse.debug.core.breakpoints
- org.eclipse.debug.core.launchConfigurationComparators
- org.eclipse.debug.core.launchConfigurationTypes
- org.eclipse.debug.core.launchDelegates
- org.eclipse.debug.core.launchers
- org.eclipse.debug.core.launchModes
- org.eclipse.debug.core.logicalStructureTypes
- org.eclipse.debug.core.processFactories
- org.eclipse.debug.core.sourceContainerTypes
- org.eclipse.debug.core.sourceLocators
- org.eclipse.debug.core.sourcePathComputers
- org.eclipse.debug.core.statusHandlers
- org.eclipse.debug.core.watchExpressionDelegates
- org.eclipse.debug.ui.breakpointOrganizers
- org.eclipse.debug.ui.consoleColorProviders
- org.eclipse.debug.ui.consoleLineTrackers
- org.eclipse.debug.ui.contextViewBindings
- org.eclipse.debug.ui.debugModelContextBindings
- org.eclipse.debug.ui.debugModelPresentations
- org.eclipse.debug.ui.launchConfigurationTabGroups
- org.eclipse.debug.ui.launchConfigurationTypeImages
- org.eclipse.debug.ui.launchGroups
- org.eclipse.debug.ui.launchShortcuts
- org.eclipse.debug.ui.memoryRenderings
- org.eclipse.debug.ui.sourceContainerPresentations
- org.eclipse.debug.ui.stringVariablePresentations
- org.eclipse.debug.ui.variableValueEditors

Console

- org.eclipse.ui.console.consoleFactories
- org.eclipse.ui.console.consolePageParticipants
- org.eclipse.ui.console.consolePatternMatchListeners

Help

- org.eclipse.help.contentProducer
- org.eclipse.help.contexts
- org.eclipse.help.toc
- org.eclipse.help.base.browser
- org.eclipse.help.base.luceneAnalyzer

Other

- org.eclipse.ant.core.antProperties
- org.eclipse.ant.core.antTasks
- org.eclipse.ant.core.antTypes
- org.eclipse.ant.core.extraClasspathEntries

- org.eclipse.compare.contentMergeViewers
- org.eclipse.compare.contentViewers
- org.eclipse.compare.streamMergers
- org.eclipse.compare.structureCreators
- org.eclipse.compare.structureMergeViewers
- org.eclipse.core.expressions.propertyTesters
- org.eclipse.core.variables.dynamicVariables
- org.eclipse.core.variables.valueVariables
- org.eclipse.ltk.core.refactoring.copyParticipants
- org.eclipse.ltk.core.refactoring.createParticipants
- org.eclipse.ltk.core.refactoring.deleteParticipants
- org.eclipse.ltk.core.refactoring.moveParticipants
- org.eclipse.ltk.core.refactoring.renameParticipants
- org.eclipse.ltk.ui.refactoring.changePreviewViewers
- org.eclipse.ltk.ui.refactoring.statusContextViewers
- org.eclipse.search.searchPages
- org.eclipse.search.searchResultSorters
- org.eclipse.search.searchResultViewPages
- org.eclipse.ui.externaltools.configurationDuplicationMaps
- org.eclipse.update.core.featureTypes
- org.eclipse.update.core.installHandlers
- org.eclipse.update.core.siteTypes

Adapters

Identifier:

org.eclipse.core.runtime.adapters

Since:

3.0

Description:

The adapters extension point allows plug−ins to declaratively register adapter factories. This information is used to by the runtime XML expression language to determine existence of adapters without causing plug−ins to be loaded. Registration of adapter factories via extension point eliminates the need to manually register adapter factories when a plug−in starts up.

Configuration Markup:

<!ELEMENT extension (factory+)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

<!ELEMENT factory (adapter+)>

<!ATTLIST factory

adaptableType CDATA #REQUIRED

class CDATA #REQUIRED>

- **adaptableType –** The fully qualified name of a class (typically implementing IAdaptable) that this factory provides adapters for.
- **class –** The fully qualified name of the adapter factory class. Must implement `org.eclipse.core.runtime.IAdapterFactory`.

<!ELEMENT adapter EMPTY>

<!ATTLIST adapter

type CDATA #REQUIRED>

- **type** – The fully qualified name of a Java class or interface that this factory can adapt to.

Examples:

Following is an example of an adapter declaration. This example declares that this plug−in will provide an adapter factory that will adapt objects of type IFile to objects of type MyFile.

<extension point=

"org.eclipse.core.runtime.adapters"

>

<factory class=

"com.xyz.MyFileAdapterFactory"

adaptableType=

"org.eclipse.core.resources.IFile"

>

<adapter type=

"com.xyz.MyFile"

/>

</factory>

</extension>

API Information:

Adapter factories registered using this extension point can be queried using the method `IAdapterManager.hasAdapter`, or retrieved using one of the `getAdapter` methods on `IAdapterFactory`. An adapter factory registered with this extension point does not need to be registered at runtime using `IAdapterFactory.registerAdapters`.

Supplied Implementation:

Several plug−ins in the platform provide adapters for a number of different IAdaptable objects.

Eclipse Public License v1.0 which accompanies this distribution, and is available at
http://www.eclipse.org/legal/epl−v10.html

Content Types

Identifier:

org.eclipse.core.runtime.contentTypes

Since:

3.0

Description:

The content types extension point allows plug−ins to contribute to the platform content type catalog. There are two forms of contributions: *content types* and *file associations*.

- a content type represents a file format and its naming conventions. Content types can be defined from scratch, or can inherit from existing ones, specializing them. Also, a content type can be made into an alias for another content type (see the `alias-for` attribute). When this feature is used:
  - if the target is absent, the alias content type is processed as a normal content type;
  - if the target is present, all references to the alias type are automatically transformed into references to the target type, and the alias type cannot be accessed nor is exposed through the API.
- a file association extends an existing content type by associating new file names and/or extensions to it

Configuration Markup:

<!ELEMENT extension (content−type* , file−association*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point −** a fully qualified identifier of the target extension point
- **id −** an optional identifier of the extension instance
- **name −** an optional name of the extension instance

<!ELEMENT content−type (describer? , property*)>

<!ATTLIST content−type

id CDATA #REQUIRED

base−type CDATA #IMPLIED

name CDATA #REQUIRED

file−extensions CDATA #IMPLIED

file−names CDATA #IMPLIED

priority (low|normal|high) "normal"

default−charset CDATA #IMPLIED

describer CDATA #IMPLIED

alias−for CDATA #IMPLIED>

- **id** − the identifier for this content type (simple id token, unique for content types within the extension namespace). The token cannot contain dot (.) or whitespace
- **base−type** − the fully qualified identifier of this content type's base type. This content type will inherit its base type's file associations, content describer and default charset, unless they are redefined
- **name** − the human−readable name of this content type
- **file−extensions** − a comma−separated list of file extensions to be associated with this content type
- **file−names** − a comma−separated list of file names to be associated with this content type
- **priority** − the priority for this content type. Priorities are used to solve conflicts (when two content types are associated to the same file name/extension)
- **default−charset** − the default charset for this content type, or an empty string, if this content type should not have a default charset even if the parent has one. This is a convenience attribute, equivalent to specifying:

  <content−type>

  <property name=

  "org.eclipse.core.runtime.charset"

  default=

  "charset−name"

  />

  </content−type>
- **describer** − the fully qualified name of a class that implements `org.eclipse.core.runtime.content.IContentDescriber` or `org.eclipse.core.runtime.content.ITextContentDescriber`, or an empty string, if this content type should not have a describer even if the parent has one
- **alias−for** − the fully qualified identifier of the content type this content type is an alias for

<!ELEMENT describer (parameter*)>

Since:

<!ATTLIST describer

class CDATA #REQUIRED

plugin CDATA #IMPLIED>

If the describer attribute is used in the content−type element, this element is ignored.

- **class** − the fully qualified name of a class that implements `org.eclipse.core.runtime.content.IContentDescriber` or `org.eclipse.core.runtime.content.ITextContentDescriber`, or an empty string, if this content type should not have a describer even if the parent has one
- **plugin** − the id for the plug−in providing the describer class

<!ELEMENT file−association EMPTY>

<!ATTLIST file−association

content−type   CDATA #REQUIRED

file−names   CDATA #IMPLIED

file−extensions CDATA #IMPLIED>

- **content−type** − the fully qualified identifier for the content type this file association contributes to
- **file−names** − a comma−separated list of file names to be associated with the target content type
- **file−extensions** − a comma−separated list of file extensions to be associated with the target content type

<!ELEMENT parameter EMPTY>

<!ATTLIST parameter

name CDATA #REQUIRED

value CDATA #REQUIRED>

- **name** − the name of this parameter made available to instances of the specified content describer class
- **value** − an arbitrary value associated with the given name and made available to instances of the specified content describer class

<!ELEMENT property EMPTY>

<!ATTLIST property

name CDATA #REQUIRED

default CDATA #IMPLIED>

Declares a property related to this content type, optionally assigning a default value. See `org.eclipse.core.runtime.content.IContentDescription` for more information on properties.

- **name** – the name of the property. If the property is being overriden and has been originally defined in a different namespace, a fully qualified property name must be used
- **default** – the default value of the property, or an empty string, if this content type should not have a default value for this property even if a parent has one

Examples:

Following is an example of a XML–based content type declaration using `org.eclipse.core.runtime.content.XMLRootElementContentDescriber`, a built–in describer:

<extension point=

"org.eclipse.core.runtime.contentTypes"

>

<content–type id=

"ABC"

base–type=

"org.eclipse.core.runtime.xml"

file–extensions=

"a,b,c"

>

<describer class=

"org.eclipse.core.runtime.content.XMLRootElementContentDescriber"

Since:

>

<parameter name=

"element"

value=

"abc"

/>

</describer>

</content−type>

</extension>

Here is an example of a simple text−based content type that has a specific file extension:

<extension point=

"org.eclipse.core.runtime.contentTypes"

>

<content−type id=

"MyText"

base−type=

"org.eclipse.core.runtime.text"

file−extensions=

"mytxt"

/>

</extension>

When there is need to associate new file names/extensions to an existing content type (as opposed to defining a new content type), a plug−in can contribute a file association as seen below. This has the effect of enhancing the definition of the text content type to include files with names following the "*.mytxt" pattern.

<extension point=

"org.eclipse.core.runtime.contentTypes"

>

<file−association content−type=

"org.eclipse.core.runtime.text"

file−extensions=

"mytxt"

/>

</extension>

Here is an example of a content type that defines properties:

<extension point=

"org.eclipse.core.runtime.contentTypes"

>

<content−type id=

"MyContentType"

file−extensions=

"dat"

>

<property name=

"file−format"

value=

"1"

/>

</content−type>

</extension>

API Information:

The value of the class attribute in the describer element must represent an implementor of `org.eclipse.core.runtime.content.IContentDescriber` or `org.eclipse.core.runtime.content.ITextContentDescriber`. `org.eclipse.core.runtime.content.IContentDescription` objects returned by the `org.eclipse.core.runtime.content` API

Supplied Implementation:

The org.eclipse.core.runtime plug−in provides the following content types:

- org.eclipse.core.runtime.text
- org.eclipse.core.runtime.xml

Other plug−ins in the platform contribute other content types.

Also, the org.eclipse.core.runtime plug−in provides ready−to−use implementations of content describers:

- org.eclipse.core.runtime.content.XMLRootElementContentDescriber
- org.eclipse.core.runtime.content.BinarySignatureDescriber

File Modification Validator

Identifier:

org.eclipse.core.resources.fileModificationValidator

Description:

For providing an implementation of an IFileModificationValidator to be used in the validate−edit and validate−save mechanism. This extension point tolerates at most one extension.

Configuration Markup:

<!ELEMENT extension (fileModificationValidator?)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT fileModificationValidator EMPTY>

<!ATTLIST fileModificationValidator

class CDATA #REQUIRED>

- **class –** a fully qualified name of a class which implements `org.eclipse.core.resources.IFileModificationValidator`.

Examples:

The following is an example of using the `fileModificationValidator` extension point:

<extension point=

"org.eclipse.core.resources.fileModificationValidator"

>

<fileModificationValidator class=

"org.eclipse.vcm.internal.VCMFileModificationValidator"

/>

</extension>

API Information:

The value of the `class` attribute must represent an implementation of `org.eclipse.core.resources.IFileModificationValidator`.

Supplied Implementation:

The Team component will generally provide the implementation of the file modification validator. The extension point should be used by any other clients.

Annotation Model Creation

Identifier:

org.eclipse.core.filebuffers.annotationModelCreation

Since:

3.0

Description:

This extension point is used to customize the annotation model creation behavior of this plug−in's default text file buffer manager. It allows to specify which annotation model factory should be used in order to create the annotation model instance of a text file buffer created for a certain file content type, file extension, or file name.

Configuration Markup:

<!ELEMENT extension (factory)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

<!ELEMENT factory EMPTY>

<!ATTLIST factory

class CDATA #REQUIRED

extensions CDATA #IMPLIED

fileNames CDATA #IMPLIED

contentTypeId CDATA #IMPLIED>

The specification of a annotation model factory. In order to find a factory for a given file the attributes of each factory specification are consulted in the following sequence: contentTypeId, fileNames, extensions. If multiple, equally specific factory specifications are found for a given file it is not specified which factory is used.

- **class** – The fully qualified name of the factory implementation class. This class must implement the org.eclipse.core.filebuffers.IAnnotationModelFactory interface.
- **extensions** – A comma separated list of file extensions for which this factory should be used.
- **fileNames** – A comma separated list of file names for which this factory should be used.
- **contentTypeId** – The id of a content type as defined by the org.eclipse.core.runtime.contentTypes extension point for which this factory should be used.

Examples:

<extension point=

"org.eclipse.core.filebuffers.annotationModelCreation"

>

<factory extensions=

"xzy"

class=

"org.eclipse.ui.texteditor.ResourceMarkerAnnotationModelFactory"

>

</factory>

</extension>

API Information:

Annotation model factories have to implement org.eclipse.core.filebuffers.IAnnotationModelFactory.

Document Creation

Identifier:

org.eclipse.core.filebuffers.documentCreation

Since:

3.0

Description:

This extension point is used to customize the document creation behavior of this plug−in's default text file buffer manager. It allows to specify which document factory should be used in order to create the document instance of a text file buffer created for a certain file content type, file extension, or file name.

Configuration Markup:

<!ELEMENT extension (factory)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

<!ELEMENT factory EMPTY>

<!ATTLIST factory

class CDATA #REQUIRED

extensions CDATA #IMPLIED

fileNames CDATA #IMPLIED

contentTypeId CDATA #IMPLIED>

The specification of a document factory. In order to find a factory for a given file the attributes of each factory specification are consulted in the following sequence: contentTypeId, fileNames, extensions. If multiple, equally specific factory specifications are found for a given file it is not specified which factory is used.

- **class** − The fully qualified name of the factory implementation class. This class must implement the org.eclipse.core.filebuffers.IDocumentFactory interface.
- **extensions** − A comma separated list of file extensions for which this factory should be used.

- **fileNames** – A comma separated list of file names for which this factory should be used.
- **contentTypeId** – The id of a content type as defined by the org.eclipse.core.runtime.contentTypes extension point for which this factory should be used.

Examples:

<extension id=

"org.eclipse.jdt.debug.ui.SnippetDocumentFactory"

name=

"%snippetDocumentFactory.name"

point=

"org.eclipse.core.filebuffers.documentCreation"

>

<factory extensions=

"jpage"

class=

"org.eclipse.jdt.internal.debug.ui.snippeteditor.SnippetDocumentFactory"

>

</factory>

</extension>

API Information:

Document factories have to implement org.eclipse.core.filebuffers.IDocumentFactory.

Document Setup

Identifier:

org.eclipse.core.filebuffers.documentSetup

Since:

3.0

Description:

This extension point is used to customize the initialization process of a document for a text file buffer manager by this plug−in's default text file buffer manager. It allows to specify which document setup participant should be involved in the initialization process for a text file buffer created for a certain file content type, file extension, or file name.

Configuration Markup:

<!ELEMENT extension (participant)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

<!ELEMENT participant EMPTY>

<!ATTLIST participant

class CDATA #REQUIRED

extensions CDATA #IMPLIED

fileNames CDATA #IMPLIED

contentTypeId CDATA #IMPLIED>

The specification of a document setup participant. In order find all participants for a given file the attributes of each participant specification are consulted in the following sequence: contentTypeId, fileNames, extensions. If multiple participants are found, the sequence in which they are called is not specified.

- **class –** The fully qualified name of the participant implementation class. This class must implement the org.eclipse.core.filebuffers.IDocumentSetupParticipant interface.

- **extensions** – A comma separated list of file extensions for which this participant should be used.
- **fileNames** – A comma separated list of file names for which this participant should be used.
- **contentTypeId** – The id of a content type as defined by the org.eclipse.core.runtime.contentTypes extension point for which this participant should be used.

Examples:

<extension id=

"JavaDocumentSetupParticipant"

name=

"%javaDocumentSetupParticipant"

point=

"org.eclipse.core.filebuffers.documentSetup"

>

<participant extensions=

"java"

class=

"org.eclipse.jdt.internal.ui.javaeditor.JavaDocumentSetupParticipant"

>

</participant>

</extension>

API Information:

Document setup participants have to implement org.eclipse.core.filebuffers.IDocumentSetupParticipant.

Annotation Types

Identifier:

org.eclipse.ui.editors.annotationTypes

Since:

3.0

Description:

An Annotation is a piece of information attached to a certain region of a text document. New kinds of annotations may be defined using this extension point. Annotations are attached to documents via their annotation model and may be displayed in text editors and views. Annotation types form a hierarchy: an annotation type may refine another type by specifying it in its `super` attribute. Some annotations serve as the UI counterpart of markers (see `org.eclipse.core.resources.IMarker`), while others exist on their own without having a persistable form. The mapping between markers and annotation types is defined by the optional `markerType` and `markerSeverity` attributes.

Configuration Markup:

<!ELEMENT extension (type)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

<!ELEMENT type EMPTY>

<!ATTLIST type

name CDATA #REQUIRED

markerType CDATA #IMPLIED

super CDATA #IMPLIED

markerSeverity (0|1|2) >

A marker type definition.

- **name –** The unique name of this annotation type.

- **markerType** – The marker type that this annotation type corresponds to, if any.
- **super** – The name of the parent type, if this type is a descendant of another annotation type.
- **markerSeverity** – The optional severity of this annotation type, used for mapping an annotation type to a marker. Any out of IMarker.SEVERITY_INFO, IMarker.SEVERITY_WARNING, SEVERITY_ERROR.

Examples:

This is an excerpt from the plugin.xml for JDT UI, which adds the java compiler error and warning annotations:

```
<extension point=

"org.eclipse.ui.editors.annotationTypes"

>

<type name=

"org.eclipse.jdt.ui.error"

super=

"org.eclipse.ui.workbench.texteditor.error"

markerType=

"org.eclipse.jdt.core.problem"

markerSeverity=

"2"

>

</type>

<type name=

"org.eclipse.jdt.ui.warning"

super=

"org.eclipse.ui.workbench.texteditor.warning"

markerType=
```

"org.eclipse.jdt.core.problem"

markerSeverity=

"1"

>

</type>

<type name=

"org.eclipse.jdt.ui.info"

super=

"org.eclipse.ui.workbench.texteditor.info"

markerType=

"org.eclipse.jdt.core.problem"

markerSeverity=

"0"

>

</type>

</extension>

API Information:

See the org.eclipse.jface.text.source.Annotation class and the org.eclipse.ui.editors.markerAnnotationSpecification extension point.

Document Providers

Identifier:

org.eclipse.ui.editors.documentProviders

Since:

3.0 (originally named org.eclipse.ui.documentProviders)

Description:

This extension point is used to define mappings between file types and document providers or between types of editor inputs and document providers that can be used by editors. Document providers must implement the interface `org.eclipse.ui.texteditor.IDocumentProvider`. Editor inputs must be instance of `org.eclipse.ui.IEditorInput`.

Configuration Markup:

<!ELEMENT extension (provider*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT provider EMPTY>

<!ATTLIST provider

extensions CDATA #IMPLIED

inputTypes CDATA #IMPLIED

class CDATA #REQUIRED

id CDATA #REQUIRED>

- **extensions –** a comma separated list of file extensions
- **inputTypes –** a comma separated list of qualified editor input class names
- **class –** the qualified name of the document provider class
- **id –** the unique id of this provider

Examples:

<extension point=

"org.eclipse.ui.editors.documentProviders"

>

<provider extensions=

".jav"

class=

"org.eclipse.ui.examples.javaeditor.JavaDocumentProvider"

id=

"org.eclipse.ui.examples.javaeditor.JavaDocumentProvider"

>

</provider>

</extension>

This example registers `org.eclipse.ui.examples.javaeditor.JavaDocumentProvider` as the default provider for files with the extension ".jav".

<extension point=

"org.eclipse.ui.editors.documentProviders"

>

<provider inputTypes=

"org.eclipse.ui.IStorageEditorInput"

class=

"org.eclipse.ui.editors.text.FileDocumentProvider"

id=

"org.eclipse.ui.editors.text.FileDocumentProvider"

Since:

>

</provider>

</extension>

This example registers `org.eclipse.ui.editors.text.FileDocumentProvider` as the default provider for all editor inputs that are `instance of org.eclipse.ui.IStorageEditorInput`.

API Information:

Document providers registered for a file extension have precedence over those registered for input types. Document providers must implement the interface `org.eclipse.ui.texteditor.IDocumentProvider`. Editor inputs must be instance of `org.eclipse.ui.IEditorInput`.

Marker Annotation Specification

Identifier:

org.eclipse.ui.editors.markerAnnotationSpecification

Since:

3.0 (originally named org.eclipse.ui.workbench.texteditor.markerAnnotationSpecification)

Description:

This extension point is used to define presentation properties of markers. Extensions provided for this extension point can be accessed using `org.eclipse.ui.texteditor.MarkerAnnotationPreferences`. Use `org.eclipse.ui.texteditor.AnnotationPreferenceLookup` to get the annotation preference for a given annotation.

Note that an extension will only be returned from `MarkerAnnotationPreferences.getAnnotationPreferences` (and thus included in the preference pages) if it contains the following four attributes in addition to the required `annotationType`:

- `colorPreferenceKey`
- `colorPreferenceValue`
- `overviewRulerPreferenceKey`
- `textPreferenceKey`

Annotation preference types that extend another annotation preference are allowed to overwrite attributes already defined in a parent preference specification, but these will not be accessible from the preference page.

Configuration Markup:

<!ELEMENT extension (specification)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

<!ELEMENT specification EMPTY>

<!ATTLIST specification

annotationType CDATA #REQUIRED

colorPreferenceKey CDATA #IMPLIED

overviewRulerPreferenceKey CDATA #IMPLIED

verticalRulerPreferenceKey CDATA #IMPLIED

textPreferenceKey CDATA #IMPLIED

label CDATA #IMPLIED

highlightPreferenceKey CDATA #IMPLIED

colorPreferenceValue CDATA #IMPLIED

presentationLayer CDATA #IMPLIED

overviewRulerPreferenceValue (true | false)

verticalRulerPreferenceValue (true | false)

textPreferenceValue (true | false)

highlightPreferenceValue (true | false)

contributesToHeader (true | false)

showInNextPrevDropdownToolbarActionKey CDATA #IMPLIED

showInNextPrevDropdownToolbarAction (true | false)

isGoToNextNavigationTargetKey CDATA #IMPLIED

isGoToNextNavigationTarget (true | false)

isGoToPreviousNavigationTargetKey CDATA #IMPLIED

isGoToPreviousNavigationTarget (true | false)

icon CDATA #IMPLIED

symbolicIcon (error|warning|info|task|bookmark)

annotationImageProvider CDATA #IMPLIED

textStylePreferenceKey CDATA #IMPLIED

textStylePreferenceValue (SQUIGGLES|BOX|UNDERLINE|IBEAM|NONE)

includeOnPreferencePage (true | false) "true">

- **annotationType –** The annotation type.
- **colorPreferenceKey –** The color preference key must be provided, otherwise this annotation type will not be included in the List returned from

MarkerAnnotationPreferences.getAnnotationPreferences() and thus not show in the preferences.

- **overviewRulerPreferenceKey –** The overview ruler preference key must be provided, otherwise this annotation type will not be included in the List returned from MarkerAnnotationPreferences.getAnnotationPreferences() and thus not show in the preferences.
- **verticalRulerPreferenceKey –** The preference key for the show in vertical ruler preference. since: 3.0
- **textPreferenceKey –** The text preference key must be provided, otherwise this annotation type will not be included in the List returned from MarkerAnnotationPreferences.getAnnotationPreferences() and thus not show in the preferences.
- **label –** The label to be used in the UI.
- **highlightPreferenceKey –** The preference key for highlighting in text. since: 3.0
- **colorPreferenceValue –** The color preference value must be provided, otherwise this annotation type will not be included in the List returned from MarkerAnnotationPreferences.getAnnotationPreferences() and thus not show in the preferences.
- **presentationLayer –** The default value for the layer in which the marker annotation will be drawn.
- **overviewRulerPreferenceValue –** The default value telling whether this marker annotation is shown in the overview ruler.
- **verticalRulerPreferenceValue –** The default value for showing in vertical ruler. since: 3.0
- **textPreferenceValue –** The default value telling whether this marker annotation is shown in the text.
- **highlightPreferenceValue –** The default value for highlighting in text. since: 3.0
- **contributesToHeader –** The default value telling whether this marker annotation contributes to the overview ruler's header summary.
- **showInNextPrevDropdownToolbarActionKey –** The preference key for the visibility in the next/previous drop down toolbar action. since: 3.0
- **showInNextPrevDropdownToolbarAction –** The default value for the visibility in the next/previous drop down toolbar action. since: 3.0
- **isGoToNextNavigationTargetKey –** The preference key for go to next navigation enablement. since: 3.0
- **isGoToNextNavigationTarget –** The default value for go to next navigation enablement. since: 3.0
- **isGoToPreviousNavigationTargetKey –** The preference key for go to previous navigation enablement. since: 3.0
- **isGoToPreviousNavigationTarget –** The default value for go to previous navigation enablement. since: 3.0
- **icon –** The path to the icon to be drawn for annotations of this annotation type.
- **symbolicIcon –** The symbolic name of the image that should be drawn to represent annotation of this annotation type. The image is only used when there is no vertical ruler icon specified for this annotation type. Possible values are: "error", "warning", "info", "task", "bookmark".
- **annotationImageProvider –** The optional annotation image provider. Must implement
- **textStylePreferenceKey –** The preference key for the text decoration property. since: 3.0
- **textStylePreferenceValue –** The default value for the "show in text" decoration style. since: 3.0
- **includeOnPreferencePage –** Defines whether this annotation type should be configurable via the standard annotation preference page. Default is true.

Marker Updaters

Identifier:

org.eclipse.ui.editors.markerUpdaters

Since:

3.0 (originally named org.eclipse.ui.markerUpdaters)

Description:

This extension point is used for registering marker update strategies with marker annotation models. A resource that is opened in a text editor is associated with a marker annotation model. For each marker attached to the resource this model manages a position that is updated with each change applied to the text in the editor. If the resource is saved, the text in the editor and the position managed for a marker are passed over to the registered marker update strategies. These strategies can then update the marker's attributes based on the text and the position. Marker update strategies are requested to implement the interface `org.eclipse.ui.texteditor.IMarkerUpdater`. The update strategies can be registered either for a particular marker type or all marker types. The latter by omitting any marker type in the extension.

Configuration Markup:

<!ELEMENT extension (updater*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point** – a fully qualified identifier of the target extension point
- **id** – an optional identifier of the extension instance
- **name** – an optional name of the extension instance

<!ELEMENT updater EMPTY>

<!ATTLIST updater

id CDATA #REQUIRED

markerType CDATA #IMPLIED

class CDATA #REQUIRED>

- **id** – the unique id of this provider
- **markerType** – the name of the marker type

- **class** – the qualified name of the marker updater class

Examples:

<extension point=

"org.eclipse.ui.editors.markerUpdaters"

>

<updater id=

"org.eclipse.jdt.ui.markerUpdaters.JavaSearchMarkerUpdater"

class=

"org.eclipse.jdt.internal.ui.search.JavaSearchMarkerUpdater"

markerType=

"org.eclipse.search.searchmarker"

>

</updater>

</extension>

This example registers `org.eclipse.jdt.internal.ui.search.JavaSearchMarkerUpdater` as a marker updater for all markers of the type `org.eclipse.search.searchmarker` including all its derived types.

<extension point=

"org.eclipse.ui.editors.markerUpdaters"

>

<updater id=

"org.eclipse.ui.texteditor.BasicMarkerUpdater"

class=

"org.eclipse.ui.texteditor.BasicMarkerUpdater"

>

```
</updater>

</extension>
```

This example registers `org.eclipse.ui.texteditor.BasicMarkerUpdater` as a marker updater independent from the type of the marker.

API Information:

Registered marker updaters have to implement the interface `org.eclipse.ui.texteditor.IMarkerUpdater`.

Editor Template

Identifier:

org.eclipse.ui.editors.templates

Since:

3.0

Description:

Templates are snippets of text or code which help the user enter reoccurring patterns into a text editor. Templates may contain variables which are resolved in the context where the template is inserted.

Configuration Markup:

<!ELEMENT extension (template* , resolver* , contextType* , include*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

<!ELEMENT contextType EMPTY>

<!ATTLIST contextType

id CDATA #REQUIRED

class CDATA #REQUIRED

name CDATA #IMPLIED>

A context type defines a context within which templates are evaluated. A context type uses its resolvers to resolve a template.

- **id –** Unambiguously identifies this context type. Use of a qualified name is recommended.
- **class –** A subclass of `org.eclipse.jface.text.templates.TemplateContextType`.
- **name –** The display name of this context.

<!ELEMENT resolver EMPTY>

<!ATTLIST resolver

contextTypeId CDATA #REQUIRED

type CDATA #REQUIRED

class CDATA #REQUIRED

description CDATA #IMPLIED

name CDATA #IMPLIED

icon CDATA #IMPLIED>

A template variable resolver can resolve a template variable in a certain context.

- **contextTypeId –** References the context type that this resolver is contributed to.
- **type –** The type of this variable resolver. This property will be set on the resolver once it gets created.
- **class –** A subclass of `org.eclipse.jface.text.templates.TemplateVariableResolver`.
- **description –** The description of this variable resolver. This property will be set on the resolver once it gets created.
- **name –** The display name of this resolver.
- **icon –** An icon that may be displayed in the user interface.

<!ELEMENT template (pattern)>

<!ATTLIST template

id CDATA #REQUIRED

contextTypeId CDATA #REQUIRED

name CDATA #REQUIRED

description CDATA #IMPLIED

icon CDATA #IMPLIED

autoinsert (true | false) "true">

A template is a snippet of code or text that will be evaluated in a given context. Variables which will be resolved in that context can be specified using the `${variable_type}` notation.

- **id –** Unambiguously identifies this template. Use of a qualified name is recommended.

- **contextTypeId** – References the context type that this template is contributed to.
- **name** – The internationalizable name of the template which will show up in the UI, such as in template proposals.
- **description** – The description of this template.
- **icon** – An icon that may be displayed in the UI for this template, for example in content assist proposals.
- **autoinsert** – `true` (default) to make the template automatically insertable, `false` to not allow automatic insertion. Since 3.1.

<!ELEMENT pattern (#PCDATA)>

The template pattern.

<!ELEMENT include EMPTY>

<!ATTLIST include

file CDATA #REQUIRED

translations CDATA #IMPLIED>

A collection of templates encoded as XML can be included as a whole via this element.

- **file** – The XML file to import templates from.
- **translations** – An optional properties file with resources for the templates specified in `file`.

Examples:

<extension point=

"org.eclipse.ui.editors.templates"

>

<template name=

"%ant.tasks.javac.name"

contextTypeId=

```
"org.eclipse.ui.examples.templateeditor.antcontext"

id=

"org.eclipse.ui.examples.templateeditor.templates.javac"

description=

"%ant.tasks.javac.description"

>

<pattern>

<javac srcdir=

"${src}"

destdir=

"${dst}"

classpath=

"${classpath}"

debug=

"${debug}"

/>

</pattern>

</template>

<resolver contextTypeId=

"org.eclipse.ui.examples.templateeditor.antcontext"

type=

"src"

class=

"org.eclipse.ui.examples.templateeditor.editors.AntVariableResolver"

>

</resolver>
```

<resolver contextTypeId=

"org.eclipse.ui.examples.templateeditor.antcontext"

type=

"dst"

class=

"org.eclipse.ui.examples.templateeditor.editors.AntVariableResolver"

>

</resolver>

</extension>

API Information:

See the `org.eclipse.jface.text.templates` package in the `org.eclipse.text` plug−in for the relevant API.

Supplied Implementation:

See the `org.eclipse.jface.text.templates` package in the `org.eclipse.text` plug−in for the relevant classes.

Reference Provider

Identifier:

org.eclipse.ui.workbench.texteditor.quickdiffReferenceProvider

Since:

3.0

Description:

Allows contributors to add reference providers for the quick diff display.

Configuration Markup:

<!ELEMENT extension (referenceprovider+)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** The fully qualified name of the extension point.
- **id –** The optional id of this extension.
- **name –** The optional name of this extension.

<!ELEMENT referenceprovider EMPTY>

<!ATTLIST referenceprovider

class CDATA #REQUIRED

label CDATA #IMPLIED

id CDATA #REQUIRED

default (true | false) >

The definition of a reference provider for the quick diff display.

- **class –** The class of the reference provider, which must implement `org.eclipse.ui.texteditor.quickdiff.IQuickDiffReferenceProvider`.

- **label** – The display label for the provider, which will show up in the menu that allows the user to set the quick diff reference to this provider.
- **id** – A string uniquely identifying this reference provider.
- **default** – If this flag is set to `true`, this reference provider will be installed per default the first time quick diff is enabled for a document. If multiple providers are installed with the flag set are encountered, the first one is taken.

Examples:

The following is an example of a reference provider definition. It contributes a provider that uses the version of a document saved on disk as a reference.

<extension point=

"quickdiff.referenceprovider"

>

<referenceprovider id=

"default"

name=

"%LastSavedProvider.name"

label=

"%quickdiff.referenceprovider.label"

class=

"org.eclipse.ui.internal.editors.quickdiff.providers.LastSaveReferenceProvider"

>

</referenceprovider>

</extension>

API Information:

There is no additional API for managing reference providers.

Supplied Implementation:

The `org.eclipse.ui.editors` plugin contributes `LastSaveReferenceProvider`. See its implementation as an example.

Since:

Spelling Engine

Identifier:

org.eclipse.ui.workbench.texteditor.spellingEngine

Since:

3.1

Description:

Allows contributors to add spelling engines.

Configuration Markup:

<!ELEMENT extension (engine+)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** The fully qualified name of the extension point.
- **id –** The optional id of this extension.
- **name –** The optional name of this extension.

<!ELEMENT engine EMPTY>

<!ATTLIST engine

class CDATA #REQUIRED

label CDATA #IMPLIED

id CDATA #REQUIRED

default (true | false)

preferencesClass CDATA #IMPLIED>

The definition of a spelling engine.

- **class –** The class of the engine, which must implement `org.eclipse.ui.texteditor.spelling.ISpellingEngine`.
- **label –** The display label for the engine, which will show up in the menu that allows the user to choose the spelling engine.
- **id –** A string uniquely identifying this engine.
- **default –** If this flag is set to `true`, this spelling engine will be installed per default. If multiple engines are installed with the flag set are encountered, the first one is taken.
- **preferencesClass –** An implementation of org.eclipse.ui.texteditor.spelling.ISpellingPreferenceBlock

Examples:

The following is an example of a spelling engine definition.

<extension point=

"org.eclipse.ui.workbench.texteditor.spellingEngine"

>

<engine default=

"true"

label=

"%defaultSpellingEngine.label"

class=

"org.eclipse.jdt.internal.ui.text.spelling.DefaultSpellingEngine"

id=

"org.eclipse.jdt.internal.ui.text.spelling.DefaultSpellingEngine"

>

</engine>

</extension>

Supplied Implementation:

The `org.eclipse.jdt.ui` plugin contributes `DefaultSpellingEngine`. See its implementation as an example.

Since:

Accelerator Configurations

Identifier:

org.eclipse.ui.acceleratorConfigurations

Since:

Release 2.0

Description:

WARNING: This extension point is DEPRECATED.
Do not use this extension point, it will be removed in future versions of this product. Instead, use the extension point org.eclipse.ui.commands

This extension point is used to register accelerator configuration extensions. Accelerator configurations are configurations to which accelerator sets may be registered. The workbench allows the user to select an accelerator configuration from the Workbench preference page. Only one accelerator configuration may be active at a time.

An accelerator configuration represents a general style or theme of accelerators for Workbench actions. For example, the Workbench provides the "Emacs" accelerator configuration. When the "Emacs" accelerator configuration is active, accelerators belonging to accelerator sets registered to the "Emacs" configuration are active. These accelerators are defined to mimic the accelerators in Emacs (a popular text editor amongst developers).

An accelerator set registers with an accelerator configuration by listing the configuration's id as the value of its "configurationId" attribute (see the Accelerator Sets extension point). Many accelerator sets can be registered to the same accelerator configuration.

Note the accelerator configuration name presented to the user is the same as the value of the attribute "name" of the extension element of org.eclipse.ui.acceleratorConfigurations extension point.

Configuration Markup:

<!ELEMENT extension (acceleratorConfiguration*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT acceleratorConfiguration EMPTY>

<!ATTLIST acceleratorConfiguration

id CDATA #REQUIRED

name CDATA #REQUIRED

description CDATA #REQUIRED>

- **id** – a unique name that can be used to identify this accelerator configuration.
- **name** – a translatable name of the accelerator configuration to be presented to the user.
- **description** – a short description of the accelerator configuration.

Examples:

Following is an example of an accelerator configuration extension:

<extension point=

"org.eclipse.ui.acceleratorConfigurations"

>

<acceleratorConfiguration id=

"org.eclipse.ui.viAcceleratorConfiguration"

name=

"VI"

description=

"VI style accelerator configuration"

>

</acceleratorConfiguration>

<acceleratorConfiguration id=

"org.eclipse.ui.jonDoeAcceleratorConfiguration"

name=

"Jon Doe"

Since:

description=

"Personal accelerator configuration for Jon Doe"

>

</acceleratorConfiguration>

</extension>

Supplied Implementation:

The workbench provides the Default and Emacs accelerator configurations.

Commands

Identifier:

org.eclipse.ui.commands

Since:

2.1

Description:

The `org.eclipse.ui.commands` extension point is used to declare commands and command categories, using the `command` and `category` elements. A command is an abstract representation of some semantic behaviour, but not it's actual implementation. This allows different developers to contribute specific behaviour for their individual parts. For example, there might be a "paste" command with one implementation in an editor and a different implementation in an explorer widget. These implementations are called handlers.

Configuration Markup:

<!ELEMENT extension (activeKeyConfiguration , category , command , keyBinding , keyConfiguration , context , scope)>

<!ATTLIST extension

id CDATA #IMPLIED

name CDATA #IMPLIED

point CDATA #REQUIRED>

- **id –** An optional identifier of the extension instance.
- **name –** An optional name of the extension instance.
- **point –** A fully qualified identifier of the target extension point.

<!ELEMENT activeKeyConfiguration EMPTY>

<!ATTLIST activeKeyConfiguration

value CDATA #IMPLIED

keyConfigurationId CDATA #IMPLIED>

This element is used to define the initial active key configuration for Eclipse. If more than one of these elements exist, only the last declared element (in order of reading the plugin registry) is considered valid.

This element has been replaced with a preference. If your application needs to change the default key configuration, then specify the following in your `plugin_customization.ini` file: `org.eclipse.ui/KEY_CONFIGURATION_ID=your.default.key.configuration.id`.

- **value –** The unique id (`id`attribute) of the keyConfiguration element one wishes to be initially active.
- **keyConfigurationId –** The unique id (`id`attribute) of the keyConfiguration element one wishes to be initially active.

<!ELEMENT category EMPTY>

<!ATTLIST category

description CDATA #IMPLIED

id CDATA #REQUIRED

name CDATA #REQUIRED>

In the UI, commands are often organized by category to make them more manageable. This element is used to define these categories. Commands can add themselves to at most one category. If more than one of these elements exist with the same `id` attribute, only the last declared element (in order of reading the plugin registry) is considered valid.

- **description –** A translatable short description of this category for display in the UI.
- **id –** The unique identifier of this category.
- **name –** The translatable name of this category for display in the UI.

<!ELEMENT command (commandParameter | defaultHandler?)>

<!ATTLIST command

category CDATA #IMPLIED

description CDATA #IMPLIED

id CDATA #REQUIRED

name CDATA #REQUIRED

categoryId CDATA #IMPLIED

defaultHandler CDATA #IMPLIED>

This element is used to define commands. A command represents an request from the user that can be handled by an action, and should be semantically unique among other commands. Do not define a command if there is already one defined with the same meaning. If more than one of these elements exist with the same `id` attribute, only the last declared element (in order of reading the plugin registry) is considered valid. See the

extension points org.eclipse.ui.actionSets and org.eclipse.ui.editorActions to understand how actions are connected to commands.

- **category –** The unique id of the category for this command. If this command does not specify a category, it will still appear in all UI, alongside other specifically categorized commands. @deprecated Please use "categoryId" instead.
- **description –** A translatable short description of this command for display in the UI.
- **id –** The unique identifier of this command.
- **name –** The translatable name of this command for display in the UI. Command are typically named in the form of an imperative verb.
- **categoryId –** The unique id of the category for this command. If this command does not specify a category, it will still appear in all UI, alongside other specifically categorized commands.
- **defaultHandler –** The default handler for this command (see the org.eclipse.ui.bindings extension point). If no other handler is active, this handler will be active. This handler will conflict with other handler definitions that specify no `activeWhen` conditions. If you are creating an `IExecutableExtension`, you can use the `defaultHandler` element instead.

<!ELEMENT keyBinding EMPTY>

<!ATTLIST keyBinding

configuration CDATA #IMPLIED

command CDATA #IMPLIED

locale CDATA #IMPLIED

platform CDATA #IMPLIED

contextId CDATA #IMPLIED

string CDATA #IMPLIED

scope CDATA #IMPLIED

keyConfigurationId CDATA #IMPLIED

commandId CDATA #IMPLIED

keySequence CDATA #IMPLIED>

This element allows one to assign key sequences to commands. Please use the `key` element in the "org.eclipse.ui.bindings" extension point instead.

- **configuration** – The unique id of the key configuration of this key binding. @deprecated Please use keyConfigurationId instead.
- **command** – The unique identifier of the command to which the key sequence specified by this key binding is assigned. If the value of this attribute is an empty string, the key sequence is assigned to an internal 'no operation' command. This is useful for 'undefining' key bindings in specific key configurations and contexts which may have been borrowed from their parents. @deprecated Please use "commandId" instead.
- **locale** – An optional attribute indicating that this key binding is only defined for the specified locale. Locales are specified according to the format declared in `java.util.Locale`.
- **platform** – An optional attribute indicating that this key binding is only defined for the specified platform. The possible values of the `platform` attribute are the set of the possible values returned by `org.eclipse.swt.SWT.getPlatform()`.
- **contextId** – The unique id of the context of this key binding.
- **string** – The key sequence to assign to the command. Key sequences consist of one or more key strokes, where a key stroke consists of a key on the keyboard, optionally pressed in combination with one or more of the following modifiers: Ctrl, Alt, Shift, and Command. Key strokes are separated by spaces, and modifiers are separated by '+' characters. @deprecated Please use "keySequence" instead.
- **scope** – The unique id of the context of this key binding. @deprecated Please use "contextId" instead. The old default scope, "org.eclipse.ui.globalScope", has been changed to "org.eclipse.ui.contexts.window". The old name is still supported, but it is deprecated.
- **keyConfigurationId** – The unique id of the key configuration of this key binding. @deprecated Please use the `schemeId` attribute on the `key` element in the new "org.eclipse.ui.bindings" extension point.
- **commandId** – The unique identifier of the command to which the key sequence specified by this key binding is assigned. If the value of this attribute is an empty string, the key sequence is assigned to an internal 'no operation' command. This is useful for 'undefining' key bindings in specific key configurations and contexts which may have been borrowed from their parents.
- **keySequence** –

  The key sequence to assign to the command. Key sequences consist of one or more key strokes, where a key stroke consists of a key on the keyboard, optionally pressed in combination with one or more of the following modifiers: Ctrl, Alt, Shift, and Command. Key strokes are separated by spaces, and modifiers are separated by '+' characters.

  The modifier keys can also be expressed in a platform−independent way. On MacOS X, for example, "Command" is almost always used in place of "Ctrl". So, we provide "M1" which will map to either "Ctrl" or "Command", as appropriate. Similarly, "M2" is "Shift"; "M3" is "Alt"; and "M4" is "Ctrl" (MacOS X). If more platforms are added, then you can count on these aliases being mapped to good platform defaults.

  The syntax for this string is defined in `org.eclipse.ui.internal.keys`. Briefly, the string is case insensitive −− though all capitals is preferred stylistically. If the key is a letter, then simply append the letter. If the key is a special key (i.e., non−ASCII), then use one of the following: ARROW_DOWN, ARROW_LEFT, ARROW_RIGHT, ARROW_UP, BREAK, CAPS_LOCK, END, F1, F2, F3, F4, F5, F6, F7, F8, F9, F10, F11, F12, F13, F14, F15, HOME, INSERT, NUM_LOCK, NUMPAD_0, NUMPAD_1, NUMPAD_2, NUMPAD_3, NUMPAD_4, NUMPAD_5, NUMPAD_6, NUMPAD_7, NUMPAD_8, NUMPAD_9, NUMPAD_ADD, NUMPAD_DECIMAL, NUMPAD_DIVIDE, NUMPAD_ENTER, NUMPAD_EQUAL, NUMPAD_MULTIPLY, NUMPAD_SUBTRACT, PAGE_UP, PAGE_DOWN, PAUSE, PRINT_SCREEN, or SCROLL_LOCK. If the key is a non−printable ASCII key, then use one of the following: BS, CR, DEL, ESC, FF, LF, NUL, SPACE, TAB, or VT. Note that the main keyboard enter/return key is CR.

Since:

<!ELEMENT keyConfiguration EMPTY>

<!ATTLIST keyConfiguration

description CDATA #IMPLIED

id CDATA #REQUIRED

name CDATA #REQUIRED

parent CDATA #IMPLIED

parentId CDATA #IMPLIED>

This element is used to define key configurations. If more than one of these elements exist with the same `id` attribute, only the last declared element (in order of reading the plugin registry) is considered valid. Please use the "org.eclipse.ui.bindings" extension point instead.

- **description –** A translatable short description of this key configuration for display in the UI.
- **id –** The unique identifier of this key configuration.
- **name –** The translatable name of this key configuration for display in the UI. If this key configuration has a parent, it is not necessary to add "(extends ...)" to the name. This will be automatically added by the UI where necessary.
- **parent –** The unique id of the parent key configuration. If this key configuration has a parent, it will borrow all key bindings from its parent, in addition to the key bindings defined in its own key configuration. @deprecated Please use parentId instead.
- **parentId –** The unique id of the parent key configuration. If this key configuration has a parent, it will borrow all key bindings from its parent, in addition to the key bindings defined in its own key configuration.

<!ELEMENT context EMPTY>

<!ATTLIST context

description CDATA #IMPLIED

id CDATA #REQUIRED

name CDATA #REQUIRED

parent CDATA #IMPLIED

parentId CDATA #IMPLIED>

Since:

This element is used to define contexts. If more than one of these elements exist with the same `id` attribute, only the last declared element (in order of reading the plugin registry) is considered valid. Please use the org.eclipse.ui.contexts extension point instead.

- **description** – A translatable short description of this context for display in the UI.
- **id** – The unique identifier of this context.
- **name** – The translatable name of this context for display in the UI. If this context has a parent, it is not necessary to add "(extends parent)" to the name. This will be automatically added by the UI where necessary.
- **parent** – The unique id of the parent context. If this context has a parent, it will borrow all key bindings from its parent, in addition to the key bindings defined in its own context. @deprecated Please use "parentId" instead.
- **parentId** – The unique id of the parent context. If this context has a parent, it will borrow all key bindings from its parent, in addition to the key bindings defined in its own context.

<!ELEMENT scope EMPTY>

<!ATTLIST scope

description CDATA #IMPLIED

id CDATA #REQUIRED

name CDATA #REQUIRED

parent CDATA #IMPLIED>

This element is used to define scopes. If more than one of these elements exist with the same `id` attribute, only the last declared element (in order of reading the plugin registry) is considered valid. @deprecated Please use the "org.eclipse.ui.contexts" extension point instead.

- **description** – A translatable short description of this scope for display in the UI. @deprecated Please use the "org.eclipse.ui.contexts" extension point instead.
- **id** – The unique identifier of this scope. @deprecated Please use the "org.eclipse.ui.contexts" extension point instead.
- **name** – The translatable name of this scope for display in the UI. If this scope has a parent, it is not necessary to add "(extends parent)" to the name. This will be automatically added by the UI where necessary. @deprecated Please use the "org.eclipse.ui.contexts" extension point instead.
- **parent** – The unique id of the parent scope. If this scope has a parent, it will borrow all key bindings from its parent, in addition to the key bindings defined in its own scope. @deprecated Please use the "org.eclipse.ui.contexts" extension point instead.

<!ELEMENT commandParameter (values)>

<!ATTLIST commandParameter

id CDATA #REQUIRED

name CDATA #REQUIRED

values CDATA #IMPLIED

optional (true | false) "true">

Defines a parameter that a command should understand. A parameter is a way to provide more information to a handler at execution time. For example, a "show view" command might take a view as a parameter. Handlers should be able to understand these parameters, so they should be treated like API.

- **id** – The unique identifier for this parameter.
- **name** – The name for the parameter. This is the name as it will be displayed to an end–user. As such, it should be translatable. The name should be short –– preferrably one word.
- **values** – The class providing a list of parameter values for the user to select. This class should implement `org.eclipse.core.commands.IParameterValues`. If this class is not specified, you must specify the more verbose `values` element. Please see `org.eclipse.core.runtime.IExecutableExtension`.
- **optional** – Whether this parameter is optional. If a parameter is optional, the handler should be able to handle the absence of the parameter. By default, all parameters are optional.

<!ELEMENT values (parameter)>

<!ATTLIST values

class CDATA #REQUIRED>

The more verbose version of the `values` attribute on the `commandParameter`.

- **class** – The class providing a list of parameter values for the user to select. This class should implement `org.eclipse.core.commands.IParameterValues`. If this class is not specified, you must specify the more verbose `values` element. Please see `org.eclipse.core.runtime.IExecutableExtension`.

<!ELEMENT parameter EMPTY>

<!ATTLIST parameter

name CDATA #REQUIRED

Since:

value CDATA #REQUIRED>

A possible value for a parameter.

- **name –** The name of the parameter to pass to the `IExecutableExtension`.
- **value –** The value of the parameter to pass to the `IExecutableExtension`.

<!ELEMENT defaultHandler (parameter)>

<!ATTLIST defaultHandler

class CDATA #REQUIRED>

The default handler for this command. If no other handler is active, this handler will be active. This handler will conflict with other handler definitions that specify no `activeWhen` conditions. If you are not creating an `IExecutableExtension`, you can use the `defaultHandler` attribute instead.

- **class –** The class which implements `org.eclipse.core.commands.IHandler`.

Examples:

The `plugin.xml` file in the `org.eclipse.ui` plugin makes extensive use of the `org.eclipse.ui.commands` extension point.

API Information:

This is no public API for declaring commands, categories, key bindings, key configurations, or contexts other than this extension point. Public API for querying and setting contexts, as well as registering actions to handle specific commands can be found in `org.eclipse.ui.IKeyBindingService`.

Action Sets

Identifier:

org.eclipse.ui.actionSets

Description:

This extension point is used to add menus, menu items and toolbar buttons to the common areas in the Workbench window. These contributions are collectively known as an *action set* and appear within the Workbench window by the user customizing a perspective.

An action's enablement and/or visibility can be defined using the elements `enablement` and `visibility` respectively. These two elements contain a boolean expression that is evaluated to determine the enablement and/or visibility.

The syntax is the same for the `enablement` and `visibility` elements. Both contain only one boolean expression sub−element. In the simplest case, this will be an `objectClass`, `objectState`, `pluginState`, or `systemProperty` element. In the more complex case, the `and`, `or`, and `not` elements can be combined to form a boolean expression. Both the `and`, and `or` elements must contain 2 sub−elements. The `not` element must contain only 1 sub−element.

Configuration Markup:

<!ELEMENT extension (actionSet+)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point −** a fully qualified identifier of the target extension point
- **id −** an optional identifier of the extension instance
- **name −** an optional name of the extension instance

<!ELEMENT actionSet (menu* , action*)>

<!ATTLIST actionSet

id CDATA #REQUIRED

label CDATA #REQUIRED

visible (true | false)

description CDATA #IMPLIED>

This element is used to define a group of actions and/or menus.

- **id –** a unique identifier for this action set.
- **label –** a translatable name used by the Workbench to represent this action set to the user.
- **visible –** an optional attribute indicating whether the action set is initially visible when a perspective is open. This option is only honoured when the user opens a perspective which has not been customized. The user can override this option from the "Customize Perspective Dialog". This attribute should be used with great care so as not to overwhelm the user with too many actions.
- **description –** a translatable description used by the Workbench to represent this action set to the user.

<!ELEMENT action (selection* | enablement?)>

<!ATTLIST action

id CDATA #REQUIRED

label CDATA #REQUIRED

accelerator CDATA #IMPLIED

definitionId CDATA #IMPLIED

menubarPath CDATA #IMPLIED

toolbarPath CDATA #IMPLIED

icon CDATA #IMPLIED

disabledIcon CDATA #IMPLIED

hoverIcon CDATA #IMPLIED

tooltip CDATA #IMPLIED

helpContextId CDATA #IMPLIED

style (push|radio|toggle|pulldown) "push"

state (true | false)

pulldown (true | false)

class CDATA #IMPLIED

retarget (true | false)

allowLabelUpdate (true | false)

Description:

enablesFor CDATA #IMPLIED>

This element defines an action that the user can invoke in the UI.

- **id –** a unique identifier used as a reference for this action.
- **label –** a translatable name used either as the menu item text or toolbar button label. The name can include mnenomic information.
- **accelerator – Deprecated:** Use the `definitionId` attribute instead.
- **definitionId –** Specifies the command that this action will handle. By specifying and action, the key binding service can assign a key sequence to this action. See the extension point org.eclipse.ui.commands for more information.
- **menubarPath –** a slash–delimited path ('/') used to specify the location of this action in the menu bar. Each token in the path, except the last one, must represent a valid identifier of an existing menu in the hierarchy. The last token represents the named group into which this action will be added. If the path is omitted, this action will not appear in the menu bar.
- **toolbarPath –** a slash–delimited path ('/') that is used to specify the location of this action in the toolbar. The first token represents the toolbar identifier (with "Normal" being the default toolbar), while the second token is the named group within the toolbar that this action will be added to. If the group does not exist in the toolbar, it will be created. If toolbarPath is omitted, the action will not appear in the toolbar.
- **icon –** a relative path of an icon used to visually represent the action in its context. If omitted and the action appears in the toolbar, the Workbench will use a placeholder icon. The path is relative to the location of the plugin.xml file of the contributing plug–in. The icon will appear in toolbars but not in menus. Enabled actions will be represented in menus by the hoverIcon.
- **disabledIcon –** a relative path of an icon used to visually represent the action in its context when the action is disabled. If omitted, the normal icon will simply appear greyed out. The path is relative to the location of the plugin.xml file of the contributing plug–in. The disabled icon will appear in toolbars but not in menus. Icons for disabled actions in menus will be supplied by the OS.
- **hoverIcon –** a relative path of an icon used to visually represent the action in its context when the mouse pointer is over the action. If omitted, the normal icon will be used. The path is relative to the location of the plugin.xml file of the contributing plug–in.
- **tooltip –** a translatable text representing the action's tool tip. Only used if the action appears in the toolbar.
- **helpContextId –** a unique identifier indicating the help context for this action. If the action appears as a menu item, then pressing F1 while the menu item is highlighted will display help.
- **style –** an attribute to define the user interface style type for the action. If omitted, then it is `push` by default. The attribute value will be one of the following:
  - **push** – as a regular menu item or tool item.
  - **radio** – as a radio style menu item or tool item. Actions with the radio style within the same menu or toolbar group behave as a radio set. The initial value is specified by the `state` attribute.
  - **toggle** – as a checked style menu item or as a toggle tool item. The initial value is specified by the `state` attribute.
  - **pulldown** – as a cascading style menu item or as a drop down menu beside the tool item.
- **state –** an optional attribute indicating the initial state (either `true` or `false`). Used only when the `style` attribute has the value `radio` or `toggle`.
- **pulldown – Deprecated:** Use the `style` attribute with the value `pulldown`.

- **class –** a fully qualified name of a class which implements `org.eclipse.ui.IWorkbenchWindowActionDelegate` or `org.eclipse.ui.IWorkbenchWindowPulldownDelegate`. The latter should be implemented in cases where the `style` attribute has the value `pulldown`. This class is the handler responsible for performing the action. If the `retarget` attribute is true, this attribute is ignored and should not be supplied.
- **retarget –** an optional attribute to retarget this action. When true, view and editor parts may supply a handler for this action using the standard mechanism for setting a global action handler on their site using this action's identifier. If this attribute is true, the `class` attribute should not be supplied.
- **allowLabelUpdate –** optional attribute indicating whether the retarget action allows the handler to override it's label and tooltip. Only applies if `retarget` attribute is true.
- **enablesFor –** a value indicating the selection count which must be met to enable the action. If specified and the condition is not met, the action is disabled. If omitted, the action enablement state is not affected. The following attribute formats are supported:

!	– 0 items selected
?	– 0 or 1 items selected
+	– 1 or more items selected
multiple, 2+	– 2 or more items selected
n	– a precise number of items selected.a precise number of items selected. For example: enablesFor=" 4" enables the action only when 4 items are selected
*	– any number of items selected

<!ELEMENT menu (separator+ , groupMarker*)>

<!ATTLIST menu

id CDATA #REQUIRED

label CDATA #REQUIRED

path CDATA #IMPLIED>

This element is used to defined a new menu.

- **id –** a unique identifier that can be used to reference this menu.
- **label –** a translatable name used by the Workbench for this new menu. The name should include mnemonic information.
- **path –** the location of the new menu starting from the root of the menu. Each token in the path must refer to an existing menu, except the last token which should represent a named group in the last menu in the path. If omitted, the new menu will be added to the `additions` named group of the menu.

<!ELEMENT separator EMPTY>

<!ATTLIST separator

name CDATA #REQUIRED>

This element is used to create a menu separator in the new menu.

- **name –** the name of the menu separator. This name can later be referenced as the last token in a menu path. Therefore, a separator also serves as named group into which actions and menus can be added.

<!ELEMENT groupMarker EMPTY>

<!ATTLIST groupMarker

name CDATA #REQUIRED>

This element is used to create a named group in the new menu. It has no visual representation in the new menu, unlike the `separator` element.

- **name –** the name of the group marker. This name can later be referenced as the last token in the menu path. It serves as named group into which actions and menus can be added.

<!ELEMENT selection EMPTY>

<!ATTLIST selection

class CDATA #REQUIRED

name CDATA #IMPLIED>

This element is used to help determine the action enablement based on the current selection. Ignored if the `enablement` element is specified.

- **class –** a fully qualified name of the class or interface that each object in the selection must implement in order to enable the action.
- **name –** an optional wild card filter for the name that can be applied to all objects in the selection. If specified and the match fails, the action will be disabled.

<!ELEMENT enablement (and | or | not | objectClass | objectState | pluginState | systemProperty)>

This element is used to define the enablement for the extension.

<!ELEMENT visibility (and | or | not | objectClass | objectState | pluginState | systemProperty)>

This element is used to define the visibility for the extension.

<!ELEMENT and (and | or | not | objectClass | objectState | pluginState | systemProperty)>

This element represent a boolean AND operation on the result of evaluating its two sub−element expressions.

<!ELEMENT or (and | or | not | objectClass | objectState | pluginState | systemProperty)>

This element represent a boolean OR operation on the result of evaluating its two sub−element expressions.

<!ELEMENT not (and | or | not | objectClass | objectState | pluginState | systemProperty)>

This element represent a boolean NOT operation on the result of evaluating its sub−element expressions.

<!ELEMENT objectClass EMPTY>

<!ATTLIST objectClass

name CDATA #REQUIRED>

This element is used to evaluate the class or interface of each object in the current selection. If each object in the selection implements the specified class or interface, the expression is evaluated as true.

- **name** − a fully qualified name of a class or interface. The expression is evaluated as true only if all objects within the selection implement this class or interface.

<!ELEMENT objectState EMPTY>

<!ATTLIST objectState

name CDATA #REQUIRED

value CDATA #REQUIRED>

This element is used to evaluate the attribute state of each object in the current selection. If each object in the selection has the specified attribute state, the expression is evaluated as true. To evaluate this type of expression, each object in the selection must implement, or adapt to, `org.eclipse.ui.IActionFilter` interface.

- **name** – the name of an object's attribute. Acceptable names reflect the object type, and should be publicly declared by the plug–in where the object type is declared.
- **value** – the required value of the object's attribute. The acceptable values for the object's attribute should be publicly declared.

<!ELEMENT pluginState EMPTY>

<!ATTLIST pluginState

id   CDATA #REQUIRED

value (installed|activated) "installed">

This element is used to evaluate the state of a plug–in. The state of the plug–in may be one of the following: `installed` (equivalent to the OSGi concept of "resolved") or `activated` (equivalent to the OSGi concept of "active").

- **id** – the identifier of a plug–in which may or may not exist in the plug–in registry.
- **value** – the required state of the plug–in. The state of the plug–in may be one of the following: `installed` (equivalent to the OSGi concept of "resolved") or `activated` (equivalent to the OSGi concept of "active").

<!ELEMENT systemProperty EMPTY>

<!ATTLIST systemProperty

name CDATA #REQUIRED

value CDATA #REQUIRED>

This element is used to evaluate the state of some system property. The property value is retrieved from the `java.lang.System`.

- **name** – the name of the system property.
- **value** – the required value of the system property.

Examples:

The following is an example of an action set (note the sub−elements and the way attributes are used):

<extension point =

"org.eclipse.ui.actionSets"

>

<actionSet id=

"com.xyz.actionSet"

label=

"My Actions"

>

<menu id=

"com.xyz.xyzMenu"

label=

"XYZ Menu"

path=

"additions"

>

<separator name=

"group1"

/>

<separator name=

"option1"

/>

</menu>

<action id=

"com.xyz.runXYZ"

label=

"&Run XYZ Tool"

style=

"toggle"

state=

"false"

menubarPath=

"com.xyz.xyzMenu/group1"

icon=

"icons/runXYZ.gif"

tooltip=

"Run XYZ Tool"

helpContextId=

"com.xyz.run_action_context"

class=

"com.xyz.actions.RunXYZ"

enablesFor=

"1"

>

<selection class=

"org.eclipse.core.resources.IFile"

name=

"*.java"

/>

</action>

<action id=

"com.xyz.runABC"

label=

"&Run ABC Tool"

style=

"push"

menubarPath=

"com.xyz.xyzMenu/group1"

toolbarPath=

"Normal/XYZ"

icon=

"icons/runABC.gif"

tooltip=

"Run ABC Tool"

helpContextId=

"com.xyz.run_abc_action_context"

retarget=

"true"

allowLabelUpdate=

"true"

>

Description:

<enablement>

<and>

<objectClass name=

"org.eclipse.core.resources.IFile"

/>

<not>

<objectState name=

"extension"

value=

"java"

/>

</not>

</and>

</enablement>

</action>

<action id=

"com.xyz.runDEF"

label=

"&Run DEF Tool"

style=

"radio"

state=

"true"

menubarPath=

"com.xyz.xyzMenu/option1"

icon=

"icons/runDEF.gif"

tooltip=

"Run DEF Tool"

class=

"com.xyz.actions.RunDEF"

helpContextId=

"com.xyz.run_def_action_context"

>

</action>

<action id=

"com.xyz.runGHI"

label=

"&Run GHI Tool"

style=

"radio"

state=

"false"

menubarPath=

"com.xyz.xyzMenu/option1"

icon=

"icons/runGHI.gif"

tooltip=

"Run GHI Tool"

class=

"com.xyz.actions.RunGHI"

helpContextId=

"com.xyz.run_ghi_action_context"

>

</action>

<action id=

"com.xyz.runJKL"

label=

"&Run JKL Tool"

style=

"radio"

state=

"false"

menubarPath=

"com.xyz.xyzMenu/option1"

icon=

"icons/runJKL.gif"

tooltip=

"Run JKL Tool"

class=

"com.xyz.actions.RunJKL"

helpContextId=

"com.xyz.run_jkl_action_context"

>

</action>

</actionSet>

</extension>

In the example above, the specified action set, named "My Actions", is not initially visible within each perspective because the `visible` attribute is not specified.

The XYZ action will appear as a check box menu item, initially not checked. It is enabled only if the selection count is 1 and if the selection contains a Java file resource.

The ABC action will appear both in the menu and on the toolbar. It is enabled only if the selection does not contain any Java file resources. Note also this is a label retarget action therefore it does not supply a `class` attribute.

The actions DEF, GHI, and JKL appear as radio button menu items. They are enabled all the time, independent of the current selection state.

API Information:

The value of the class attribute must be the fully qualified name of a class that implements `org.eclipse.ui.IWorkbenchWindowActionDelegate` or `org.eclipse.ui.IWorkbenchWindowPulldownDelegate`. The latter should be implemented in cases where the `style` attribute has the value `pulldown`. This class is the handler responsible for performing the action. If the `retarget` attribute is true, this attribute is ignored and should not be supplied. This class is loaded as late as possible to avoid loading the entire plug−in before it is really needed.

The enablement criteria for an action extension is initially defined by `enablesFor`, and also either `selection` or `enablement`. However, once the action delegate has been instantiated, it may control the action enable state directly within its `selectionChanged` method.

It is important to note that the workbench does not generate menus on a plug−in's behalf. Menu paths must reference menus that already exist.

Action and menu labels may contain special characters that encode mnemonics using the following rules:

1. Mnemonics are specified using the ampersand ('&') character in front of a selected character in the translated text. Since ampersand is not allowed in XML strings, use `&` character entity.

If two or more actions are contributed to a menu or toolbar by a single extension the actions will appear in the reverse order of how they are listed in the plugin.xml file. This behavior is admittedly unintuitive. However, it was discovered after the Eclipse Platform API was frozen. Changing the behavior now would break every plug−in which relies upon the existing behavior.

The `selection` and `enablement` elements are mutually exclusive. The `enablement` element can replace the `selection` element using the sub−elements `objectClass` and `objectState`. For example, the following:

<selection class=

"org.eclipse.core.resources.IFile"

name=

"*.java"

>

</selection>

can be expressed using:

<enablement>

<and>

<objectClass name=

"org.eclipse.core.resources.IFile"

/>

<objectState name=

"extension"

value=

"java"

/>

</and>

</enablement>

Supplied Implementation:

Plug−ins may use this extension point to add new top level menus. Plug−ins can also define named groups which allow other plug−ins to contribute their actions into them.

Top level menus are created by using the following values for the path attribute:

- additions − represents a group immediately to the left of the `Window` menu.

Omitting the path attribute will result in adding the new menu into the additions menu bar group.

The default groups in a workbench window are defined in the `IWorkbenchActionConstants` interface. These constants can be used in code for dynamic contribution. The values can also be copied into an XML file for fine grained integration with the existing workbench menus and toolbar.

Various menu and toolbar items within the workbench window are defined algorithmically. In these cases a separate mechanism must be used to extend the window. For example, adding a new workbench view results

in a new menu item appearing in the Perspective menu. Import, Export, and New Wizards extensions are also added automatically to the window.

Bindings

Identifier:

org.eclipse.ui.bindings

Since:

3.1

Description:

The `org.eclipse.ui.bindings` extension point is used to declare bindings and schemes. Schemes are sets of one or more bindings. A binding is a mapping between a certain group of conditions, some user input and a triggered command.

All bindings require a trigger of some kind, a context in which they are active and scheme in which they exist. If you're not sure which context to chose, then just leave it blank. It will default to `"org.eclipse.ui.contexts.window"` context. This context means that the binding will apply in any Eclipse main window. When the context becomes active, the binding will become active as well. Bindings from child contexts will override bindings from parent contexts. For more information about contexts, please see the `org.eclipse.ui.contexts` extension point.

If a binding does not define a command identifier, then it is a deletion marker. This means that if all the conditions are met, it will cancel any bindings with the same trigger in the same context. This mechanism can be used, for example, to change a binding on a particular platform.

One type of binding is a `key` binding (i.e., a keyboard shortcut). For example, binding `Ctrl+C` to `Copy` is considered a `key` binding. The trigger for a key binding is a sequence of key strokes.

A scheme is a group of these bindings into a set that the end user can select. For example, a user might want to use the default scheme, but they might also want an Emacs−style scheme or a Brief−style scheme.

Configuration Markup:

<!ELEMENT extension (scheme , key)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

<!ELEMENT scheme EMPTY>

<!ATTLIST scheme

id CDATA #REQUIRED

name CDATA #REQUIRED

description CDATA #IMPLIED

parentId CDATA #IMPLIED>

A scheme is a grouping of bindings that an end user can chose to use.

It is possible for schemes to inherit bindings from a parent scheme. This is intended to make it easier for plug−in developers to create customized binding sets. An active binding defined in a child scheme will always override an active binding in a parent scheme, if they have the same trigger. This technique is used to provide the Emacs scheme in the workbench.

- **id −** The unique identifier for this scheme.
- **name −** The name for this scheme, as it should be displayed to an end−user. This value should be translated.
- **description −** The description for this scheme, as it would be displayed to an end user. This value should be translated.
- **parentId −** The identifier for the parent of this scheme. If there is no parent, then do not specify this attribute.

<!ELEMENT key (parameter)>

<!ATTLIST key

sequence CDATA #REQUIRED

schemeId CDATA #REQUIRED

contextId CDATA "org.eclipse.ui.contexts.window"

commandId CDATA #IMPLIED

platform CDATA #IMPLIED

locale CDATA #IMPLIED>

A binding between some keyboard input and the triggering of a command.

- **sequence −**

  The key sequence for this binding. This key sequence should consist of one or more key strokes. Key strokes are separated by spaces. Key strokes consist of one or more keys held down at the same time.

This should be zero or more modifier keys, and one other key. The keys are separated by the + character.

The recognized modifiers keys are `M1`, `M2`, `M3`, `M4`, `ALT`, `COMMAND`, `CTRL`, and `SHIFT`. The "M" modifier keys are a platform−independent way of representing keys, and these are generally preferred. `M1` is the `COMMAND` key on MacOS X, and the `CTRL` key on most other platforms. `M2` is the `SHIFT` key. `M3` is the `Option` key on MacOS X, and the `ALT` key on most other platforms. `M4` is the `CTRL` key on MacOS X, and is undefined on other platforms.

The actual key is generally specified simply as the ASCII character, in uppercase. So, for example `F` or `,` are examples of such keys. However, there are some special keys; keys that have no printable ASCII representation. The following is a list of the current special keys: `ARROW_DOWN`, `ARROW_LEFT`, `ARROW_RIGHT`, `ARROW_UP`, `BREAK`, `BS`, `CAPS_LOCK`, `CR`, `DEL`, `END`, `ESC`, `F1`, `F2`, `F3`, `F4`, `F5`, `F6`, `F7`, `F8`, `F9`, `F10`, `F11`, `F12`, `F13`, `F14`, `F15`, `FF`, `HOME`, `INSERT`, `LF`, `NUL`, `NUM_LOCK`, `NUMPAD_0`, `NUMPAD_1`, `NUMPAD_2`, `NUMPAD_3`, `NUMPAD_4`, `NUMPAD_5`, `NUMPAD_6`, `NUMPAD_7`, `NUMPAD_8`, `NUMPAD_9`, `NUMPAD_ADD`, `NUMPAD_DECIMAL`, `NUMPAD_DIVIDE`, `NUMPAD_ENTER`, `NUMPAD_EQUAL`, `NUMPAD_MULTIPLY`, `NUMPAD_SUBTRACT`, `PAGE_UP`, `PAGE_DOWN`, `PAUSE`, `PRINT_SCREEN`, `SCROLL_LOCK`, `SPACE`, `TAB` and `VT`.

We also understand some alternative names for some common special keys. For example, we accept both `ESC` and `ESCAPE`, and `CR`, `ENTER` and `RETURN` are all the same.

It is also strongly recommended that you keep the key sequences short. One or two is the most you should need. Use contexts to give key sequences different meanings in different parts of your application. At the very most, you should not use any key sequence that contains more than four key strokes.

- **schemeId –** The identifier of the scheme in which this key binding is active.
- **contextId –** The identifier of the context in which this key binding is active. Please see the `org.eclipse.ui.contexts` extension point. If this is not specified, then it defaults to `org.eclipse.ui.contexts.window`.
- **commandId –**

  The identifier of the command which should be executed when this binding is triggered.

  If no command identifier is specified, this is a deletion marker. This means that any binding in the same context with the same sequence, platform and locale will become inactive when this binding becomes active. If the platform or locale on a deletion is not specified, then it matches any platform or locale.
- **platform –** The platform on which this binding applies. The platform should be specified in the same way as the string from `SWT.getPlatform()`. For example, the following strings are considered valid: `win32`, `gtk`, `motif`, `carbon` and `photon`.
- **locale –** The locale on which this bindings applies. This is useful for changing bindings that conflict with locale−specific input method editors (IMEs). The locale is specified in the same way as `Locale.toString()`. For example, `"en"` or `"en_CA"` are both understood.

<!ELEMENT parameter EMPTY>

<!ATTLIST parameter

Since:

id CDATA #IMPLIED

value CDATA #IMPLIED>

A parameter name and value that should be passed to the command when it is executed. This allows for the command to be qualified in some way. For example, a "Show View" command might accept the view id as a parameter.

- **id** – The parameter name as understood by the command. This is not a translatable name, but the key to name–value map.
- **value** – The value for the parameter. This value is a free–form string, but it should be parseable by the command. Consult the command to see what format it expects these values to take.

Examples:

<extension point=

"org.eclipse.ui.bindings"

>

<key sequence=

"M2+F5"

commandId=

"commandId"

schemeId=

"default"

contextId=

"windows"

/>

<scheme name=

"Default"

description=

"Default shortcuts for Eclipse"

id=

"default"

/>

</extension>

API Information:

There is no public API for defining bindings. To try to achieve stability for the user, bindings are only defined through the extension points. If you are an RCP application, you should be able to override this behaviour in the `WorkbenchAdvisor`.

For bindings to work, you must have defined a command. For the binding to work, the command must have an active handler. Handlers can be registered programmatically; please see the org.eclipse.ui.handlers extension point.

Contexts

Identifier:

org.eclipse.ui.contexts

Since:

3.0

Description:

The `org.eclipse.ui.contexts` extension point is used to declare contexts and associated elements.

Configuration Markup:

<!ELEMENT extension (context)>

<!ATTLIST extension

id CDATA #IMPLIED

name CDATA #IMPLIED

point CDATA #REQUIRED>

- **id –** An optional identifier of the extension instance.
- **name –** An optional name of the extension instance.
- **point –** A fully qualified identifier of the target extension point.

<!ELEMENT context EMPTY>

<!ATTLIST context

description CDATA #IMPLIED

id CDATA #REQUIRED

name CDATA #REQUIRED

parentId CDATA #IMPLIED>

This element is used to define contexts. If more than one of these elements exist with the same `id` attribute, only the last declared element (in order of reading the registry) is considered valid.

- **description –** A translatable short description of this context for display in the UI.
- **id –** The unique identifier of this context.

- **name** – The translatable name of this context for display in the UI.
- **parentId** – The unique identifier of the parent of this context.

Examples:

The `plugin.xml` file in the `org.eclipse.ui` plugin makes use of the `org.eclipse.ui.contexts` extension point.

API Information:

There is currently no public API for declaring contexts or associated elements other than this extension point.

Accelerator Scopes

Identifier:

org.eclipse.ui.acceleratorScopes

Since:

Release 2.0

Description:

WARNING: This extension point is DEPRECATED.
Do not use this extension point, it will be removed in future versions of this product. Instead, use the extension point org.eclipse.ui.commands

This extension point is used to register accelerator scope extensions. Accelerator scopes are scopes for which accelerator sets may be applicable. For example, if an accelerator set is applicable for the scope entitled "Text Editor Scope", the accelerators of that accelerator set will only operate if the "Text Editor Scope" or one of its children is active (in other words, if the active part is a participating text editor).

An accelerator set declares what scope it is applicable for by listing the scope's id as the value of its "scopeId" attribute (see the Accelerator Sets extension point). Many accelerator sets can be applicable for the same accelerator scope.

Configuration Markup:

<!ELEMENT extension (acceleratorScope*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT acceleratorScope EMPTY>

<!ATTLIST acceleratorScope

id CDATA #REQUIRED

name CDATA #REQUIRED

description CDATA #REQUIRED

parentScope CDATA #IMPLIED>

- **id** – a unique name that can be used to identify this accelerator scope.
- **name** – a translatable name of the accelerator scope.
- **description** – a short description of the accelerator scope.
- **parentScope** – an optional attribute which represents a scope which is active whenever this scope is active. For most scopes, org.eclipse.ui.globalScope will be the parent scope

Examples:

Following is an example of an accelerator scope extension:

<extension point=

"org.eclipse.ui.acceleratorScopes"

>

<acceleratorScope id=

"org.eclipse.ui.globalScope"

name=

"Global"

description=

"Action accelerator key applicable to all views and editors unless explicitly overridden."

>

</acceleratorScope>

<acceleratorScope id=

"org.eclipse.ui.javaEditorScope"

name=

"Java Editor"

description=

"Action accelerator key applicable only when java editor active."

parentScope=

"org.eclipse.ui.globalScope"

>

</acceleratorScope>

</extension>

API Information:

The method public IKeyBindingService getKeyBindingService() was added to IEditorSite.

Supplied Implementation:

The workbench provides the Global accelerator scope and the Text Editor accelerator scope.

Accelerator Sets

Identifier:

org.eclipse.ui.acceleratorSets

Since:

Release 2.0

Description:

WARNING: This extension point is DEPRECATED.
Do not use this extension point, it will be removed in future versions of this product. Instead, use the extension point org.eclipse.ui.commands

This extension point is used to register accelerator set extensions. Accelerator sets are just what the name implies, sets of accelerators. An accelerator is an association between one or more sequences of accelerator keys and a workbench action. An accelerator key sequence may be of length one or greater.

An accelerator set is registered with an accelerator configuration (see the Accelerator Configuration extension point) and is applicable for an accelerator scope (see the Accelerator Scope extension point).

Configuration Markup:

<!ELEMENT extension (acceleratorSet*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT acceleratorSet (accelerator+)>

<!ATTLIST acceleratorSet

configurationId CDATA #REQUIRED

scopeId CDATA #REQUIRED>

@deprecated Please use the "org.eclipse.ui.bindings" extension point instead. Use the `key` element.

- **configurationId** – a unique name that identifies the accelerator configuration to which this accelerator set is registered.
- **scopeId** – a unique name that identifies the accelerator scope for which this accelerator set is applicable. @deprecated When migrating to the new bindings extension point, it is recommended that you change any occurrences of "org.eclipse.ui.globalScope" to "org.eclipse.ui.contexts.window".

<!ELEMENT accelerator EMPTY>

<!ATTLIST accelerator

id CDATA #IMPLIED

key CDATA #REQUIRED

locale CDATA #IMPLIED

platform CDATA #IMPLIED>

@deprecated Please use the "org.eclipse.ui.bindings" extension point instead. Use the `key` element.

- **id** – the unique identifier of the action definition of the action associated with this accelerator. If the id is not specified this accelerator deletes any mappings with the same key. This is used to delete a key binding for a specific Locale.
- **key** – an attribute representing the sequence(s) of accelerator keys used to perform the action associated with this accelerator. Sequences are separated by '||', and individual keys in a sequence are separated by a space. A key may be modified by the CTRL, ALT, or SHIFT keys. Depending on keyboard layout, some keys ('?' for example) may need the SHIFT to be accessed but the accelerator should be specified without the SHIFT so it will be independent of keyboard layout. E.g. if CTRL+? is specified as an accelerator, the user may have to press CTRL+SHIFT+? depending on the keyboard layout.
- **locale** – an optional attribute which specifies a locale for which the accelerator is applicable. If this attribute is not specified, the accelerator is applicable for all locales.
- **platform** – an optional attribute which specifies a platform on which the accelerator is applicable. If this attribute is not specified, the accelerator is applicable on all platforms.

Examples:

Following is an example of an accelerator set extension:

<extension point=

"org.eclipse.ui.acceleratorSets"

>

<acceleratorSet configurationId=

"org.eclipse.ui.exampleAcceleratorConfiguration"

scopeId=

"org.eclipse.ui.globalScope"

>

<accelerator id=

"org.eclipse.ui.ExampleActionA"

key=

"CTRL+R CTRL+A"

>

</accelerator>

<accelerator id=

"org.eclipse.ui.ExampleActionB"

key=

"CTRL+R CTRL+B"

>

</accelerator>

<accelerator id=

"org.eclipse.ui.ExampleActionC"

key=

"CTRL+R CTRL+C || CTRL+SHIFT+DELETE"

>

</accelerator>

</acceleratorSet>

</extension>

API Information:

More than one accelerator may be specified for the same action in the accelerator set but only one will be used.

If the locale and/or the platform is specified, the accelerator that better matches the current locale and platform will be used. The current locale is determined by the API Locale.getDefault() and the platform by the API SWT.getPlatform(). If the platform and/or the locale is specified and it does not match the current locale and/or platform, the accelerator is discarded. If accelerator A defines only the locale and B defines only the platform, B is used. If accelerator A defines "ja" as its locale and B defines "ja_JP", B is used in case the current locale is "ja_JP".

If two accelerators are defined in accelerators sets in different plugins, the chosen accelerator will depend on the plugins. If plugin A depends on B, the accelerators defined in B is used. If A and B don't depend on each other, they will be alphabetically sorted by the plugin id.

If two accelerators are defined in different scopes, the accelerator defined in the current scope will be used. If an accelerator is not defined in the current scope or one of its parents it is discarded. If an accelerator is defined in a parent and child scope, the one in the child is used.

Supplied Implementation:

The workbench provides accelerator sets for the Default and Emacs accelerator configurations.

Action Definitions

Identifier:

org.eclipse.ui.actionDefinitions

Since:

Release 2.0

Description:

WARNING: This extension point is DEPRECATED.
Do not use this extension point, it will be removed in future versions of this product. Instead, use the extension point org.eclipse.ui.commands

This extension point is used to register action definitions. Accelerators (see the Accelerator Sets extension point) use action definitions to reference actions. An action associates itself with a given accelerator by registering with that accelerator's associated action definition. An action registers itself with an action definition by calling the setActionDefinitionId(String id) method and supplying the action definition's id as an argument.

Configuration Markup:

<!ELEMENT extension (actionDefinition*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT actionDefinition EMPTY>

<!ATTLIST actionDefinition

id CDATA #REQUIRED

name CDATA #IMPLIED

description CDATA #IMPLIED>

- **id –** a unique name that can be used to identify this action.

- **name** – the name of the action as displayed to the user.
- **description** – a short description of the action to display to the user.

Examples:

Following is an example of an action definition extension:

```
<extension point=
"org.eclipse.ui.actionDefinitions"
>
<actionDefinition id=
"org.eclipse.ui.file.save"
>
</actionDefinition>
<actionDefinition id=
"org.eclipse.ui.file.saveAll"
>
</actionDefinition>
<actionDefinition id=
"org.eclipse.ui.file.close"
>
</actionDefinition>
<actionDefinition id=
"org.eclipse.ui.file.closeAll"
>
</actionDefinition>
<actionDefinition id=
```

"org.eclipse.ui.file.print"

>

</actionDefinition>

</extension>

API Information:

The methods public void setActionDefinitionId(String id) and public String getActionDefinitionId() have been added to IAction.

NOTE – other attributes may be added in the future, as needed.

Supplied Implementation:

The workbench provides many action definitions.

Action Set Part Associations

Identifier:

org.eclipse.ui.actionSetPartAssociations

Description:

This extension point is used to define an action set which should be added to a perspective when a part (view or editor) is opened in the perspective. In the case of an editor, the action set will remain visible while the editor is the current editor. In the case of a view, the action set will be visible when the view is the active part.

Configuration Markup:

<!ELEMENT extension (actionSetPartAssociation*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT actionSetPartAssociation (part*)>

<!ATTLIST actionSetPartAssociation

targetID CDATA #REQUIRED>

- **targetID –** the unique identifier of the action set (as specified in the registry) which is to be associated with particular workbench views and/or editors.

<!ELEMENT part EMPTY>

<!ATTLIST part

id CDATA #REQUIRED>

- **id –** the unique identifier of the part (view or editor) to be associated with the action set.

Examples:

The following is an example of an action set part association (note the subelement and the way attributes are used):

```
<extension point=

"org.eclipse.ui.actionSetPartAssociations"

>

<actionSetPartAssociation targetID=

"org.eclipse.jdt.ui.refactoring.actionSet"

>

<part id=

"org.eclipse.jdt.ui.PackageExplorer"

/>

<part id=

"org.eclipse.jdt.ui.CompilationUnitError"

/>

</actionSetPartAssociation>

</extension>
```

In the example above, a view or editor are associated with the refactoring action set.

API Information:

The user may override these associations using the customize perspective dialog. Regardless of these associations, action sets which the user turns off will never appear and action sets which the user turns on will always be visible.

Activities

Identifier:

org.eclipse.ui.activities

Since:

3.0

Description:

The `org.eclipse.ui.activities` extension point is used to declare activities and associated elements. Activities are used by the platform to filter certain plugin contributions from the users view until such a time that they express interest in them. This allows Eclipse to grow dynamically based on the usage pattern of a user.

Configuration Markup:

<!ELEMENT extension (activity , activityRequirementBinding , activityPatternBinding , category , categoryActivityBinding , defaultEnablement)*>

<!ATTLIST extension

id CDATA #IMPLIED

name CDATA #IMPLIED

point CDATA #REQUIRED>

- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance
- **point –** a fully qualified identifier of the target extension point

<!ELEMENT activity EMPTY>

<!ATTLIST activity

description CDATA #IMPLIED

id CDATA #REQUIRED

name CDATA #REQUIRED>

This element is used to define activities. If more than one of these elements exist with the same `id` attribute, only the last declared element (in order of reading the registry) is considered valid.

- **description** – a translatable short description of this activity for display in the UI
- **id** – the unique identifier of this activity
- **name** – the translatable name of this activity for display in the UI

<!ELEMENT activityRequirementBinding EMPTY>

<!ATTLIST activityRequirementBinding

requiredActivityId CDATA #REQUIRED

activityId CDATA #REQUIRED>

This element allows one to bind activities to activities. The relationship is such that if the activityId is ever enabled then the requiredActivityId is enabled as well.

- **requiredActivityId** – the unique identifier of required activity to bind
- **activityId** – the unique identifier of the activity to bind

<!ELEMENT activityPatternBinding EMPTY>

<!ATTLIST activityPatternBinding

activityId CDATA #REQUIRED

pattern CDATA #REQUIRED>

This element allows one to bind activities to patterns.

- **activityId** – the unique identifier of the activity to bind
- **pattern** – the pattern to be bound. Patterns are regular expressions which match unique identifiers. Please see the Java documentation for `java.util.regex.Pattern` for further details.

<!ELEMENT category EMPTY>

<!ATTLIST category

description CDATA #IMPLIED

id CDATA #REQUIRED

name CDATA #REQUIRED>

Since:

This element is used to define categories. If more than one of these elements exist with the same `id` attribute, only the last declared element (in order of reading the registry) is considered valid.

- **description** – a translatable short description of this category for display in the UI
- **id** – the unique identifier of this category
- **name** – the translatable name of this category for display in the UI

<!ELEMENT categoryActivityBinding EMPTY>

<!ATTLIST categoryActivityBinding

activityId CDATA #REQUIRED

categoryId CDATA #REQUIRED>

This element allows one to bind categories to activities.

- **activityId** – the unique identifier of the activity to bind
- **categoryId** – the unique identifier of the category to bind

<!ELEMENT defaultEnablement EMPTY>

<!ATTLIST defaultEnablement

id CDATA #REQUIRED>

This element allows one to specify that a given activity should be enabled by default.

- **id** – the unique identifier of the activity

Examples:

The following is an example of several activity and category definitions as well as associated bindings.

<extension point=

"org.eclipse.ui.activities"

```
>
<activity id=
"com.xyz.Activity"
description=
"Filters contributions from com.xyz"
name=
"My Activity"
/>
<activity id=
"com.xyz.OtherActivity"
description=
"Filters other contributions from com.xyz"
name=
"My Other Activity"
/>
<!-- other activity requires activity -->
<activityRequirementBinding activityId=
"com.xyz.OtherActivity"
requiredActivityId=
"com.xyz.Activity"
/>
<category id=
"com.xyz.Category"
description=
"com.xyz Activities"
name=
```

```
"My Category"
/>
<!-- put the activity in the category -->
<categoryActivityBinding activityId=
"com.xyz.Activity"
categoryId=
"com.xyz.Category"
/>
<!-- bind all contributions from plugin com.xyz -->
<activityPatternBinding id=
"com.xyz.Activity"
pattern=
"com\.xyz/.*"
/>
<!-- bind my.contribution from plugin com.xyz.other -->
<activityPatternBinding id=
"com.xyz.OtherActivity"
pattern=
"com\.xyz\.other/my.contribution"
/>
<!-- our activity should be enabled by default -->
<defaultEnablement id=
"com.xyz.Activity"
/>
</extension>
```

API Information:

There is currently no public API for declaring activities or associated elements other than this extension point. The state of activities in the workbench is accessible via `org.eclipse.ui.IWorkbench.getActivitySupport()`. From here you may query and update the set of currently enabled activities.

Supplied Implementation:

There are no "default activities" provided by the workbench. Activities are intended to be defined at the product level, such as the Eclipse SDK, so as to tightly integrate all of the (known) components that product contains.

Activity Support

Identifier:

org.eclipse.ui.activitySupport

Since:

3.1

Description:

This extension point is used to register various support extensions relating to the activities infrastucture.

Configuration Markup:

<!ELEMENT extension (triggerPoint | triggerPointAdvisor | triggerPointAdvisorProductBinding | categoryImageBinding | activityImageBinding)*>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT triggerPoint (hint*)>

<!ATTLIST triggerPoint

id CDATA #REQUIRED>

Specifies a trigger point. A trigger point is an identifer that is used by the activity engine to determine whether or not an action should cause the enablement of activities.

- **id –** a unique identifier for this trigger point

<!ELEMENT hint EMPTY>

<!ATTLIST hint

id CDATA #REQUIRED

value CDATA #REQUIRED>

A hint represents some data that may be used by the eclipse infrastructure to determine the behaviour of the activity support relating to the hosting trigger point.

The following hints are "well known" and it is expected that trigger point advisors acknowledge them:

interactive
: Whether this trigger point is hint in an "interactive" way. Ie: it is the explicit result of an action undertaken by the user such as activating a wizard in the wizard dialog. Accepted values are `true` and `false`

- **id –** a unique identifier for this hint
- **value –** the value of this hint

<!ELEMENT triggerPointAdvisor EMPTY>

<!ATTLIST triggerPointAdvisor

id CDATA #REQUIRED

class CDATA #REQUIRED>

A trigger point advisor is a policy mechanism that is consulted whenever the system undertakes an action that has disabled activities associated with it. It is the advisors responsibility to answer whether an action should proceed, and if it can, what activities to enable.

- **id –** a unique identifier for this trigger point advisor
- **class –** a fully qualified name of the class implementing the `org.eclipse.ui.activities.ITriggerPointAdvisor` interface.

<!ELEMENT triggerPointAdvisorProductBinding EMPTY>

<!ATTLIST triggerPointAdvisorProductBinding

productId CDATA #REQUIRED

triggerPointAdvisorId CDATA #REQUIRED>

Specifies a binding between a product and an advisor. These bindings determine which advisor is appropriate for the current product (as defined by `org.eclipse.core.runtime.Platform.getProduct()`).

Since:

- **productId –** unique id of a product
- **triggerPointAdvisorId –** unique id of a trigger point advisor

<!ELEMENT categoryImageBinding EMPTY>

<!ATTLIST categoryImageBinding

id CDATA #REQUIRED

icon CDATA #REQUIRED>

This element allows binding of icons to categories These icons may be used by user interface components that wish to visualize categories in some way.

- **id –** the id of the category to bind an icon to
- **icon –** the name of the icon that will be used for this category

<!ELEMENT activityImageBinding EMPTY>

<!ATTLIST activityImageBinding

id CDATA #REQUIRED

icon CDATA #REQUIRED>

This element allows binding of icons to activities. These icons may be used by user interface components that wish to visualize activities in some way.

- **id –** the id of the activity to bind an icon to
- **icon –** the name of the icon that will be used for this activity

Examples:

The following is an example of a non−interactive trigger point:

<extension point=

"org.eclipse.ui.activitySupport"

>

<triggerPoint id=

"com.example.xyz.myTriggerPoint"

>

<hint id=

"interactive"

value=

"false"

/>

</triggerPoint>

</extension>

The following is an example of a trigger point advisor bound to a particular product:

<extension point=

"org.eclipse.ui.activitySupport"

>

<triggerPointAdvisor id=

"com.example.xyz.myTriggerPointAdvisor"

class=

"com.example.xyz.AdvisorImpl"

/>

<triggerPointAdvisorProductBinding productId=

"myProduct"

triggerPointAdvisorId=

"com.example.xyz.myTriggerPointAdvisor"

/>

</extension>

The following is an example of binding images to activities and categories:

<extension point=

"org.eclipse.ui.activitySupport"

>

<activityImageBinding id=

"some.activity.id"

icon=

"icons/someIcon.gif"

/>

<categoryImageBinding id=

"some.category.id"

icon=

"icons/someIcon.gif"

/>

</extension>

API Information:

The value of the `class` attribute of the triggerPointAdvisor tag must be the fully qualified name of a class that implements the `org.eclipse.ui.activities.ITriggerPointAdvisor`.

Supplied Implementation:

The workbench implementation (`org.eclipse.ui.activities.WorkbenchTriggerPointAdvisor`) is available for clients to subclass and reuse.

Browser Support

Identifier:

org.eclipse.ui.browserSupport

Since:

3.1

Description:

This extension point is used to contribute workbench browser support. The support is responsible for opening URLs for all the Eclipse plug−ins. Workbench provides a very rudimentary implementation with a more complete implementation available as an optional RCP plug−in.

Contributions that are meant to be shipped with the product as the standard support should be marked as `default`. This way, it is possible to override the support with another contribution that is not marked as `default`. Note however that only one support can be active at any point in time. In case of multiple default and/or non−default contributions, the result is non−deterministic.

Configuration Markup:

<!ELEMENT extension (support+)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point** − A fully qualified identifier of the target extension point.
- **id** − An optional identifier of the extension instance.
- **name** − An optional name of the extension instance.

<!ELEMENT support EMPTY>

<!ATTLIST support

class CDATA #REQUIRED

default (true | false) "false">

- **class** − The class providing the browser support for the workbench. This class should extend `org.eclipse.ui.browser.AbstractWorkbenchBrowserSupport`.
- **default** − indicates whether this support is the default. Browser support should be marked as default if it is normally shipped with the product as the standard browser support. Browser supports that need to

override the default support should have this flag set to `false`. When workbench encounters two extensions, it will pick a non−default over a default one.

Examples:

The following is an example of a browser support contribution:

<extension point=

"org.eclipse.ui.browserSupport"

>

<support default=

"true"

class=

"com.example.xyz.MyBrowserSupport"

>

</support>

</extension>

API Information:

The contributors are expected to provide a class that extends `org.eclipse.ui.browser.AbstractWorkbenchBrowserSupport`.

Supplied Implementation:

The workbench provides a simple implementation of the browser support that is used when no contributions are found in the registry.

Cheat Sheet Content

Identifier:

org.eclipse.ui.cheatsheets.cheatSheetContent

Since:

3.0

Description:

This extension point is used to register cheat sheet content contributions. Cheat sheets appear as choices from the "Help" menu or from within the cheat sheet view, and are typically used to aid a user through a series of comlex tasks to accomplish an overall goal.

The cheat sheets are organized into categories which usually reflect a particular problem domain. For instance, a Java oriented plug−in may define a category called "Java" which is appropriate for cheat sheets that would aid a user with any of the Java tools. The categories defined by one plug−in can be referenced by other plug−ins using the category attribute of a cheatsheet element. Uncategorized cheat sheets, as well as cheat sheets with invalid category paths, will end up in an "Other" category.

Cheat sheets may optionally specify a description subelement whose body should contain short text about the cheat sheet.

Configuration Markup:

<!ELEMENT extension (category | cheatsheet)*>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point −** a fully qualified identifier of the target extension point
- **id −** an optional identifier of the extension instance
- **name −** an optional name of the extension instance

<!ELEMENT category EMPTY>

<!ATTLIST category

id CDATA #REQUIRED

name CDATA #REQUIRED

parentCategory CDATA #IMPLIED>

A category element in the cheatsheetContent extension point creates a new category in the cheat sheet menu. The cheat sheet menu is available from the help menu in the workbench. If a new category is specified, cheat sheets may be targeted to that category and will appear under it in the cheat sheet selection dialog.

- **id** – a unique name that can be used to identify this category
- **name** – a translatable name of the category that will be used in the dialog box
- **parentCategory** – a path to another category if this category should be added as a child

<!ELEMENT cheatsheet (description?)>

<!ATTLIST cheatsheet

id CDATA #REQUIRED

name CDATA #REQUIRED

category CDATA #IMPLIED

contentFile CDATA #IMPLIED

listener CDATA #IMPLIED>

A cheatsheet element is put into the cheatsheetContent extension point if there is a cheat sheet to be contributed to the workbench. A cheat sheet element must specify an id, a translatable name to appear in the selection options, a category id to specify which category this cheat sheet will be included in, and a content file. The cheat sheet content file is an XML file that describes the steps and actions that the cheat sheet has.

- **id** – a unique name that can be used to identify this cheat sheet
- **name** – a translatable name of the cheat sheet that will be used in the help menu and the selection dialog box
- **category** – a slash–delimited path ('/') of category IDs. Each token in the path must represent a valid category ID previously defined by this or some other plug–in. If omitted, the wizard will be added to the "Other" category.
- **contentFile** – the path of a cheat sheet content file. The content file is an XML file that contains the specifics of the cheat sheet (cheat sheet content file format specification). The content file is parsed at run time by the cheat sheet framework. Based on the settings in this file, a certain number of steps, actions, descriptions, and help links are shown to the user when the cheat sheet is opened. The path is interpreted as relative to the plug–in that declares the extension; the path may include special variables. In particular, use "nl" as the first segment of the path to indicate that there are locale–specific translations of the content file in subdirectories below "nl/". For more detail about the special variables, you can read the Java API document for Platform.find.
- **listener** – listener is a fully qualified name of a Java class which must subclass `org.eclipse.ui.cheatsheets.CheatSheetListener`.

Since:

<!ELEMENT description (#PCDATA)>

a short description of the cheat sheet

Examples:

Here is a sample usage of the cheatSheetContent extension point:

<extension point=

"org.eclipse.ui.cheatsheets.cheatSheetContent"

>

<category name=

"Example category"

id=

"com.example.category"

>

</category>

<cheatsheet name=

"Example cheat sheet"

category=

"com.example.category"

id=

"com.example.cheatSheet"

contentFile=

"ExampleCheatSheet.xml"

>

<description>

This is a descriptive bit of text for my cheat sheet description.

</description>

</cheatsheet>

</extension>

API Information:

For further details see the spec for the org.eclipse.ui.cheatsheets API package.

Supplied Implementation:

There are no built−in cheat sheets.

Cheat Sheet Content File XML Format

Version 3.0

This document describes the cheat sheet content file structure as a series of DTD fragments (machine readable XML schema).

cheatsheet

```
<!ELEMENT cheatsheet (intro, item+)>
<!ATTLIST cheatsheet
  title               CDATA #REQUIRED
>
```

The <cheatsheet> element defines the body of the cheat sheet content file. <cheatsheet> attributes are as follows:

- **title –** the title of the cheat sheet

intro

```
<!ELEMENT intro (description)>
<!ATTLIST intro
  contextId           CDATA #IMPLIED
  href                CDATA #IMPLIED
>
```

The <intro> element is used to describe the cheat sheet introduction to be displayed. The <description> subelement contains the body of the introduction. <intro> attributes are as follows:

- **contextId –** The optional help context id of the documentation for this cheat sheet. If supplied, context help for the given fully−qualified context id is shown to the user (typically in a small pop−up window) when they clicks the introduction's help link. If this attribute is supplied, the **href** attribute should not be supplied (**href** will be ignored if both are present).
- **href –** The optional help document describing this cheat sheet. If supplied, this help document is shown to the user (typically in a help browser shown in a separate window) when they clicks the introduction's help link. If this attribute is supplied, the **contextId** attribute should not be supplied (**href** will be ignored if both are present).

description

```
<!ELEMENT description EMPTY>
<!ATTLIST description
>
```

The <description> element holds the description of a cheat sheet or of a cheat sheet item. The description consists of text interspersed with simple formatting tags. The cheat sheet automatically formats and lays out the text to make it show up reasonably in the UI. Within the text, balanced ****...**** tags cause the enclosed text to be rendered in a bold font, and the **
** element can be used to force a line break. These are the only formatting tags supported at this time (however, others may be added in the future). Certain characters in the

text have special significance for XML parsers; in particular, to write "<", ">", "&", """, and "'" (quotation mark) instead write "<", ">", "&", "'", and """ respectively. Whitespace (spaces and line breaks) is treated as a word separator; adjacent spaces and line breaks are treated as single unit and rendered as a single space or a line break. Whitespace immediately after the **<description>** and **
** tags is ignored, as is whitespace immediately before the **</description>** tag.

item

```
<!ELEMENT item (description ([action|perform-when] | (subitem|repeated-subitem|conditional-subite
<!ATTLIST item
  title               CDATA #REQUIRED
  skip                ("true" | "false") "false"
  contextId           CDATA #IMPLIED
  href                CDATA #IMPLIED
>
```

Each <item> element describes one top−level step in a cheat sheet. The <item> is either simple or composite. <item> attributes are as follows:

- **title** − The title of the cheat sheet item.
- **skip** − skip="true" means that the whole step can be skipped; the UI generally shows a button that the user can press to indicate that they are skipping this step
- **contextId** − The optional help context id of the documentation for this cheat sheet step. If supplied, context help for the given fully−qualified context id is shown to the user (typically in a small pop−up window) when they clicks the step's help link. If this attribute is supplied, the **href** attribute should not be supplied (**href** will be ignored if both are present).
- **href** − The optional help document describing this cheat sheet step. If supplied, this help document is shown to the user (typically in a help browser shown in a separate window) when they clicks the step's help link. If this attribute is supplied, the **contextId** attribute should not be supplied (**href** will be ignored if both are present).

The org.eclipse.ui.cheatsheets.cheatSheetItemExtension allows additional custom controls for the item to be displayed in the UI. Contributions to this extension point declare the names of additional, string−valued attributes that may appear on <item> elements.

Simple items have a description and an optional action. In the typical presentation, the titles of cheat sheet items are shown to the user most of the time. An item's description is only shown while the step is in the process of being executed. The presence of an <action> (or <perform−when>) element is typically associated with a button that the user can press to perform the step's action. If no action is present, the step is one that the user must carry out manually and then overtly indicate that they have successfully completed the step.

Composite steps are broken down into sub−steps as specified by the <subitem> subelements. Unlike items, which the user must follow in strict sequence, the sub−items of a given item can be performed in any order. All sub−items within an item have to be attempted (or skipped) before progressing to the next item. (Which means actions that must be performed in a required sequence cannot be represented as sub−items.)

A <conditional−subitem> subelement allow a step to tailor the presentation of a sub−step based on cheat sheet variables whose values are acquired in earlier steps. A <repeated−subitem> subelement allows a step to include a set of similar sub−steps. Again, the exact set of sub−steps may be based on cheat sheet variables whose value are acquired in earlier steps.

subitem

```
<!ELEMENT subitem ( [action|perform-when] )>
<!ATTLIST subitem
  label               CDATA #REQUIRED
  skip                ("true" | "false") "false"
  when                CDATA #IMPLIED
>
```

Each <subitem> element describes a sub−step in a cheat sheet. A <subitem> carries a simple text label, but has neither a lengthy description nor further sub−items. <subitem> attributes are as follows:

- **label −** The title of the cheat sheet sub−item. If the string contains substring occurrences of the form "${*var*}", they are considered references to cheat sheet variables. All such occurrences in the string value will be replaced by the value of the corresponding variable in the context of the execution of the cheat sheet, or the empty string for variables that are unbound. The values of the variables are as of the beginning of the execution of the main step (when the <item> element is elaborated), rather than when the individual sub−step are run.
- **skip −** skip="true" means that the sub−step can be skipped. The UI generally shows a button that the user can press to indicate that they are skipping this sub−step.
- **when −** Indicates this subitem is to be used if and only if the value of the condition attribute of the containing <conditional−subitem> element matches this string value. This attribute is ignored if the <subitem> element is not a child of a <conditional−subitem> element.

Sub−items have an optional action. The presence of an <action> (or <perform−when>) element is typically associated with a button that the user can press to perform the sub−step's action. If no action is present, the sub−step is one that the user must carry out manually and then overtly indicate that they have successfully completed the step.

Unlike items, which must be followed in strict sequence, the sub−items of a given item can be performed in any order. All sub−items within an item have to be attempted (or skipped) before progressing to the next item. (Which means actions that must be performed in a required sequence should not be represented as sub−items.)

conditional−subitem

```
<!ELEMENT conditional-subitem (subitem+)>
<!ATTLIST conditional-subitem
  condition              CDATA #REQUIRED
>
```

Each <conditional−subitem> element describes a single sub−step whose form can differ based on a condition known at the time the item is expanded. <conditional−subitem> attributes are as follows:

- **condition −** Arbitrary string value used to select which child <subitem> will be used. If the attribute string has the form "${*var*}", it is considered a reference to a cheat sheet variable *var*, and value of the condition will be the value of the variable for the cheat sheet at the start of execution of the containing <item> element (or the empty string if the variable is unbound at that time).

The **condition** attribute on the <conditional−subitem> element provides a string value (invariably this value comes from a cheat sheet variable). Each of the <subitem> children must carry a **when** attribute with a distinct string value. When the item is expanded, the <conditional−subitem> element is replaced by the <subitem>

element with the matching value. It is considered an error if there is no <subitem> element with a matching value.

For example, if the cheat sheet variable named "v1" has the value "b" when the following item is expanded

```
<item ...>
  <conditional-subitem condition="${v1}">
     <subitem when="a" label="Step for A." />
     <subitem when="b" label="Step for B." />
  </conditional-subitem>
</item>
```

then the second sub−item is selected and the item expands to something equivalent to

```
<item ...>
  <subitem label="Step for B."/>
</item>
```

repeated-subitem

```
<!ELEMENT repeated-subitem (subitem)>
<!ATTLIST repeated-subitem
  values               CDATA #REQUIRED
>
```

Each <repeated−subitem> element describes a sub−item that expands into 0, 1, or more similar sub−steps. <repeated−subitem> attributes are as follows:

- **values** − A string containing a comma−separated list of values. If the attribute string has the form "${*var*}", it is considered a reference to a cheat sheet variable *var*, and value of the condition will be the value of the variable for the cheat sheet at the start of execution of the containing <item> element (or the empty string if the variable is unbound at that time).

The **values** attribute provides a list of comma−separated strings; the <subitem> child provide the template. When the item is expanded, the <repeated−subitem> element is replaced by copies of the <subitem> element with occurrences of the variable "this" replaced by the corresponding string value.

For example, if the cheat sheet variable named "v1" has the value "1,b,three" when the following item is expanded

```
<item ...>
  <repeated-subitem values="${v1}">
     <subitem label="Step ${this}.">
        <action class="com.xyz.myaction" pluginId="com.xyz" param1="${this}"/>
     </subitem>
  </repeated-subitem>
</item>
```

then the item expands to something equivalent to:

```
<item ...>
  <subitem label="Step 1.">
     <action class="com.xyz.myaction" pluginId="com.xyz" param1="1"/>
  </subitem>
  <subitem label="Step b.">
```

```
    <action class="com.xyz.myaction" pluginId="com.xyz" param1="b"/>
  </subitem>
  <subitem label="Step three.">
    <action class="com.xyz.myaction" pluginId="com.xyz" param1="three"/>
  </subitem>
</item>
```

action

```
<!ELEMENT action EMPTY>
<!ATTLIST action
  class               CDATA #REQUIRED
  pluginId            CDATA #REQUIRED
  param1              CDATA #IMPLIED
  ...
  param9              CDATA #IMPLIED
  confirm             ("true" | "false") "false"
  when                CDATA #IMPLIED
>
```

Each <action> element describes an action in a cheat sheet. <action> attributes are as follows:

- **class –** The fully−qualified name of the Java class implementing `org.eclipse.jface.action.IAction`. If this action also implements `org.eclipse.ui.cheatsheets.ICheatSheetAction` it will be invoked via its run(String[],ICheatSheetManager) method and be passed the cheat sheet manager and action parameters. The pluginId attribute must be present whenever this attribute is present. It is strongly recommended that actions intended to be invoked from cheat sheets should report success/fail outcome if running the action might fail (perhaps because the user cancels the action from its dialog). (See org.eclipse.jface.action.Action.notifyResult(boolean) for details.)
- **pluginId –** The id of the plug−in which contains the Java class of the action class. This attribute must be present whenever the class attribute is present.
- **param*N* –** For action classes that also implement `org.eclipse.ui.cheatsheets.ICheatSheetAction`, the string values of these attributes are passed to the action when it is invoked. You can pass up to 9 parameters to a cheat sheet action (**param1**, **param2**, etc.). The parameters supplied must start with parameter 1 and be contiguous; that is, it is illegal to specify **param2** without **param1** also being present. If the attribute string has the form "${*var*}", it is considered a reference to a cheat sheet variable *var*, and value of the condition will be the value of the variable for the cheat sheet at the start of execution of the containing <item> element (or the empty string if the variable is unbound at that time).
- **confirm –** "true" indicates this step (or sub−step) requires the user to manually confirm that the action has been completed.
- **when –** Indicates this action is to be used if and only if the value of the condition attribute of the containing <perform−when> element matches this string value. This attribute is ignored if the <action> element is not a child of a <perform−when> element.

perform−when

```
<!ELEMENT perform-when (action+)>
<!ATTLIST perform-when
  condition               CDATA #REQUIRED
>
```

Each <perform−when> element describes an action in a cheat sheet. <perform−when> attributes are as follows:

- **condition −** Arbitrary string value used to select which child <action> will be performed. If the attribute string has the form "${*var*}", it is considered a reference to a cheat sheet variable *var*, and value of the condition will be the value of the variable for the cheat sheet at the start of execution of the containing <item> element (or the empty string if the variable is unbound at that time).

The **condition** attribute on the <conditional−subitem> element provides a string value (invariably this value comes from a cheat sheet variable). Each of the <subitem> children must carry a **when** attribute with a distinct string value. When the item is expanded, the <conditional−subitem> element is replaced by the <subitem> element with the matching value. It is considered an error if there is no <subitem> element with a matching value.

For example, if the cheat sheet variable named "v1" has the value "b" when the following item is expanded

```
<item ...>
  <subitem label="Main step">
     <perform-when condition="${v1}">
        <action when="a" class="com.xyz.action1" pluginId="com.xyz" />
        <action when="b" class="com.xyz.action2" pluginId="com.xyz" />
     </conditional-subitem>
  </subitem>
</item>
```

then the second action is selected and the item expands to something equivalent to

```
<item ...>
  <subitem label="Main step">
     <action class="com.xyz.action2" pluginId="com.xyz" />
  </subitem>
</item>
```

Example

The following is an example of a very simple cheat sheet content file:

```
<?xml version="1.0" encoding="UTF-8"?>
<cheatsheet title="Example">
  <intro>
    <description>Example cheat sheet with two steps.</description>
  </intro>
  <item title="Step 1">
     <description>This is a step with an action.</description>
     <action class="com.xyz.myaction" pluginId="com.xyz"/>
  </item>
  <item title="Step 2">
     <description>This is a fully manual step.</description>
  </item>
</cheatsheet>
```

Eclipse Public License v1.0 which accompanies this distribution, and is available at
http://www.eclipse.org/legal/epl−v10.html

Cheat Sheet Item Extension

Identifier:

org.eclipse.ui.cheatsheets.cheatSheetItemExtension

Since:

3.0

Description:

This extension point should be used when an extra button needs to be added to a step in the cheat sheet. You can put a new attribute into the "item" tag in the cheat sheet content file, and when that value is read by the cheat sheet framework, it will check to see if there is a class registered through this extension point that will handle this attribute. The attribute name found in the cheat sheet content file is matched against all of the values found in the "itemAttribute" attribute of all of the registered cheatsheetItemExtension point implementations. If there is a match, the class specified to handle this item attribute is loaded by the cheat sheet framework and is called to handle the attribute specified in the cheat sheet content file. After having parsed the value of the item attribute, the class remains available to the cheat sheets framework. When the item is rendered for the cheat sheets view, the class is once again called to handle the addition of components to a Composite. The items that are added to this composite are displayed in the cheat sheet step (currently, beside the help icon). It is displayed only for the step that is described by the "item" tag that the attribute appeared in the cheat sheet content file. The suggested use of this extension point is adding a small (16x16) button with a graphic that opens a dialog box when pressed.

Configuration Markup:

<!ELEMENT extension (itemExtension)*>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT itemExtension EMPTY>

<!ATTLIST itemExtension

itemAttribute CDATA #REQUIRED

class CDATA #REQUIRED>

Use this item extension to add elements to cheat sheet steps. You can use this extension point to add icons and buttons (currently, beside the help icon) for a step in the cheat sheet. You specify the name of an attribute that you will put into the cheat sheet item tag. You also specify a class that will handle the parsing of the attribute value from the cheat sheet content file when the cheat sheet is loaded. The attribute valuemust be a string. The specified class must subclass `org.eclipse.ui.cheatsheets.AbstractItemExtensionElement`. After the cheat sheet content file is parsed and loaded, the class specified in the extension point is called again through the interface to add graphics or buttons to the step in the cheat sheet (currently, next to the help button).

- **itemAttribute –** This attribute value must be the string value of an attribute name that is put into an item tag in the cheat sheet content file. If this attribute string matches an attribute parsed from the item tag in the cheat sheet content file, the class specified will be loaded and will be called to parse the full value of the attribute using the w3 DOM specification. It will later be called to add controls to a Composite, and the added components (usually graphics or buttons) will appear in the step of the cheat sheet for the item specified (currently, beside the help icon for that step).
- **class –** The fully qualified class name of the class that subclasses `org.eclipse.ui.cheatsheets.AbstractItemExtensionElement` to handle unknown attributes in the cheat sheet content file and extend the steps in the cheat sheet. The class must be public, and have a public 1−argument constructor that accepts the attribute name (a `String`).

Examples:

Here is an example implementation of this extension point:

<extension point=

"org.eclipse.ui.cheatsheets.cheatSheetItemExtension"

>

<itemExtension itemAttribute=

"xyzButton"

class=

"com.example.HandleParsingAndAddButton"

>

</itemExtension>

</extension>

And here is the item attribute for that extension:

<item title=

"XYZ Title"

xyzButton=

"/icon/button.gif"

/>

Note that the value of the attribute in the item tag can be ANYTHING. It can be anything because the class that parses that attribute is the class HandleParsingAndAddButton, which in this example parses a string /icon/button.gif from the attribute. It later will use that info to load the gif and use it as the icon for a new button.

API Information:

See the Javadoc information for org.eclipse.ui.cheatsheets.AbstractItemExtensionElement for API details.

Supplied Implementation:

There is no supplied implementation at this time.

Decorators

Identifier:

org.eclipse.ui.decorators

Since:

Release 2.0

Description:

This extension point is used to add decorators to views that subscribe to a decorator manager. As of 2.1 there is the concept of a lightweight decorator that will handle the image management for the decorator. It is also possible to declare a lightweight decorator that simply overlays an icon when enabled that requires no implementation from the plug−in.

An action's enablement and/or visibility can be defined using the elements `enablement` and `visibility` respectively. These two elements contain a boolean expression that is evaluated to determine the enablement and/or visibility.

The syntax is the same for the `enablement` and `visibility` elements. Both contain only one boolean expression sub−element. In the simplest case, this will be an `objectClass`, `objectState`, `pluginState`, or `systemProperty` element. In the more complex case, the `and`, `or`, and `not` elements can be combined to form a boolean expression. Both the `and`, and `or` elements must contain 2 sub−elements. The `not` element must contain only 1 sub−element.

Configuration Markup:

<!ELEMENT extension (decorator*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point −** a fully qualified identifier of the target extension point
- **id −** an optional identifier of the extension instance
- **name −** an optional name of the extension instance

<!ELEMENT decorator (description? , enablement?)>

<!ATTLIST decorator

id CDATA #REQUIRED

label CDATA #REQUIRED

class CDATA #IMPLIED

objectClass CDATA #IMPLIED

adaptable (true | false)

state (true | false)

lightweight (true|false)

icon CDATA #IMPLIED

location (TOP_LEFT|TOP_RIGHT|BOTTOM_LEFT|BOTTOM_RIGHT|UNDERLAY) >

- **id –** a unique name that will be used to identify this decorator.
- **label –** a translatable name that will be used in the workbench window menu to represent this decorator.
- **class –** a fully qualified name of a class which implements `org.eclipse.jface.viewers.ILabelDecorator` if ***lightweight*** is false or `org.eclipse.jface.viewers.ILightweightLabelDecorator` if lightweight is true. The default value is false. If there is no ***class*** element it is assumed to be true.
- **objectClass –** a fully qualified name of a class which this decorator will be applied to. Deprecated in 2.1. Make this value part of the enablement.
- **adaptable –** a flag that indicates if types that adapt to IResource should use this object contribution. This flag is used only if objectClass adapts to IResource. Default value is false.
- **state –** a flag that indicates if the decorator is on by default. Default value is false.
- **lightweight –** The lightweight flag indicates that the decorator is either declarative or implements `org.eclipse.jface.viewers.ILightweightLabelDecorator`.
- **icon –** if the decorator is ***lightweight*** and the ***class*** is not specified this is the path to the overlay image to apply
- **location –** if the decorator is ***lightweight*** this is the location to apply the decorator to. Defaults to BOTTOM_RIGHT.

<!ELEMENT description (#PCDATA)>

an optional subelement whose body should contain text providing a short description of the decorator. This will be shown in the Decorators preference page so it is recommended that this is included. Default value is an empty String.

<!ELEMENT enablement (and | or | not | objectClass | objectState | pluginState | systemProperty)>

This element is used to define the enablement for the extension.

<!ELEMENT visibility (and | or | not | objectClass | objectState | pluginState | systemProperty)>

This element is used to define the visibility for the extension.

<!ELEMENT and (and | or | not | objectClass | objectState | pluginState | systemProperty)>

This element represent a boolean AND operation on the result of evaluating its two sub−element expressions.

<!ELEMENT or (and | or | not | objectClass | objectState | pluginState | systemProperty)>

This element represent a boolean OR operation on the result of evaluating its two sub−element expressions.

<!ELEMENT not (and | or | not | objectClass | objectState | pluginState | systemProperty)>

This element represent a boolean NOT operation on the result of evaluating its sub−element expressions.

<!ELEMENT objectClass EMPTY>

<!ATTLIST objectClass

name CDATA #REQUIRED>

This element is used to evaluate the class or interface of each object in the current selection. If each object in the selection implements the specified class or interface, the expression is evaluated as true.

- **name** − a fully qualified name of a class or interface. The expression is evaluated as true only if all objects within the selection implement this class or interface.

<!ELEMENT objectState EMPTY>

<!ATTLIST objectState

name CDATA #REQUIRED

value CDATA #REQUIRED>

This element is used to evaluate the attribute state of each object in the current selection. If each object in the selection has the specified attribute state, the expression is evaluated as true. To evaluate this type of expression, each object in the selection must implement, or adapt to, `org.eclipse.ui.IActionFilter` interface.

- **name** – the name of an object's attribute. Acceptable names reflect the object type, and should be publicly declared by the plug–in where the object type is declared.
- **value** – the required value of the object's attribute. The acceptable values for the object's attribute should be publicly declared.

<!ELEMENT pluginState EMPTY>

<!ATTLIST pluginState

id   CDATA #REQUIRED

value (installed|activated) "installed">

This element is used to evaluate the state of a plug–in. The state of the plug–in may be one of the following: `installed` (equivalent to the OSGi concept of "resolved") or `activated` (equivalent to the OSGi concept of "active").

- **id** – the identifier of a plug–in which may or may not exist in the plug–in registry.
- **value** – the required state of the plug–in. The state of the plug–in may be one of the following: `installed` (equivalent to the OSGi concept of "resolved") or `activated` (equivalent to the OSGi concept of "active").

<!ELEMENT systemProperty EMPTY>

<!ATTLIST systemProperty

name CDATA #REQUIRED

value CDATA #REQUIRED>

This element is used to evaluate the state of some system property. The property value is retrieved from the `java.lang.System`.

Since:

- **name** – the name of the system property.
- **value** – the required value of the system property.

Examples:

The following are example of decorators:

A full decorator. The plug−in developer must handle their own image support.

<extension point=

"org.eclipse.ui.decorators"

>

<decorator id=

"com.xyz.decorator"

label=

"XYZ Decorator"

state=

"true"

class=

"com.xyz.DecoratorContributor"

>

<enablement>

<objectClass name=

"org.eclipse.core.resources.IResource"

/>

</enablement>

</decorator>

</extension>

A lightweight decorator. There is a concrete class but as it is an ILightweightLabelDecorator it only needs to supply text and an ImageDescriptor and therefore needs no resource handling.

<extension point=

"org.eclipse.ui.decorators"

>

<decorator id=

"com.xyz.lightweight.decorator"

label=

"XYZ Lightweight Decorator"

state=

"false"

class=

"com.xyz.LightweightDecoratorContributor"

lightweight=

"true"

>

<enablement>

<objectClass name=

"org.eclipse.core.resources.IResource"

/>

</enablement>

</decorator>

</extension>

A declarative lightweight decorator. There is no concrete class so it supplies an icon and a quadrant to apply that icon.

```
<extension point=
"org.eclipse.ui.decorators"
>
<decorator id=
"com.xyz.lightweight.declarative.decorator"
label=
"XYZ Lightweight Declarative Decorator"
state=
"false"
lightweight=
"true"
icon=
"icons/full/declarative.gif"
location=
"TOP_LEFT"
>
<enablement>
<objectClass name=
"org.eclipse.core.resources.IResource"
/>
</enablement>
</decorator>
</extension>
```

API Information:

The value of the `class` attribute must be the fully qualified name of a class that implements `org.eclipse.jface.viewers.ILabelDecorator` (if lightweight is false) or `org.eclipse.jface.viewers.ILightweightLabelDecorator`. This class is loaded as late as

possible to avoid loading the entire plug−in before it is really needed. Declarative decorators do not entail any plug−in activation and should be used whenever possible. Non−lightweight decorators will eventually be deprecated.

Supplied Implementation:

Plug−ins may use this extension point to add new decorators to be applied to views that use the decorator manager as their label decorator. To use the decorator manager, use the result of IViewPart.getDecoratorManager() as the decorator for an instance of DecoratingLabelProvider. This is currently in use by the Resource Navigator.

Drop Actions

Identifier:

org.eclipse.ui.dropActions

Description:

This extension point is used to add drop behaviour to views defined by other plugins.

Due to the UI layering imposed by the plugin mechanism, views are often not aware of the content and nature of other views. This can make drag and drop operations between plugins difficult. For example, one may wish to provide Java refactoring support whereby the user drags a method from the Java editor's content outliner into another java file in the resource navigator. Since the resource navigator doesn't know anything about Java content, it doesn't know how to behave when java methods are dropped onto it. Similarly, an ISV may want to drop some of their content into one of the Java viewers.

The `org.eclipse.ui.dropActions` extension point is provided by the Platform to address these situations. This mechanism delegates the drop behaviour back to the originator of the drag operation. This behaviour is contained in an action that must implement `org.eclipse.ui.part.IDropActionDelegate`. The viewer that is the source of the drag operation must support `the org.eclipse.ui.part.PluginTransfer` transfer type, and place a `PluginTransferData` object in the drag event. See org.eclipse.jface.viewers.StructuredViewer#addDragSupport to learn how to add drag support to a viewer.

Configuration Markup:

<!ELEMENT extension (action*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT action EMPTY>

<!ATTLIST action

id CDATA #REQUIRED

class CDATA #REQUIRED>

- **id** – a unique identifier that can be used to reference this action
- **class** – the name of the fully qualified class that implements `org.eclipse.ui.part.IDropActionDelegate`.

Examples:

The following is an example of a drop action extension:

<extension point=

"org.eclipse.ui.dropActions"

>

<action id=

"my_drop_action"

class=

"com.xyz.eclipse.TestDropAction"

>

</action>

</extension>

Here is an example of a drag listener that makes use of the drop action defined above.

```
class MyDragListener extends DragSourceAdapter {
    public void dragSetData(DragSourceEvent event) {
        if (PluginTransfer.getInstance().isSupportedType(event.dataType)) {
            byte[] dataToSend = ...//enter the data to be sent.
            event.data = new PluginTransferData(
                "my_drop_action", dataToSend);
        }
    }
}
```

For a more complete example, see the Platform readme example. In that example, a drop action is defined in ReadmeDropActionDelegate, and it is used by ReadmeContentOutlineDragListener.

API Information:

The value of the class attribute must be a fully qualified name of a Java class that implements `org.eclipse.ui.part.IDropActionDelegate`. This class is loaded as late as possible to avoid loading the entire plug–in before it is really needed

Supplied Implementation:

The workbench does not provide an implementation for this extension point. Plug−ins can contribute to this extension point to add drop behavior to views defined by other plugins.

Editor Menus, Toolbars and Actions

Identifier:

org.eclipse.ui.editorActions

Description:

This extension point is used to add actions to the menu and toolbar for editors registered by other plug−ins.

The initial contribution set for an editor is defined by another extension point (org.eclipse.ui.editors). One set of actions is created and shared by all instances of the same editor type. When invoked, these action act upon the active editor. This extension point follows the same pattern. Each action extension is created and shared by all instances of the same editor type. The action class is required to implement `org.eclipse.ui.IEditorActionDelegate`. The active editor is passed to the delegate by invoking `IEditorActionDelegate.setActiveEditor`.

An action's enablement and/or visibility can be defined using the elements `enablement` and `visibility` respectively. These two elements contain a boolean expression that is evaluated to determine the enablement and/or visibility.

The syntax is the same for the `enablement` and `visibility` elements. Both contain only one boolean expression sub−element. In the simplest case, this will be an `objectClass`, `objectState`, `pluginState`, or `systemProperty` element. In the more complex case, the `and`, `or`, and `not` elements can be combined to form a boolean expression. Both the `and`, and `or` elements must contain 2 sub−elements. The `not` element must contain only 1 sub−element.

Configuration Markup:

<!ELEMENT extension (editorContribution+)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point** − a fully qualified identifier of the target extension point
- **id** − an optional identifier of the extension instance
- **name** − an optional name of the extension instance

<!ELEMENT editorContribution (menu* , action*)>

<!ATTLIST editorContribution

id CDATA #REQUIRED

targetID CDATA #REQUIRED>

This element is used to define a group of editor actions and/or menus.

- **id –** a unique identifier used to reference this contribution.
- **targetID –** a unique identifier of a registered editor that is the target of this contribution.

<!ELEMENT action (selection* | enablement?)>

<!ATTLIST action

id CDATA #REQUIRED

label CDATA #REQUIRED

accelerator CDATA #IMPLIED

definitionId CDATA #IMPLIED

menubarPath CDATA #IMPLIED

toolbarPath CDATA #IMPLIED

icon CDATA #IMPLIED

disabledIcon CDATA #IMPLIED

hoverIcon CDATA #IMPLIED

tooltip CDATA #IMPLIED

helpContextId CDATA #IMPLIED

style (push|radio|toggle) "push"

state (true | false)

class CDATA #REQUIRED

enablesFor CDATA #IMPLIED

actionID CDATA #IMPLIED>

This element defines an action that the user can invoke in the UI.

- **id** – a unique identifier used as a reference for this action. The following IDs have special meaning for the text editor framework:
  - "RulerClick": the contributed action which is invoked upon single–click in the (left) vertical ruler
  - "RulerDoubleClick": the contributed action which is invoked upon single–click in the (left) vertical ruler

  If multiple extensions contribute ruler actions to the same text editor, the extension whose plug–in is at the top of the prerequisite tree wins. If there are multiple such plug–ins, the first wins.
- **label** – a translatable name used either as the menu item text or toolbar button label. The name can include mnenomic information.
- **accelerator** – **Deprecated:** Use the `definitionId` attribute instead.
- **definitionId** – Specifies the command that this action will handle. By specifying and action, the key binding service can assign a key sequence to this action. See the extension point org.eclipse.ui.commands for more information.
- **menubarPath** – a slash–delimited path ('/') used to specify the location of this action in the menu bar. Each token in the path, except the last one, must represent a valid identifier of an existing menu in the hierarchy. The last token represents the named group into which this action will be added. If the path is omitted, this action will not appear in the menu bar.
- **toolbarPath** – a slash–delimited path ('/') that is used to specify the location of this action in the toolbar. The first token represents the toolbar identifier (with "Normal" being the default toolbar), while the second token is the named group within the toolbar that this action will be added to. If the group does not exist in the toolbar, it will be created. If toolbarPath is omitted, the action will not appear in the toolbar.
- **icon** – a relative path of an icon used to visually represent the action in its context. If omitted and the action appears in the toolbar, the Workbench will use a placeholder icon. The path is relative to the location of the plugin.xml file of the contributing plug–in. The icon will appear in toolbars but not in menus. Enabled actions will be represented in menus by the hoverIcon.
- **disabledIcon** – a relative path of an icon used to visually represent the action in its context when the action is disabled. If omitted, the normal icon will simply appear greyed out. The path is relative to the location of the plugin.xml file of the contributing plug–in. The disabled icon will appear in toolbars but not in menus. Icons for disabled actions in menus will be supplied by the OS.
- **hoverIcon** – a relative path of an icon used to visually represent the action in its context when the mouse pointer is over the action. If omitted, the normal icon will be used. The path is relative to the location of the plugin.xml file of the contributing plug–in.
- **tooltip** – a translatable text representing the action's tool tip. Only used if the action appears in the toolbar.
- **helpContextId** – a unique identifier indicating the help context for this action. If the action appears as a menu item, then pressing F1 while the menu item is highlighted will display help.
- **style** – an optional attribute to define the user interface style type for the action. If defined, the attribute value will be one of the following:

  push – as a regular menu item or tool item.

  radio – as a radio style menu item or tool item. Actions with the radio style within the same menu or toolbar group behave as a radio set. The initial value is specified by the `state` attribute.

  toggle – as a checked style menu item or as a toggle tool item. The initial value is specified by the `state` attribute.
- **state** – an optional attribute indicating the initial state (either `true` or `false`), used when the `style` attribute has the value `radio` or `toggle`.
- **class** – the name of the fully qualified class that implements `org.eclipse.ui.IEditorActionDelegate`

- **enablesFor** – a value indicating the selection count which must be met to enable the action. If this attribute is specified and the condition is met, the action is enabled. If the condition is not met, the action is disabled. If no attribute is specified, the action is enabled for any number of items selected. The following attribute formats are supported:

!	– 0 items selected
?	– 0 or 1 items selected
+	– 1 or more items selected
multiple, 2+	– 2 or more items selected
n	– a precise number of items selected.a precise number of items selected. For example: enablesFor=" 4" enables the action only when 4 items are selected
*	– any number of items selected

- **actionID** – Internal tag for use by the text editors. Should not be used by plug–in developers.

<!ELEMENT menu (separator+ , groupMarker*)>

<!ATTLIST menu

id CDATA #REQUIRED

label CDATA #REQUIRED

path CDATA #IMPLIED>

This element is used to defined a new menu.

- **id** – a unique identifier that can be used to reference this menu.
- **label** – a translatable name used by the Workbench for this new menu. The name should include mnemonic information.
- **path** – the location of the new menu starting from the root of the menu. Each token in the path must refer to an existing menu, except the last token which should represent a named group in the last menu in the path. If omitted, the new menu will be added to the `additions` named group of the menu.

<!ELEMENT separator EMPTY>

<!ATTLIST separator

name CDATA #REQUIRED>

This element is used to create a menu separator in the new menu.

- **name** – the name of the menu separator. This name can later be referenced as the last token in a menu path. Therefore, a separator also serves as named group into which actions and menus can be added.

<!ELEMENT groupMarker EMPTY>

<!ATTLIST groupMarker

name CDATA #REQUIRED>

This element is used to create a named group in the new menu. It has no visual representation in the new menu, unlike the `separator` element.

- **name** – the name of the group marker. This name can later be referenced as the last token in the menu path. It serves as named group into which actions and menus can be added.

<!ELEMENT selection EMPTY>

<!ATTLIST selection

class CDATA #REQUIRED

name CDATA #IMPLIED>

This element is used to help determine the action enablement based on the current selection. Ignored if the `enablement` element is specified.

- **class** – a fully qualified name of the class or interface that each object in the selection must implement in order to enable the action.
- **name** – an optional wild card filter for the name that can be applied to all objects in the selection. If specified and the match fails, the action will be disabled.

<!ELEMENT enablement (and | or | not | objectClass | objectState | pluginState | systemProperty)>

This element is used to define the enablement for the extension.

<!ELEMENT visibility (and | or | not | objectClass | objectState | pluginState | systemProperty)>

This element is used to define the visibility for the extension.

<!ELEMENT and (and | or | not | objectClass | objectState | pluginState | systemProperty)>

This element represent a boolean AND operation on the result of evaluating its two sub−element expressions.

<!ELEMENT or (and | or | not | objectClass | objectState | pluginState | systemProperty)>

This element represent a boolean OR operation on the result of evaluating its two sub−element expressions.

<!ELEMENT not (and | or | not | objectClass | objectState | pluginState | systemProperty)>

This element represent a boolean NOT operation on the result of evaluating its sub−element expressions.

<!ELEMENT objectClass EMPTY>

<!ATTLIST objectClass

name CDATA #REQUIRED>

This element is used to evaluate the class or interface of each object in the current selection. If each object in the selection implements the specified class or interface, the expression is evaluated as true.

- **name** − a fully qualified name of a class or interface. The expression is evaluated as true only if all objects within the selection implement this class or interface.

<!ELEMENT objectState EMPTY>

<!ATTLIST objectState

name CDATA #REQUIRED

value CDATA #REQUIRED>

This element is used to evaluate the attribute state of each object in the current selection. If each object in the selection has the specified attribute state, the expression is evaluated as true. To evaluate this type of expression, each object in the selection must implement, or adapt to,

`org.eclipse.ui.IActionFilter` interface.

- **name** – the name of an object's attribute. Acceptable names reflect the object type, and should be publicly declared by the plug–in where the object type is declared.
- **value** – the required value of the object's attribute. The acceptable values for the object's attribute should be publicly declared.

<!ELEMENT pluginState EMPTY>

<!ATTLIST pluginState

id CDATA #REQUIRED

value (installed|activated) "installed">

This element is used to evaluate the state of a plug–in. The state of the plug–in may be one of the following: `installed` (equivalent to the OSGi concept of "resolved") or `activated` (equivalent to the OSGi concept of "active").

- **id** – the identifier of a plug–in which may or may not exist in the plug–in registry.
- **value** – the required state of the plug–in. The state of the plug–in may be one of the following: `installed` (equivalent to the OSGi concept of "resolved") or `activated` (equivalent to the OSGi concept of "active").

<!ELEMENT systemProperty EMPTY>

<!ATTLIST systemProperty

name CDATA #REQUIRED

value CDATA #REQUIRED>

This element is used to evaluate the state of some system property. The property value is retrieved from the `java.lang.System`.

- **name** – the name of the system property.
- **value** – the required value of the system property.

Examples:

The following is an example of an editor action extension:

```
<extension point=
"org.eclipse.ui.editorActions"
>
<editorContribution id=
"com.xyz.xyzContribution"
targetID=
"com.ibm.XMLEditor"
>
<menu id=
"XYZ"
label=
"&amp;amp;XYZ Menu"
>
<separator name=
"group1"
/>
</menu>
<action id=
"com.xyz.runXYZ"
label=
"&amp;amp;Run XYZ Tool"
menubarPath=
"XYZ/group1"
```

```
toolbarPath=

"Normal/additions"

style=

"toggle"

state=

"true"

icon=

"icons/runXYZ.gif"

tooltip=

"Run XYZ Tool"

helpContextId=

"com.xyz.run_action_context"

class=

"com.xyz.actions.RunXYZ"

>

<selection class=

"org.eclipse.core.resources.IFile"

name=

"*.java"

/>

</action>

</editorContribution>

</extension>
```

In the example above, the specified action will appear as a check box item in the new top−level menu named "XYZ Menu", and as a toggle button in the toolbar. The action is enabled if the selection contains only Java file resources.

The following is an other example of an editor action extension:

```
<extension point=
"org.eclipse.ui.editorActions"
>
<editorContribution id=
"com.xyz.xyz2Contribution"
targetID=
"com.ibm.XMLEditor"
>
<menu id=
"XYZ2"
label=
"&amp;amp;XYZ2 Menu"
path=
"edit/additions"
>
<separator name=
"group1"
/>
</menu>
<action id=
"com.xyz.runXYZ2"
label=
"&amp;amp;Run XYZ2 Tool"
menubarPath=
"edit/XYZ2/group1"
```

```
style=
"push"
icon=
"icons/runXYZ2.gif"
tooltip=
"Run XYZ2 Tool"
helpContextId=
"com.xyz.run_action_context2"
class=
"com.xyz.actions.RunXYZ2"
>
<enablement>
<and>
<objectClass name=
"org.eclipse.core.resources.IFile"
/>
<not>
<objectState name=
"extension"
value=
"java"
/>
</not>
</and>
</enablement>
</action>
```

</editorContribution>

</extension>

In the example above, the specified action will appear as a menu item in the sub−menu named "XYZ2 Menu" in the top level "Edit" menu. The action is enabled if the selection contains no Java file resources.

API Information:

The value of the class attribute must be a fully qualified name of a Java class that implements `org.eclipse.ui.IEditorActionDelegate`. This class is loaded as late as possible to avoid loading the entire plug−in before it is really needed. The method `setActiveEditor` will be called each time an editor of the specified type is activated. Only one set of actions and menus will be created for all instances of the specified editor type, regardless of the number of editor instances currently opened in the Workbench.

This extension point can be used to contribute actions into menus previously created by the target editor. In addition, menus and actions can be contributed to the Workbench window. The identifiers for actions and major groups within the Workbench window are defined in `org.eclipse.ui.IWorkbenchActionConstants`. These should be used as a reference point for the addition of new actions. Top level menus are created by using the following values for the path attribute:

- additions − represents a named group immediately to the left of the `Window` menu.

Omitting the path attribute will result in adding the new menu into the additions menu bar group.

Actions and menus added into these paths will only be shown while the associated editor is active. When the editor is closed, menus and actions will be removed.

The enablement criteria for an action extension is initially defined by `enablesFor`, and also either `selection` or `enablement`. However, once the action delegate has been instantiated, it may control the action enable state directly within its `selectionChanged` method.

Action and menu labels may contain special characters that encode mnemonics using the following rules:

1. Mnemonics are specified using the ampersand ('&') character in front of a selected character in the translated text. Since ampersand is not allowed in XML strings, use `&` character entity.

If two or more actions are contributed to a menu or toolbar by a single extension the actions will appear in the reverse order of how they are listed in the plugin.xml file. This behavior is admittedly unintuitive. However, it was discovered after the Eclipse Platform API was frozen. Changing the behavior now would break every plug−in which relies upon the existing behavior.

The `selection` and `enablement` elements are mutually exclusive. The `enablement` element can replace the `selection` element using the sub−elements `objectClass` and `objectState`. For example, the following:

<selection class=

"org.eclipse.core.resources.IFile"

name=

"*.java"

>

</selection>

can be expressed using:

<enablement>

<and>

<objectClass name=

"org.eclipse.core.resources.IFile"

/>

<objectState name=

"extension"

value=

"java"

/>

</and>

</enablement>

Supplied Implementation:

The Workbench provides a built−in "Default Text Editor". Plug−ins can contribute into this default editor or editors provided by other plug−ins.

Internal and External Editors

Identifier:

org.eclipse.ui.editors

Description:

This extension point is used to add new editors to the workbench. A editor is a visual component within a workbench page. It is typically used to edit or browse a document or input object. To open an editor, the user will typically invoke "Open" on an `IFile`. When this action is performed the workbench registry is consulted to determine an appropriate editor for the file type and then a new instance of the editor type is created. The actual result depends on the type of the editor. The workbench provides support for the creation of internal editors, which are tightly integrated into the workbench, and external editors, which are launched in a separate frame window. There are also various level of integration between these extremes.

In the case of an internal editor tight integration can be achieved between the workbench window and the editor part. The workbench menu and toolbar are pre−loaded with a number of common actions, such as cut, copy, and paste. The active part, view or editor, is expected to provide the implementation for these actions. An internal editor may also define new actions which appear in the workbench window. These actions only appear when the editor is active.

The integration between the workbench and external editors is more tenuous. In this case the workbench may launch an editor but after has no way of determining the state of the external editor or collaborating with it by any means except through the file system.

Configuration Markup:

<!ELEMENT extension (editor*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point −** a fully qualified identifier of the target extension point
- **id −** an optional identifier of the extension instance
- **name −** an optional name of the extension instance

<!ELEMENT editor (contentTypeBinding*)>

<!ATTLIST editor

id CDATA #REQUIRED

name CDATA #REQUIRED

icon CDATA #IMPLIED

extensions CDATA #IMPLIED

class CDATA #IMPLIED

command CDATA #IMPLIED

launcher CDATA #IMPLIED

contributorClass CDATA #IMPLIED

default (true | false) "false"

filenames CDATA #IMPLIED

symbolicFontName CDATA #IMPLIED

matchingStrategy CDATA #IMPLIED>

- **id –** a unique name that will be used to identify this editor
- **name –** a translatable name that will be used in the UI for this editor
- **icon –** a relative name of the icon that will be used for all resources that match the specified extensions. An icon is not required if you specify a command rather than a class. In that case, the workbench will use the icon provided by the operating system.
- **extensions –** an optional field containing the list of file types understood by the editor. This is a string containing comma separate file extensions. For instance, an editor which understands hypertext documents may register for "htm, html".
- **class –** the name of a class that implements `org.eclipse.ui.IEditorPart`. The attributes `class`, `command`, and `launcher` are mutually exclusive. If this attribute is defined then `contributorClass` should also be defined.
- **command –** a command to run in order to launch an external editor. The executable command must be located on the system path or in the plug–in's directory. The attributes `class`, `command`, and `launcher` are mutually exclusive.
- **launcher –** the name of a class which that implements `org.eclipse.ui.IEditorLauncher`. A launcher will open an external editor. The attributes `class`, `command`, and `launcher` are mutually exclusive.
- **contributorClass –** the name of a class that implements `org.eclipse.ui.IEditorActionBarContributor`. This attribute should only be defined if the `class` attribute is defined. This class is used to add new actions to the workbench menu and tool bar which reflect the features of the editor type.
- **default –** if true, this editor will be used as the default editor for the type. This is only relevant in a case where more than one editor is registered for the same type. If an editor is not the default for the type, it can still be launched using "Open with..." submenu for the selected resource.

  Please note that this attribute is only honored for filename and extension associations at this time. It will not be honored for content type bindings. Content type–based resolution will occur on a first come, first serve basis and is not explicitly specified.
- **filenames –** an optional field containing the list of file names understood by the editor. This is a string containing comma separate file names. For instance, an editor which understands specific hypertext documents may register for "ejb.htm, ejb.html".

- **symbolicFontName** – the symbolic name of a font. The symbolic font name must be the id of a defined font (see org.eclipse.ui.fontDefinitions). If this attribute is missing or invalid then the font name is the value of "org.eclipse.jface.textfont" in the editor's preferences store. If there is no preference store or the key is not defined then the JFace text font will be used. The editor implementation decides if it uses this symbolic font name to set the font.
- **matchingStrategy** – the name of a class that implements `org.eclipse.ui.IEditorMatchingStrategy`. This attribute should only be defined if the `class` attribute is defined. This allows the editor extension to provide its own algorithm for matching the input of one of its editors to a given editor input.

<!ELEMENT contentTypeBinding EMPTY>

<!ATTLIST contentTypeBinding

contentTypeId CDATA #REQUIRED>

Advertises that the containing editor understands the given content type and is suitable for editing files of that type.

- **contentTypeId** – the content type identifier

Examples:

The following is an example of an internal editor extension definition:

<extension point=

"org.eclipse.ui.editors"

>

<editor id=

"com.xyz.XMLEditor"

name=

"Fancy XYZ XML editor"

icon=

"./icons/XMLEditor.gif"

extensions=

"xml"

class=

"com.xyz.XMLEditor"

contributorClass=

"com.xyz.XMLEditorContributor"

symbolicFontName=

"org.eclipse.jface.textfont"

default=

"false"

>

</editor>

</extension>

API Information:

If the command attribute is used, it will be treated as an external program command line that will be executed in a platform−dependent manner.

If the launcher attribute is used the editor will also be treated as an external program. In this case the specified class must implement `org.eclipse.ui.IEditorLauncher`. The launcher will be instantiated and then `open(IFile file)` will be invoked to launch the editor.

If the class attribute is used, the workbench will assume that it is an internal editor and the specified class must implement `org.eclipse.ui.IEditorPart`. It is common practice to `subclass org.eclipse.ui.EditorPart` when defining a new editor type. It is also necessary to define a `contributorClass` attribute. The specified class must implement `org.eclipse.ui.IEditorActionBarContributor`, and is used to add new actions to the workbench menu and tool bar which reflect the features of the editor type.

Within the workbench there may be more than one open editor of a particular type. For instance, there may be one or more open Java Editors. To avoid the creation of duplicate actions and action images the editor concept has been split into two. An `IEditorActionBarContributor` is responsible for the creation of actions. The editor is responsible for action implementation. Furthermore, the contributor is shared by each open editor. As a result of this design there is only one set of actions for one or more open editors.

The contributor will add new actions to the workbench menu and toolbar which reflect the editor type. These actions are shared and, when invoked, act upon the active editor. The active editor is passed to the contributor by invoking `IEditorActionBarContributor.setActiveEditor`. The identifiers for actions and

major groups within the workbench window are defined in `org.eclipse.ui.IWorkbenchActionConstants`. These should be used as a reference point for the addition of new actions. Top level menus are created by using the following values for the path attribute:

- additions – represents a group to the left of the Window menu.

Actions and menus added into these paths will only be shown while the associated editor is active. When the editor is closed, menus and actions will be removed.

Supplied Implementation:

The workbench provides a "Default Text Editor". The end user product may contain other editors as part of the shipping bundle. In that case, editors will be registered as extensions using the syntax described above.

Element Factories

Identifier:

org.eclipse.ui.elementFactories

Description:

This extension point is used to add element factories to the workbench. An element factory is used to recreate `IAdaptable` objects which are persisted during workbench shutdown.

As an example, the element factory is used to persist editor input. The input for an editor must implement `org.eclipse.ui.EditorInput`. The life cycle of an `IEditorInput` within an editor has a number of phases.

1. The initial input for an editor is passed in during editor creation.
2. On shutdown the workbench state is captured. In this process the workbench will create a memento for each open editor and its input. The input is saved as a two part memento containing a factory ID and any primitive data required to recreate the element on startup. For more information see the documentation on `org.eclipse.ui.IPersistableElement`.
3. On startup the workbench state is read and the editors from the previous session are recreated. In this process the workbench will recreate the input element for each open editor. To do this it will map the original factory ID for the input element to a concrete factory class defined in the registry. If a mapping exists, and the factory class is valid, an instance of the factory class is created. Then the workbench asks the factory to recreate the original element from the remaining primitive data within the memento. The resulting `IAdaptable` is cast to an `IEditorInput` and passed to the new editor.

Configuration Markup:

<!ELEMENT extension (factory*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT factory EMPTY>

<!ATTLIST factory

id CDATA #REQUIRED

class CDATA #REQUIRED>

- **id** – a unique name that will be used to identify this factory.
- **class** – a fully qualified name of a class that implements `org.eclipse.ui.IElementFactory`

Examples:

The following is an example of an element factory extension:

<extension point =

"org.eclipse.ui.elementFactories"

>

<factory id =

"com.xyz.ElementFactory"

class=

"com.xyz.ElementFactory"

>

</factory>

</extension>

API Information:

The value of the `class` attribute must be a fully qualified name of a class that implements `org.eclipse.ui.IElementFactory`. An instance of this class must create an `IAdaptable` object from a workbench memento.

Supplied Implementation:

The workbench provides an `IResource` factory. Additional factories should be added to recreate other `IAdaptable` types commonly found in other object models, such as the Java Model.

Encodings

Identifier:

org.eclipse.ui.encodings

Since:

3.1

Description:

A list of encodings that will be made available to the user for selection of file encodings. This list corresponds to the values of Charset#getName() for a series of Charsets. Those that are specified but not avaiable will be indicated as such in the workbench and will not be selectable by the user.

Configuration Markup:

<!ELEMENT extension (encoding*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

<!ELEMENT encoding EMPTY>

<!ATTLIST encoding

name CDATA #REQUIRED>

- **name –** The name of the charset. If this value is sent to Charset#getName(String) the corresponding Charset would be returned.

Examples:

[Enter extension point usage example here.]

API Information:

[Enter API information here.]

Supplied Implementation:

[Enter information about supplied implementation of this extension point.]

Export Wizards

Identifier:

org.eclipse.ui.exportWizards

Description:

This extension point is used to register export wizard extensions. Export wizards appear as choices within the "Export Dialog", and are used to export resources from the workbench.

Wizards may optionally specify a description subelement whose body should contain short text about the wizard.

Configuration Markup:

<!ELEMENT extension (wizard*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT wizard (description? , selection*)>

<!ATTLIST wizard

id CDATA #REQUIRED

name CDATA #REQUIRED

class CDATA #REQUIRED

icon CDATA #IMPLIED>

an element that will be used to create export wizard

- **id –** a unique name that will be used to identify this wizard
- **name –** a translatable name that will be used in the dialog box to represent this wizard

- **class** – a fully qualified name of the class that implements `org.eclipse.ui.IExportWizard` interface
- **icon** – a relative name of the icon that will be used alongside the wizard name in the export engine listing.

<!ELEMENT description (#PCDATA)>

an optional subelement whose body should represent a short description of the export engine functionality.

<!ELEMENT selection EMPTY>

<!ATTLIST selection

name CDATA #IMPLIED

class CDATA #REQUIRED>

an optional element that restricts the types and names of objects that can be selected when the wizard is invoked.

- **name** – an optional name filter. Each object in the workbench selection must match the name filter to be passed to the wizard.
- **class** – a fully qualified class name. If each object in the workbench selection implements this interface the selection will be passed to the wizard. Otherwise, an empty selection is passed.

Examples:

The following is an example of an export extension definition:

<extension point=

"org.eclipse.ui.exportWizards"

>

<wizard id=

"com.xyz.ExportWizard1"

name=

"XYZ Web Exporter"

class=

"com.xyz.exports.ExportWizard1"

icon=

"./icons/import1.gif"

>

<description>

A simple engine that exports Web project

</description>

<selection class=

"org.eclipse.core.resources.IProject"

/>

</wizard>

</extension>

API Information:

The value of the `class` attribute must be a name of the class that implements `org.eclipse.ui.IExportWizard`.

Supplied Implementation:

The workbench comes preloaded with basic export engines for files and directories.

Font Definitions

Identifier:

org.eclipse.ui.fontDefinitions

Since:

Release 2.1

Description:

This extension point is used to register fonts with the JFace FontRegistry and with the workbench preference store for use by the Fonts preference page. This extension point has been deprecated in 3.0. You should now add fontDefinition elements to org.eclipse.ui.themes.

Configuration Markup:

<!ELEMENT extension (fontDefinition*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT fontDefinition (description?)>

<!ATTLIST fontDefinition

id CDATA #REQUIRED

label CDATA #REQUIRED

value CDATA #IMPLIED

categoryId CDATA #IMPLIED

defaultsTo CDATA #IMPLIED>

- **id –** a unique name that can be used to identify this font definition.
- **label –** a translatable name of the font to be presented to the user.
- **value –**

the font value. This is in the form: `fontname-style-height` where `fontname` is the name of a font, `style` is a font style (one of `"regular"`, `"bold"`, `"italic"`, or `"bold italic"`) and `height` is an integer representing the font height.

Example: `Times New Roman-bold-36`.

Only one (or neither) of value or defaultsTo may be used.

- **categoryId –** the optional id of the presentation category this font belongs to.
- **defaultsTo –** the id of another font definition that is the default setting for the receiver. When there is no preference for this font the font registry will have the value of defaultsTo set for it in the registry.

  Only one or neither of value or defaultsTo may be used.

<!ELEMENT description EMPTY>

a short description of the fonts usage

Examples:

Following is an example of an a font definition extension:

<extension point=

"org.eclipse.ui.fontDefinition"

>

<fontDefinition id=

"org.eclipse.examples.textFont"

label=

"Text"

>

<description>

The text font

</description>

</fontDefinition>

Since:

```
<fontDefinition id=

"org.eclipse.examples.userFont"

label=

"User"

defaultsTo=

"org.eclipse.jface.textFont"

>

<description>

The user font

</description>

</fontDefinition>

</extension>
```

API Information:

The defaultsTo tag is used as a directive by the Workbench to set the value of the font definition to the value of defaultsTo whenever the defaultsTo fontDefinition is updated. This only occurs if the fontDefinition is at its default value − once it is set by the user this updates will not occur. The workbench provides 4 fonts:

org.eclipse.jface.bannerfont. The banner font is used in wizard banners.
org.eclipse.jface.dialogfont. The dialog font is the font for widgets in dialogs.
org.eclipse.jface.headerfont. The header font is used for section headers in composite text pages.
org.eclipse.jface.textfont. The text font is used by text editors.

Supplied Implementation:

The workbench provides the font definitions for the text, dialog, banner and header fonts.

Handlers

Identifier:

org.eclipse.ui.handlers

Since:

3.1

Description:

The handlers extension point is an elaboration of the experimental `handlerSubmission` element defined in Eclipse 3.0. A handler is the behaviour of a command at a particular point in time. A command can have zero or more handlers associated with it. At any one point in time, however, a command will either have no active handler or one active handler. The active handler is the one which is currently responsible for carrying out the behaviour of the command. This is very similar to the concept of an action handler and a retargettable action.

The handlers extension point allows a plug−in developer to specify a handler that should become active and/or enabled under certain conditions. If a handler is inactive, then no command will delegate its behaviour to the handler. If a handler is disabled, then the handler will not be asked to execute; execution of the handler is blocked. The conditions are defined using the expression language facility added during 3.0. They are expressed using `activeWhen` and `enabledWhen` clauses.

The workbench provides some variables that these expressions can rely on. The variables supported are: the active contexts, the active editor, the active part and the current selection. While not supported in this initial design, it is easy to see how it would be possible to add other variables or even allow plug−in developers to contribute other variables.

A handler that specifies no conditions is a default handler. A default handler is only active if no other handler has all of its conditions satisfied. If two handlers still have conditions that are satisfied, then the conditions are compared. The idea is to select a handler whose condition is more specific or more local. To do this, the variables referred to by the condition are looked at. The condition that refers to the most specific variable "wins". The order of specificity (from least specific to most specific) is defined in `org.eclipse.ui.ISources`.

If this still doesn't resolve the conflict, then no handler is active. If a particular tracing option is turned on, then this leads to a message in the log. A conflict can also occur if there are two default handlers. It is the responsibility of the plug−in developers and integration testers to ensure that this does not happen. These conditions are used to avoid unnecessary plug−in loading. These handler definitions are wrapped in a proxy. For a proxy to load its underlying handler, two things must happen: the conditions for the proxy must be met so that it becomes active, and the command must be asked to do something which it must delegate (e.g., execute()).

Configuration Markup:

<!ELEMENT extension (handler)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

<!ELEMENT handler (activeWhen? | class? | enabledWhen?)>

<!ATTLIST handler

commandId CDATA #REQUIRED

class CDATA #IMPLIED>

- **commandId –**
- **class –**

<!ELEMENT activeWhen (not | and | or | instanceof | test | systemTest | equals | count | with | resolve | adapt | iterate)>

<!ELEMENT enabledWhen (not | and | or | instanceof | test | systemTest | equals | count | with | resolve | adapt | iterate)>

<!ELEMENT class (parameter)>

<!ATTLIST class

class CDATA #IMPLIED>

- **class –**

<!ELEMENT parameter EMPTY>

<!ATTLIST parameter

name CDATA #REQUIRED

value CDATA #REQUIRED>

Since:

- **name** –
- **value** –

<!ELEMENT enablement (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

A generic root element. The element can be used inside an extension point to define its enablement expression. The children of an enablement expression are combined using the and operator.

<!ELEMENT not (not | and | or | instanceof | test | systemTest | equals | count | with | resolve | adapt | iterate)>

This element represent a NOT operation on the result of evaluating it's sub−element expression.

<!ELEMENT and (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

This element represent an AND operation on the result of evaluating all it's sub−elements expressions.

<!ELEMENT or (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

This element represent an OR operation on the result of evaluating all it's sub−element expressions.

<!ELEMENT instanceof EMPTY>

<!ATTLIST instanceof

value CDATA #REQUIRED>

This element is used to perform an instanceof check of the object in focus. The expression returns EvaluationResult.TRUE if the object's type is a sub type of the type specified by the attribute value. Otherwise EvaluationResult.FALSE is returned.

- **value** – a fully qualified name of a class or interface.

<!ELEMENT test EMPTY>

<!ATTLIST test

property CDATA #REQUIRED

args CDATA #IMPLIED

value CDATA #IMPLIED>

This element is used to evaluate the property state of the object in focus. The set of testable properties can be extended using the propery tester extension point. The test expression returns EvaluationResult.NOT_LOADED if teh property tester doing the actual testing isn't loaded yet.

- **property** – the name of an object's property to test.
- **args** – additional arguments passed to the property tester. Multiple arguments are seperated by commas. Each individual argument is converted into a Java base type using the same rules as defined for the value attribute of the test expression.
- **value** – the expected value of the property. Can be omitted if the property is a boolean property. The test expression is supposed to return EvaluationResult.TRUE if the property matches the value and EvaluationResult.FALSE otherwise. The value attribute is converted into a Java base type using the following rules:
  - the string "true" is converted into Boolean.TRUE
  - the string "false" is converted into Boolean.FALSE
  - if the string contains a dot then the interpreter tries to convert the value into a Float object. If this fails the string is treated as a java.lang.String
  - if the string only consists of numbers then the interpreter converts the value in an Integer object.
  - in all other cases the string is treated as a java.lang.String
  - the conversion of the string into a Boolean, Float, or Integer can be suppressed by surrounding the string with single quotes. For example, the attribute value="'true'" is converted into the string "true"

<!ELEMENT systemTest EMPTY>

<!ATTLIST systemTest

property CDATA #REQUIRED

value CDATA #REQUIRED>

Tests a system property by calling the System.getProperty method and compares the result with the value specified through the value attribute.

Since:

- **property** – the name of an system property to test.
- **value** – the expected value of the property. The value is interpreted as a string value.

<!ELEMENT equals EMPTY>

<!ATTLIST equals

value CDATA #REQUIRED>

This element is used to perform an equals check of the object in focus. The expression returns EvaluationResult.TRUE if the object is equal to the value provided by the attribute value. Otherwise EvaluationResult.FALSE is returned.

- **value** – the operatand of the equals tests. The value provided as a string is converted into a Java base type using the same rules as for the value attribute of the test expression.

<!ELEMENT count EMPTY>

<!ATTLIST count

value CDATA #REQUIRED>

This element is used to test the number of elements in a collection.

- **value** – an expression to specify the number of elements in a list. Following wildcard characters can be used:
  *
  : any number of elements
  ?
  : no elements or one element
  +
  : one or more elements
  !
  : no elements
  integer value
  : the list must contain the exact number of elements

<!ELEMENT with (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST with

variable CDATA #REQUIRED>

This element changes the object to be inspected for all its child element to the object referneced by the given variable. If the variable can not be resolved then the expression will throw a ExpressionException when evaluating it. The children of a with expression are combined using the and operator.

- **variable –** the name of the variable to be used for further inspection. It is up to the evaluator of an extension point to provide the variable in the variable pool.

<!ELEMENT resolve (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST resolve

variable CDATA #REQUIRED

args CDATA #IMPLIED>

This element changes the object to be inspected for all its child element to the object referneced by the given variable. If the variable can not be resolved then the expression will throw a ExpressionException when evaluating it. The children of a with expression are combined using the and operator.

- **variable –** the name of the variable to be resolved. This variable is then used as the object in focus for child element evaluation. It is up to the evaluator of an extension point to provide a corresponding variable resolver (see IVariableResolver) through the evaluation context passed to the root expression element when evaluating the expression.
- **args –** additional arguments passed to the variable resolver. Multiple arguments are seperated by commas. Each individual argument is converted into a Java base type using the same rules as defined for the value attribute of the test expression.

<!ELEMENT adapt (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST adapt

type CDATA #REQUIRED>

This element is used to adapt the object in focus to the type specified by the attribute type. The expression returns not loaded if either the adapter or the type referenced isn't loaded yet. It throws a ExpressionException during evaluation if the type name doesn't exist at all. The children of an adapt expression are combined using the and operator.

- **type** – the type to which the object in focus is to be adapted.

<!ELEMENT iterate (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST iterate

operator (orland) >

This element is used to iterate over a variable that is of type java.util.Collection. If the object in focus is not of type java.util.Collection then an ExpressionException will be thrown while evaluating the expression.

- **operator** – either "and" or "or". The operator defines how the child elements will be combined. If not specified, "and" will be used.

Examples:

<extension point=

"org.eclipse.ui.handlers"

>

<handler commandId=

"commandId"

class=

"org.eclipse.compare.Command"

>

<activeWhen>

<with variable=

"selection"

>

<count value=

"1"

/>

<iterate operator=

"and"

>

<adapt type=

"IResource"

/>

</iterate>

</with>

</activeWhen>

</handler>

</extension>

To further avoid plug−in loading, it is possible to specify when the handler is enabled. If the proxy has not yet loaded the handler, then only the expressions syntax is used to decide if the handler is enabled. If the proxy has loaded the handler, then the expressions syntax is consulted first. If the expressions syntax evaluates to true, then the handler is asked if it is enabled. (This is a short−circuit Boolean "and" operation between the expressions syntax and the handler's enabled state.)

<extension point=

"org.eclipse.ui.handlers"

>

<handler commandId=

"commandId"

class=

"org.eclipse.Handler"

>

<enabledWhen>

<with variable=

"context"

>

<property id=

"id"

value=

"debugging"

/>

</with>

</enabledWhen>

</handler>

</extension>

API Information:

All handlers implement `org.eclipse.core.commands.IHandler`. Within the workbench, it is possible to activate and deactivate handlers using the `org.eclipse.ui.handlers.IHandlerService` interface. This interface can be retrieved from supporting workbench objects, such as `IWorkbench` itself. To retrieve the service, you would make a call like `IWorkbench.getAdapter(IHandlerService.class)`.

It is also possible to activate and deactive handlers using legacy code in the workbench. This can be done through the legacy mechanism shown below. This mechanism is useful to clients who are using actions to contribute to menus or toolbars.

```
IWorkbenchPartSite mySite;
IAction myAction;

myAction.setActionDefinitionId(commandId);
IKeyBindingService service = mySite.getKeyBindingService();
service.registerAction(myAction);
```

HelpSupport

Identifier:

org.eclipse.ui.helpSupport

Since:

3.0 (originally named org.eclipse.help.support)

Description:

This extension point is for contributing the help system UI. The platform should be configured with no more than one help system UI.

Configuration Markup:

<!ELEMENT extension (config?)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

<!ELEMENT config EMPTY>

<!ATTLIST config

class CDATA #REQUIRED>

- **class** – the implementation class for displaying online and context−sensitive help. This class must implement the `org.eclipse.ui.help.AbstractHelpUI` interface.

Examples:

The following is a sample usage of the help support extension point:

<extension point=

"org.eclipse.ui.helpSupport"

>

<config class=

"com.example.XYZHelpUI"

/>

</extension>

API Information:

The supplied class must implement a subclass of `org.eclipse.ui.help.AbstractHelpUI`. Implementation of the abstract methods in that class determine what happens when a user asks for online help or context−sensitive help. The implementation should access contributed help information using org.eclipse.help.HelpSystem.

Supplied Implementation:

The `org.eclipse.help.ui` plug−in contains an implementation of the help system UI.

Marker Help

Identifier:

org.eclipse.ui.ide.markerHelp

Since:

3.0 (originally added in release 2.0 as org.eclipse.ui.markerHelp)

Description:

This extension point is used to associate a help context id with a specific "kind" of marker (a marker of a certain type or having certain attribute values).

Configuration Markup:

<!ELEMENT extension (markerHelp*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT markerHelp (attribute*)>

<!ATTLIST markerHelp

markerType CDATA #IMPLIED

helpContextId CDATA #REQUIRED>

- **markerType –** the unique type of the marker for which the help context applies.
- **helpContextId –** the unique id of the help context.

<!ELEMENT attribute EMPTY>

<!ATTLIST attribute

name CDATA #REQUIRED

value CDATA #REQUIRED>

- **name** – the name of the attribute whose value is being specified.
- **value** – the specified value of the attribute.

Examples:

The following is an example of a marker help extension (note the sub−element and the way attributes are used):

```
<extension point=

"org.eclipse.ui.ide.markerHelp"

>

<markerHelp markerType=

"org.eclipse.ui.examples.readmetool.readmemarker"

helpContextId=

"org.eclipse.ui.examples.readmetool.marker_example1_context"

>

<attribute name=

"org.eclipse.ui.examples.readmetool.id"

value=

"1234"

/>

</markerHelp>

</extension>
```

In the example above, a help context id is associated with markers of type org.eclipse.ui.examples.readmetool.readmemarker whose org.eclipse.ui.examples.readmetool.id attribute has a value of 1234.

API Information:

It is up to the developer to ensure that only a single help context id is supplied for a given marker. If two or more help context ids are supplied for a given kind of marker, the workbench does not define which will be returned. However the workbench does define that the "most specific" context id will always be returned for a given marker. That is, a context id associated with three matching attribute values will be returned before a context id associated with only two.

Marker Image Providers

Identifier:

org.eclipse.ui.ide.markerImageProviders

Since:

3.0 (originally added in release 2.1 as org.eclipse.ui.markerImageProviders)

Description:

The markerImageProvider extension point is the point for specifying the images for marker types in the defining plug−in.

Configuration Markup:

<!ELEMENT extension (imageprovider*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

<!ELEMENT imageprovider EMPTY>

<!ATTLIST imageprovider

id CDATA #REQUIRED

markertype CDATA #REQUIRED

icon CDATA #IMPLIED

class CDATA #IMPLIED>

- **id –** a unique name that can be used to identify this markerImageProvider.
- **markertype –** The markertype is the id of the type defined in `org.eclipse.core.resources.IMarker` that this definition is applied to.
- **icon –** If there is no class defined the icon attribute is used to define the icon that will be applied to this type of marker.
- **class –** The class is the fully qualifed name of the class that will be used to look up an image. This class must implement `IMarkerImageProvider`.

Examples:

The following an example of the two forms of marker image providers. The first one is one where the image does not change and is declared directly. For the second one the image must be determined by an instance of IMarkerImageProvider.

```
<extension point=

"org.eclipse.ui.ide.markerImageProviders"

>

<imageprovider markertype=

"org.eclipse.core.resources.taskmarker"

icon=

"taskicon.gif"

id=

"myPlugin.declarativeMarkerProvider"

>

</imageprovider>

<imageprovider markertype=

"org.eclipse.core.resources.problemmarker"

class=

"myPlugin.MyIMarkerImageProvider"

id=

"myPlugin.implementedMarkerProvider"

>

</imageprovider>

</extension>
```

API Information:

[Enter API information here.]

Supplied Implementation:

[Enter information about supplied implementation of this extension point.]

Marker Resolutions

Identifier:

org.eclipse.ui.ide.markerResolution

Since:

3.0 (originally added in release 2.0 as org.eclipse.ui.markerResolution)

Description:

This extension point is used to associate a marker resolution generator with a specific "kind" of marker. (a marker of a certain type or having certain attribute values).

Configuration Markup:

<!ELEMENT extension (markerResolutionGenerator*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT markerResolutionGenerator (attribute*)>

<!ATTLIST markerResolutionGenerator

class CDATA #REQUIRED

markerType CDATA #IMPLIED>

- **class –** the name of the class implementing IMarkerResolutionGenerator
- **markerType –** the type of marker for which the help context applies.

<!ELEMENT attribute EMPTY>

<!ATTLIST attribute

name CDATA #REQUIRED

value CDATA #REQUIRED>

- **name** – the name of the attribute whose value is being specified.
- **value** – the specified value of the attribute.

Examples:

The following is an example of a marker resolution generator extension (note the sub–element and the way attributes are used):

<extension point=

"org.eclipse.ui.ide.markerResolution"

>

<markerResolutionGenerator class=

"org.eclipse.ui.examples.readmetool.ReadmeMarkerResolutionGenerator"

markerType=

"org.eclipse.ui.examples.readmetool.readmemarker"

>

<attribute name=

"org.eclipse.ui.examples.readmetool.id"

value=

"1234"

/>

</markerResolutionGenerator>

</extension>

In the example above, a marker resolution generator is associated with markers of type org.eclipse.ui.examples.readmetool.readmemarker whose org.eclipse.ui.examples.redmetool.id attribute has a value of 1234.

API Information:

More than one marker help generator may be supplied for a given marker.

Project Nature Images

Identifier:

org.eclipse.ui.ide.projectNatureImages

Since:

3.0 (originally added in release 1.0 as org.eclipse.ui.projectNatureImages)

Description:

This extension point is used to associate an image with a project nature. The supplied image is used to form a composite image consisting of the standard project image combined with the image of its nature. The supplied image is drawn over the top right corner of the base image.

Configuration Markup:

<!ELEMENT extension (image*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT image (description?)>

<!ATTLIST image

id CDATA #REQUIRED

natureId CDATA #REQUIRED

icon CDATA #REQUIRED>

- **id –** a unique name that will be used to identify this nature image.
- **natureId –** the unique name of the nature for which the image is being supplied.
- **icon –** a relative name of the image that will be associated with this perspective.

<!ELEMENT description (#PCDATA)>

a short description of what this image represents.

Examples:

The following is an example of a nature image extension:

```
<extension point=
"org.eclipse.ui.ide.projectNatureImages"
>
<image id=
"org.eclipse.ui.javaNatureImage"
natureId=
"Resource"
icon=
"icons/javaNature.gif"
>
</image>
</extension>
```

API Information:

The value of the `natureId` attribute is the nature id as defined by the plugin creating the project.

Resource Filters

Identifier:

org.eclipse.ui.ide.resourceFilters

Since:

3.0 (originally added in release 1.0 as org.eclipse.ui.resourceFilters)

Description:

This extension point is used to add predefined filters to views which show resources, such as the Navigator view. These filters can be selected to hide resources whose names match the filter's pattern.

Configuration Markup:

<!ELEMENT extension (filter*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT filter (description?)>

<!ATTLIST filter

pattern CDATA #REQUIRED

selected (true | false) "false">

- **pattern –** the pattern to match. May contain * and ? wildcards.
- **selected –** "true" if the pattern should be selected by default, "false" or undefined if not.

<!ELEMENT description (#PCDATA)>

the description of the purpose of this filter.

Examples:

The following is an example of a resource filter extension, which filters out class files, and is selected by default:

<extension point=

"org.eclipse.ui.ide.resourceFilters"

>

<filter pattern=

"*.class"

selected=

"true"

/>

</extension>

Import Wizards

Identifier:

org.eclipse.ui.importWizards

Description:

This extension point is used to register import wizard extensions. Import wizards appear as choices within the "Import Dialog" and are used to import resources into the workbench.

Wizards may optionally specify a description subelement whose body should contain short text about the wizard.

Configuration Markup:

<!ELEMENT extension (wizard*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT wizard (description? , selection*)>

<!ATTLIST wizard

id CDATA #REQUIRED

name CDATA #REQUIRED

class CDATA #REQUIRED

icon CDATA #IMPLIED>

an element that will be used to create import wizard

- **id –** a unique name that will be used to identify this wizard
- **name –** a translatable name that will be used in the dialog box to represent this wizard

- **class** – a fully qualified name of the class that implements `org.eclipse.ui.IImportWizard` interface
- **icon** – a relative name of the icon that will be used alongside the wizard name in the import engine listing.

<!ELEMENT description (#PCDATA)>

an optional subelement whose body should represent a short description of the import engine functionality.

<!ELEMENT selection EMPTY>

<!ATTLIST selection

name CDATA #IMPLIED

class CDATA #REQUIRED>

an optional element that restricts the types and names of objects that can be selected when the wizard is invoked.

- **name** – an optional name filter. Each object in the workbench selection must match the name filter to be passed to the wizard.
- **class** – fully qualified class name. If each object in the workbench selection implements this interface the selection will be passed to the wizard. Otherwise, an empty selection is passed.

Examples:

The following is an example of an import extension definition:

<extension point=

"org.eclipse.ui.importWizards"

>

<wizard id=

"com.xyz.ImportWizard1"

name=

"XYZ Web Scraper"

class=

"com.xyz.imports.ImportWizard1"

icon=

"./icons/import1.gif"

>

<description>

A simple engine that searches the Web and imports files

</description>

<selection class=

"org.eclipse.core.resources.IResource"

/>

</wizard>

</extension>

API Information:

The value of the `class` attribute must represent a name of the class that implements `org.eclipse.ui.IImportWizard`.

Supplied Implementation:

The workbench comes preloaded with the basic import engines for files and directories.

Intro Part Configuration

Identifier:

org.eclipse.ui.intro.config

Since:

3.0

Description:

This extension point is used to register an intro configuration. This configuration provides presentation implementations and content for a given intro contribution. An intro appears when the workbench is first launched and as a choice from the "Help" menu. The intro is typically used to introduce a user to a product built on Eclipse.

The intros are organized into pages which usually reflect a particular category of introductory material. For instance, a What's New page may introduce new concepts or functionality since previous versions. The content defined by one intro configuration can be referenced and extended from other plug−ins using the org.eclipse.ui.intro.configExtension extension point.

Configuration Markup:

<!ELEMENT extension (handle? , config+)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point −** a fully qualified identifier of the target extension point
- **id −** an optional identifier of the extension instance
- **name −** an optional name of the extension instance

<!ELEMENT config (presentation)>

<!ATTLIST config

introId CDATA #REQUIRED

id CDATA #REQUIRED

content CDATA #REQUIRED>

A config element can be used to configure a customizable Intro Part. A config element must specify an id, an introId, and a content file. The intro content file is an XML file that describes the pages, groups and links that the intro has.

- **introId** – the id of an intro part contribution that this configuration will be associated with.
- **id** – a unique name that can be used to identify this intro configuration
- **content** – an intro content file. The content file is an XML file that contains the specifics of the intro (intro content file format specification). The content file is parsed at run time by the intro framework. Based on the settings in this file, a certain number of pages, groups, and links are shown to the user when the intro is opened.

<!ELEMENT presentation (implementation+ , launchBar?)>

<!ATTLIST presentation

home−page−id CDATA #REQUIRED

standby−page−id CDATA #IMPLIED>

Presentation element that defines all the possible implementations of an intro part's presentation. It can have one or more implementation defined in it. Only one implementation will be chosen at startup, based the os/ws attributes of the implementations. Otherwise, the first one with no os/ws attributes defined will be chosen.

- **home−page−id** – the id of the home (root) page, which is the first page of the introduction. This page can be used as an entry point to the other main pages that make up the intro.
- **standby−page−id** – an optional attribute to define the id of the standby page. The standby page will be shown to the user when the Intro is set to standby.

<!ELEMENT implementation (head?)>

<!ATTLIST implementation

kind (swt|html)

style CDATA #IMPLIED

os CDATA #IMPLIED

ws CDATA #IMPLIED>

The presentation of the Platform's out of the box experience has two implementations. One of them is SWT Browser based, while the other is UI Forms based. The customizable intro part can be configured to pick one of those two presentations based on the current OS and WS. The type of the implementation can be swt or

html.

- **kind –** Specifies the type of this implementation. The swt kind indicates a UI Forms based implementation, and the html kind indicates an SWT Browser based implementation
- **style –** The shared style that will be applied to all pages presented by this intro presentation implementation.
- **os –** The optional operating system specification used when choosing the presentation's implementation. It can be any of the os designators defined by Eclipse, e.g., win32, linux, etc (see Javadoc for `org.eclipse.core.runtime.Platform`).
- **ws –** The optional windowing system specification used when choosing the presentation's implementation. It can be any of the ws designators defined by Eclipse (see Javadoc for `org.eclipse.core.runtime.Platform`).

<!ELEMENT launchBar (handle? , shortcut*)>

<!ATTLIST launchBar

location (left|bottom|right|fastview) "fastview"

bg CDATA #IMPLIED

fg CDATA #IMPLIED>

Launch bar works in conjuction with 'switchToLaunchBar' command. When executed, the command will close intro and create a launch bar in one of the trim areas of the main window. The launch bar will contain at least one button that would allow users to return to the last intro page. Optionally, it can contain additional buttons defined using `shortcut` elements. These buttons can be used to quickly go to a certain intro page. since 3.1

- **location –** the location of the launch bar relative to the content area in the main window (one of `left`, `bottom` or `right`). If the choice is set to `fastview` (the default), the launch bar will be initially created where the fast view bar is.
- **bg –** background color that will be applied to the launch bar if specified. Expected format is hex, as in: #rrggbb (for example, #80a4a1).
- **fg –** foreground color that will be used to paint the border of the launch bar if specified. Expected format is hex, as in: #rrggbb (for example, #80a4a1).

<!ELEMENT shortcut EMPTY>

<!ATTLIST shortcut

tooltip CDATA #IMPLIED

Since:

icon CDATA #REQUIRED

url CDATA #REQUIRED>

a button will be created in the intro launch bar for each shortcut element. The URL defined in the shortcut will be executed when the button is pressed. since 3.1

- **tooltip –** a tooltip to be used for this shortcut
- **icon –** a relative path to the image icon that should be used for this shortcut in the intro launch bar.
- **url –** the URL to execute when the shortcut in the launch bar is activated

<!ELEMENT handle EMPTY>

<!ATTLIST handle

close (true | false) "true"

image CDATA #IMPLIED>

an optional element that controls how launch bar handle is rendered. since 3.1

- **close –** an optional flag that controls if the handle will have a close button (default is `true`).
- **image –** a plug−in−relative path of the branding image to be tiled on the handle.

<!ELEMENT introContent (page+ , group* , extensionContent*)>

The introContent element defines the body of the intro content file. The content file is made up of pages, shared groups that can be included in multiple pages, and extensions to anchor points defined in other configurations.

<!ELEMENT page (group* | link* | text* | head* | img* | include* | html* | title? | anchor* | contentProvider*)>

<!ATTLIST page

url CDATA #IMPLIED

id CDATA #REQUIRED

style CDATA #IMPLIED

Since:

alt−style CDATA #IMPLIED

filteredFrom (swt|html)

content CDATA #IMPLIED

style−id CDATA #IMPLIED

shared−style (true | false) >

This element is used to describe a page to be displayed. The intro can display both dynamic and static pages. Content for dynamic pages is generated from the subelements of the page, described below. The style or alt−style will be applied depending on the presentation. The styles can be further enhanced by referencing the id or class−id.
Static pages allow for the reuse of existing HTML documents within one's introduction, and can be linked to from any static or dynamic page. Static pages are not defined in a page element, they are simply html files that can be linked to by other pages.
The home page, whose id is specified in the presentation element of the intro config extension point, can have a url indicating that it is a static page. If no url is specified then the home page is assumed to be dynamic. All other pages described using the page element are dynamic.
Also note that when the SWT presentation is used and a static page is to be displayed, an external brower is launched and the current page remains visible.
The subelements used in a dynamic page are as follows: A **group** subelement is used to group related content and apply style across the grouped content. A **link** subelement defines a link which can be used to link to a static or dynamic page and run an intro action/command. A link is normally defined at the page level to navigate between main pages versus links within a page. A **text** subelement defines textual content at the page level. A **head** subelement is only applicable for the Web based presentation and allows for additional html to be added to the HTML **head** section. This is useful for adding java scripts or extra style sheets. An **img** subelement defines image content for the page level. An **include** subelement allows for reuse of any element other than a page. An **html** subelement is only applicable for the Web based presentation and allows for the embedding or inclusion of html into the page's content. Embedding allows for a fully defined html file to be embeded within an HTML **object** by referencing the html file. Inclusion allows for including an html snippet directly from an html file. A **title** subelement defines the title of the page. An **anchor** subelement defines a point where external contributions can be made by an <extensionContent> element.

- **url −** The optional relative path to an HTML file. When using the Web based presentation, this HTML file will be displayed instead of any content defined for this page. This attribute is only applicable to the home page, which is identified in the presentation element of the intro config extension point. It is ignored for all other pages.
- **id −** A unique name that can be used to identify this page.
- **style −** A relative path to a CSS file which is applied to the page only when using the Web based presentation. The path is relative to the location of this xml content file.
  Since 3.1, styles can also be a comma separated list of styles. These styles will be injected into the HTML HEAD element in the order in which they are listed in the style attribute.
- **alt−style −** A relative path to a SWT presentation properies file which is applied to the page only when using the SWT based presentation. The path is relative to the location of this xml content file.
  Since 3.1, styles can also be a comma separated list of styles. These styles will be used when creating the SWT presentation of the welcome page.

- **filteredFrom –** an optional attribute that allows for filtering a given element out of a specific implementation. For example, if a group has filteredFrom = swt, it means that this group will not appear as content in the swt implementation.
- **content –** an optional attribute which can define the location of an introContent.xml file that represents the content of this page. When this attribute is defined, all children and attributes in this page element, except id, are ignored. This is because the content of this page is now assumed to reside in the xml file pointed to by the content file attribute. When resolving to the content of this file, the page with an id that matches the id defined in this page element is chosen. This seperation of pages can be used when performance is an issue, as the content of a page is now loaded more lazily. Since 3.1, if the content of the external file is XHTML 1.0, then the page is rendered as is.
- **style–id –** A means to classifiy the page into a given category so that a common style may be applied.
- **shared–style –** a boolean flag that controls the addition of the shared style into this page's list of styles. If `true` (the default), the shared style is added to this page's styles. If `false`, the shared style defined in the Intro configuration will not be injected into the styles of this page.

<!ELEMENT group (group* | link* | text* | img* | include* | html* | anchor* | contentProvider*)>

<!ATTLIST group

id CDATA #REQUIRED

label CDATA #IMPLIED

style–id CDATA #IMPLIED

filteredFrom (swt|html) >

Used to group related content, content that should have similar style applied, or content that will be included together in other pages.

- **id –** unique identifier of the group
- **label –** a label or heading for this group
- **style–id –** A means to classifiy this group into a given category so that a common style may be applied.
- **filteredFrom –** an optional attribute that allows for filtering a given element out of a specific implementation. For example, if a group has filteredFrom = swt, it means that this group will not appear as content in the swt implementation.

<!ELEMENT head EMPTY>

<!ATTLIST head

src CDATA #REQUIRED

encoding CDATA #IMPLIED>

Since:

Direct HTML to include in a page's HEAD content area. It allows for additional html to be added to the HTML HEAD section. This is useful for adding java srcipts or extra styles sheets. This markup is only to be used with an HTML based intro part implementation. It is simply ignored in the case of a UI Forms implementation. A page can have more than one head element. An implementation can have one and only one head element (since it is a shared across all pages).

- **src** – relative or absolute URL to a file containing HTML to include directly into the HTML head section.
- **encoding** – an optional attribute to specify the encoding of the inlined file containing the head snippet. Default is UTF–8. Since 3.0.1

<!ELEMENT title EMPTY>

<!ATTLIST title

id CDATA #IMPLIED

style–id CDATA #IMPLIED

filteredFrom (swt|html) >

a snippet of text that can optionally contain escaped HTML tags. It is only used as a Page Title, and so a given page can have a maximum of one title element.

- **id** – unique identifier of this title.
- **style–id** – A means to classifiy this element into a given category so that a common style may be applied
- **filteredFrom** – an optional attribute that allows for filtering a given element out of a specific implementation. For example, if a group has filteredFrom = swt, it means that this group will not appear as content in the swt implementation.

<!ELEMENT link (text? , img?)>

<!ATTLIST link

id CDATA #IMPLIED

label CDATA #IMPLIED

url CDATA #REQUIRED

style–id CDATA #IMPLIED

Since:

filteredFrom (swt|html) >

Can link to a static HTML file, an external web site, or can run an Intro URL action.

- **id** – A unique id that can be used to identify this link
- **label** – The text name of this link
- **url** – A valid URL to an external web site, a static html file, or an Intro URL that represents an Intro action. All intro URLs have the following form: http://org.eclipse.ui.intro/<action name>?param1=value1&param2=value2 and will be processed by the intro framework.
  The predefined actions will be described using this format:

  action name – descripton of action
  action parameter1 – description of parameter
  action parameter2 (optional) – description of parameter
  action parameter3 (optional) = ("true" "false") "false" – description of parameter, choice of either true or false and "false" is the default

  The following predefined actions are included in the intro framework:

  close – closes the intro part
  no parameters required

  navigate – navigate through the intro pages in a given direction or return to the home page
  direction = ("backward" "forward" "home") – specifies the direction to navigate

  openBrowser – open the url in an external browser. Since 3.1, this action relies on the workbench Browser support. This means that any user preferences set for the browser will be honored.
  url – a valid URL to an external web site or a local HTML file
  pluginId (optional) – if the url is relative, then it is relative to a plugin. Specify here the id of the plug–in containing the file.

  openURL – open the url embedded in the Welcome page. In the case of SWT presentation, the url is displayed in an external browser (similar to the openBrowser action above). since 3.1
  url – a valid URL to an external web site or to a local HTML file
  pluginId (optional) – if the url is relative, then this specifies the id of the plug–in containing the file.

  runAction – runs the specified action
  class – the fully qualified class name of the class that implements one of `org.eclipse.ui.intro.config.IIntroAction`, `org.eclipse.jface.actino.IAction`, or `org.eclipse.ui.IActionDelegate`
  pluginId – The id of the plug–in which contains the class.
  standby (optional) = ("true" "false") "false" – indicate whether to set the intro into standby mode after executing the action
  additional parameters – any additional parameters are passed to actions that implement `org.eclipse.ui.intro.config.IIntroAction`

  setStandbyMode – sets the state of the intro part
  standby = ("true" "false") – true to put the intro part in its partially visible standy mode, and false to

make it fully visible

showHelp – Open the help system.
no parameters required

showHelpTopic – Open a help topic.
id – the URL of the help resource. (See Javadoc for `org.eclipse.ui.help.WorkbenchHelp.displayHelpResource`)
embed (optional) = ("true" "false") "true" – indicates that the help resource needs to be displayed embedded as part of the Welcome pages. Default is false. This flag is simply ignored in the case of the SWT presentation. This is equivalent to openURL() command, but for Help System topics. The embedded URL occupies the full real–estate of the current page. since 3.1
embedTarget (optional) – the path to a div in the current Welcome page that will hold the content of the Help topic. If specified, then *embed* is true by default and the embedded URL is inserted inside the div with the specified path. The path is relative to the page and so it should not start with the page id. The children of the div are replaced by the content of the URL. Only one div per page can be used as an embed target. This flag is simply ignored in the case of the SWT presentation. It is also unsupported when using XHTML as intro content. since 3.1

showMessage – Displays a message to the user using a standard information dialog.
message – the message to show the user

showStandby – Sets the intro part to standby mode and shows the standbyContentPart with the given input
partId – the id of the standbyContentPart to show
input – the input to set on the standbyContentPart

showPage – show the intro page with the given id
id – the id of the intro page to show
standby (optional) = ("true" "false") "false" – indicate whether to set the intro into standby mode after showing the page

If any of the parameters passed to these actions have special characters (ie: characters that are illegal in a URL), then they should be encoded using UTF–8 url encoding. To recieve these parametrs in there decoded state a special parameter, *decode = ("true" "false")* can be used to force a decode of these parameters when the Intro framework processes them.
For example, the following Intro url:
http://org.eclipse.ui.intro/showMessage?message=This+is+a+message
will process the message parameter as "This+is+a+message"
whereas
http://org.eclipse.ui.intro/showMessage?message=This+is+a+message&decode=true
will process the message parameter as "This is a message".

- **style–id** – A means to classifiy this link into a given category so that a common style may be applied.
- **filteredFrom** – an optional attribute that allows for filtering a given element out of a specific implementation. For example, if a group has filteredFrom = swt, it means that this group will not appear as content in the swt implementation.

<!ELEMENT text EMPTY>

Since:

<!ATTLIST text

id CDATA #IMPLIED

style−id CDATA #IMPLIED

filteredFrom (swt|html) >

a snippet of text that can optionally contain escaped HTML tags. It can include b and li tags. It can also contain anchors for urls. If multiple paragraphs are needed, then the text can be divided into multiple sections each begining and ending with the p tag.

- **id −** unique identifier of this text.
- **style−id −** A means to classifiy this element into a given category so that a common style may be applied
- **filteredFrom −** an optional attribute that allows for filtering a given element out of a specific implementation. For example, if a group has filteredFrom = swt, it means that this group will not appear as content in the swt implementation.

<!ELEMENT img EMPTY>

<!ATTLIST img

id CDATA #REQUIRED

src CDATA #REQUIRED

alt CDATA #IMPLIED

style−id CDATA #IMPLIED

filteredFrom (swt|html) >

An image that represents intro content and not presentation (as opposed to decoration images defined in styles).

- **id −** unique identifier of this image
- **src −** the file to load the image from
- **alt −** the alternative text to use when the image can not be loaded and as tooltip text for the image.
- **style−id −** A means to classifiy this image into a given category so that a common style may be applied.
- **filteredFrom −** an optional attribute that allows for filtering a given element out of a specific implementation. For example, if a group has filteredFrom = swt, it means that this group will not appear as content in the swt implementation.

<!ELEMENT html (img | text)>

<!ATTLIST html

id CDATA #REQUIRED

src CDATA #REQUIRED

type (inline|embed)

style−id CDATA #IMPLIED

filteredFrom (swt|html)

encoding CDATA #IMPLIED>

direct HTML to include in the page either by embedding the entire document, or inlining a snippet of HTML in−place. A fallback image or text must be defined for alternative swt presentation rendering.
Embedding allows for a fully defined html file to be embedded within the dynamic page's content. An HTML **object** element is created that references the html file.
Inclusion allows for including an html snippet directly from a file into the dynamic html page.

- **id** − unique identifier of this HTML element
- **src** − relative or absolute URL to a file containing HTML
- **type** − if 'embed', a valid (full) HTML document will be embedded using HTML 'OBJECT' tag. If 'inline', value of 'src' will be treated as a snippet of HTML to emit 'in−place'. (if type is not specified, this html object is ignored by the intro configuration).
- **style−id** − A means to classifiy this HTML element into a given category so that a common style may be applied.
- **filteredFrom** − an optional attribute that allows for filtering a given element out of a specific implementation. For example, if a group has filteredFrom = swt, it means that this group will not appear as content in the swt implementation.
- **encoding** − an optional attribute to specify the encoding of the inlined file (in the case where type=inline is specified). If not specified, the default is UTF−8. Since 3.0.1

<!ELEMENT include EMPTY>

<!ATTLIST include

configId CDATA #IMPLIED

path CDATA #REQUIRED

merge−style (true | false) >

Since:

expands an element targeted by the given path and optional configId attributes. Path should uniquely address an element within the specified configuration. It could point to a shared group defined at the configuration level, or any element in a page.

- **configId** – identifier of a configuration where the included element is defined. If specified, it is assumed that the element to be included is specified in another configuration, and not the enclosing configuration. In this case, that external config is loaded and the element is resolved from that new config. If not specified, enclosing (parent) configuration of this include is assumed.
- **path** – the path that uniquely represents the target element within the configuration (e.g. page/group1/group2). It may be a group element, or any element that may be contained in a group. You can not include a page.
- **merge–style** – if `true`, style belonging to the page that owns the included element will be added to list of styles of the including page. If `false` (the default), the including page is responsible for controlling properties of the included element.

<!ELEMENT anchor EMPTY>

<!ATTLIST anchor

id CDATA #REQUIRED>

an anchor is the element used to declare extensibility. It is a location in the configurtaion that allows for external contributions. Only anchors are valid target values for the path attribute in an extensionContent

- **id** – unique id to identify this anchor.

<!ELEMENT extensionContent (text | group | link | html | include)>

<!ATTLIST extensionContent

style CDATA #IMPLIED

alt–style CDATA #IMPLIED

path CDATA #REQUIRED

content CDATA #IMPLIED>

The content to be added to the target anchor. Only one extensionContent is allowed in a given configExtension because if this extension could not be resolved (if the config could not be found, or the target anchor element could not be found) then the pages and/or groups in the extension need to be ingnored.

Since:

- **style –** A relative path to a CSS file which is applied to the page only when using the Web based presentation. The path is relative to the location of this xml content file.
  Since 3.1, styles can also be a comma separated list of styles. These styles will be injected into the HTML HEAD element in the order in which they are listed in the style attribute.
- **alt–style –** A relative path to a SWT presentation properies file which is applied to the page only when using the SWT based presentation. The path is relative to the location of this xml content file.
  Since 3.1, styles can also be a comma separated list of styles. These styles will be used when creating the SWT presentation of the welcome page.
- **path –** the path that uniquely represents the path to an anchor. (e.g. page/group1/group2/anchorId) within the target configuration to be extended. It can only be an anchor which can be in any page or group, including shared groups at configuration level.
- **content –** if content is defined, it is assumed that the extension content is defined in an external XHTML file. In that case the resource pointed to by this content attribute is loaded and the path attribute is now resolved in this external file. since 3.1

<!ELEMENT contentProvider (text?)>

<!ATTLIST contentProvider

id CDATA #REQUIRED

pluginId CDATA #IMPLIED

class CDATA #REQUIRED>

A proxy for an intro content provider, which allows an intro page to dynamically pull data from various sources (e.g., the web, eclipse, etc) and provide content at runtime based on this dynamic data. If the IIntroContentProvider class that is specified in the class attribute can not be loaded, then the contents of the text element will be rendered instead. This is a dynamic version of the html intro tag. While the html tag allows for embedding or inlining a static html content into the generated html intro page, the contentProvider tag allows for dynamic creation of that content at runtime. Another difference between the tags is that the html tag is only supported for the HTML presentation, while this contentProvider tag is supported for both the HTML and SWT presentations. Since 3.0.1

- **id –** unique identifier of this content provider element. It is a required attribute because having a unique id is what prevents the intro framework from reinstantiating this content provider class and recreating its content.
- **pluginId –** The id of the plugin that contains the IContentProvider class specified by the class attribute. This is an optional attribute that should be used if the class doesn't come from the same plugin that defined the markup.
- **class –** A class that implements the IContentProvider interface

Examples:

Here is a sample usage of the config extension point.

```
<extension id=
"intro"
point=
"org.eclipse.ui.intro.config"
>
<config introId=
"com.org.xyz.intro"
id=
"com.org.xyz.introConfig"
content=
"introContent.xml"
>
<presentation home−page−id=
"root"
title=
"%intro.title"
>
<implementation ws=
"win32"
style=
"css/shared.css"
kind=
"html"
```

os=

"win32"

>

</implementation>

<implementation style=

"css/shared_swt.properties"

kind=

"swt"

>

</implementation>

</presentation>

</config>

</extension>

API Information:

For further details see the spec for the org.eclipse.ui.intro.config API package.

Supplied Implementation:

The intro contributed by the org.eclipse.platform plugin is the only implementation within Eclipse.

Intro Part Configuration Extension

Identifier:

org.eclipse.ui.intro.configExtension

Since:

3.0

Description:

This extension point is used to extend an existing intro configuration by providing more content, additional StandbyContentParts or additional IntroUrl actions.

Configuration Markup:

<!ELEMENT extension (configExtension+ , standbyContentPart* , action*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT configExtension EMPTY>

<!ATTLIST configExtension

configId CDATA #REQUIRED

content CDATA #REQUIRED>

Defines an extension to an intro configuration. Any page or group in an intro part configuration can be extended, if it has declared extensability by defining anchors.

- **configId –** the id of an intro contribution that will be extended
- **content –** an intro content file. The content file is an XML file that contains the specifics of the intro (intro content file format specification). The content file is parsed at run time by the intro framework. Based on the settings in this file, a certain number of pages, groups, and links are shown to the user when the intro is opened.

<!ELEMENT standbyContentPart EMPTY>

<!ATTLIST standbyContentPart

id CDATA #REQUIRED

pluginId CDATA #REQUIRED

class CDATA #REQUIRED>

standbyContentPart registration. Once registered, standby parts can be launched through an introURL action of the following format:

```
http://org.eclipse.ui.intro/showStandby?partId=&lt;id of standbyContentPart&gt;
```

- **id –** a unique id that identifies this standbyContentPart.
- **pluginId –** the name of the plugin that holds the class defined in the "class" attribute.
- **class –** the fully qualified class name of the class that implements `org.eclipse.ui.intro.config.IStandbyContentPart` to handle displaying alternative standby content, such as a cheat sheet.

<!ELEMENT action EMPTY>

<!ATTLIST action

name CDATA #REQUIRED

replaces CDATA #REQUIRED>

custom Intro URL action registration. This can be used to create new Intro URL actions or a shortCut to predefined actions.

- **name –** a unique name that identifies this action.
- **replaces –** the macro which replaces the action name in the Intro URL.

Examples:

Here is an example implementation of this extension point:

<extension point=

```
"org.eclipse.ui.intro.configExtension"

>

<configExtension configId=

"com.org.xyz.introConfig"

content=

"extensionContent.xml"

/>

<standbyPart id=

"com.org.xyz.myStandbyPart"

class=

"com.org.xyz.internal.MyStandbyContent"

pluginId=

"com.org.xyz"

/>

<action name=

"shortcutAction"

replaces=

"http://org.eclipse.ui.intro/showStandby?partId=com.org.xyz.myStandbyPart"

/>

<action name=

"customAction"

replaces=

"runAction?pluginId=com.org.xyz&amp;amp;class=com.org.xyz.CustomAction&amp;amp;param1=value1"

/>

</extension>
```

API Information:

For further details see the spec for the org.eclipse.ui.intro.config API package.

Supplied Implementation:

There are three supplied implementations:

- org.eclipse.jdt, makes use of configExtension
- org.eclipse.pde, makes use of configExtension
- org.eclipse.platform, makes use of standbyPoint

Intro Content File XML Format

Version 3.1.0

This document describes the intro content file structure as a series of DTD fragments.

introContent

<!ELEMENT introContent (page+ , group* , extensionContent*)>

The introContent element defines the body of the intro content file. The content file is made up of pages, shared groups that can be included in multiple pages, and extensions to anchor points defined in other configurations.

page

<!ELEMENT page (group* | link* | text* | head* | img* | include* | html* | title? | anchor* | contentProvider*)>

<!ATTLIST page

url CDATA #IMPLIED

id CDATA #REQUIRED

style CDATA #IMPLIED

alt−style CDATA #IMPLIED

filteredFrom (swt|html)

content CDATA #IMPLIED

style−id CDATA #IMPLIED>

This element is used to describe a page to be displayed. The intro can display both dynamic and static pages.

Content for dynamic pages is generated from the sub elements of the page, described below. The style or alt−style will be applied depending on the presentation. The styles can be further enhanced by referencing the id or class−id.

Static pages allow for the reuse of existing HTML documents within one's introduction, and can be linked to

from any static or dynamic page. Static pages are not defined in a page element, they are simply html files that can be linked to by other pages.

The home page, whose id is specified in the presentation element of the intro config extension point, can have a url indicating that it is a static page. If no url is specified then the home page is assumed to be dynamic. All other pages described using the page element are dynamic.
Also note that when the SWT presentation is used and a static page is to be displayed, an external browser is launched and the current page remains visible.

The subelements used in a dynamic page are as follows: A **group** subelement is used to group related content and apply style across the grouped content. A **link** subelement defines a link which can be used to link to a static or dynamic page and run an intro action/command. A link is normally defined at the page level to navigate between main pages versus links within a page. A **text** subelement defines textual content at the page level. A **head** subelement is only applicable for the Web based presentation and allows for additional html to be added to the HTML **head** section. This is useful for adding java scripts or extra style sheets. An **img** subelement defines image content for the page level. An **include** subelement allows for reuse of any element other than a page. An **html** subelement is only applicable for the Web based presentation and allows for the embedding or inclusion of html into the page's content. Embedding allows for a fully defined html file to be embeded within an HTML **object** by referencing the html file. Inclusion allows for including an html snippet directly from an html file. A **title** subelement defines the title of the page. An **anchor** subelement defines a point where external contributions can be made by an <extensionContent> element.

- **url** – The optional relative path to an HTML file. When using the Web based presentation, this HTML file will be displayed instead of any content defined for this page. This attribute is only applicable to the home page, which is identified in the presentation element of the intro config extension point. It is ignored for all other pages.
- **id** – A unique name that can be used to identify this page.
- **style** – A relative path to a CSS file which is applied to the page only when using the Web based presentation. The path is relative to the location of this xml content file.
  Since 3.1, styles can also be a comma separated list of styles. These styles will be injected into the HTML HEAD element in the order in which they are listed in the style attribute.
- **alt–style** – A relative path to a SWT presentation properties file which is applied to the page only when using the SWT based presentation. The path is relative to the location of this xml content file. Since 3.1, styles can also be a comma separated list of styles. These styles will be used when creating the SWT presentation of the welcome page.
- **filteredFrom** – an optional attribute that allows for filtering a given element out of a specific implementation. For example, if a group has filteredFrom = swt, it means that this group will not appear as content in the swt implementation.
- **content** – an optional attribute which can define the location of an introContent.xml file that represents the content of this page. When this attribute is defined, all children and attributes in this page element, except id, are ignored. This is because the content of this page is now assumed to reside in the xml file pointed to by the content file attribute. When resolving to the content of this file, the page with an id that matches the id defined in this page element is chosen. This separation of pages can be used when performance is an issue, as the content of a page is now loaded more lazily.Since 3.1, if the content of the external file is XHTML 1.0, then the page is rendered as is.
- **style–id** – A means to classify the page into a given category so that a common style may be applied.
- ***shared–style*** – a boolean flag that controls the addition of the shared style into this page's list of styles. If true (the default), the shared style is added to this page's styles. If false , the shared style defined in the Intro configuration will not be injected into the styles of this page.
-

group

<!ELEMENT group (group* | link* | text* | img* | include* | html* | anchor*)>

<!ATTLIST group

id CDATA #REQUIRED

label CDATA #IMPLIED

style−id CDATA #IMPLIED

filteredFrom (swt|html) >

Used to group related content, content that should have similar style applied, or content that will be included together in other pages.

- **id −** unique identifier of the group
- **label −** a label or heading for this group
- **style−id −** A means to classify this group into a given category so that a common style may be applied.
- **filteredFrom −** an optional attribute that allows for filtering a given element out of a specific implementation. For example, if a group has filteredFrom = swt, it means that this group will not appear as content in the swt implementation.

link

<!ELEMENT link (text? , img?)>

<!ATTLIST link

id CDATA #IMPLIED

label CDATA #IMPLIED

url CDATA #REQUIRED

style−id CDATA #IMPLIED

filteredFrom (swt|html) >

Can link to a static HTML file, an external web site, or can run an Intro URL action.

- **id** – A unique id that can be used to identify this link
- **label** – The text name of this link
- **url** – A valid URL to an external web site, a static html file, or an Intro URL that represents an Intro action. All intro URLs have the following form: http://org.eclipse.ui.intro/<action name>?param1=value1&param2=value2 and will be processed by the intro framework.

The predefined actions will be described using this format:

action name – description of action
action parameter1 – description of parameter
action parameter2 (optional) – description of parameter
action parameter3 (optional) = ("true" | "false") "false" – description of parameter, choice of either true or false and "false" is the default

The following predefined actions are included in the intro framework:

close – closes the intro part
no parameters required

navigate – navigate through the intro pages in a given direction or return to the home page
direction = ("backward" | "forward" | "home") – specifies the direction to navigate

openBrowser – open the url in an external browser. Since 3.1, this action relies on the workbench Browser support. This means that any user preferences set for the browser will be honored.
url – a valid URL to an external web site or a static HTML file
pluginId (optional) – only required if a static HTML file is specified. This is the id of the plug–in containing the file.

openURL – open the url embedded in the Welcome page. In the case of SWT presentation, the url is displayed in an external browser (similar to the openBrowser action above). since 3.1
url – a valid URL to an external web site or to a local HTML file
pluginId (optional) – if the url is relative, then this specifies the id of the plug–in containing the file.

runAction – runs the specified action
class – the fully qualified class name of the class that implements one of `org.eclipse.ui.intro.config.IIntroAction`, `org.eclipse.jface.action.IAction`, or `org.eclipse.ui.IActionDelegate`
pluginId – The id of the plug–in which contains the class.
standby (optional) = ("true" | "false") "false" – indicate whether to set the intro into standby mode after executing the action
additional parameters – any additional parameters are passed to actions that implement `org.eclipse.ui.intro.config.IIntroAction`

setStandbyMode – sets the state of the intro part
standby = ("true" | "false") – true to put the intro part in its partially visible standby mode, and false to make it fully visible

showHelp – Open the help system.

no parameters required

showHelpTopic – Open a help topic.
id – the URL of the help resource. (See Javadoc for `org.eclipse.ui.help.WorkbenchHelp.displayHelpResource`
embed (optional) = ("true" | "false") "true" – indicates that the help resource needs to be displayed embedded as part of the welcome pages. Default is false. This flag is simply ignored in the case of the SWT presentation. since 3.1
embedTarget (optional) – the path to a div in the current Welcome page that will hold the content of the Help topic. If specified, then *embed* is true by default and the embedded URL is inserted inside the div with the specified path. The path is relative to the page and so it should not start with the page id. The children of the div are replaced by the content of the URL. Only one div per page can be used as an embed target. This flag is simply ignored in the case of the SWT presentation. It is also unsupported when using XHTML as intro content. since 3.1

showMessage – Displays a message to the user using a standard information dialog.
message – the message to show the user

showStandby – Sets the intro part to standby mode and shows the standbyContentPart with the given input
partId – the id of the standbyContentPart to show
input – the input to set on the standbyContentPart

showPage – show the intro page with the given id
id – the id of the intro page to show
standby (optional) = ("true" | "false") "false" – indicate whether to set the intro into standby mode after showing the page

If any of the parameters passed to these actions have special characters (ie: characters that are illegal in a URL), then they should be encoded using UTF–8 url encoding. To recieve these parametrs in there decoded state a special parameter, *decode = ("true" "false")* can be used to force a decode of these parameters when the Intro framework processes them.
For example, the following Intro url:
http://org.eclipse.ui.intro/showMessage?message=This+is+a+message
will process the message parameter as "This+is+a+message"
whereas
http://org.eclipse.ui.intro/showMessage?message=This+is+a+message&decode=true
will process the message parameter as "This is a message".

- **style–id** – A means to classify this link into a given category so that a common style may be applied.
- **filteredFrom** – an optional attribute that allows for filtering a given element out of a specific implementation. For example, if a group has filteredFrom = swt, it means that this group will not appear as content in the swt implementation.

`html`

<!ELEMENT html (img | text)>

<!ATTLIST html

id CDATA #REQUIRED

src CDATA #REQUIRED

type (inline|embed)

style−id CDATA #IMPLIED

filteredFrom (swt|html) >

encoding CDATA #IMPLIED

direct HTML to include in the page either by embedding the entire document, or inlining a snippet of HTML in−place. A fallback image or text must be defined for alternative swt presentation rendering.
Embedding allows for a fully defined html file to be embedded within the dynamic page's content. An HTML **object** element is created that references the html file.
Inclusion allows for including an html snippet directly from a file into the dynamic html page.

- **id −** unique identifier of this HTML element
- **src −** relative or absolute URL to a file containing HTML
- **type −** if 'embed', a valid (full) HTML document will be embedded using HTML 'OBJECT' tag. If 'inline', value of 'src' will be treated as a snippet of HTML to emit 'in−place'. (if type is not specified, this html object is ignored by the intro configuration).
- **style−id −** A means to classify this HTML element into a given category so that a common style may be applied.
- **filteredFrom −** an optional attribute that allows for filtering a given element out of a specific implementation. For example, if a group has filteredFrom = swt, it means that this group will not appear as content in the swt implementation.
- **encoding −** an optional attribute to specify the encoding of the inlined file (in the case where type=inline is specified). If not specified, the default is UTF−8. Since 3.0.1

title

<!ELEMENT title EMPTY>

<!ATTLIST title

id CDATA #IMPLIED

style−id CDATA #IMPLIED

filteredFrom (swt|html) >

a snippet of text that can optionally contain escaped HTML tags. It is only used as a Page Title, and so a given page can have a maximum of one title element.

- **id –** unique identifier of this title.
- **style–id –** A means to classifiy this element into a given category so that a common style may be applied
- **filteredFrom –** an optional attribute that allows for filtering a given element out of a specific implementation. For example, if a group has filteredFrom = swt, it means that this group will not appear as content in the swt implementation.

text

<!ELEMENT text EMPTY>

<!ATTLIST text

id CDATA #IMPLIED

style–id CDATA #IMPLIED

filteredFrom (swt|html) >

a snippet of text that can optionally contain escaped HTML tags. It can include b and li tags. It can also contain anchors for urls. If multiple paragraphs are needed, then the text can be divided into multiple sections each beginning and ending with the p tag.

- **id –** unique identifier of this text.
- **style–id –** A means to classify this element into a given category so that a common style may be applied
- **filteredFrom –** an optional attribute that allows for filtering a given element out of a specific implementation. For example, if a group has filteredFrom = swt, it means that this group will not appear as content in the swt implementation.

include

<!ELEMENT include EMPTY>

<!ATTLIST include

configId CDATA #IMPLIED

path CDATA #REQUIRED

merge−style (true | false) >

expands an element targeted by the given path and optional configId attributes. Path should uniquely address an element within the specified configuration. It could point to a shared group defined at the configuration level, or any element in a page.

- **configId −** identifier of a configuration where the included element is defined. If specified, it is assumed that the element to be included is specified in another configuration, and not the enclosing configuration. In this case, that external config is loaded and the element is resolved from that new config. If not specified, enclosing (parent) configuration of this include is assumed.
- **path −** the path that uniquely represents the target element within the configuration (e.g. page/group1/group2). It may be a group element, or any element that may be contained in a group. You can not include a page.
- **merge−style −** if `true`, style belonging to the page that owns the included element will be added to list of styles of the including page. If `false` (the default), the including page is responsible for controlling properties of the included element.

head

<!ELEMENT head EMPTY>

<!ATTLIST head

src CDATA #REQUIRED>

encoding CDATA #IMPLIED

Direct HTML to include in a page's HEAD content area. It allows for additional html to be added to the HTML HEAD section. This is useful for adding java scripts or extra styles sheets. This markup is only to be used with an HTML based intro part implementation. It is simply ignored in the case of a UI Forms implementation. A page can have more than one head element. An implementation can have one and only one head element (since it is a shared across all pages).

- **src −** relative or absolute URL to a file containing HTML to include directly into the HTML head section.
- **encoding −** an optional attribute to specify the encoding of the inlined file containing the head snippet. Default is UTF−8. Since 3.0.1

img

<!ELEMENT img EMPTY>

<!ATTLIST img

id CDATA #REQUIRED

src CDATA #REQUIRED

alt CDATA #IMPLIED

style–id CDATA #IMPLIED

filteredFrom (swt|html) >

An image that represents intro content and not presentation (as opposed to decoration images defined in styles).

- **id –** unique identifier of this image
- **src –** the file to load the image from
- **alt –** the alternative text to use when the image can not be loaded and as tooltip text for the image.
- **style–id –** A means to classify this image into a given category so that a common style may be applied.
- **filteredFrom –** an optional attribute that allows for filtering a given element out of a specific implementation. For example, if a group has filteredFrom = swt, it means that this group will not appear as content in the swt implementation.

extensionContent

<!ELEMENT extensionContent (text | group | link | html | include)>

<!ATTLIST extensionContent

style CDATA #IMPLIED

alt–style CDATA #IMPLIED

path CDATA #REQUIRED>

The content to be added to the target anchor. Only one extensionContent is allowed in a given configExtension because if this extension could not be resolved (if the config could not be found, or the target anchor element could not be found) then the pages and/or groups in the extension need to be ignored.

- **style –** A relative path to a CSS file which is applied to the page only when using the Web based presentation. The path is relative to the location of this xml content file.
  Since 3.1, styles can also be a comma separated list of styles. These styles will be injected into the HTML HEAD element in the order in which they are listed in the style attribute.

- **alt−style** − A relative path to a SWT presentation properies file which is applied to the page only when using the SWT based presentation. The path is relative to the location of this xml content file. Since 3.1, styles can also be a comma separated list of styles. These styles will be used when creating the SWT presentation of the welcome page.
- **path** − the path that uniquely represents the path to an anchor. (e.g. page/group1/group2/anchorId) within the target configuration to be extended. It can only be an anchor which can be in any page or group, including shared groups at configuration level
- ***content*** − if content is defined, it is assumed that the extension content is defined in an external XHTML file. In that case the resource pointed to by this content attribute is loaded and the path attribute is now resolved in this external file. since 3.1
-

anchor

<!ELEMENT anchor EMPTY>

<!ATTLIST anchor

id CDATA #REQUIRED>

an anchor is the element used to declare extensibility. It is a location in the configuration that allows for external contributions. Only anchors are valid target values for the path attribute in an extensionContent

- **id** − unique id to identify this anchor.

contentProvider

<!ELEMENT contentProvider (text)>

<!ATTLIST contentProvider

id CDATA #REQUIRED

pluginId CDATA #IMPLIED

class CDATA #REQUIRED>

A proxy for an intro content provider, which allows an intro page to dynamically pull data from various sources (e.g., the web, eclipse, etc) and provide content at runtime based on this dynamic data. If the IIntroContentProvider class that is specified in the class attribute can not be loaded, then the contents of the text element will be rendered instead. This is a dynamic version of the html intro tag. While the html tag allows for embedding or inlining a static html content into the generated html intro page, the contentProvider tag allows for dynamic creation of that content at runtime. Another difference between the tags is that the html

tag is only supported for the HTML presentation, while this contentProvider tag is supported for both the HTML and SWT presentations. Since 3.0.1

- **id** – unique identifier of this content provider element.
- **pluginId** – The id of the plug–in that contains the IContentProvider class specified by the class attribute. This is an optional attribute that should be used if the class doesn't come from the same plug–in that defined the markup.
- **class** – A class that implements the IContentProvider interface

Keywords

Identifier:

org.eclipse.ui.keywords

Since:

3.1

Description:

The keywords extension point defines keywords and a unique id for reference by other schemas. See propertyPages and preferencePages.

Configuration Markup:

<!ELEMENT extension (keyword*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT keyword EMPTY>

<!ATTLIST keyword

id CDATA #REQUIRED

label CDATA #REQUIRED>

- **id –** The id is the unique id used to reference the keyword.
- **label –** The human readable label of the keyword

Creation Wizards

Identifier:

org.eclipse.ui.newWizards

Description:

This extension point is used to register resource creation wizard extensions. Creation wizards appear as choices within the "New Dialog", and are typically used to create folders and files.

In the "New Dialog", wizards are organized into categories which usually reflect a particular problem domain. For instance, a Java oriented plugin may define a category called "Java" which is appropriate for "Class" or "Package" creation wizards. The categories defined by one plug−in can be referenced by other plug−ins using the category attribute. Uncategorized wizards, as well as wizards with invalid category paths, will end up in an "Other" category.

Wizards may optionally specify a description subelement whose body should contain short text about the wizard.

Configuration Markup:

<!ELEMENT extension (category | wizard | primaryWizard)*>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point** − a fully qualified identifier of the target extension point
- **id** − an optional identifier of the extension instance
- **name** − an optional name of the extension instance

<!ELEMENT category EMPTY>

<!ATTLIST category

id CDATA #REQUIRED

name CDATA #REQUIRED

parentCategory CDATA #IMPLIED>

- **id** − a unique name that can be used to identify this category
- **name** − a translatable name of the category that will be used in the dialog box
- **parentCategory** − a path to another category if this category should be added as a child

<!ELEMENT wizard (description? , selection*)>

<!ATTLIST wizard

id CDATA #REQUIRED

name CDATA #REQUIRED

icon CDATA #IMPLIED

category CDATA #IMPLIED

class CDATA #REQUIRED

project (true | false)

finalPerspective CDATA #IMPLIED

preferredPerspectives CDATA #IMPLIED

helpHref CDATA #IMPLIED

descriptionImage CDATA #IMPLIED

canFinishEarly (true | false)

hasPages (true | false) >

- **id –** a unique name that can be used to identify this wizard
- **name –** a translatable name of the wizard that will be used in the dialog box
- **icon –** a relative path of an icon that will be used together with the name to represent the wizard as one of the choices in the creation dialog box.
- **category –** a slash–delimited path ('/') of category IDs. Each token in the path must represent a valid category ID previously defined by this or some other plug–in. If omitted, the wizard will be added to the "Other" category.
- **class –** a fully qualified name of the Java class implementing `org.eclipse.ui.INewWizard`.
- **project –** an optional attribute indicating the wizard will create an IProject resource. Also causes the wizard to appear as a choice within the "New Project Dialog".
- **finalPerspective –** an optional attribute which identifies a perspective to activate when IProject resource creation is finished.
- **preferredPerspectives –** an optional attribute specifying a comma–separated list of perspective IDs. If the current perspective is in this list, then no perspective activation occurs when IProject resource creation is finished.
- **helpHref –** a help url that can describe this wizard in detail.

  Since 3.0
- **descriptionImage –** a larger image that can help describe this wizard.

  Since 3.0

- **canFinishEarly –** whether the wizard is capable of finishing without ever showing pages to the user.
- **hasPages –** whether the wizard provides any pages.

<!ELEMENT description (#PCDATA)>

an optional subelement whose body contains a short text describing what the wizard will do when started

<!ELEMENT selection EMPTY>

<!ATTLIST selection

class CDATA #REQUIRED

name CDATA #IMPLIED>

- **class –** a fully qualified class name. If each object in the workbench selection implements this interface the selection will be passed to the wizard. Otherwise, an empty selection is passed
- **name –** an optional name filter. Each object in the workbench selection must match the name filter to be passed to the wizard

<!ELEMENT primaryWizard EMPTY>

<!ATTLIST primaryWizard

id CDATA #REQUIRED>

a means of declaring that a wizard is "primary" in the UI. A primary wizard is emphasized in the new wizard dialog. Please note that this element is not intended to be used by plug in developers! This element exists so that product managers may emphasize a set of wizards for their products.

- **id –** the id of a wizard that should be made primary.

Examples:

Following is an example of creation wizard configuration:

<extension point=

"org.eclipse.ui.newWizards"

>

<category id=

"com.xyz.XYZ"

name=

"XYZ Wizards"

>

</category>

<category id=

"com.xyz.XYZ.Web"

name=

"Web Wizards"

parentCategory=

"com.xyz.XYZ"

>

</category>

<wizard id=

"com.xyz.wizard1"

name=

"XYZ artifact"

category=

"com.xyz.XYZ/com.xyz.XYZ.Web"

icon=

"./icons/XYZwizard1.gif"

class=

"com.xyz.XYZWizard1"

>

<description>

Create a simple XYZ artifact and set initial content

</description>

<selection class=

"org.eclipse.core.resources.IResource"

/>

</wizard>

</extension>

API Information:

The value of the class attribute must represent a class that implements `org.eclipse.ui.INewWizard`. If the wizard is created from within the New Wizard it will be inserted into the existing wizard. If the wizard is launched as a shortcut (from the File New menu or a toolbar button) it will appear standalone as a separate dialog box.

Supplied Implementation:

The workbench comes with wizards for creating empty resources of the following types: project, folder and file. These wizards are registered using the same mechanism as described above. Additional wizards may also appear, subject to particular platform installation.

Perspective Extensions

Identifier:

org.eclipse.ui.perspectiveExtensions

Description:

This extension point is used to extend perspectives registered by other plug−ins. A perspective defines the initial contents of the window action bars (menu and toolbar) and the initial set of views and their layout within a workbench page. Other plug−ins may contribute actions or views to the perspective which appear when the perspective is selected. Optional additions by other plug−ins are appended to the initial definition.

Configuration Markup:

<!ELEMENT extension (perspectiveExtension*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point −** a fully qualified identifier of the target extension point
- **id −** an optional identifier of the extension instance
- **name −** an optional name of the extension instance

<!ELEMENT perspectiveExtension (actionSet | viewShortcut | perspectiveShortcut | newWizardShortcut | view | showInPart)*>

<!ATTLIST perspectiveExtension

targetID CDATA #REQUIRED>

- **targetID −** the unique identifier of the perspective (as specified in the registry) into which the contribution is made.

<!ELEMENT actionSet EMPTY>

<!ATTLIST actionSet

id CDATA #REQUIRED>

- **id −** the unique identifier of the action set which will be added to the perspective.

<!ELEMENT viewShortcut EMPTY>

<!ATTLIST viewShortcut

id CDATA #REQUIRED>

- **id** – the unique identifier of the view which will be added to the perspective's "Show View" submenu of the "Window" menu.

<!ELEMENT perspectiveShortcut EMPTY>

<!ATTLIST perspectiveShortcut

id CDATA #REQUIRED>

- **id** – the unique identifier of the perspective which will be added to the perspective's "Open Perspective" submenu of the "Window" menu.

<!ELEMENT newWizardShortcut EMPTY>

<!ATTLIST newWizardShortcut

id CDATA #REQUIRED>

- **id** – the unique identifier of the new wizard which will be added to the perspective's "New" submenu of the "File" menu.

<!ELEMENT showInPart EMPTY>

<!ATTLIST showInPart

id CDATA #IMPLIED>

- **id** – the unique identifier of the view which will be added to the perspective's "Show In..." prompter in the Navigate menu.

<!ELEMENT view EMPTY>

<!ATTLIST view

id CDATA #REQUIRED

relative CDATA #IMPLIED

relationship (stack|left|right|top|bottom|fast)

ratio CDATA #IMPLIED

visible (true | false)

closeable (true | false)

moveable (true | false)

standalone (true | false)

showTitle (true | false) >

- **id –** the unique identifier of the view which will be added to the perspective layout.
- **relative –** the unique identifier of a view which already exists in the perspective. This will be used as a reference point for placement of the view. The relationship between these two views is defined by `relationship`. Ignored if relationship is "fast".
- **relationship –** specifies the relationship between `id` and `relative`. The following values are supported:
  fast – the view extension will be created as a fast view.
  stack – the view extension will be stacked with the relative view in a folder.
  left, right, top, bottom – the view extension will be placed beside the relative view. In this case a `ratio` must also be defined.
- **ratio –** the percentage of area within the relative view which will be donated to the view extension. If the view extension is a fast view, the ratio is the percentage of the workbench the fast view will cover when active. This must be defined as a floating point value and lie between 0.05 and 0.95.
- **visible –** whether the view is initially visible when the perspective is opened. This attribute should have a value of "true" or "false" if used. If this attribute is not used, the view will be initially visible by default.
- **closeable –** whether the view is closeable in the target perspective. This attribute should have a value of "true" or "false" if used. If this attribute is not used, the view will be closeable, unless the perspective itself is marked as fixed.
- **moveable –** whether the view is moveable. A non−moveable view cannot be moved either within the same folder, or moved between folders in the perspective. This attribute should have a value of "true" or "false" if used. If this attribute is not used, the view will be moveable, unless the perspective itself is marked as fixed.
- **standalone –** whether the view is a standalone view. A standalone view cannot be docked together with others in the same folder. This attribute should have a value of "true" or "false" if used. This attribute is ignored if the relationship attribute is "fast" or "stacked". If this attribute is not used, the view will be a regular view, not a standalone view (default is "false").
- **showTitle –** whether the view's title is shown. This attribute should have a value of "true" or "false" if used. This attribute only applies to standalone views. If this attribute is not used, the view's title will be shown (default is "true").

Examples:

The following is an example of a perspective extension (note the subelements and the way attributes are used):

```
<extension point=

"org.eclipse.ui.perspectiveExtensions"

>

<perspectiveExtension targetID=

"org.eclipse.ui.resourcePerspective"

>

<actionSet id=

"org.eclipse.jdt.ui.JavaActionSet"

/>

<viewShortcut id=

"org.eclipse.jdt.ui.PackageExplorer"

/>

<newWizardShortcut id=

"org.eclipse.jdt.ui.wizards.NewProjectCreationWizard"

/>

<perspectiveShortcut id=

"org.eclipse.jdt.ui.JavaPerspective"

/>

<view id=

"org.eclipse.jdt.ui.PackageExplorer"

relative=

"org.eclipse.ui.views.ResourceNavigator"
```

relationship=

"stack"

/>

<view id=

"org.eclipse.jdt.ui.TypeHierarchy"

relative=

"org.eclipse.ui.views.ResourceNavigator"

relationship=

"left"

ratio=

"0.50"

/>

</perspectiveExtension>

</extension>

In the example above, an action set, view shortcut, new wizard shortcut, and perspective shortcut are contributed to the initial contents of the Resource Perspective. In addition, the Package Explorer view is stacked on the Resource Navigator and the Type Hierarchy View is added beside the Resource Navigator.

API Information:

The items defined within the perspective extension are contributed to the initial contents of the target perspective. Following this, the user may remove any contribution or add others to a perspective from within the workbench user interface.

Perspectives

Identifier:

org.eclipse.ui.perspective

Description:

This extension point is used to add perspective factories to the workbench. A perspective factory is used to define the initial layout and visible action sets for a perspective. The user can select a perspective by invoking the "Open Perspective" submenu of the "Window" menu.

Configuration Markup:

<!ELEMENT extension (perspective*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT perspective (description?)>

<!ATTLIST perspective

id CDATA #REQUIRED

name CDATA #REQUIRED

class CDATA #REQUIRED

icon CDATA #IMPLIED

fixed (true | false) >

- **id –** a unique name that will be used to identify this perspective.
- **name –** a translatable name that will be used in the workbench window menu bar to represent this perspective.
- **class –** a fully qualified name of the class that implements `org.eclipse.ui.IPerspectiveFactory` interface.
- **icon –** a relative name of the icon that will be associated with this perspective.

- **fixed** – indicates whether the layout of the perspective is fixed. If true, then views created by the perspective factory are not closeable, and cannot be moved. The default is false.

<!ELEMENT description (#PCDATA)>

an optional subelement whose body should contain text providing a short description of the perspective.

Examples:

The following is an example of a perspective extension:

```
<extension point=
"org.eclipse.ui.perspectives"
>
<perspective id=
"org.eclipse.ui.resourcePerspective"
name=
"Resource"
class=
"org.eclipse.ui.internal.ResourcePerspective"
icon=
"icons/MyIcon.gif"
>
</perspective>
</extension>
```

API Information:

The value of the `class` attribute must be the fully qualified name of a class that implements `org.eclipse.ui.IPerspectiveFactory`. The class must supply the initial layout for a perspective when asked by the workbench.

The `plugin_customization.ini` file is used to define the default perspective. The *default perspective* is the first perspective which appears when the product is launched after install. It is also used when the user opens a page or window with no specified perspective. The default perspective is defined as a property within the plugin_customization.ini, as shown below. The user may also override this perspective from the workbench perspectives preference page.

```
defaultPerspectiveId = org.eclipse.ui.resourcePerspective
```

The perspectives which appear in the "Open Perspective" menu are shortcuts for perspective selection. This set is defined by the active perspective itself, and extensions made through the perspectiveExtensions extension point.

Supplied Implementation:

The workbench provides a "Resource Perspective". Additional perspectives may be added by plug−ins. They are selected using the "Open Perspective" submenu of the "Window" menu.

Pop−up Menus

Identifier:

org.eclipse.ui.popupMenus

Description:

This extension point is used to add new actions to context menus owned by other plug−ins. Action contributions may be made against a specific object type (`objectContribution`) or against a specific context menu of a view or editor part (`viewerContribution`). When using `objectContribution`, the contribution will appear in all view or editor part context menus where objects of the specified type are selected. In contrast, using `viewerContribution`, the contribution will only appear in the specified view or editor part context menu, regardless of the selection.

When the selection is heterogeneous, the contribution will be applied if registered against a common type of the selection, if possible. If a direct match is not possible, matching against superclasses and superinterfaces will be attempted.

Selection can be further constrained through the use of a name filter. If used, all the objects in the selection must match the filter in order to apply the contribution.

Individual actions in an object contribution can use the attribute `enablesFor` to specify if it should only apply for a single, multiple, or any other selection type.

If these filtering mechanisms are inadequate an action contribution may use the `filter` mechanism. In this case the attributes of the target object are described in a series of name−value pairs. The attributes which apply to the selection are type−specific and beyond the domain of the workbench itself, so the workbench will delegate filtering at this level to the actual selection.

An action's enablement and/or visibility can be defined using the elements `enablement` and `visibility` respectively. These two elements contain a boolean expression that is evaluated to determine the enablement and/or visibility.

The syntax is the same for the `enablement` and `visibility` elements. Both contain only one boolean expression sub−element. In the simplest case, this will be an `objectClass`, `objectState`, `pluginState`, or `systemProperty` element. In the more complex case, the `and`, `or`, and `not` elements can be combined to form a boolean expression. Both the `and`, and `or` elements must contain 2 sub−elements. The `not` element must contain only 1 sub−element.

Configuration Markup:

<!ELEMENT extension (objectContribution , viewerContribution)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT objectContribution (filter* , visibility? , enablement? , menu* , action*)>

<!ATTLIST objectContribution

id CDATA #REQUIRED

objectClass CDATA #REQUIRED

nameFilter CDATA #IMPLIED

adaptable (true | false) "false">

This element is used to define a group of actions and/or menus for any viewer context menus for which the objects of the specified type are selected.

- **id –** a unique identifier used to reference this contribution
- **objectClass –** a fully qualified name of the class or interface that all objects in the selection must subclass or implement.
- **nameFilter –** an optional wild card filter for the name that can be applied to all objects in the selection. No contribution will take place if there is no match.
- **adaptable –** a flag that indicates if types that adapt to IResource should use this object contribution. This flag is used only if objectClass adapts to IResource. Default value is false.

<!ELEMENT viewerContribution (visibility? , menu* , action*)>

<!ATTLIST viewerContribution

id CDATA #REQUIRED

targetID CDATA #REQUIRED>

This element is used to define a group of actions and/or menus for a specific view or editor part context menu.

- **id –** a unique identifier used to reference this contribution
- **targetID –** the unique identifier of a context menu inside a view or editor part.

<!ELEMENT action (selection* , enablement?)>

<!ATTLIST action

id CDATA #REQUIRED

label CDATA #REQUIRED

definitionId CDATA #IMPLIED

menubarPath CDATA #IMPLIED

icon CDATA #IMPLIED

helpContextId CDATA #IMPLIED

style (push|radio|toggle|pulldown)

state (true | false)

class CDATA #REQUIRED

enablesFor CDATA #IMPLIED

overrideActionId CDATA #IMPLIED

tooltip CDATA #IMPLIED>

This element defines an action that the user can invoke in the UI.

- **id –** a unique identifier used as a reference for this action.
- **label –** a translatable name used as the menu item text. The name can include mnenomic information.
- **definitionId –** This specifies the command that this action is handling. This is used to decide which key binding to display in the pop–up menu.
- **menubarPath –** a slash–delimited path ('/') used to specify the location of this action in the context menu. Each token in the path, except the last one, must represent a valid identifier of an existing menu in the hierarchy. The last token represents the named group into which this action will be added. If the path is omitted, this action will be added to the standard additions group defined by `IWorkbenchActionConstants.MB_ADDITIONS`.
- **icon –** a relative path of an icon used to visually represent the action in its context. The path is relative to the location of the plugin.xml file of the contributing plug–in.
- **helpContextId –** a unique identifier indicating the help context for this action. On some platforms, if the action appears as a menu item, then pressing the appropriate help key while the menu item is highlighted will display help. Not all platforms support this behaviour.
- **style –** an optional attribute to define the user interface style type for the action. If defined, the attribute value will be one of the following:

  push – as a regular menu item or tool item.

  radio – as a radio style menu item or tool item. Actions with the radio style within the same menu or toolbar group behave as a radio set. The initial value is specified by the `state` attribute.

toggle – as a checked style menu item or as a toggle tool item. The initial value is specified by the `state` attribute.

pulldown – as a cascading style menu item.

- **state** – an optional attribute indicating the initial state (either `true` or `false`), used when the `style` attribute has the value `radio` or `toggle`.
- **class** – a name of the fully qualified class that implements `org.eclipse.ui.IObjectActionDelegate` (for object contributions), `org.eclipse.ui.IViewActionDelegate` (for viewer contributions to a view part), or `org.eclipse.ui.IEditorActionDelegate` (for viewer contributions to an editor part). For backwards compatibility, `org.eclipse.ui.IActionDelegate` may be implemented for object contributions.
- **enablesFor** – a value indicating the selection count which must be met to enable the action. If this attribute is specified and the condition is met, the action is enabled. If the condition is not met, the action is disabled. If no attribute is specified, the action is enabled for any number of items selected. The following attribute formats are supported:

  ! – 0 items selected

  ? – 0 or 1 items selected

  + – 1 or more items selected

  multiple, 2+ – 2 or more items selected

  n – a precise number of items selected.a precise number of items selected. For example: enablesFor=" 4" enables the action only when 4 items are selected

  * – any number of items selected

  The enablement criteria for an action extension are initially defined by `enablesFor`, `selection` and `enablement`. However, once the action delegate has been instantiated it may control the action enable state directly within its `selectionChanged` method.
- **overrideActionId** – an optional attribute that specifies the identifier of an action which this action overrides. The action represented by this identifier will not be contributed to the context menu. The action identifier must be from a prerequisite plug–in only. This attribute is only applicable to action elements of an object contribution.
- **tooltip** – a translatable text representing the action's tool tip. Only used if the action appears in the toolbar.

<!ELEMENT filter EMPTY>

<!ATTLIST filter

name CDATA #REQUIRED

value CDATA #REQUIRED>

This element is used to evaluate the attribute state of each object in the current selection. A match only if each object in the selection has the specified attribute state. Each object in the selection must implement, or adapt to, `org.eclipse.ui.IActionFilter`.

- **name** – the name of an object's attribute. Acceptable names reflect the object type, and should be publicly declared by the plug–in where the object type is declared.
- **value** – the required value of the object's attribute. The acceptable values for the object's attribute should be publicly declared.

<!ELEMENT menu (separator+ , groupMarker*)>

<!ATTLIST menu

id CDATA #REQUIRED

label CDATA #REQUIRED

path CDATA #IMPLIED>

This element is used to defined a new menu.

- **id** – a unique identifier that can be used to reference this menu.
- **label** – a translatable name used by the Workbench for this new menu. The name should include mnemonic information.
- **path** – the location of the new menu starting from the root of the menu. Each token in the path must refer to an existing menu, except the last token which should represent a named group in the last menu in the path. If omitted, the new menu will be added to the `additions` named group of the menu.

<!ELEMENT separator EMPTY>

<!ATTLIST separator

name CDATA #REQUIRED>

This element is used to create a menu separator in the new menu.

- **name** – the name of the menu separator. This name can later be referenced as the last token in a menu path. Therefore, a separator also serves as named group into which actions and menus can be added.

<!ELEMENT groupMarker EMPTY>

<!ATTLIST groupMarker

name CDATA #REQUIRED>

Description:

This element is used to create a named group in the new menu. It has no visual representation in the new menu, unlike the `separator` element.

- **name –** the name of the group marker. This name can later be referenced as the last token in the menu path. It serves as named group into which actions and menus can be added.

<!ELEMENT selection EMPTY>

<!ATTLIST selection

class CDATA #REQUIRED

name CDATA #IMPLIED>

This element is used to help determine the action enablement based on the current selection. Ignored if the `enablement` element is specified.

- **class –** a fully qualified name of the class or interface that each object in the selection must implement in order to enable the action.
- **name –** an optional wild card filter for the name that can be applied to all objects in the selection. If specified and the match fails, the action will be disabled.

<!ELEMENT enablement (and | or | not | objectClass | objectState | pluginState | systemProperty)>

This element is used to define the enablement for the extension.

<!ELEMENT visibility (and | or | not | objectClass | objectState | pluginState | systemProperty)>

This element is used to define the visibility for the extension.

<!ELEMENT and (and | or | not | objectClass | objectState | pluginState | systemProperty)>

This element represent a boolean AND operation on the result of evaluating its two sub−element expressions.

<!ELEMENT or (and | or | not | objectClass | objectState | pluginState | systemProperty)>

This element represent a boolean OR operation on the result of evaluating its two sub−element expressions.

<!ELEMENT not (and | or | not | objectClass | objectState | pluginState | systemProperty)>

This element represent a boolean NOT operation on the result of evaluating its sub−element expressions.

<!ELEMENT objectClass EMPTY>

<!ATTLIST objectClass

name CDATA #REQUIRED>

This element is used to evaluate the class or interface of each object in the current selection. If each object in the selection implements the specified class or interface, the expression is evaluated as true.

- **name −** a fully qualified name of a class or interface. The expression is evaluated as true only if all objects within the selection implement this class or interface.

<!ELEMENT objectState EMPTY>

<!ATTLIST objectState

name CDATA #REQUIRED

value CDATA #REQUIRED>

This element is used to evaluate the attribute state of each object in the current selection. If each object in the selection has the specified attribute state, the expression is evaluated as true. To evaluate this type of expression, each object in the selection must implement, or adapt to, `org.eclipse.ui.IActionFilter` interface.

- **name −** the name of an object's attribute. Acceptable names reflect the object type, and should be publicly declared by the plug−in where the object type is declared.
- **value −** the required value of the object's attribute. The acceptable values for the object's attribute should be publicly declared.

<!ELEMENT pluginState EMPTY>

<!ATTLIST pluginState

id CDATA #REQUIRED

value (installed|activated) "installed">

This element is used to evaluate the state of a plug−in. The state of the plug−in may be one of the following: `installed` (equivalent to the OSGi concept of "resolved") or `activated` (equivalent to the OSGi concept of "active").

- **id** − the identifier of a plug−in which may or may not exist in the plug−in registry.
- **value** − the required state of the plug−in. The state of the plug−in may be one of the following: `installed` (equivalent to the OSGi concept of "resolved") or `activated` (equivalent to the OSGi concept of "active").

<!ELEMENT systemProperty EMPTY>

<!ATTLIST systemProperty

name CDATA #REQUIRED

value CDATA #REQUIRED>

This element is used to evaluate the state of some system property. The property value is retrieved from the `java.lang.System`.

- **name** − the name of the system property.
- **value** − the required value of the system property.

<!ELEMENT enablement (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

A generic root element. The element can be used inside an extension point to define its enablement expression. The children of an enablement expression are combined using the and operator.

<!ELEMENT not (not | and | or | instanceof | test | systemTest | equals | count | with | resolve | adapt | iterate)>

This element represent a NOT operation on the result of evaluating it's sub−element expression.

<!ELEMENT and (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

This element represent an AND operation on the result of evaluating all it's sub−elements expressions.

<!ELEMENT or (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

This element represent an OR operation on the result of evaluating all it's sub−element expressions.

<!ELEMENT instanceof EMPTY>

<!ATTLIST instanceof

value CDATA #REQUIRED>

This element is used to perform an instanceof check of the object in focus. The expression returns EvaluationResult.TRUE if the object's type is a sub type of the type specified by the attribute value. Otherwise EvaluationResult.FALSE is returned.

- **value** – a fully qualified name of a class or interface.

<!ELEMENT test EMPTY>

<!ATTLIST test

property CDATA #REQUIRED

args CDATA #IMPLIED

value CDATA #IMPLIED>

This element is used to evaluate the property state of the object in focus. The set of testable properties can be extended using the propery tester extension point. The test expression returns EvaluationResult.NOT_LOADED if teh property tester doing the actual testing isn't loaded yet.

- **property** – the name of an object's property to test.
- **args** – additional arguments passed to the property tester. Multiple arguments are seperated by commas. Each individual argument is converted into a Java base type using the same rules as defined for the value attribute of the test expression.
- **value** – the expected value of the property. Can be omitted if the property is a boolean property. The test expression is supposed to return EvaluationResult.TRUE if the property matches the value and EvaluationResult.FALSE otherwise. The value attribute is converted into a Java base type using the following rules:
    - the string "true" is converted into Boolean.TRUE
    - the string "false" is converted into Boolean.FALSE
    - if the string contains a dot then the interpreter tries to convert the value into a Float object. If this fails the string is treated as a java.lang.String
    - if the string only consists of numbers then the interpreter converts the value in an Integer object.
    - in all other cases the string is treated as a java.lang.String
    - the conversion of the string into a Boolean, Float, or Integer can be suppressed by surrounding the string with single quotes. For example, the attribute value="'true'" is converted into the string "true"

<!ELEMENT systemTest EMPTY>

<!ATTLIST systemTest

property CDATA #REQUIRED

value CDATA #REQUIRED>

Tests a system property by calling the System.getProperty method and compares the result with the value specified through the value attribute.

- **property** – the name of an system property to test.
- **value** – the expected value of the property. The value is interpreted as a string value.

<!ELEMENT equals EMPTY>

<!ATTLIST equals

value CDATA #REQUIRED>

This element is used to perform an equals check of the object in focus. The expression returns EvaluationResult.TRUE if the object is equal to the value provided by the attribute value. Otherwise EvaluationResult.FALSE is returned.

- **value** – the operatand of the equals tests. The value provided as a string is converted into a Java base type using the same rules as for the value attribute of the test expression.

<!ELEMENT count EMPTY>

<!ATTLIST count

value CDATA #REQUIRED>

This element is used to test the number of elements in a collection.

- **value** – an expression to specify the number of elements in a list. Following wildcard characters can be used:
  *
  any number of elements
  ?
  no elements or one element
  +
  one or more elements
  !
  no elements
  integer value
  the list must contain the exact number of elements

<!ELEMENT with (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST with

variable CDATA #REQUIRED>

This element changes the object to be inspected for all its child element to the object referneced by the given variable. If the variable can not be resolved then the expression will throw a ExpressionException when evaluating it. The children of a with expression are combined using the and operator.

- **variable** – the name of the variable to be used for further inspection. It is up to the evaluator of an extension point to provide the variable in the variable pool.

<!ELEMENT resolve (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST resolve

variable CDATA #REQUIRED

args CDATA #IMPLIED>

This element changes the object to be inspected for all its child element to the object referneced by the given variable. If the variable can not be resolved then the expression will throw a ExpressionException when evaluating it. The children of a with expression are combined using the and operator.

- **variable –** the name of the variable to be resolved. This variable is then used as the object in focus for child element evaluation. It is up to the evaluator of an extension point to provide a corresponding variable resolver (see IVariableResolver) through the evaluation context passed to the root expression element when evaluating the expression.
- **args –** additional arguments passed to the variable resolver. Multiple arguments are seperated by commas. Each individual argument is converted into a Java base type using the same rules as defined for the value attribute of the test expression.

<!ELEMENT adapt (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST adapt

type CDATA #REQUIRED>

This element is used to adapt the object in focus to the type specified by the attribute type. The expression returns not loaded if either the adapter or the type referenced isn't loaded yet. It throws a ExpressionException during evaluation if the type name doesn't exist at all. The children of an adapt expression are combined using the and operator.

- **type –** the type to which the object in focus is to be adapted.

<!ELEMENT iterate (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST iterate

operator (or|and) >

This element is used to iterate over a variable that is of type java.util.Collection. If the object in focus is not of type java.util.Collection then an ExpressionException will be thrown while evaluating the expression.

- **operator** – either "and" or "or". The operator defines how the child elements will be combined. If not specified, "and" will be used.

Examples:

The following is an example of a pop–up menu extension point:

<extension point=

"org.eclipse.ui.popupMenus"

>

<objectContribution id=

"com.xyz.C1"

objectClass=

"org.eclipse.core.resources.IFile"

nameFilter=

"*.java"

>

<menu id=

"com.xyz.xyzMenu"

path=

"additions"

label=

"&XYZ Java Tools"

>

<separator name=

"group1"

/>

```
</menu>
<action id=
"com.xyz.runXYZ"
label=
"&amp;amp;Run XYZ Tool"
style=
"push"
menubarPath=
"com.xyz.xyzMenu/group1"
icon=
"icons/runXYZ.gif"
helpContextId=
"com.xyz.run_action_context"
class=
"com.xyz.actions.XYZToolActionDelegate"
enablesFor=
"1"
/>
</objectContribution>
<viewerContribution id=
"com.xyz.C2"
targetID=
"org.eclipse.ui.views.TaskList"
>
<action id=
"com.xyz.showXYZ"
```

label=

"&Show XYZ"

style=

"toggle"

state=

"true"

menubarPath=

"additions"

icon=

"icons/showXYZ.gif"

helpContextId=

"com.xyz.show_action_context"

class=

"com.xyz.actions.XYZShowActionDelegate"

/>

</viewerContribution>

</extension>

In the example above, the specified object contribution action will only enable for a single selection (`enablesFor` attribute). In addition, each object in the selection must implement the specified interface (`IFile`) and must be a Java file. This action will be added into a submenu previously created. This contribution will be effective in any view that has the required selection.

In contrast, the viewer contribution above will only appear in the Tasks view context menu, and will not be affected by the selection in the view.

The following is an example of the filter mechanism. In this case the action will only appear for IMarkers which are completed and have high priority.

<extension point=

"org.eclipse.ui.popupMenus"

```
>
<objectContribution id=
"com.xyz.C3"
objectClass=
"org.eclipse.core.resources.IMarker"
>
<filter name=
"done"
value=
"true"
/>
<filter name=
"priority"
value=
"2"
/>
<action id=
"com.xyz.runXYZ"
label=
"High Priority Completed Action Tool"
icon=
"icons/runXYZ.gif"
class=
"com.xyz.actions.MarkerActionDelegate"
>
</action>
```

```
</objectContribution>

</extension>
```

The following is an other example of using the visibility element:

```
<extension point=

"org.eclipse.ui.popupMenus"

>

<viewerContribution id=

"com.xyz.C4"

targetID=

"org.eclipse.ui.views.TaskList"

>

<visibility>

<and>

<pluginState id=

"com.xyz"

value=

"activated"

/>

<systemProperty name=

"ADVANCED_MODE"

value=

"true"

/>

</and>

</visibility>
```

```
<action id=

"com.xyz.showXYZ"

label=

"&amp;amp;Show XYZ"

style=

"push"

menubarPath=

"additions"

icon=

"icons/showXYZ.gif"

helpContextId=

"com.xyz.show_action_context"

class=

"com.xyz.actions.XYZShowActionDelegate"

>

</action>

</viewerContribution>

</extension>
```

In the example above, the specified action will appear as a menu item in the Task view context menu, but only if the "com.xyz" plug−in is active and the specified system property is set to true.

API Information:

The value of the action attribute `class` must be a fully qualified class name of a Java class that implements `org.eclipse.ui.IObjectActionDelegate` in the case of object contributions, `org.eclipse.ui.IViewActionDelegate` for contributions to context menus that belong to views, or `org.eclipse.ui.IEditorActionDelegate` for contributions to context menus that belong to editors. In all cases, the implementing class is loaded as late as possible to avoid loading the entire plug−in before it is really needed.

Note: For backwards compatibility, `org.eclipse.ui.IActionDelegate` may be implemented for object contributions.

Conext menu extension within a part is only possible when the target part publishes a menu for extension. This is strongly encouraged, as it improves the extensibility of the product. To accomplish this each part should publish any context menus which are defined by calling `IWorkbenchPartSite.registerContextMenu`. Once this has been done the workbench will automatically insert any action extensions which exist.

A menu id must be provided for each registered menu. For consistency across parts the following strategy should be adopted by all part implementors.

- If the target part has only one context menu it should be registered with id == part id. This can be done easily by calling `registerContextMenu(MenuManager, ISelectionProvider)`. Extenders may use the part id itself as the targetID for the action extension.
- If the target part has more than one context menu a unique id should be defined for each. Prefix each id with the part id and publish these id's within the javadoc for the target part. Register each menu at runtime by calling `registerContextMenu(String, MenuManager, ISelectionProvider)`. Extenders will use the unique menu id as the targetID for the action extension.

Any context menu which is registered with the workbench also should contain a standard insertion point with id `IWorkbenchActionConstants.MB_ADDITIONS`. Other plug−ins will use this value as a reference point for insertion. The insertion point may be defined by adding a GroupMarker to the menu at an appropriate location for insertion.

An object in the workbench which is the selection in a context menu may define an `org.eclipse.ui.IActionFilter`. This is a filtering strategy which can perform type specific filtering. The workbench will retrieve the filter for the selection by testing to see if it implements `IActionFilter`. If that fails, the workbench will ask for a filter through the `IAdaptable` mechanism.

Action and menu labels may contain special characters that encode mnemonics which are specified using the ampersand ('&') character in front of a selected character in the translated text. Since ampersand is not allowed in XML strings, use `&` character entity.

If two or more actions are contributed to a menu by a single extension the actions will appear in the reverse order of how they are listed in the plugin.xml file. This behavior is admittedly unintuitive. However, it was discovered after the Eclipse Platform API was frozen. Changing the behavior now would break every plug−in which relies upon the existing behavior.

The `selection` and `enablement` elements are mutually exclusive. The `enablement` element can replace the `selection` element using the sub−elements `objectClass` and `objectState`. For example, the following:

<selection class=

"org.eclipse.core.resources.IFile"

name=

"*.java"

>

</selection>

can be expressed using:

<enablement>

<and>

<objectClass name=

"org.eclipse.core.resources.IFile"

/>

<objectState name=

"extension"

value=

"java"

/>

</and>

</enablement>

Supplied Implementation:

The workbench views have built−in context menus that already come loaded with a number of actions. Plug−ins can contribute to these menus. If a viewer has reserved slots for these contributions and they are made public, slot names can be used as paths. Otherwise, actions and submenus will be added at the end of the pop−up menu.

Preference Pages

Identifier:

org.eclipse.ui.preferencePages

Description:

The workbench provides one common dialog box for preferences. The purpose of this extension point is to allow plug–ins to add pages to the preference dialog box. When preference dialog box is opened (initiated from the menu bar), pages contributed in this way will be added to the dialog box.

The preference dialog box provides for hierarchical grouping of the pages. For this reason, a page can optionally specify a `category` attribute. This attribute represents a path composed of parent page IDs separated by '/'. If this attribute is omitted or if any of the parent nodes in the path cannot be found, the page will be added at the root level.

Configuration Markup:

<!ELEMENT extension (page*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT page (keywordReference*)>

<!ATTLIST page

id CDATA #REQUIRED

name CDATA #REQUIRED

class CDATA #REQUIRED

category CDATA #IMPLIED>

- **id –** a unique name that will be used to identify this page.
- **name –** a translatable name that will be used in the UI for this page.
- **class –** a name of the fully qualified class that implements `org.eclipse.ui.IWorkbenchPreferencePage`.

- **category** – a path indicating the location of the page in the preference tree. The path may either be a parent node ID or a sequence of IDs separated by '/', representing the full path from the root node.

<!ELEMENT keywordReference EMPTY>

<!ATTLIST keywordReference

id CDATA #REQUIRED>

A reference by a preference page to a keyword. See the keywords extension point.

- **id** – The id of the keyword being referred to.

Examples:

The following is an example for the preference extension point:

<extension point=

"org.eclipse.ui.preferencePages"

>

<page id=

"com.xyz.prefpage1"

name=

"XYZ"

class=

"com.xyz.prefpages.PrefPage1"

>

<keywordReference id=

"xyz.Keyword"

/>

```
</page>

<page id=

"com.xyz.prefpage2"

name=

"Keyboard Settings"

class=

"com.xyz.prefpages.PrefPage2"

category=

"com.xyz.prefpage1"

>

</page>

</extension>
```

API Information:

The value of the attribute class must represent a fully qualified name of the class that implements `org.eclipse.ui.IWorkbenchPreferencePage`.

Supplied Implementation:

The workbench adds several pages for setting the preferences of the platform. Pages registered through this extension will be added after them according to their category information.

Preference Transfer

Identifier:

org.eclipse.ui.preferenceTransfer

Since:

3.1

Description:

The workbench provides support for maintaining preferences. The purpose of this extension point is to allow plug−ins to add specific support for saving and loading specific groups of preferences. Typically this is used for operations such as Import and Export.

Configuration Markup:

<!ELEMENT extension (transfer*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point −** a fully qualified identifier of the target extension point
- **id −** an optional identifier of the extension instance
- **name −** an optional name of the extension instance

<!ELEMENT transfer (mapping+ , description?)>

<!ATTLIST transfer

id CDATA #REQUIRED

name CDATA #REQUIRED

icon CDATA #IMPLIED>

- **id −** a unique name that will be used to identify this transfer
- **name −** a translatable name that will be used in UI when listing this item
- **icon −** a relative name of the icon that will be used when displaying the transfer

<!ELEMENT description (#PCDATA)>

an optional subelement whose body should represent a short description of the transfer engine functionality.

<!ELEMENT mapping (entry*)>

<!ATTLIST mapping

scope CDATA #REQUIRED>

a subelement which represents a `org.eclipse.core.runtime.preferences.IPreferenceFilter`. It specifies 1 or more mappings from a scope `org.eclipse.core.runtime.preferences.IScope` to 0 or more nodes where 0 or more keys are specified per node. The rules for export and import behaviour for a filter can be found in the specifications in `org.eclipse.core.runtime.preferences.IPreferenceFilter`.

- **scope –** an attribute specifying the name of the scope to map the provided nodes and keys to (for example "instance")

<!ELEMENT entry (key*)>

<!ATTLIST entry

node CDATA #IMPLIED>

a subelement specifing the nodes and keys that are to be transferred for a given scope

- **node –** an attribute specifing the preference node within the given scope to be transferred

<!ELEMENT key EMPTY>

<!ATTLIST key

name CDATA #REQUIRED>

- **name –** an attribute specifing a comma separated list of preference keys within the given node to be transferred, specifying the key attribute without specifying a matching node will result in a runtime error.

Examples:

Example that export all tranfers, exports all nodes for specified scopes.

<extension point=

"org.eclipse.ui.preferenceTransfer"

>

<transfer icon=

"XYZ.gif"

name=

"Export All Transfer Test"

id=

"org.eclipse.ui.tests.all"

>

<mapping scope=

"instance"

>

</mapping>

<mapping scope=

"configuration"

>

</mapping>

<mapping scope=

"project"

>

</mapping>

<description>

Export all tranfer, exports all nodes for specified scopes

</description>

</transfer>

</extension>

Very Simple Transfer only provides required info and no more.

<extension point=

"org.eclipse.ui.preferenceTransfer"

>

<transfer name=

"Bare Bones Transfer Test"

id=

"org.eclipse.ui.tests.all"

>

<mapping scope=

"instance"

>

</mapping>

</transfer>

</extension>

Example that exports many combinations of keys and nodes

<extension point=

"org.eclipse.ui.preferenceTransfer"

>

<transfer icon=

"XYZ.gif"

name=

"Export many preferences"

id=

"org.eclipse.ui.tests.all"

>

<mapping scope=

"instance"

>

<entry node=

"org.eclipse.ui"

>

<key name=

"showIntro,DOCK_PERSPECTIVE_BAR"

/>

</entry>

<entry node=

"org.eclipse.ui.workbench"

>

<key name=

"bogus,RUN_IN_BACKGROUND"

/>

</entry>

<entry node=

"org.eclipse.ui.ide"

/>

<entry node=

"org.eclipse.core.resources"

/>

</mapping>

<mapping scope=

"configuration"

>

</mapping>

<description>

Export many combinations of keys and nodes

</description>

</transfer>

</extension>

Presentation Factories

Identifier:

org.eclipse.ui.workbench.presentationFactories

Since:

3.0

Description:

This extension point is used to add presentation factories to the workbench. A presentation factory defines the overall look and feel of the workbench, including how views and editors are presented.

Configuration Markup:

<!ELEMENT extension (factory*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

<!ELEMENT factory EMPTY>

<!ATTLIST factory

class CDATA #REQUIRED

id CDATA #REQUIRED

name CDATA #REQUIRED>

- **class –** Specify the fully qualified class to be used for the presentation factory. The specified value must be a subclass of `org.eclipse.ui.presentations.AbstractPresentationFactory`.
- **id –** a unique name that will be used to identify this presentation factory
- **name –** a translatable name that can be used to show this presentation factory in the UI

Examples:

The following is an example of a presentationFactory extension:

<extension point=

"org.eclipse.ui.presentationFactories"

>

<factory class=

"org.eclipse.ui.workbench.ExampleWorkbenchPresentationFactory"

/>

</extension>

API Information:

The class specified in the factory element must be a concrete subclass of `org.eclipse.ui.presentations.AbstractPresentationFactory`.

Supplied Implementation:

If a presentation factory is not specified or is missing then the implementation in `org.eclipse.ui.presentations.WorkbenchPresentationFactory` will be used.

Property Pages

Identifier:

org.eclipse.ui.propertyPages

Description:

This extension point is used to add additional property page for objects of a given type. Once defined, these property pages will appear in the Properties Dialog for objects of that type.

A property page is a user friendly way to interact with the properties of an object. Unlike the Properties view, which restricts the space available for editing an object property, a property page may benefit from the freedom to define larger, more complex controls with labels, icons, etc. Properties which logically go together may also be clustered in a page, rather than scattered in the property sheet. However, in most applications it will be appropriate to expose some properties of an object via the property sheet and some via the property pages.

Property pages are shown in a dialog box that is normally visible when the "Properties" menu item is selected on a pop−up menu for an object. In addition to the object class, the name filter can optionally be supplied to register property pages only for specific object types.

If these filtering mechanisms are inadequate a property page may use the filter mechanism. In this case the attributes of the target object are described in a series of key value pairs. The attributes which apply to the selection are type specific and beyond the domain of the workbench itself, so the workbench will delegate filtering at this level to the actual selection.

Configuration Markup:

<!ELEMENT extension (page*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point** − a fully qualified identifier of the target extension point
- **id** − an optional identifier of the extension instance
- **name** − an optional name of the extension instance

<!ELEMENT page (filter* , keywordReference*)>

<!ATTLIST page

id CDATA #REQUIRED

name CDATA #REQUIRED

icon CDATA #IMPLIED

objectClass CDATA #REQUIRED

class CDATA #REQUIRED

nameFilter CDATA #IMPLIED

adaptable (true | false)

category CDATA #IMPLIED>

- **id –** a unique name that will be used to identify this page
- **name –** a translatable name that will be used in the UI for this page
- **icon –** a relative path to an icon that will be used in the UI in addition to the page name
- **objectClass –** a fully qualified name of the class for which the page is registered.
- **class –** a fully qualified name of the class that implements `org.eclipse.ui.IWorkbenchPropertyPage`.
- **nameFilter –** an optional attribute that allows registration conditional on wild card match applied to the target object name.
- **adaptable –** a flag that indicates if types that adapt to IResource should use this property page. This flag is used if objectClass adapts to IResource. Default value is false.
- **category –** A path indicating the location of the page in the properties tree. The path may either be a parent node ID or a sequence of IDs separated by '/', representing the full path from the root node.

<!ELEMENT filter EMPTY>

<!ATTLIST filter

name CDATA #REQUIRED

value CDATA #REQUIRED>

This element is used to evaluate the attribute state of each object in the current selection. A match only if each object in the selection has the specified attribute state. Each object in the selection must implement, or adapt to, `org.eclipse.ui.IActionFilter`.

- **name –** the name of an object attribute.
- **value –** the value of an object attribute. In combination with the name attribute, the name value pair is used to define the target object for a property page.

<!ELEMENT keywordReference EMPTY>

<!ATTLIST keywordReference

id CDATA #IMPLIED>

A reference by a property page to a keyword. See the keywords extension point.

- **id** – The id of the keyword being referred to.

Examples:

The following is an example of the property page definition:

<extension point=

"org.eclipse.ui.propertyPages"

>

<page id=

"com.xyz.projectPage"

name=

"XYZ Java Properties"

objectClass=

"org.eclipse.core.resources.IFile"

class=

"com.xyz.ppages.JavaPropertyPage"

nameFilter=

"*.java"

>

<filter name=

"readOnly"

value=

"true"

/>

</page>

</extension>

API Information:

The attribute `class` must specify a fully qualified name of the class that implements `org.eclipse.ui.IWorkbenchPropertyPage`.

Supplied Implementation:

Some objects provided by the workbench may have property pages registered. Plug−ins are allowed to add more property pages for these objects. Property pages are not limited to workbench resources: all objects showing up in the workbench (even domain specific objects created by the plug−ins) may have property pages and other plug−ins are allowed to register property pages for them.

Startup

Identifier:

org.eclipse.ui.startup

Since:

Release 2.0

Description:

This extension point is used to register plugins that want to be activated on startup. The plugin class or the class given as the attribute on the startup element must implement the interface `org.eclipse.ui.IStartup`. Once the workbench is started, the method earlyStartup() will be called from a separate thread. If the startup element has a class attribute, the class will be instantiated and earlyStartup() will be called on the result. Otherwise, this method will be called on the plug−in class. Do not specify the plug−in class as the value of the class attribute, or it will be instantiated twice (once by regular plug−in activation, and once by this mechanism). These plugins are listed in the workbench preferences and the user may disable any plugin from early startup.

Configuration Markup:

<!ELEMENT extension (startup*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point** − a fully qualified identifier of the target extension point
- **id** − an optional identifier of the extension instance
- **name** − an optional name of the extension instance

<!ELEMENT startup EMPTY>

<!ATTLIST startup

class CDATA #IMPLIED>

- **class** − a fully qualified name of the class that implements `org.eclipse.ui.IStartup`. If not specified, the plug−in class is used. Do not specify the plug−in class as an explicit value, or it will be instantiated twice (once by regular plug−in activation, and once by this mechanism). Since release 3.0.

Examples:

Following is an example of a startup extension:

<extension point=

"org.eclipse.ui.startup"

>

<startup class=

"org.eclipse.example.StartupClass"

/>

</extension>

API Information:

See interface `org.eclipse.ui.IStartup`.

System Summary Sections

Identifier:

org.eclipse.ui.systemSummarySections

Since:

3.0

Description:

The Eclipse UI provides an AboutDialog that can be branded and reused by client product plugins. This dialog includes a SystemSummary dialog that contains configuration details. By extending the org.eclipse.ui.systemSummarySections extension point clients are able to put their own information into the log.

Configuration Markup:

<!ELEMENT extension (section+)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT section EMPTY>

<!ATTLIST section

id CDATA #IMPLIED

sectionTitle CDATA #REQUIRED

class CDATA #REQUIRED>

- **id –** an optional, unique name that will be used to identify this system summary section
- **sectionTitle –** a translatable name that will be displayed as the title of this section in the system summary
- **class –** The fully qualified name of a class the implements `org.eclipse.ui.about.ISystemSummarySection`. The class must provide a default constructor.

Examples:

Following is an example of a systemSummarySections extension:

<extension point=

"org.eclipse.ui.systemSummarySections"

>

<section id=

"RCPBrowser.CookieDetails"

sectionTitle=

"Browser Cookies"

class=

"org.eclipse.ui.examples.rcp.browser.CookieConfigDetails"

/>

</extension>

API Information:

The class specified in the section element must be a concrete subclass of `org.eclipse.ui.about.ISystemSummarySection`.

Supplied Implementation:

The Workbench uses this extension point to provide the following sections in the system summary dialog:

- System properties: The properties returned by java.lang.System.getProperties().
- Features: The installed features.
- Plug−in Registry: The installed plug−ins and their status.
- User Preferences: The preferences that have been modified from their default values.

Themes

Identifier:

org.eclipse.ui.themes

Since:

3.0

Description:

This extension point is used to customize the appearance of the UI. It allows definition of color and font entities as well as theme entitities. Themes allow applications to selectivly override default color and font specifications for particular uses.

Configuration Markup:

<!ELEMENT extension (theme* , colorDefinition* , fontDefinition* , themeElementCategory* , data* , categoryPresentationBinding*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT theme (colorOverride* , fontOverride* , description? , data*)>

<!ATTLIST theme

id CDATA #REQUIRED

name CDATA #IMPLIED>

A collection of font, color and data overrides. Such a collection may be used to alter the appearance of the workbench. Many theme elements may exist with the same id. This allows compontent authors to contribute to existing themes.

- **id –** a unique name that will be used to identify this theme

- **name** – a translatable name of the theme to be presented to the user. At least one theme definition with any given id should contain this attribute.

<!ELEMENT themeElementCategory (description)>

<!ATTLIST themeElementCategory

id CDATA #REQUIRED

parentId CDATA #IMPLIED

class CDATA #IMPLIED

label CDATA #IMPLIED>

A logical grouping of theme element definitions. This category may include colors and fonts.

- **id** – the id for this category
- **parentId** – the id of the parent category, if any.
- **class** – a class that implements `org.eclipse.ui.themes.IThemePreview`
- **label** – a translatable name of the theme element category to be presented to the user

<!ELEMENT colorDefinition (colorFactory? , colorValue* , description?)>

<!ATTLIST colorDefinition

id CDATA #IMPLIED

label CDATA #REQUIRED

defaultsTo CDATA #IMPLIED

value CDATA #IMPLIED

categoryId CDATA #IMPLIED

colorFactory CDATA #IMPLIED

isEditable (true | false) >

A symbolic color definition.

- **id** – a unique id that can be used to identify this color definition.

Since:

- **label –** a translatable name of the color to be presented to the user.
- **defaultsTo –** the id of another color definition that is the default setting for the receiver. When there is no preference for this color the color registry will have the value of defaultsTo set for it in the registry. Only one of `defaultsTo`, `value` or `colorFactory` may be defined.
- **value –** The default value of this color. The value may be specified in the following ways:
  - ♦ a String containing comma seperated integers in the form red,green,blue
  - ♦ a String that maps to an SWT color constant (ie: COLOR_RED).

  Only one of `defaultsTo`, `value` or `colorFactory` may be defined. If `value` is specified, additional value definitions may be specified on a per platform/windowing system basis via the `colorValue` element.
- **categoryId –** the optional id of the `themeElementCategory` this color belongs to.
- **colorFactory –** a class that implements `org.eclipse.ui.themes.IColorFactory`. This may be used instead of `value` to specify the default value of the color. Please be advised that this should be used with caution – usage of this attribute will cause plugin activation on workbench startup.
- **isEditable –** whether the user should be allowed to edit this color in the preference page. If this is `false` then the contribution is not shown to the user.

<!ELEMENT fontDefinition (fontValue* , description?)>

<!ATTLIST fontDefinition

id CDATA #REQUIRED

label CDATA #REQUIRED

value CDATA #IMPLIED

categoryId CDATA #IMPLIED

defaultsTo CDATA #IMPLIED

isEditable (true | false) >

A symbolic font definition.

- **id –** a unique name that can be used to identify this font definition.
- **label –** a translatable name of the font to be presented to the user.
- **value –**

  the font value. This is in the form: `fontname-style-height` where `fontname` is the name of a font, `style` is a font style (one of `"regular"`, `"bold"`, `"italic"`, or `"bold italic"`) and `height` is an integer representing the font height.

  Example: `Times New Roman-bold-36`.

  Only one (or neither) of `value` or `defaultsTo` may be used.

Since:

If `value` is specified, additional value definitions may be specified on a per platform/windowing system basis via the `fontValue` element.

- **categoryId –** the optional id of the `themeElementCategory` this font belongs to.
- **defaultsTo –** the id of another font definition that is the default setting for the receiver. When there is no preference for this font the font registry will have the value of defaultsTo set for it in the registry.

  Only one (or neither) of `value` or `defaultsTo` may be used.
- **isEditable –** whether the user should be allowed to edit this color in the preference page. If this is `false` then the contribution is not shown to the user.

<!ELEMENT colorOverride (colorFactory? , colorValue*)>

<!ATTLIST colorOverride

id CDATA #REQUIRED

value CDATA #IMPLIED

colorFactory CDATA #IMPLIED>

Allows overriding of colors defined in `colorDefinition` elements. These colors will be applied when the theme is in use.

- **id –** a unique id that can be used to identify this color definition override. This should match an existing font identifier. Strictly speaking, you may override colors that do not exist in the base theme although this practice is not recommended. In effect, such overrides will have behaviour similar to `colorDefinitions` that have `isEditable` set to `false`.
- **value –** the overriding value of this color. The value may be specified in the following ways:
  - ♦ a String containing comma seperated integers in the form red,green,blue
  - ♦ a String that maps to an SWT color constant (ie: COLOR_RED).

  Only one of `value` or `colorFactory` may be defined. Unlike a `colorDefinition`, you may not supply a `defaultsTo` for an override.
- **colorFactory –** a class that implements `org.eclipse.ui.themes.IColorFactory`. This may be used instead of `value` to specify the default value of the color. Please be advised that this should be used with caution – usage of this attribute will cause plugin activation on workbench startup.

<!ELEMENT fontOverride (fontValue*)>

<!ATTLIST fontOverride

id CDATA #REQUIRED

value CDATA #REQUIRED>

Since:

Allows overriding of fonts defined in `fontsDefinition` elements. These fonts will be applied when the theme is in use.

- **id –** a unique id that can be used to identify this font definition override. This should match an existing font identifier. Strictly speaking, you may override fonts that do not exist in the base theme although this practice is not recommended. In effect, such overrides will have behaviour similar to `fontDefinition`s that have `isEditable` set to `false`.
- **value –**

  the overriding value of this font. This is in the form: `fontname-style-height` where `fontname` is the name of a font, `style` is a font style (one of "`regular`", "`bold`", "`italic`", or "`bold italic`") and `height` is an integer representing the font height.

  Example: `Times New Roman-bold-36`.

  `value` must be defined for a font override. Unlike a `fontDefinition`, you may not supply a `defaultsTo` for a `fontOverride`.

<!ELEMENT description (#PCDATA)>

A short description of the elements usage.

<!ELEMENT colorFactory (parameter*)>

<!ATTLIST colorFactory

class CDATA #REQUIRED

plugin CDATA #IMPLIED>

The element version of the `colorFactory` attribute. This is used when the colorFactory implements `org.eclipse.core.runtime.IExecutableExtension` and there is parameterized data that you wish used in its initialization.

- **class –** a class that implements `org.eclipse.ui.themes.IColorFactory`. It may also implement `org.eclipse.core.runtime.IExecutableExtension`.
- **plugin –** the identifier of the plugin that contains the class

<!ELEMENT parameter EMPTY>

<!ATTLIST parameter

name CDATA #REQUIRED

value CDATA #REQUIRED>

A parameter element to be used within the colorFactory element. This will be passed as initialization data to the colorFactory class.

- **name** – the parameter name
- **value** – the parameter value

<!ELEMENT data EMPTY>

<!ATTLIST data

name CDATA #REQUIRED

value CDATA #REQUIRED>

An element that allows arbitrary data to be associated with a theme or the default theme. This data may be gradient directions or percentages, labels, author information, etc.

This element has behaviour similar to definitions and overrides. If a key is present in both the default theme and an identified theme, then the identified themes value will be used when that theme is accessed. If the identified theme does not supply a value then the default is used.

- **name** – the data name,
- **value** – the data value

<!ELEMENT colorValue (colorFactory?)>

<!ATTLIST colorValue

os   CDATA #IMPLIED

ws   CDATA #IMPLIED

value   CDATA #IMPLIED

colorFactory CDATA #IMPLIED>

This element allows specification of a color value on a per−platform basis.

Since:

- **os –** an optional os string to enable choosing of color based on current OS. eg: win32,linux
- **ws –** an optional os string to enable choosing of color based on current WS. eg: win32,gtk
- **value –** The default value of this color. The value may be specified in the following ways:
  - ♦ a String containing comma seperated integers in the form red,green,blue
  - ♦ a String that maps to an SWT color constant (ie: COLOR_RED).

  Only one of `value` or `colorFactory` may be defined.
- **colorFactory –** a class that implements `org.eclipse.ui.themes.IColorFactory`. This may be used instead of `value` to specify the value of the color. Please be advised that this should be used with caution – usage of this attribute will cause plugin activation on workbench startup.

<!ELEMENT fontValue EMPTY>

<!ATTLIST fontValue

os CDATA #IMPLIED

ws CDATA #IMPLIED

value CDATA #REQUIRED>

This element allows specification of a font value on a per−platform basis.

- **os –** an optional os string to enable choosing of font based on current OS. eg: win32,linux
- **ws –** an optional os string to enable choosing of font based on current WS. eg: win32,gtk
- **value –**

  the font value. This is in the form: `fontname-style-height` where `fontname` is the name of a font, `style` is a font style (one of `"regular"`, `"bold"`, `"italic"`, or `"bold italic"`) and `height` is an integer representing the font height.

  Example: `Times New Roman-bold-36`.

<!ELEMENT categoryPresentationBinding EMPTY>

<!ATTLIST categoryPresentationBinding

categoryId CDATA #REQUIRED

presentationId CDATA #REQUIRED>

This element allows a category to be bound to a specific presentation as described by the `org.eclipse.ui.presentationFactory` extension point. If a category has any presentation bindings then it (and it's children) is only configurable by the user if it is bound to the active presentation. This

is useful for removing unused items from user consideration.

- **categoryId** – the id of the category to bind
- **presentationId** – the id of the presentation to bind to

Examples:

The following is an example of several color and font definitions as well as a theme that overrides them.

```
<extension point=
"org.eclipse.ui.themes"
>
<themeElementCategory id=
"com.xyz.ThemeCategory"
class=
"com.xyz.XYZPreview"
label=
"XYZ Elements"
/>
<colorDefinition id=
"com.xyz.Forground"
categoryId=
"com.xyz.ThemeCategory"
label=
"XYZ Foreground Color"
value=
"COLOR_BLACK"
```

Since:

>

<!−− white should be used on GTK −−>

<colorValue value=

"COLOR_WHITE"

os=

"linux"

ws=

"gtk"

/>

<description>

This color is used for the forground color of the XYZ plugin editor.

</description>

</colorDefinition>

<colorDefinition id=

"com.xyz.Background"

categoryId=

"com.xyz.ThemeCategory"

label=

"XYZ Background Color"

>

<colorFactory class=

"org.eclipse.ui.themes.RGBBlendColorFactory"

plugin=

"org.eclipse.ui"

>

<parameter name=

"color1"

value=

"COLOR_WHITE"

/>

<parameter name=

"color2"

value=

"COLOR_BLUE"

/>

</colorFactory>

<!−− black should be used on GTK −−>

<colorValue value=

"COLOR_BLACK"

os=

"linux"

ws=

"gtk"

/>

<description>

This color is used for the background color of the XYZ plugin editor.

</description>

</colorDefinition>

<fontDefinition id=

"com.xyz.TextFont"

categoryId=

"com.xyz.ThemeCategory"

label=

"XYZ Editor Font"

defaultsTo=

"org.eclipse.jface.textfont"

>

<description>

This font is used by the XYY plugin editor.

</description>

</fontDefinition>

<data name=

"com.xyz.EditorMarginWidth"

value=

"5"

/>

<theme id=

"com.xyz.HarshTheme"

label=

"Harsh Colors for XYZ"

>

<colorOverride id=

"com.xyz.Forground"

value=

"COLOR_CYAN"

/>

<colorOverride id=

"com.xyz.Background"

value=

"COLOR_MAGENTA"

/>

<data name=

"com.xyz.EditorMarginWidth"

value=

"1"

/>

</theme>

</extension>

API Information:

The `org.eclipse.ui.IWorkbench.getThemeManager()` provides an instance of `org.eclipse.ui.themes.IThemeManager` that may be used to obtain a named theme (by id, including the default theme which has an id of `IThemeManager.DEFAULT_THEME`) or the current theme. From an `org.eclipse.ui.themes.ITheme` you may obtain a `org.eclipse.jface.resources.ColorRegistry`, an `org.eclipse.jface.resources.FontRegistry` and the arbitrary data associated with a theme.

Supplied Implementation:

The workbench provides the font definitions for the text, dialog, banner, header and part title fonts. it also supplies color definitions for the hyperlink, active hyperlink, error, active part (background gradient parts and forground) and the inactive part (background gradient parts and forground). The workbench also provides data constants for the title gradient percentages (active and inactive) and the gradient directions (active and inactive). The workbench does not provide any named themes.

View Menus,Toolbars and Actions

Identifier:

org.eclipse.ui.viewActions

Description:

This extension point is used to add actions to the pulldown menu and toolbar for views registered by other plug−ins. Each view has a local pulldown menu normally activated by clicking on the top right triangle button. Other plug−ins can contribute submenus and actions to this menu. Plug−ins may also contribute actions to a view toolbar. View owners are first given a chance to populate these areas. Optional additions by other plug−ins are appended.

An action's enablement and/or visibility can be defined using the elements `enablement` and `visibility` respectively. These two elements contain a boolean expression that is evaluated to determine the enablement and/or visibility.

The syntax is the same for the `enablement` and `visibility` elements. Both contain only one boolean expression sub−element. In the simplest case, this will be an `objectClass`, `objectState`, `pluginState`, or `systemProperty` element. In the more complex case, the `and`, `or`, and `not` elements can be combined to form a boolean expression. Both the `and`, and `or` elements must contain 2 sub−elements. The `not` element must contain only 1 sub−element.

Configuration Markup:

<!ELEMENT extension (viewContribution+)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point −** a fully qualified identifier of the target extension point
- **id −** an optional identifier of the extension instance
- **name −** an optional name of the extension instance

<!ELEMENT viewContribution (menu* , action*)>

<!ATTLIST viewContribution

id CDATA #REQUIRED

targetID CDATA #REQUIRED>

This element is used to define a group of view actions and/or menus.

- **id –** a unique identifier used to reference this contribution.
- **targetID –** a unique identifier of a registered view that is the target of this contribution.

<!ELEMENT action (selection* | enablement?)>

<!ATTLIST action

id CDATA #REQUIRED

label CDATA #REQUIRED

menubarPath CDATA #IMPLIED

toolbarPath CDATA #IMPLIED

icon CDATA #IMPLIED

disabledIcon CDATA #IMPLIED

hoverIcon CDATA #IMPLIED

tooltip CDATA #IMPLIED

helpContextId CDATA #IMPLIED

style (push|radioltoggle) "push"

state (true | false)

class CDATA #REQUIRED

enablesFor CDATA #IMPLIED>

This element defines an action that the user can invoke in the UI.

- **id –** a unique identifier used as a reference for this action.
- **label –** a translatable name used either as the menu item text or toolbar button label. The name can include mnenomic information.
- **menubarPath –** a slash–delimited path ('/') used to specify the location of this action in the pulldown menu. Each token in the path, except the last one, must represent a valid identifier of an existing menu in the hierarchy. The last token represents the named group into which this action will be added. If the path is omitted, this action will not appear in the pulldown menu.
- **toolbarPath –** a named group within the local toolbar of the target view. If the group does not exist, it will be created. If omitted, the action will not appear in the local toolbar.

Description:

- **icon –** a relative path of an icon used to visually represent the action in its context. If omitted and the action appears in the toolbar, the Workbench will use a placeholder icon. The path is relative to the location of the plugin.xml file of the contributing plug−in. The icon will appear in the toolbar but not in the pulldown menu.
- **disabledIcon –** a relative path of an icon used to visually represent the action in its context when the action is disabled. If omitted, the normal icon will simply appear greyed out. The path is relative to the location of the plugin.xml file of the contributing plug−in. The disabled icon will appear in the toolbar but not in the pulldown menu.
- **hoverIcon –** a relative path of an icon used to visually represent the action in its context when the mouse pointer is over the action. If omitted, the normal icon will be used. The path is relative to the location of the plugin.xml file of the contributing plug−in.
- **tooltip –** a translatable text representing the action's tool tip. Only used if the action appears in the toolbar.
- **helpContextId –** a unique identifier indicating the help context for this action. On some platforms, if the action appears as a menu item, then pressing the appropriate help key while the menu item is highlighted will display help. Not all platforms support this behaviour.
- **style –** an optional attribute to define the user interface style type for the action. If defined, the attribute value will be one of the following:

  push – as a regular menu item or tool item.

  radio – as a radio style menu item or tool item. Actions with the radio style within the same menu or toolbar group behave as a radio set. The initial value is specified by the `state` attribute.

  toggle – as a checked style menu item or as a toggle tool item. The initial value is specified by the `state` attribute.
- **state –** an optional attribute indicating the initial state (either `true` or `false`), used when the `style` attribute has the value `radio` or `toggle`.
- **class –** name of the fully qualified class that implements `org.eclipse.ui.IViewActionDelegate`.
- **enablesFor –** a value indicating the selection count which must be met to enable the action. If this attribute is specified and the condition is met, the action is enabled. If the condition is not met, the action is disabled. If no attribute is specified, the action is enabled for any number of items selected. The following attribute formats are supported:

  ! – 0 items selected

  ? – 0 or 1 items selected

  + – 1 or more items selected

  multiple, 2+ – 2 or more items selected

  n – a precise number of items selected.a precise number of items selected. For example: enablesFor=" 4" enables the action only when 4 items are selected

  ***** – any number of items selected

<!ELEMENT menu (separator+ , groupMarker*)>

<!ATTLIST menu

id CDATA #REQUIRED

Description:

label CDATA #REQUIRED

path CDATA #IMPLIED>

This element is used to defined a new menu.

- **id –** a unique identifier that can be used to reference this menu.
- **label –** a translatable name used by the Workbench for this new menu. The name should include mnemonic information.
- **path –** the location of the new menu starting from the root of the menu. Each token in the path must refer to an existing menu, except the last token which should represent a named group in the last menu in the path. If omitted, the new menu will be added to the `additions` named group of the menu.

<!ELEMENT separator EMPTY>

<!ATTLIST separator

name CDATA #REQUIRED>

This element is used to create a menu separator in the new menu.

- **name –** the name of the menu separator. This name can later be referenced as the last token in a menu path. Therefore, a separator also serves as named group into which actions and menus can be added.

<!ELEMENT groupMarker EMPTY>

<!ATTLIST groupMarker

name CDATA #REQUIRED>

This element is used to create a named group in the new menu. It has no visual representation in the new menu, unlike the `separator` element.

- **name –** the name of the group marker. This name can later be referenced as the last token in the menu path. It serves as named group into which actions and menus can be added.

<!ELEMENT selection EMPTY>

<!ATTLIST selection

class CDATA #REQUIRED

name CDATA #IMPLIED>

This element is used to help determine the action enablement based on the current selection. Ignored if the `enablement` element is specified.

- **class –** a fully qualified name of the class or interface that each object in the selection must implement in order to enable the action.
- **name –** an optional wild card filter for the name that can be applied to all objects in the selection. If specified and the match fails, the action will be disabled.

<!ELEMENT enablement (and | or | not | objectClass | objectState | pluginState | systemProperty)>

This element is used to define the enablement for the extension.

<!ELEMENT visibility (and | or | not | objectClass | objectState | pluginState | systemProperty)>

This element is used to define the visibility for the extension.

<!ELEMENT and (and | or | not | objectClass | objectState | pluginState | systemProperty)>

This element represent a boolean AND operation on the result of evaluating its two sub−element expressions.

<!ELEMENT or (and | or | not | objectClass | objectState | pluginState | systemProperty)>

This element represent a boolean OR operation on the result of evaluating its two sub−element expressions.

<!ELEMENT not (and | or | not | objectClass | objectState | pluginState | systemProperty)>

This element represent a boolean NOT operation on the result of evaluating its sub−element expressions.

<!ELEMENT objectClass EMPTY>

<!ATTLIST objectClass

name CDATA #REQUIRED>

This element is used to evaluate the class or interface of each object in the current selection. If each object in the selection implements the specified class or interface, the expression is evaluated as true.

- **name** – a fully qualified name of a class or interface. The expression is evaluated as true only if all objects within the selection implement this class or interface.

<!ELEMENT objectState EMPTY>

<!ATTLIST objectState

name CDATA #REQUIRED

value CDATA #REQUIRED>

This element is used to evaluate the attribute state of each object in the current selection. If each object in the selection has the specified attribute state, the expression is evaluated as true. To evaluate this type of expression, each object in the selection must implement, or adapt to, `org.eclipse.ui.IActionFilter` interface.

- **name** – the name of an object's attribute. Acceptable names reflect the object type, and should be publicly declared by the plug−in where the object type is declared.
- **value** – the required value of the object's attribute. The acceptable values for the object's attribute should be publicly declared.

<!ELEMENT pluginState EMPTY>

<!ATTLIST pluginState

id   CDATA #REQUIRED

value (installed|activated) "installed">

This element is used to evaluate the state of a plug−in. The state of the plug−in may be one of the following: `installed` (equivalent to the OSGi concept of "resolved") or `activated` (equivalent to the OSGi

concept of "active").

- **id** – the identifier of a plug–in which may or may not exist in the plug–in registry.
- **value** – the required state of the plug–in. The state of the plug–in may be one of the following: `installed` (equivalent to the OSGi concept of "resolved") or `activated` (equivalent to the OSGi concept of "active").

<!ELEMENT systemProperty EMPTY>

<!ATTLIST systemProperty

name CDATA #REQUIRED

value CDATA #REQUIRED>

This element is used to evaluate the state of some system property. The property value is retrieved from the `java.lang.System`.

- **name** – the name of the system property.
- **value** – the required value of the system property.

Examples:

The following is an example of a view action extension:

<extension point=

"org.eclipse.ui.viewActions"

>

<viewContribution id=

"com.xyz.xyzViewC1"

targetID=

"org.eclipse.ui.views.navigator.ResourceNavigator"

>

```
<menu id=
"com.xyz.xyzMenu"
label=
"XYZ Menu"
path=
"additions"
>
<separator name=
"group1"
/>
</menu>
<action id=
"com.xyz.runXYZ"
label=
"&amp;amp;Run XYZ Tool"
menubarPath=
"com.xyz.xyzMenu/group1"
toolbarPath=
"Normal/additions"
style=
"toggle"
state=
"true"
icon=
"icons/runXYZ.gif"
tooltip=
```

"Run XYZ Tool"

helpContextId=

"com.xyz.run_action_context"

class=

"com.xyz.actions.RunXYZ"

>

<selection class=

"org.eclipse.core.resources.IFile"

name=

"*.java"

/>

</action>

</viewContribution>

</extension>

In the example above, the specified action will only enable for a single selection (`enablesFor` attribute). In addition, the object in the selection must be a Java file resource.

The following is an other example of a view action extension:

<extension point=

"org.eclipse.ui.viewActions"

>

<viewContribution id=

"com.xyz.xyzViewC1"

targetID=

"org.eclipse.ui.views.navigator.ResourceNavigator"

>

<menu id=

"com.xyz.xyzMenu"

label=

"XYZ Menu"

path=

"additions"

>

<separator name=

"group1"

/>

</menu>

<action id=

"com.xyz.runXYZ2"

label=

"&Run XYZ2 Tool"

menubarPath=

"com.xyz.xyzMenu/group1"

style=

"push"

icon=

"icons/runXYZ2.gif"

tooltip=

"Run XYZ2 Tool"

helpContextId=

"com.xyz.run_action_context2"

class=

"com.xyz.actions.RunXYZ2"

>

<enablement>

<and>

<objectClass name=

"org.eclipse.core.resources.IFile"

/>

<not>

<objectState name=

"extension"

value=

"java"

/>

</not>

</and>

</enablement>

</action>

</viewContribution>

</extension>

In the example above, the specified action will appear as a menu item. The action is enabled if the selection contains no Java file resources.

API Information:

The value of the `class` attribute must be a fully qualified name of a Java class that implements `org.eclipse.ui.IViewActionDelegate`. This class is loaded as late as possible to avoid loading the entire plug−in before it is really needed.

The interface `org.eclipse.ui.IViewActionDelegate` extends `org.eclipse.ui.IActionDelegate` and adds an additional method that allows the delegate to initialize with the view instance it is contributing into.

This extension point can be used to contribute actions into menus previously created by the target view. Omitting the menu path attribute will result in adding the new menu or action at the end of the pulldown menu.

The enablement criteria for an action extension is initially defined by `enablesFor`, and also either `selection` or `enablement`. However, once the action delegate has been instantiated, it may control the action enable state directly within its `selectionChanged` method.

Action and menu labels may contain special characters that encode mnemonics using the following rules:

1. Mnemonics are specified using the ampersand ('&') character in front of a selected character in the translated text. Since ampersand is not allowed in XML strings, use `&` character entity.

If two or more actions are contributed to a menu or toolbar by a single extension the actions will appear in the reverse order of how they are listed in the plugin.xml file. This behavior is admittedly unintuitive. However, it was discovered after the Eclipse Platform API was frozen. Changing the behavior now would break every plug−in which relies upon the existing behavior.

The `selection` and `enablement` elements are mutually exclusive. The `enablement` element can replace the `selection` element using the sub−elements `objectClass` and `objectState`. For example, the following:

<selection class=

"org.eclipse.core.resources.IFile"

name=

"*.java"

>

</selection>

can be expressed using:

<enablement>

<and>

<objectClass name=

"org.eclipse.core.resources.IFile"

/>

<objectState name=

"extension"

value=

"java"

/>

</and>

</enablement>

Supplied Implementation:

Each view normally comes with a number of standard items on the pulldown menu and local toolbar. Additions from other plug−ins will be appended to the standard complement.

Working Sets

Identifier:

org.eclipse.ui.workingSets

Since:

Release 2.0

Description:

This extension point is used to define a working set wizard page. Working sets contain a number of elements of type IAdaptable and can be used to group elements for presentation to the user or for operations on a set of elements. A working set wizard page is used to create and edit working sets that contain elements of a specific type.

To select a working set the user is presented with a list of working sets that exist in the workbench. From that list a working set can be selected and edited using one of the wizard pages defined using this extension point. An existing working set is always edited with the wizard page that was used to create it or with the default resource based working set page if the original page is not available.

A new working set can be defined by the user from the same working set selection dialog. When a new working set is defined, the plugin provided wizard page is preceded by a page listing all available working set types. This list is made up of the name attribute values of each working set extension.

Views provide a user interface to open the working set selection dialog and must store the selected working set.

The resource navigator uses a working set to filter elements from the navigator view. Only parents and children of working set elements are shown in the view, in addition to the working set elements themselves.

Configuration Markup:

<!ELEMENT extension (workingSet*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT workingSet EMPTY>

<!ATTLIST workingSet

id CDATA #REQUIRED

name CDATA #REQUIRED

icon CDATA #IMPLIED

pageClass CDATA #IMPLIED

updaterClass CDATA #IMPLIED>

- **id** – a unique name that can be used to identify this working set dialog.
- **name** – the name of the element type that will be displayed and edited by the working set page. This should be a descriptive name like "Resource" or "Java Element".
- **icon** – the relative path of an image that will be displayed in the working set type list on the first page of the working set creation wizard as well as in the working set selection dialog.
- **pageClass** – the fully qualified name of a Java class implementing `org.eclipse.ui.dialogs.IWorkingSetPage`.
- **updaterClass** – the fully qualified name of a Java class implementing `org.eclipse.ui.IWorkingSetUpdater`.

Examples:

Following is an example of how the resource working set dialog extension is defined to display and edit generic IResource elements:

<extension point=

"org.eclipse.ui.workingSets"

>

<workingSet id=

"org.eclipse.ui.resourceWorkingSetPage"

name=

"Resource"

icon=

"icons/resworkset.gif"

pageClass=

"org.eclipse.ui.internal.dialogs.ResourceWorkingSetPage"

>

</workingSet>

</extension>

API Information:

The value of the pageClass attribute must represent a class that implements the `org.eclipse.ui.dialogs.IWorkingSetPage` interface.

Supplied Implementation:

The workbench provides a working set wizard page for creating and editing resource based working sets.

Synchronize Participants

Identifier:

org.eclipse.team.ui.synchronizeParticipants

Since:

3.0

Description:

This extension point is used to register a synchronize participant. A synchronize participant is a component that displays changes between resources and typically allows the user to manipulate the changes. For example, CVS defines a workspace synchronize participant that allows showing changes between workspace resources and the resources at a remote location that is used to share those resources. Synchronize participants are typically created via a synchronize participant wizard or they can be created via a plug−in action and then registered with the ISynchronizeManager. The Synchronize View displays synchronize participants.

A participant is a generic component that provides access to creating a page and is shown to the user and a configuration that defines common configuration parameters for the page. The Synchronize View doesn't enforce any restrictions on how changes are shown to the user, and instead only manages the participants.

Configuration Markup:

<!ELEMENT extension (participant)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point −** A fully qualified identifier of the target extension point
- **id −** An optional identifier of the extension instance.
- **name −** An optional name for this extension instance.

<!ELEMENT participant EMPTY>

<!ATTLIST participant

icon CDATA #IMPLIED

id CDATA #REQUIRED

class CDATA #REQUIRED

name CDATA #REQUIRED

persistent (true | false) "false">

- **icon** – An icon that will be used when showing this participant in lists and menus.
- **id** – A unique name that will be used to identify this type of participant.
- **class** – A fully qualified name of the class the extends `org.eclipse.team.ui.synchronize.AbstractSynchronizeParticipant`.
- **name** – The name of the participant. This will be shown in the UI.
- **persistent** – By default participants will be persisted between sessions. Set this attribute to false if this participant should not be persisted between sessions.

Examples:

<extension point=

"org.eclipse.team.ui.synchronizeParticipants"

>

<participant name=

"CVS Workspace"

icon=

"icons/full/cview16/server.gif"

class=

"org.eclipse.team.internal.ccvs.ui.subscriber.WorkspaceSynchronizeParticipant"

id=

"org.eclipse.team.cvs.ui.cvsworkspace−participant"

>

</participant>

<participant name=

"CVS Merge"

icon=

"icons/full/obj16/prjversions_rep.gif"

class=

"org.eclipse.team.internal.ccvs.ui.subscriber.MergeSynchronizeParticipant"

type=

"dynamic"

id=

"org.eclipse.team.cvs.ui.cvsmerge−participant"

>

</participant>

</extension>

API Information:

The value of the `class` attribute must represent a class that implements `org.eclipse.team.ui.synchronize.AbstractSynchronizeParticipant`.

Breakpoints

Identifier:

org.eclipse.debug.core.breakpoints

Description:

This extension point defines a mechanism for defining new types of breakpoints.

Configuration Markup:

<!ELEMENT extension (breakpoint*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT breakpoint EMPTY>

<!ATTLIST breakpoint

id CDATA #REQUIRED

markerType CDATA #REQUIRED

class CDATA #REQUIRED

name CDATA #IMPLIED>

- **id –** specifies a unique identifier for this breakpoint type.
- **markerType –** specifies the fully qualified identifier (id) of the corresponding marker definition for breakpoints of this type.
- **class –** specifies the fully qualified name of the Java class that implements `IBreakpoint`.
- **name –** specifies a user−presentable name for this breakpoint type. For example, "Java Line Breakpoint". This attribute was added in 3.1 to support automatic grouping of breakpoints by breakpoint type. When this attribute is unspecified, breakpoints of this type cannot be automatically grouped by type.

Examples:

The following is an example of a launch configuration type extension point:

```
<extension point=
"org.eclipse.debug.core.breakpoints"
>
<breakpoint id=
"com.example.ExampleBreakpoint"
markerType=
"com.example.ExampleBreakpointMarker"
class=
"com.example.ExampleBreakpointImpl"
>
</breakpoint>
</extension>
```

In the example above, the specified type of breakpoint is implemented by the class "com.example.BreakpointImpl". There is an associated marker definition for "com.example.ExampleBreakpointMarker", defining the attributes of this breakpoint.

API Information:

Value of the attribute **class** must be a fully qualified name of a Java class that implements the interface **org.eclipse.debug.core.model.IBreakpoint**.

Launch Configuration Comparators

Identifier:

org.eclipse.debug.core.launchConfigurationComparators

Description:

This extension point provides a configurable mechanism for comparing specific attributes of a launch configuration. In general, launch configuration attributes can be compared for equality via the default implementation of `java.lang.Object.equals(Object)`. However, attributes that require special handling should implement this extension point. For example, when an attribute is stored as XML, it is possible that two strings representing an equivalent attribute have different whitespace formatting.

Configuration Markup:

<!ELEMENT extension (launchConfigurationComparator*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT launchConfigurationComparator EMPTY>

<!ATTLIST launchConfigurationComparator

id CDATA #REQUIRED

attribute CDATA #REQUIRED

class CDATA #REQUIRED>

- **id –** specifies a unique identifier for this extension.
- **attribute –** specifies the launch configuration attribute name that this comparator compares.
- **class –** specifies a fully–qualified name of a class that implements `java.util.Comparator`.

Examples:

The following is an example of a launch configuration comparator extension point:

```
<extension point=
"org.eclipse.debug.core.launchConfigurationComparators"
>
<launchConfigurationComparator id=
"com.example.ExampleIdentifier"
attribute=
"com.example.ExampleAttributeName"
class=
"com.example.ComparatorImplementation"
>
</launchConfigurationComparator>
</extension>
```

In the example above, the specified type of launch configuration comparator will be consulted when comparing the equality of attributes keyed with name `com.example.ExampleAttributeName`.

API Information:

Value of the attribute **class** must be a fully−qualified name of a Java class that implements the interface **java.util.Comparator**.

Launch Configuration Types

Identifier:

org.eclipse.debug.core.launchConfigurationTypes

Description:

This extension point provides a configurable mechanism for launching applications. Each launch configuration type has a name, supports one or more modes (run and/or debug), and specifies a delegate responsible for the implementation of launching an application.

Configuration Markup:

<!ELEMENT extension (launchConfigurationType*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT launchConfigurationType (fileExtension+)>

<!ATTLIST launchConfigurationType

id CDATA #REQUIRED

delegate CDATA #REQUIRED

modes CDATA #REQUIRED

name CDATA #REQUIRED

public (true | false)

category CDATA #IMPLIED

sourcePathComputerId CDATA #IMPLIED

sourceLocatorId CDATA #IMPLIED>

- **id –** specifies a unique identifier for this launch configuration type.

- **delegate** – specifies the fully qualified name of the Java class that implements `ILaunchConfigurationDelegate`. Launch configuration instances of this type will delegate to instances of this class to perform launching.
- **modes** – specifies a comma–separated list of the modes this type of lauch configuration supports – "run" and/or "debug".
- **name** – specifies a human–readable name for this type of launch configuration.
- **public** – specifies whether this launch configuration type is accessible by users. Defaults to `true` if not specified.
- **category** – an optional attribute that specifies this launch configuration type's category. The default value is unspecified (`null`). Categories are client defined. This attribute was added in the 2.1 release.
- **sourcePathComputerId** – The unique identifier of a sourcePathComputer extension that is used to compute a default source lookup path for launch configurations of this type. This atttribute was added in the 3.0 release.
- **sourceLocatorId** – The unique identifier of a sourceLocator extension that is used to create the source locator for sessions launched using launch configurations of this type. This atttribute was added in the 3.0 release.

<!ELEMENT fileExtension EMPTY>

<!ATTLIST fileExtension

extension CDATA #REQUIRED

default (true | false) >

- **extension** – specifies a file extension that this launch configuration type can be used for.
- **default** – specifies whether this launch configuration type should be the default launch configuration type for the specified file extension. Defaults to `false` if not specified.

Examples:

The following is an example of a launch configuration type extension point:

<extension point=

"org.eclipse.debug.core.launchConfigurationTypes"

>

<launchConfigurationType id=

"com.example.ExampleIdentifier"

delegate=

"com.example.ExampleLaunchConfigurationDelegate"

modes=

"run,debug"

name=

"Example Application"

>

<fileExtension extension=

"txt"

default=

"true"

/>

<fileExtension extension=

"gif"

default=

"false"

/>

</launchConfigurationType>

</extension>

In the example above, the specified type of launch configuration supports both run and debug modes. The launch configuration is applicable to .txt and .gif files, and is the default launch configuration for .txt files.

API Information:

Value of the attribute **delegate** must be a fully qualified name of a Java class that implements the interface **org.eclipse.debug.core.model.ILaunchConfigurationDelegate**.

Launch Delegates

Identifier:

org.eclipse.debug.core.launchDelegates

Since:

3.0

Description:

This extension point provides a mechanism for contributing a launch delegate to an existing launch configuration type for one or more launch modes. Since launch modes are extensible, it may be neccessary to contribute additional launch delegates to an existing launch configuration type. Each launch delegate is contributed for a specific launch configuration type. A launch delegate supports one or more launch modes, and specifies a delegate responsible for the implementation of launching.

Configuration Markup:

<!ELEMENT extension (launchDelegate*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT launchDelegate EMPTY>

<!ATTLIST launchDelegate

id CDATA #REQUIRED

delegate CDATA #REQUIRED

modes CDATA #REQUIRED

type CDATA #REQUIRED

sourcePathComputerId CDATA #IMPLIED

sourceLocatorId CDATA #IMPLIED>

- **id** – specifies a unique identifier for this launch delegate.
- **delegate** – specifies the fully qualified name of the Java class that implements `ILaunchConfigurationDelegate`. Launch configuration instances of this delegate's type will delegate to instances of this class to perform launching.
- **modes** – specifies a comma–separated list of the modes this lauch delegate supports.
- **type** – identifier of an existing launch configuration type that this launch delegate is capable of launching.
- **sourcePathComputerId** – The unique identifier of a sourcePathComputer extension that is used to compute a default source lookup path for launch configurations of this type. Since 3.1, this attribute cab be specified in a launchDelegate extension when unspecified in the assocaited launchConfigurationType extension. Only one source path computer can be specified per launch configuration type.
- **sourceLocatorId** – The unique identifier of a sourceLocator extension that is used to create the source locator for sessions launched using launch configurations of this type. Since 3.1, this attribute can be specified in a launchDelegate extension when unspecified in the assocaited launchConfigurationType extension. Only one source locater can be specified per launch configuration type.

Examples:

The following is an example of a launch delegate extension point:

<extension point=

"org.eclipse.debug.core.launchDelegates"

>

<launchDelegate id=

"com.example.ExampleProfileDelegate"

delegate=

"com.example.ExampleProfileDelegate"

type=

"org.eclipse.jdt.launching.localJavaApplication"

modes=

"profile"

>

</launchDelegate>

Since:

</extension>

In the example above, the specified launch delegate is contributed to launch Java applications in profile mode.

API Information:

Value of the attribute **delegate** must be a fully qualified name of a Java class that implements the interface **org.eclipse.debug.core.model.ILaunchConfigurationDelegate**.

Launcher (Obsolete)

Identifier:

org.eclipse.debug.core.launchers

Description:

This extension point has been replaced by the launchConfigurationTypes extension point. Extensions of this type are obsolete as of release 2.0 and are ignored. This extension point was used to contribute launchers. A launcher was responsible for initiating a debug session or running a program and registering the result with the launch manager.

Configuration Markup:

<!ELEMENT extension (launcher*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT launcher EMPTY>

<!ATTLIST launcher

id CDATA #REQUIRED

class CDATA #REQUIRED

modes CDATA #REQUIRED

label CDATA #REQUIRED

wizard CDATA #IMPLIED

public (true | false)

description CDATA #IMPLIED

perspective CDATA #IMPLIED

icon CDATA #IMPLIED>

- **id –** a unique identifier that can be used to reference this launcher.
- **class –** fully qualified name of the Java class that implements `org.eclipse.debug.core.model.ILauncherDelegate`.
- **modes –** A comma separated list of modes this launcher supports. The two supported modes are "run" and "debug" – as defined in `org.eclipse.debug.core.ILaunchManager`. A launcher may be capable of launching in one or both modes.
- **label –** a label to use for the launcher. This attribute is used by the debug UI.
- **wizard –** fully qualified name of the class that implements `org.eclipse.debug.ui.ILaunchWizard`. This attribute is used by the debug UI. A launcher may contribute a wizard that allows users to configure and launch specific attributes.
- **public –** whether a launcher is publically visible in the debug UI. If `true`, the launcher will be available from the debug UI – the launcher will appear as a choice for a default launcher, launches created by this launcher will appear in the launch history, and the launcher will be available from the drop–down run/debug toolbar actions.
- **description –** a description of the launcher. Currently only used if the wizard attribute is specified.
- **perspective –** the identifier of the perspective that will be switched to on a successful launch. Default value is the identifier for the debug perspective. This attribute is used by the debug UI.
- **icon –** a relative path of an icon that will represent the launcher in the UI if specified.

Examples:

The following is an example of a launcher extension point:

<extension point =

"org.eclipse.debug.core.launchers"

>

<launcher id =

"com.example.ExampleLauncher"

class =

"com.example.launchers.ExampleLauncher"

modes =

"run, debug"

label =

"Example Launcher"

wizard =

"com.example.launchers.ui.ExampleLaunchWizard"

public =

"true"

description =

"Launches example programs"

perspective=

"com.example.JavaPerspective"

>

</launcher>

</extension>

In the example above, the specified launcher supports both run and debug modes. Following a successful launch, the debug UI will change to the Java perspective. When the debug UI presents the user with a list of launchers to choose from, "Example Launcher" will appear as one of the choices with the "Launches example programs" as the description, and the wizard specified by `com.example.launchers.ui.ExampleLaunchWizard` will be used to configure any launch specific details.

API Information:

Value of the attribute `class` must be a fully qualified class name of a Java class that implements the interface `org.eclipse.debug.core.ILauncherDelegate`. Value of the attribute `wizard` must be a fully qualified class name of a Java class that implements `org.eclipse.debug.ui.ILaunchWizard`.

Launch Modes

Identifier:

org.eclipse.debug.core.launchModes

Since:

3.0

Description:

This extension point provides a mechanism for contributing launch modes to the debug platform. The debug platform defines modes for "run", "debug", and "profile".

Configuration Markup:

<!ELEMENT extension (launchMode*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT launchMode EMPTY>

<!ATTLIST launchMode

mode CDATA #REQUIRED

label CDATA #REQUIRED>

- **mode –** specifies a unique identifier for this launch mode. The launch modes contributed by the debug plaform are "run", "debug", and "profile".
- **label –** A human−readable label that describes the launch mode

Examples:

The following is an example of a launch delegate extension point:

<extension point=

"org.eclipse.debug.core.launchModes"

>

<launchMode mode=

"profile"

label=

"Profile"

>

</launchMode>

</extension>

In the example above, the profile launch mode is contributed.

Logical Structure Types

Identifier:

org.eclipse.debug.core.logicalStructureTypes

Since:

3.0

Description:

This extension point allows debuggers to present alternative logical structures of values. Often, complex data structures are more convenient to navigate in terms of their logical structure, rather than in terms of their implementation. For example, no matter how a list is implemented (linked, array, etc.), a user may simply want to see the elements in the list in terms of an ordered collection. This extension point allows the contribution of logical structure types, to provide translations from a raw implementation value to a logical value.

Configuration Markup:

<!ELEMENT extension (logicalStructureType*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT logicalStructureType EMPTY>

<!ATTLIST logicalStructureType

id CDATA #REQUIRED

description CDATA #IMPLIED

class CDATA #REQUIRED

modelIdentifier CDATA #REQUIRED>

- **id –** a unique identifier for this logical structure type

- **description** – a description of this logical structure. Since 3.1, this attribute is optional. When unspecified, a logical structure type delegate must also implement `ILogicalStructureTypeDelegate2` to provide descriptions for values that logical structures are provided for.
- **class** – fully qualified name of a Java class that implements `ILogicalStructureTypeDelegate`. The class may optionally implement `ILogicalStructureTypeDelegate2`.
- **modelIdentifier** – identifier of the debug model this logical structure type is associated with

Examples:

The following is an example of a logical structure type extension point:

<extension point=

"org.eclipse.debug.core.logicalStructureTypes"

>

<logicalStructureType id=

"com.example.ExampleLogicalStructure"

class=

"com.example.ExampleLogicalStructureDelegate"

modelIdentifier=

"com.example.debug.model"

description=

"Ordered Collection"

>

</logicalStructureType>

</extension>

In the example above, the specified logical structure type will be consulted for alternative logical values for values from the `com.example.debug.model` debug model as they are displayed in the variables view.

API Information:

Value of the attribute **class** must be a fully qualified name of a Java class that implements the interface **org.eclipse.debug.core.model.ILogicalStructureTypeDelegate**.

Process Factories

Identifier:

org.eclipse.debug.core.processFactories

Since:

3.0

Description:

This extension point provides a mechanism for specifying a process factory to be used with a launch configuration to create the appropriate instance of **IProcess**. The launch configuration will require the **DebugPlugin.ATTR_PROCESS_FACTORY_ID** attribute set to the appropriate process factory ID that will be used to create the **IProcess**

Configuration Markup:

<!ELEMENT extension (processFactory*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT processFactory EMPTY>

<!ATTLIST processFactory

id CDATA #REQUIRED

class CDATA #REQUIRED>

- **id –** specifies a unique identifier for this process factory.
- **class –** specifies the fully qualified name of the Java class that implements `IProcessFactory`.

Examples:

The following is an example of a process factory extension point:

<extension point=

"org.eclipse.debug.core.processFactories"

>

<processFactory id=

"com.example.ExampleIdentifier"

class=

"com.example.ExampleProcessFactory"

>

</processFactory>

</extension>

API Information:

Value of the attribute **class** must be a fully qualified name of a Java class that implements the interface **org.eclipse.debug.core.IProcessFactory**.

Source Container Types

Identifier:

org.eclipse.debug.core.sourceContainerTypes

Since:

3.0

Description:

This extension point allows for an extensible set of source container types to be contributed by the debug platform source lookup facilities.

Configuration Markup:

<!ELEMENT extension (sourceContainerType*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT sourceContainerType EMPTY>

<!ATTLIST sourceContainerType

id CDATA #REQUIRED

name CDATA #REQUIRED

class CDATA #REQUIRED

description CDATA #IMPLIED>

- **id –** The unique id used to refer to this type
- **name –** The name of this source container type used for presentation purposes.
- **class –** A class that implements `ISourceContainerTypeDelegate`
- **description –** A short description of this source container for presentation purposes.

Examples:

The following is an example of a source container type definition:

<extension point=

"org.eclipse.debug.core.sourceContainerTypes"

>

<sourceContainerType name=

"Project"

class=

"org.eclipse.debug.internal.core.sourcelookup.containers.ProjectSourceContainerType"

id=

"org.eclipse.debug.core.containerType.project"

description=

"A project in the workspace"

>

</sourceContainerType>

</extension>

API Information:

Value of the attribute **class** must be a fully qualified name of a Java class that implements the interface **ISourceContainerType**.

Source Locators

Identifier:

org.eclipse.debug.core.sourceLocators

Description:

This extension point provides a mechanism specifying a source locator to be used with a launch configuration.

Configuration Markup:

<!ELEMENT extension (sourceLocator*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT sourceLocator EMPTY>

<!ATTLIST sourceLocator

id CDATA #REQUIRED

class CDATA #REQUIRED

name CDATA #REQUIRED>

- **id –** specifies a unique identifier for this source locator.
- **class –** specifies the fully qualified name of the Java class that implements `IPersistableSourceLocator`.
- **name –** a human–readable name, describing the type of this source locator.

Examples:

The following is an example of a source locator extension point:

<extension point=

"org.eclipse.debug.core.sourceLocators"

>

<sourceLocator id=

"com.example.ExampleIdentifier"

class=

"com.example.ExampleSourceLocator"

name=

"Example Source Locator"

>

</sourceLocator>

</extension>

API Information:

Value of the attribute **class** must be a fully qualified name of a Java class that implements the interface **org.eclipse.debug.core.model.IPersistableSourceLocator**.

Source Path Computers

Identifier:

org.eclipse.debug.core.sourcePathComputers

Since:

3.0

Description:

Defines an extension point to register a computer that can describe a default source lookup path for a launch configuration. Source path computers are associated with launch configuration types via the launchConfigurationTypes extension point. As well, a source path computer can be associated with a specific launch configuration via the launch configuration attribute ATTR_SOURCE_PATH_COMPUTER_ID.

Configuration Markup:

<!ELEMENT extension (sourcePathComputer*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT sourcePathComputer EMPTY>

<!ATTLIST sourcePathComputer

id CDATA #REQUIRED

class CDATA #REQUIRED>

Defines an extension point to register a computer that can describe a default source lookup path for a launch configuration.

- **id –** The unique id used to refer to this computer.
- **class –** A class that implements `ISourcePathComputerDelegate`.

Examples:

Following is an example source path computer definition:

<extension point=

"org.eclipse.debug.core.sourcePathComputers"

>

<sourcePathComputer id=

"org.eclipse.example.exampleSourcePathComputer"

class=

"org.eclipse.example.SourcePathComputer"

>

</sourcePathComputer>

</extension>

API Information:

Value of the attribute **class** must be a fully qualified name of a Java class that implements the interface **ISourcePathComputer**.

Status Handlers

Identifier:

org.eclipse.debug.core.statusHandlers

Description:

This extension point provides a mechanism for separating the generation and resolution of an error. The interaction between the source of the error and the resolution is client−defined. It is a client responsibility to look up and delegate to status handlers when an error condition occurs.

Configuration Markup:

<!ELEMENT extension (statusHandler*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point −** a fully qualified identifier of the target extension point
- **id −** an optional identifier of the extension instance
- **name −** an optional name of the extension instance

<!ELEMENT statusHandler EMPTY>

<!ATTLIST statusHandler

id CDATA #REQUIRED

class CDATA #REQUIRED

plugin CDATA #REQUIRED

code CDATA #REQUIRED>

- **id −** specifies a unique identifier for this status handler.
- **class −** specifies the fully qualified name of the Java class that implements `IStatusHandler`.
- **plugin −** Plug−in identifier that corresponds to the plug−in of the status this handler is registered for. (i.e. `IStatus.getPlugin()`).
- **code −** specifies the status code this handler is registered for.

Examples:

The following is an example of a status handler extension point:

```
<extension point=
"org.eclipse.debug.core.statusHandlers"
>
<statusHandler id=
"com.example.ExampleIdentifier"
class=
"com.example.ExampleStatusHandler"
plugin=
"com.example.ExamplePluginId"
code=
"123"
>
</statusHandler>
</extension>
```

In the example above, the specified status handler will be registered for to handle status objects with a plug−in identifier of `com.example.ExamplePluginId` and a status code of `123`.

API Information:

Value of the attribute **class** must be a fully qualified name of a Java class that implements the interface **org.eclipse.debug.core.IStatusHandler**.

watchExpressionDelegates

Identifier:

org.eclipse.debug.core.watchExpressionDelegates

Since:

3.0

Description:

This extension provides a mechanism for providing delegates to evaluate watch expressions on a per debug model basis. Watch expression delegates perform evaluations for watch expressions and report the results asynchronously.

Configuration Markup:

<!ELEMENT extension (watchExpressionDelegate*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT watchExpressionDelegate EMPTY>

<!ATTLIST watchExpressionDelegate

debugModel CDATA #REQUIRED

delegateClass CDATA #REQUIRED>

- **debugModel –** specifies the id of the debug model that this delegate provides evaluations for
- **delegateClass –** specifies a Java class which implements `org.eclipse.debug.core.model.IWatchExpressionDelegate`, which is used to evaluate the value of an expression.

Examples:

The following is the definition of a watch expression delegate for the com.example.foo plug-in:

<extension point=

"org.eclipse.debug.core.watchExpressionDelegates"

>

<watchExpressionDelegate debugModel=

"org.eclipse.jdt.debug"

delegateClass=

"org.eclipse.jdt.internal.debug.ui.JavaWatchExpressionDelegate"

/>

</extension>

API Information:

Value of the attribute **delegateClass** must be a fully qualified name of a Java class that implements the interface **org.eclipse.debug.core.model.IWatchExpressionDelegate**.

Breakpoint Organizers

Identifier:

org.eclipse.debug.ui.breakpointOrganizers

Since:

3.1

Description:

Breakpoint organizers categorize breakpoints based on some specific criteria. For example, a breakpoint organizer is provided to categorize breakpoints by project. Organizers with the specified name will be automatically created by the Debug Platform and presented to the user as options for grouping breakpoints. The supplied class, which must implement org.eclipse.debug.ui.IBreakpointOrganizerDelegate, will be loaded only as necessary, to avoid early plugin activation.

Configuration Markup:

<!ELEMENT extension (breakpointOrganizer+)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT breakpointOrganizer EMPTY>

<!ATTLIST breakpointOrganizer

id CDATA #REQUIRED

class CDATA #REQUIRED

label CDATA #REQUIRED

icon CDATA #IMPLIED

othersLabel CDATA #IMPLIED>

- **id –** Unique identifier for this breakpoint organizer.

- **class** – Implementation of `org.eclipse.debug.ui.IBreakpointOrganizerDelegate` that performs categorization
- **label** – Label for this organizer which is suitable for presentation to the user.
- **icon** – Optional path to an icon which can be shown for this organizer.
- **othersLabel** – Optional label for this organizer which is suitable for presentation to the user to describe breakpoints that do not fall into a category supplied by this organizer. For example, if an organizer categorizes breakpoints by working sets, but a breakpoint does not belong to a working set, this label will be used. When unspecified, "Others" is used.

Examples:

Following is an example of a breakpoint organizer extension.

<extension point=

"org.eclipse.debug.ui.breakpointOrganizers"

>

<breakpointOrganizer class=

"com.example.BreakpointOrganizer"

id=

"com.example.BreakpointOrganizer"

label=

"Example Organizer"

icon=

"icons/full/obj16/example_org.gif"

/>

</extension>

In the above example, the supplied factory will be included in the list of options for grouping breakpoints ("Group By > Example Organizer"). When selected, the associated organizer will be used to categorize breakpoints.

API Information:

Value of the attribute **class** must be a fully qualified name of a Java class that implements the interface **org.eclipse.debug.ui.IBreakpointOrganizerDelegate**.

Since:

Supplied Implementation:

The Debug Platform provides breakpoint organizers for projects, files, breakpoint types, and working sets.

Console Color Providers

Identifier:

org.eclipse.debug.ui.consoleColorProviders

Since:

2.1

Description:

This extension point provides a mechanism for contributing a console document coloring scheme for a process. The color provider will be used to color output in the console.

Configuration Markup:

<!ELEMENT extension (consoleColorProvider*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point** – a fully qualified identifier of the target extension point
- **id** – an optional identifier of the extension instance
- **name** – an optional name of the extension instance

<!ELEMENT consoleColorProvider EMPTY>

<!ATTLIST consoleColorProvider

id CDATA #REQUIRED

class CDATA #REQUIRED

processType CDATA #REQUIRED>

- **id** – specifies a unique identifier for this console color provider.
- **class** – specifies a fully qualified name of a Java class that implements `IConsoleColorProvider`
- **processType** – specifies the type of process this color provider is for. This attribute corresponds to the process attribute `IProcess.ATTR_PROCESS_TYPE`.

Examples:

The following is an example of a console color provider extension point:

<extension point=

"org.eclipse.debug.ui.consoleColorProviders"

>

<consoleColorProvider id=

"com.example.ExampleConsoleColorProvider"

class=

"com.example.ExampleConsoleColorProvider"

processType=

"ExampleProcessType"

>

</consoleColorProvider>

</extension>

In the above example, the contributed color provider will be used for processes of type "ExampleProcessType", which corresponds to the process attribute IProcess.ATTR_PROCESS_TYPE. Process types are client defined, and are set by clients that create processes.

API Information:

Value of the attribute **class** must be a fully qualified name of a Java class that implements the interface **org.eclipse.debug.ui.console.IConsoleColorProvider**.

Console Line Trackers

Identifier:

org.eclipse.debug.ui.consoleLineTrackers

Since:

2.1

Description:

This extension point provides a mechanism to listen to console output for a type of process.

Configuration Markup:

<!ELEMENT extension (consoleLineTracker*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT consoleLineTracker EMPTY>

<!ATTLIST consoleLineTracker

id CDATA #REQUIRED

class CDATA #REQUIRED

processType CDATA #REQUIRED>

- **id –** specifies a unique identifier for this console line tracker.
- **class –** specifies a fully qualified name of a Java class that implements `IConsoleLineTracker`
- **processType –** specifies the type of process this line tracker is for. This attribute corresponds to the process attribute `IProcess.ATTR_PROCESS_TYPE`.

Examples:

The following is an example of a console line tracker extension point:

<extension point=

"org.eclipse.debug.ui.consoleLineTrackers"

>

<consoleLineTracker id=

"com.example.ExampleConsoleLineTracker"

class=

"com.example.ExampleConsoleLineTracker"

processType=

"ExampleProcessType"

>

</consoleLineTracker>

</extension>

In the above example, the contributed line tracker will be notified as lines are appended to the console for processes of type "ExampleProcessType", which corresponds to the process attribute IProcess.ATTR_PROCESS_TYPE. Process types are client defined, and are set by clients that create processes.

API Information:

Value of the attribute **class** must be a fully qualified name of a Java class that implements the interface **org.eclipse.debug.ui.console.IConsoleLineTracker**.

Context View Bindings

Identifier:

org.eclipse.debug.ui.contextViewBindings

Since:

3.0

Description:

This extension point provides a mechanism for associating a view with a context identifier. When a context is activated by the Debug view, views associated with it (and also views associated with any parent contexts) are opened, closed, or activated. Contributors have the option to override the automatic open and close behavior.

Configuration Markup:

<!ELEMENT extension (contextViewBinding*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT contextViewBinding EMPTY>

<!ATTLIST contextViewBinding

contextId CDATA #REQUIRED

viewId CDATA #REQUIRED

autoOpen (true | false)

autoClose (true | false) >

- **contextId –** Specifies the context identifier that this binding is for.
- **viewId –** Specifies the identifier of the view which should be associated with the specified context. When the specified context is enabled, this view will be automatically brought to the front. When elements are selected in the Debug view, contexts associated with those elements (as specified by extensions of the debugModelContextBindings extension point) are automatically enabled. Note that

this only occurs in perspectives for which the user has requested "automatic view management" via the preferences (by default, only in the Debug perspective).

- **autoOpen –** Specifies whether the view should be automatically opened when the given context is enabled. If unspecified, the value of this attribute is `true`. If this attribute is specified `false`, the view will not be automatically opened, but it will still be brought to the front if it is open when the given context is enabled. Clients are intended to specify `false` to avoid cluttering the perspective with views that are used infrequently.
- **autoClose –** Clients are not intended to specify this attribute except in rare cases. Specifies whether the view should be automatically closed when the given context is disabled (this occurs when all debug targets that contained the specified context have terminated). When unspecified, the value of this attribute is `true`. This attribute should only be specified `false` in the unlikely case that a debugging view must remain open even when the user is not debugging.

Examples:

The following is an example of a context view binding contribution:

<extension point=

"org.eclipse.debug.ui.contextViewBindings"

>

<contextViewBinding contextId=

"com.example.mydebugger.debugging"

viewId=

"com.example.view"

autoOpen=

"true"

autoClose=

"false"

>

</contextViewBinding>

</extension>

In the above example, when a context with the specified identifier is activated by the Debug view, the given view will be automatically opened. When a context which is bound to a different debug model is activated that isn't associated with the view, the view will not be automatically closed.

Since:

Debug Model Context Bindings

Identifier:

org.eclipse.debug.ui.debugModelContextBindings

Since:

3.0

Description:

This extension point provides a mechanism for specifying a context that should be associated with the given debug model. The Debug view uses these bindings to automatically enable contexts. When an element in the Debug view which provides an `IDebugModelProvider` adapter or a stack frame with the specified debug model identifier is selected, the context with the given identifier will be enabled.

Configuration Markup:

<!ELEMENT extension (modelContextBinding*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT modelContextBinding EMPTY>

<!ATTLIST modelContextBinding

debugModelId CDATA #REQUIRED

contextId CDATA #REQUIRED>

- **debugModelId –** specifies the debug model identifier this binding is for
- **contextId –** specifies the context identifier of the context that should be associated with the given debug model

Examples:

The following is an example of a debug model context binding contribution:

<extension point=

"org.eclipse.debug.ui.debugModelContextBindings"

>

<modelContextBinding contextId=

"com.example.myLanguage.debugging"

debugModelId=

"com.example.myLanguageDebugModel"

>

</modelContextBinding>

</extension>

In the above example, when a stack frame with the debug model identifier of "com.example.myLanguageDebugModel" is selected, the context with the identifier "com.example.myLanguage.debugging" will be enabled.

Debug Model Presentation

Identifier:

org.eclipse.debug.ui.debugModelPresentations

Description:

This extension point allows tools to handle the presentation aspects of a debug model. A debug model presentation is responsible for providing labels, images, and editors for elements in a specific debug model.

Configuration Markup:

<!ELEMENT extension (debugModelPresentation*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT debugModelPresentation EMPTY>

<!ATTLIST debugModelPresentation

class CDATA #REQUIRED

id CDATA #REQUIRED

detailsViewerConfiguration CDATA #IMPLIED>

- **class –** fully qualifed name of a Java class that implements the `org.eclipse.debug.ui.IDebugModelPresentation` interface. Since 3.1, debug model presentations may optionally implement `IColorProvider` and `IFontProvider` to override default fonts and colors for debug elements.
- **id –** the identifier of the debug model this presentation is responsible for
- **detailsViewerConfiguration –** the fully qualified name of the Java class that is an instance of `org.eclipse.jface.text.source.SourceViewerConfiguration`. When specified, the source viewer configuration will be used in the "details" area of the variables and expressions view when displaying the details of an element from the debug model associated with this debug model presentation. When unspecified, a default configuration is used.

Examples:

The following is an example of a debug model presentations extension point:

<extension point =

"org.eclipse.debug.ui.debugModelPresentations"

>

<debugModelPresentation class =

"com.example.JavaModelPresentation"

id =

"com.example.JavaDebugModel"

>

</debugModelPresentation>

</extension>

In the example above, the class `com.example.JavaModelPresentation` will be used to render and present debug elements originating from the debug model identified by `com.example.JavaDebugModel`.

API Information:

Value of the action attribute `class` must be a fully qualified class name of a Java class that implements `org.eclipse.debug.ui.IDebugModelPresentation`. Since 3.1, debug model presentations may optionally implement `IColorProvider` and `IFontProvider` to override default fonts and colors for debug elements.

Launch Configuration Tab Groups

Identifier:

org.eclipse.debug.ui.launchConfigurationTabGroups

Description:

This extension point provides a mechanism for contributing a group of tabs to the launch configuration dialog for a type of launch configuration.

Configuration Markup:

<!ELEMENT extension (launchConfigurationTabGroup*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT launchConfigurationTabGroup (launchMode*)>

<!ATTLIST launchConfigurationTabGroup

id CDATA #REQUIRED

type CDATA #REQUIRED

class CDATA #REQUIRED

helpContextId CDATA #IMPLIED

description CDATA #IMPLIED>

- **id –** specifies a unique identifier for this launch configuration tab group.
- **type –** specifies a launch configuration type that this tab group is applicable to (corresponds to the id of a launch configuration type extension).
- **class –** specifies a fully qualified name of a Java class that implements `ILaunchConfigurationTabGroup`.
- **helpContextId –** an optional identifier that specifies the help context to associate with this tab group's launch configuration type
- **description –** A description of the Launch Configuration Type

<!ELEMENT launchMode EMPTY>

<!ATTLIST launchMode

mode CDATA #REQUIRED

perspective CDATA #IMPLIED

description CDATA #IMPLIED>

A launch mode element specifies a launch mode this tab group is specific to. A tab group can be associated with one or more launch modes. For backwards compatibility (previous to 3.0), a launch mode does not need to be specified. When unspecified, a tab group is registered as the default tab group for the associated launch configration type (i.e. applicable to all supported launch modes for which a specific tab group has not been contributed).

- **mode –** identifier for a launch mode this tab group is specific to.
- **perspective –** the default perspective identifier associated with this launch configuration type and launch mode. This allows an extension to cause a perspective switch (or open) when a corresponding launch is registered with the debug plug–in. When unspecified, it indicates that by default, no perspective switch should occurr.
- **description –** A description of the Launch Configuration Type specific to this launchMode.

Examples:

The following is an example of a launch configuration tab group extension point:

<extension point=

"org.eclipse.debug.ui.launchConfigurationTabGroups"

>

<launchConfigurationTabGroup id=

"com.example.ExampleTabGroup"

type=

"com.example.ExampleLaunchConfigurationTypeIdentifier"

class=

"com.example.ExampleLaunchConfigurationTabGroupClass"

>

</launchConfigurationTabGroup>

</extension>

In the above example, the contributed tab group will be shown for the launch configuration type with identifier `com.example.ExampleLaunchConfigurationTypeIdentifier`.

API Information:

Value of the attribute `class` must be a fully qualified name of a Java class that implements the interface `org.eclipse.debug.ui.ILaunchConfigurationTabGroup`.

Launch Configuration Type Images

Identifier:

org.eclipse.debug.ui.launchConfigurationTypeImages

Description:

This extension point provides a way to associate an image with a launch configuration type.

Configuration Markup:

<!ELEMENT extension (launchConfigurationTypeImage*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT launchConfigurationTypeImage EMPTY>

<!ATTLIST launchConfigurationTypeImage

id CDATA #REQUIRED

configTypeID CDATA #REQUIRED

icon CDATA #REQUIRED>

- **id –** specifies a unique identifier for this launch configuration type image.
- **configTypeID –** specifies the fully qualified ID of a launch configuration type.(in 2.1, this attribute can also be specified using the "type" attribute, to be consistent with the launchConfigurationTabGroups extension point).
- **icon –** specifies the plugin–relative path of an image file.

Examples:

The following is an example of a launch configuration type image extension point:

```
<extension point=

"org.eclipse.debug.ui.launchConfigurationTypeImages"

>

<launchConfigurationTypeImage id=

"com.example.FirstLaunchConfigurationTypeImage"

configTypeID=

"com.example.FirstLaunchConfigurationType"

icon=

"icons/FirstLaunchConfigurationType.gif"

>

</launchConfigurationTypeImage>

</extension>
```

Launch Groups

Identifier:

org.eclipse.debug.ui.launchGroups

Since:

2.1

Description:

This extension point provides support for defining a group of launch configurations to be viewed together in the launch configuration dialog, and support a launch history (recent and favorite launches).

Configuration Markup:

<!ELEMENT extension (launchGroup*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT launchGroup EMPTY>

<!ATTLIST launchGroup

id CDATA #REQUIRED

mode CDATA #REQUIRED

category CDATA #IMPLIED

label CDATA #REQUIRED

image CDATA #REQUIRED

bannerImage CDATA #REQUIRED

public CDATA #IMPLIED

title CDATA #IMPLIED>

- **id** – specifies a unique identifier for this launch group.
- **mode** – specifies the launch mode associated with this group – i.e. run or debug.
- **category** – specifies the category of launch configurations in this group. When unspecified, the category is `null`.
- **label** – specifies a translatable label used to render this group.
- **image** – specifies a plug–in relative path to an image used to render this group in trees, lists, tabs, etc.
- **bannerImage** – specifies a plug–in relative path to an image used to render this group in a wizard or dialog banner area.
- **public** – specifies whether this launch group is public and should be have a visible launch history tab in the debug preferences. The implied value is `true`, when not specified.
- **title** – title to display in the launch wizard when this launch group is opened – for example "Select or configure an application to debug" (since 3.1)

Examples:

The following is an example of a launch group extension point:

<extension point=

"org.eclipse.debug.ui.launchGroups"

>

<launchGroup id=

"com.example.ExampleLaunchGroupId"

mode=

"run"

label=

"Run"

image=

"icons\run.gif"

bannerImage=

"icons\runBanner.gif"

>

Since:

</launchGroup>

</extension>

In the above example, the launch group will consist of all launch configurations with no category that support run mode.

Launch Shortcuts

Identifier:

org.eclipse.debug.ui.launchShortcuts

Description:

This extension point provides support for selection sensitive launching. Extensions register a shortcut which appears in the run and/or debug cascade menus to launch the workbench selection or active editor.

Configuration Markup:

<!ELEMENT extension (shortcut*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT shortcut (perspective* , contextualLaunch? , enablement?)>

<!ATTLIST shortcut

id CDATA #REQUIRED

modes CDATA #REQUIRED

class CDATA #REQUIRED

label CDATA #REQUIRED

icon CDATA #IMPLIED

category CDATA #IMPLIED

helpContextId CDATA #IMPLIED

path CDATA #IMPLIED>

- **id –** specifies a unique identifier for this launch shortcut.
- **modes –** specifies a comma−separated list of modes this shortcut supports.

- **class –** specifies the fully qualified name of a class which implements `org.eclipse.debug.ui.ILaunchShortcut`.
- **label –** specifies a label used to render this shortcut.
- **icon –** specifies a plugin–relative path to an image used to render this shortcut. Icon is optional because it is up to other plugins (i.e. Views) to render it.
- **category –** specifies the launch configuration type category this shortcut is applicable for. When unspecified, the category is `null` (default).
- **helpContextId –** an optional identifier that specifies the help context to associate with this launch shortcut
- **path –** an optional menu path used to group launch shortcuts in menus. Launch shortcuts are grouped alphabetically based on the `path` attribute, and then sorted alphabetically within groups based on the `label` attribute. When unspecified, a shortcut appears in the last group. This attribute was added in the 3.0.1 release.

<!ELEMENT perspective EMPTY>

<!ATTLIST perspective

id CDATA #REQUIRED>

The `perspective` element has been **deprecated** in the 3.1 release. The top level Run/Debug/Profile cascade menus now support contextual (selection sensitive) launching, and clients should provide a `contextualLaunch` element instead.

- **id –** the unique identifier of a perspective in which a menu shortcut for this launch shortcut will appear.

<!ELEMENT contextualLaunch (contextLabel* , enablement?)>

Holds all descriptions for adding shortcuts to the selection sensitive Run/Debug/Profile cascade menus.

<!ELEMENT contextLabel EMPTY>

<!ATTLIST contextLabel

mode (run|debug|profile)

label CDATA #REQUIRED>

Specify the label for a contextual launch mode.

- **mode** – specifies a mode from the set {"run","debug","profile"}
- **label** – specifies the label to appear in the contextual launch menu.

<!ELEMENT enablement (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

A generic root element. The element can be used inside an extension point to define its enablement expression. The children of an enablement expression are combined using the and operator.

<!ELEMENT not (not | and | or | instanceof | test | systemTest | equals | count | with | resolve | adapt | iterate)>

This element represent a NOT operation on the result of evaluating it's sub−element expression.

<!ELEMENT and (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

This element represent an AND operation on the result of evaluating all it's sub−elements expressions.

<!ELEMENT or (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

This element represent an OR operation on the result of evaluating all it's sub−element expressions.

<!ELEMENT instanceof EMPTY>

<!ATTLIST instanceof

value CDATA #REQUIRED>

This element is used to perform an instanceof check of the object in focus. The expression returns EvaluationResult.TRUE if the object's type is a sub type of the type specified by the attribute value. Otherwise EvaluationResult.FALSE is returned.

- **value** – a fully qualified name of a class or interface.

<!ELEMENT test EMPTY>

<!ATTLIST test

property CDATA #REQUIRED

args CDATA #IMPLIED

value CDATA #IMPLIED>

This element is used to evaluate the property state of the object in focus. The set of testable properties can be extended using the propery tester extension point. The test expression returns EvaluationResult.NOT_LOADED if teh property tester doing the actual testing isn't loaded yet.

- **property –** the name of an object's property to test.
- **args –** additional arguments passed to the property tester. Multiple arguments are seperated by commas. Each individual argument is converted into a Java base type using the same rules as defined for the value attribute of the test expression.
- **value –** the expected value of the property. Can be omitted if the property is a boolean property. The test expression is supposed to return EvaluationResult.TRUE if the property matches the value and EvaluationResult.FALSE otherwise. The value attribute is converted into a Java base type using the following rules:
  - the string "true" is converted into Boolean.TRUE
  - the string "false" is converted into Boolean.FALSE
  - if the string contains a dot then the interpreter tries to convert the value into a Float object. If this fails the string is treated as a java.lang.String
  - if the string only consists of numbers then the interpreter converts the value in an Integer object.
  - in all other cases the string is treated as a java.lang.String
  - the conversion of the string into a Boolean, Float, or Integer can be suppressed by surrounding the string with single quotes. For example, the attribute value="'true'" is converted into the string "true"

<!ELEMENT systemTest EMPTY>

<!ATTLIST systemTest

property CDATA #REQUIRED

value CDATA #REQUIRED>

Tests a system property by calling the System.getProperty method and compares the result with the value specified through the value attribute.

- **property** – the name of an system property to test.
- **value** – the expected value of the property. The value is interpreted as a string value.

<!ELEMENT equals EMPTY>

<!ATTLIST equals

value CDATA #REQUIRED>

This element is used to perform an equals check of the object in focus. The expression returns EvaluationResult.TRUE if the object is equal to the value provided by the attribute value. Otherwise EvaluationResult.FALSE is returned.

- **value** – the operatand of the equals tests. The value provided as a string is converted into a Java base type using the same rules as for the value attribute of the test expression.

<!ELEMENT count EMPTY>

<!ATTLIST count

value CDATA #REQUIRED>

This element is used to test the number of elements in a collection.

- **value** – an expression to specify the number of elements in a list. Following wildcard characters can be used:
  *
  : any number of elements
  ?
  : no elements or one element
  +
  : one or more elements
  !
  : no elements
  integer value
  : the list must contain the exact number of elements

<!ELEMENT with (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST with

variable CDATA #REQUIRED>

This element changes the object to be inspected for all its child element to the object referneced by the given variable. If the variable can not be resolved then the expression will throw a ExpressionException when evaluating it. The children of a with expression are combined using the and operator.

- **variable –** the name of the variable to be used for further inspection. It is up to the evaluator of an extension point to provide the variable in the variable pool.

<!ELEMENT resolve (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST resolve

variable CDATA #REQUIRED

args CDATA #IMPLIED>

This element changes the object to be inspected for all its child element to the object referneced by the given variable. If the variable can not be resolved then the expression will throw a ExpressionException when evaluating it. The children of a with expression are combined using the and operator.

- **variable –** the name of the variable to be resolved. This variable is then used as the object in focus for child element evaluation. It is up to the evaluator of an extension point to provide a corresponding variable resolver (see IVariableResolver) through the evaluation context passed to the root expression element when evaluating the expression.
- **args –** additional arguments passed to the variable resolver. Multiple arguments are seperated by commas. Each individual argument is converted into a Java base type using the same rules as defined for the value attribute of the test expression.

<!ELEMENT adapt (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST adapt

type CDATA #REQUIRED>

This element is used to adapt the object in focus to the type specified by the attribute type. The expression returns not loaded if either the adapter or the type referenced isn't loaded yet. It throws a ExpressionException during evaluation if the type name doesn't exist at all. The children of an adapt expression are combined using the and operator.

- **type** – the type to which the object in focus is to be adapted.

<!ELEMENT iterate (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST iterate

operator (orland) >

This element is used to iterate over a variable that is of type java.util.Collection. If the object in focus is not of type java.util.Collection then an ExpressionException will be thrown while evaluating the expression.

- **operator** – either "and" or "or". The operator defines how the child elements will be combined. If not specified, "and" will be used.

Examples:

The following is an example of a launch shortcut extension point:

<extension point=

"org.eclipse.debug.ui.launchShortcuts"

>

<shortcut id=

"com.example.ExampleLaunchShortcutId"

modes=

"run,debug"

class=

"com.example.ExampleLaunchShortcutImpl"

label=

"Example Launch Shortcut"

icon=

"icons/examples.gif"

>

<perspective id=

"org.eclipse.jdt.ui.JavaPerspective"

/>

<perspective id=

"org.eclipse.debug.ui.DebugPerspective"

/>

</shortcut>

</extension>

In the above example, a launch shortcut will be shown in the run and debug cascade menus with the label "Example Launch Shortcut", in the JavaPerspective and the DebugPerspective.

API Information:

Value of the attribute **class** must be a fully qualified name of a Java class that implements the interface **org.eclipse.debug.ui.ILaunchShortcut**.

Since 3.1, the debug platform registers a command handler for each launch shortcut and its applicable modes to allow clients to define key−bindings for a launch shortcut. The command id for a handler is generated as the launch shortcut id attribute, followed by a dot and the applicable launch mode. For example, the command id for the above launch shortcut example would be `com.example.ExampleLaunchShortcutId.debug` for launching in debug mode. A command and key binding could be defined as follows, to bind "`ALT−SHIFT−D, E`" to the launch shortcut.

<extension point=

"org.eclipse.ui.commands"

>

<command name=

"Debug Example Launch"

description=

"Debug Example Launch"

```
categoryId=

"org.eclipse.debug.ui.category.run"

id=

"com.example.ExampleLaunchShortcutId.debug"

>

</command>

<keyBinding keySequence=

"M3+M2+D E"

contextId=

"org.eclipse.ui.globalScope"

commandId=

"com.example.ExampleLaunchShortcutId.debug"

keyConfigurationId=

"org.eclipse.ui.defaultAcceleratorConfiguration"

>

</keyBinding>

</extension>
```

Memory Renderings

Identifier:

org.eclipse.debug.ui.memoryRenderings

Since:

3.1 – replacement for memoryRenderingTypes extension point which was considered experimental in 3.0

Description:

Allows plug−ins to contribute arbitrary renderings for memory blocks and bind memory blocks to renderings. For example, a rendering may translate raw bytes of a memory block into ASCII characters.

Configuration Markup:

<!ELEMENT extension (renderingType* | renderingBindings*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point** – a fully qualified identifier of the target extension point
- **id** – an optional identifier of the extension instance
- **name** – an optional name of the extension instance

<!ELEMENT renderingType EMPTY>

<!ATTLIST renderingType

name CDATA #REQUIRED

id CDATA #REQUIRED

class CDATA #REQUIRED>

This element defines a rendering type.

- **name** – human−readable name of this rendering type
- **id** – unique identifier for this rendering
- **class** – fully qualified name of the Java class that implements `org.eclipse.debug.ui.memory.IMemoryRenderingTypeDelegate`

<!ELEMENT renderingBindings (enablement*)>

<!ATTLIST renderingBindings

renderingIds CDATA #IMPLIED

primaryId CDATA #IMPLIED

class CDATA #IMPLIED

defaultIds CDATA #IMPLIED>

Binds memory blocks with available renderings.

- **renderingIds –** comma delimited list of memory rendering type identifiers, specifying available rendering types for memory blocks this binding is enabled for. Must not be specified when `class` is provided.
- **primaryId –** memory rendering type identifier, specifying the default rendering type to be considered primary for memory blocks this binding is enabled for. When there is more than one default rendering type bound to a memory block, the UI may use the information to determine which rendering should be made visible (i.e the primrary one). Must not be specified when `class` is provided. Clients should be careful to specify only one primary rendering type per memory block.
- **class –** fully qualified name of the Java class that implements `org.eclipse.debug.ui.memory.IMemoryRenderingBindingsProvider`, allowing for dynamic rendering bindings. When specified, `renderingIds`, `defaultIds`, and `primaryId` must not be specified.
- **defaultIds –** comma delimited list of memory rendering type identifiers, specifying default rendering types for memory blocks this binding is enabled for. Must not be specified when `class` is provided.

<!ELEMENT enablement (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

A generic root element. The element can be used inside an extension point to define its enablement expression. The children of an enablement expression are combined using the and operator.

<!ELEMENT not (not | and | or | instanceof | test | systemTest | equals | count | with | resolve | adapt | iterate)>

This element represent a NOT operation on the result of evaluating it's sub−element expression.

<!ELEMENT and (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

This element represent an AND operation on the result of evaluating all it's sub−elements expressions.

<!ELEMENT or (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

This element represent an OR operation on the result of evaluating all it's sub−element expressions.

<!ELEMENT instanceof EMPTY>

<!ATTLIST instanceof

value CDATA #REQUIRED>

This element is used to perform an instanceof check of the object in focus. The expression returns EvaluationResult.TRUE if the object's type is a sub type of the type specified by the attribute value. Otherwise EvaluationResult.FALSE is returned.

- **value −** a fully qualified name of a class or interface.

<!ELEMENT test EMPTY>

<!ATTLIST test

property CDATA #REQUIRED

args CDATA #IMPLIED

value CDATA #IMPLIED>

This element is used to evaluate the property state of the object in focus. The set of testable properties can be extended using the propery tester extension point. The test expression returns EvaluationResult.NOT_LOADED if teh property tester doing the actual testing isn't loaded yet.

- **property −** the name of an object's property to test.
- **args −** additional arguments passed to the property tester. Multiple arguments are seperated by commas. Each individual argument is converted into a Java base type using the same rules as defined

for the value attribute of the test expression.

- **value** – the expected value of the property. Can be omitted if the property is a boolean property. The test expression is supposed to return EvaluationResult.TRUE if the property matches the value and EvaluationResult.FALSE otherwise. The value attribute is converted into a Java base type using the following rules:
  - the string "true" is converted into Boolean.TRUE
  - the string "false" is converted into Boolean.FALSE
  - if the string contains a dot then the interpreter tries to convert the value into a Float object. If this fails the string is treated as a java.lang.String
  - if the string only consists of numbers then the interpreter converts the value in an Integer object.
  - in all other cases the string is treated as a java.lang.String
  - the conversion of the string into a Boolean, Float, or Integer can be suppressed by surrounding the string with single quotes. For example, the attribute value="'true'" is converted into the string "true"

<!ELEMENT systemTest EMPTY>

<!ATTLIST systemTest

property CDATA #REQUIRED

value CDATA #REQUIRED>

Tests a system property by calling the System.getProperty method and compares the result with the value specified through the value attribute.

- **property** – the name of an system property to test.
- **value** – the expected value of the property. The value is interpreted as a string value.

<!ELEMENT equals EMPTY>

<!ATTLIST equals

value CDATA #REQUIRED>

This element is used to perform an equals check of the object in focus. The expression returns EvaluationResult.TRUE if the object is equal to the value provided by the attribute value. Otherwise EvaluationResult.FALSE is returned.

- **value** – the operatand of the equals tests. The value provided as a string is converted into a Java base type using the same rules as for the value attribute of the test expression.

<!ELEMENT count EMPTY>

<!ATTLIST count

value CDATA #REQUIRED>

This element is used to test the number of elements in a collection.

- **value** – an expression to specify the number of elements in a list. Following wildcard characters can be used:
  *
  any number of elements
  ?
  no elements or one element
  +
  one or more elements
  !
  no elements
  integer value
  the list must contain the exact number of elements

<!ELEMENT with (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST with

variable CDATA #REQUIRED>

This element changes the object to be inspected for all its child element to the object referneced by the given variable. If the variable can not be resolved then the expression will throw a ExpressionException when evaluating it. The children of a with expression are combined using the and operator.

- **variable** – the name of the variable to be used for further inspection. It is up to the evaluator of an extension point to provide the variable in the variable pool.

<!ELEMENT resolve (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST resolve

variable CDATA #REQUIRED

args CDATA #IMPLIED>

This element changes the object to be inspected for all its child element to the object referneced by the given variable. If the variable can not be resolved then the expression will throw a ExpressionException when evaluating it. The children of a with expression are combined using the and operator.

- **variable –** the name of the variable to be resolved. This variable is then used as the object in focus for child element evaluation. It is up to the evaluator of an extension point to provide a corresponding variable resolver (see IVariableResolver) through the evaluation context passed to the root expression element when evaluating the expression.
- **args –** additional arguments passed to the variable resolver. Multiple arguments are seperated by commas. Each individual argument is converted into a Java base type using the same rules as defined for the value attribute of the test expression.

<!ELEMENT adapt (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST adapt

type CDATA #REQUIRED>

This element is used to adapt the object in focus to the type specified by the attribute type. The expression returns not loaded if either the adapter or the type referenced isn't loaded yet. It throws a ExpressionException during evaluation if the type name doesn't exist at all. The children of an adapt expression are combined using the and operator.

- **type –** the type to which the object in focus is to be adapted.

<!ELEMENT iterate (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST iterate

operator (or|and) >

This element is used to iterate over a variable that is of type java.util.Collection. If the object in focus is not of type java.util.Collection then an ExpressionException will be thrown while evaluating the expression.

- **operator –** either "and" or "or". The operator defines how the child elements will be combined. If not specified, "and" will be used.

Examples:

The following is an example for defining a new rendering type and binding.

<extension point=

"org.eclipse.debug.ui.memoryRenderings"

>

<renderingType name=

"Sample Rendering"

id=

"com.example.sampleRendering"

class=

"com.example.SampleRenderingTypeDelegate"

>

</renderingType>

<renderingBindings renderingIds=

"com.example.sampleRendering"

>

<enablement>

<instanceof value=

"com.example.SampleMemoryBlock"

/>

</enablement>

</renderingBindings>

</extension>

In the above example, a new rendering type, Sample Rendering, is defined. The class `com.example.SampleRenderingTypeDelegate` implements

`org.eclipse.debug.ui.memory.IMemoryRenderingTypeDelegate` and will be used to create new renderings of this type. The rendering is bound to instances of `com.exmaple.SampleMemoryBlock`.

API Information:

- Value of the attribute **class** in a **renderingType** element must be a fully qualifed name of a Java class that implements **org.eclipse.debug.ui.memory.IMemoryRenderingTypeDelegate**.
- Value of the attribute **class** in a **renderingBindings** element must be a fully qualifed name of a Java class that implements **org.eclipse.debug.ui.memory.IMemoryRenderingBindingsProvider**.

Supplied Implementation:

The debug platform provides the following rendering types:
- Hex Rendering (rendering id: org.eclipse.debug.ui.rendering.raw_memory)
- ASCII Rendering (rendering id: org.eclipse.debug.ui.rendering.ascii)
- Signed Integer Rendering (rendering id: org.eclipse.debug.ui.rendering.signedint)
- Unsigned Integer Rendering (rendering id: org.eclipse.debug.ui.rendering.unsignedint)

The debug platform provides a memory view to host renderings.

Source Container Presentations

Identifier:

org.eclipse.debug.ui.sourceContainerPresentations

Since:

3.0

Description:

Extension point to define a presentation aspects of a source container type.

Configuration Markup:

<!ELEMENT extension (sourceContainerPresentation*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT sourceContainerPresentation EMPTY>

<!ATTLIST sourceContainerPresentation

id CDATA #REQUIRED

icon CDATA #IMPLIED

containerTypeID CDATA #REQUIRED

browserClass CDATA #IMPLIED>

An extension point to define presentation aspects of a source container type.

- **id –** The unique id used to refer to this implementation.
- **icon –** The icon that should be displayed for the source container type and instances.

- **containerTypeID** – The unique identifier of the source container type for which this presentation is being provided.
- **browserClass** – A class that can be called to display a browse dialog for the source container type. Must implement `ISourceLocationBrowser`.

Examples:

Following is an example of an source container presentation definition.

<extension point=

"org.eclipse.debug.ui.sourceContainerPresentations"

>

<sourceContainerPresentation browserClass=

"org.eclipse.debug.internal.ui.sourcelookup.browsers.ProjectSourceContainerBrowser"

containerTypeID=

"org.eclipse.debug.core.containerType.project"

icon=

"icons/full/obj16/prj_obj.gif"

id=

"org.eclipse.debug.ui.containerPresentation.project"

>

</sourceContainerPresentation>

</extension>

API Information:

Value of the attribute **browserClass** must be a fully qualified name of a Java class that implements the interface **ISourceLocationBrowser**.

http://www.eclipse.org/legal/epl−v10.html

String Variable Presentations

Identifier:

org.eclipse.debug.ui.stringVariablePresentations

Since:

2.1

Description:

This extension point provides a mechanism for contributing a user interface/presentation for a string substitution variable (i.e. a context variable or value variable).

Configuration Markup:

<!ELEMENT extension (variablePresentation*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT variablePresentation EMPTY>

<!ATTLIST variablePresentation

variableName CDATA #REQUIRED

argumentSelector CDATA #REQUIRED>

- **variableName –** specifies the variable this presentation is for
- **argumentSelector –** specifies a fully qualified name of a Java class that implements `IArgumentSelector`

Examples:

The following is an example of a variable presentation contribution:

<extension point=

"org.eclipse.debug.ui.stringVariablePresentations"

>

<variablePresentation variableName=

"example_variable"

argumentSelector=

"com.example.ExampleArgumentChooser"

>

</variablePresentation>

</extension>

In the above example, the contributed presentation will be used for the variable named "example_variable". An argument selector is specified to configure an argument applicable to the variable.

API Information:

Value of the attribute **argumentSelector** must be a fully qualified name of a Java class that implements the interface **org.eclipse.debug.ui.stringsubstitution.IArgumentSelector**.

Variable Value Editors

Identifier:

org.eclipse.debug.ui.variableValueEditors

Since:

3.1

Description:

This extension point provides a mechanism for contributing variable value editors for a particular debug model. Variable value editors are used to edit/save variable values. When the user invokes the "Change Value..." or "Assign Value" (details pane) actions on a variable in the Variables view, the contributed `org.eclipse.debug.ui.actions.IVariableValueEditor` will be invoked to change the value.

Configuration Markup:

<!ELEMENT extension (variableValueEditor)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT variableValueEditor EMPTY>

<!ATTLIST variableValueEditor

class CDATA #REQUIRED

modelId CDATA #REQUIRED>

- **class –** Implementation of `org.eclipse.debug.ui.actions.IVariableValueEditor`
- **modelId –** The debug model identifier for which the given variable value editor is applicable.

Examples:

<extension point=

"org.eclipse.debug.ui.variableValueEditors"

>

<variableEditor modelId=

"com.examples.myDebugModel"

class=

"com.examples.variables.MyVariableValueEditor"

/>

</extension>

API Information:

Value of the attribute **class** must be a fully qualified name of a Java class that implements the interface **org.eclipse.debug.ui.actions.IVariableValueEditor**.

Console Factories

Identifier:

org.eclipse.ui.console.consoleFactories

Since:

3.1

Description:

A console factory creates or activates a console, and appears as an action in the console view.

Configuration Markup:

<!ELEMENT extension (consoleFactory)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT consoleFactory EMPTY>

<!ATTLIST consoleFactory

label CDATA #REQUIRED

icon CDATA #IMPLIED

class CDATA #REQUIRED>

- **label –** action label to appear in menu
- **icon –** optional plug–in relative path to an icon to appear with the action
- **class –** class implementing `org.eclipse.ui.console.IConsoleFactory` that will be called when the associated action is invoked

Examples:

The following is an example of a console factory extension point:

```
<extension point=
"org.eclipse.ui.console.consoleFactories"
>
<consoleFactory label=
"Command Console"
class=
"com.example.CommandConsoleFactory"
icon=
"icons/cmd_console.gif"
>
</consoleFactory>
</extension>
```

This extension will cause an entry to appear in the console view's **Open Console** drop−down menu labelled **Command Console** with the specified icon. When the action is invoked by a user, the method **openConsole()** on the class **com.example.CommandConsoleFactory** will be called. The factory can the decide to create a new console or activate an existing console.

API Information:

Value of the attribute **class** must be a fully qualified name of a Java class that implements the interface **org.eclipse.ui.console.IConsoleFactory**.

Supplied Implementation:

The console plug−in provides a console factory to open a new console view.

Console Page Participants

Identifier:

org.eclipse.ui.console.consolePageParticipants

Since:

3.1

Description:

This extension point provides a mechanism for contributing a console page participant. A console page participant is notified of page lifecycle events such as creation, activation, deactivation and disposal. A page participant can also provide adapters for a page.

Configuration Markup:

<!ELEMENT extension (consolePageParticipant)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT consolePageParticipant (enablement)>

<!ATTLIST consolePageParticipant

id CDATA #REQUIRED

class CDATA #REQUIRED>

- **id –** specifies a unique identifier for this Console Page Participant
- **class –** specifies a fully qualified name of a Java class that implements `IConsolePageParticipant`

<!ELEMENT enablement (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

A generic root element. The element can be used inside an extension point to define its enablement expression. The children of an enablement expression are combined using the and operator.

<!ELEMENT not (not | and | or | instanceof | test | systemTest | equals | count | with | resolve | adapt | iterate)>

This element represent a NOT operation on the result of evaluating it's sub−element expression.

<!ELEMENT and (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

This element represent an AND operation on the result of evaluating all it's sub−elements expressions.

<!ELEMENT or (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

This element represent an OR operation on the result of evaluating all it's sub−element expressions.

<!ELEMENT instanceof EMPTY>

<!ATTLIST instanceof

value CDATA #REQUIRED>

This element is used to perform an instanceof check of the object in focus. The expression returns EvaluationResult.TRUE if the object's type is a sub type of the type specified by the attribute value. Otherwise EvaluationResult.FALSE is returned.

- **value −** a fully qualified name of a class or interface.

<!ELEMENT test EMPTY>

<!ATTLIST test

property CDATA #REQUIRED

Since:

args CDATA #IMPLIED

value CDATA #IMPLIED>

This element is used to evaluate the property state of the object in focus. The set of testable properties can be extended using the propery tester extension point. The test expression returns EvaluationResult.NOT_LOADED if teh property tester doing the actual testing isn't loaded yet.

- **property** – the name of an object's property to test.
- **args** – additional arguments passed to the property tester. Multiple arguments are seperated by commas. Each individual argument is converted into a Java base type using the same rules as defined for the value attribute of the test expression.
- **value** – the expected value of the property. Can be omitted if the property is a boolean property. The test expression is supposed to return EvaluationResult.TRUE if the property matches the value and EvaluationResult.FALSE otherwise. The value attribute is converted into a Java base type using the following rules:
  - the string "true" is converted into Boolean.TRUE
  - the string "false" is converted into Boolean.FALSE
  - if the string contains a dot then the interpreter tries to convert the value into a Float object. If this fails the string is treated as a java.lang.String
  - if the string only consists of numbers then the interpreter converts the value in an Integer object.
  - in all other cases the string is treated as a java.lang.String
  - the conversion of the string into a Boolean, Float, or Integer can be suppressed by surrounding the string with single quotes. For example, the attribute value="'true'" is converted into the string "true"

<!ELEMENT systemTest EMPTY>

<!ATTLIST systemTest

property CDATA #REQUIRED

value CDATA #REQUIRED>

Tests a system property by calling the System.getProperty method and compares the result with the value specified through the value attribute.

- **property** – the name of an system property to test.
- **value** – the expected value of the property. The value is interpreted as a string value.

<!ELEMENT equals EMPTY>

<!ATTLIST equals

value CDATA #REQUIRED>

This element is used to perform an equals check of the object in focus. The expression returns EvaluationResult.TRUE if the object is equal to the value provided by the attribute value. Otherwise EvaluationResult.FALSE is returned.

- **value** – the operatand of the equals tests. The value provided as a string is converted into a Java base type using the same rules as for the value attribute of the test expression.

<!ELEMENT count EMPTY>

<!ATTLIST count

value CDATA #REQUIRED>

This element is used to test the number of elements in a collection.

- **value** – an expression to specify the number of elements in a list. Following wildcard characters can be used:
  *
  any number of elements
  ?
  no elements or one element
  +
  one or more elements
  !
  no elements
  integer value
  the list must contain the exact number of elements

<!ELEMENT with (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST with

variable CDATA #REQUIRED>

This element changes the object to be inspected for all its child element to the object referneced by the given variable. If the variable can not be resolved then the expression will throw a ExpressionException when evaluating it. The children of a with expression are combined using the and operator.

- **variable –** the name of the variable to be used for further inspection. It is up to the evaluator of an extension point to provide the variable in the variable pool.

<!ELEMENT resolve (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST resolve

variable CDATA #REQUIRED

args CDATA #IMPLIED>

This element changes the object to be inspected for all its child element to the object referneced by the given variable. If the variable can not be resolved then the expression will throw a ExpressionException when evaluating it. The children of a with expression are combined using the and operator.

- **variable –** the name of the variable to be resolved. This variable is then used as the object in focus for child element evaluation. It is up to the evaluator of an extension point to provide a corresponding variable resolver (see IVariableResolver) through the evaluation context passed to the root expression element when evaluating the expression.
- **args –** additional arguments passed to the variable resolver. Multiple arguments are seperated by commas. Each individual argument is converted into a Java base type using the same rules as defined for the value attribute of the test expression.

<!ELEMENT adapt (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST adapt

type CDATA #REQUIRED>

This element is used to adapt the object in focus to the type specified by the attribute type. The expression returns not loaded if either the adapter or the type referenced isn't loaded yet. It throws a ExpressionException during evaluation if the type name doesn't exist at all. The children of an adapt expression are combined using the and operator.

- **type –** the type to which the object in focus is to be adapted.

<!ELEMENT iterate (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST iterate

operator (or|and) >

This element is used to iterate over a variable that is of type java.util.Collection. If the object in focus is not of type java.util.Collection then an ExpressionException will be thrown while evaluating the expression.

- **operator –** either "and" or "or". The operator defines how the child elements will be combined. If not specified, "and" will be used.

Examples:

The following is an example of a console page participant extension point:

<extension point=

"org.eclipse.ui.console.consolePageParticipant"

>

<consolePageParticipant class=

"com.example.ExamplePageParticipant"

id=

"com.example.ExamplePageParticipant"

>

<enablement>

<instanceof value=

"com.example.ExampleConsole"

/>

</enablement>

</consolePageParticipant>

</extension>

In the above example, the contributed console page participant will be used for all consoles of type "com.example.ExampleConsole."

API Information:

Value of the attribute **class** must be a fully qualified name of a Java class that implements the interface **org.eclipse.debug.ui.console.IConsolePageParticipantDelegate**.

Console Pattern Match Listeners

Identifier:

org.eclipse.ui.console.consolePatternMatchListeners

Since:

3.1

Description:

Provides regular expression matching for text consoles. Pattern match listeners can be contributed to specific consoles by using an enablement expression. Listeners are notified as matches are found.

Configuration Markup:

<!ELEMENT extension (consolePatternMatchListener*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT consolePatternMatchListener (enablement)>

<!ATTLIST consolePatternMatchListener

id CDATA #REQUIRED

class CDATA #REQUIRED

regex CDATA #REQUIRED

flags CDATA #IMPLIED

qualifier CDATA #IMPLIED>

- **id –** specifies a unique identifier for this console pattern match listener
- **class –** specifies a fully qualified name of a Java class that implements `IPatternMatchListenerDelegate`
- **regex –** specifies the regular expression to be matched

- **flags –** specifies flags to be used when matching the pattern. Acceptable flags are defined in `java.util.regex.Pattern` and should be specified as Strings (eg "Pattern.MULTILINE" or "MULTILINE")
- **qualifier –** a simple regular expression used to identify a line that may contain this pattern match listener's complete regular expression `regex`. When a line is found containing this expression, a search is performed from the beginning of the line for this pattern matcher's complete `regex`. Use of this attribute is optional but can greatly improve performance as lines not containing this expression are disqualified from the search.

<!ELEMENT enablement (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

A generic root element. The element can be used inside an extension point to define its enablement expression. The children of an enablement expression are combined using the and operator.

<!ELEMENT not (not | and | or | instanceof | test | systemTest | equals | count | with | resolve | adapt | iterate)>

This element represent a NOT operation on the result of evaluating it's sub−element expression.

<!ELEMENT and (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

This element represent an AND operation on the result of evaluating all it's sub−elements expressions.

<!ELEMENT or (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

This element represent an OR operation on the result of evaluating all it's sub−element expressions.

<!ELEMENT instanceof EMPTY>

<!ATTLIST instanceof

value CDATA #REQUIRED>

This element is used to perform an instanceof check of the object in focus. The expression returns EvaluationResult.TRUE if the object's type is a sub type of the type specified by the attribute value. Otherwise

EvaluationResult.FALSE is returned.

- **value –** a fully qualified name of a class or interface.

<!ELEMENT test EMPTY>

<!ATTLIST test

property CDATA #REQUIRED

args CDATA #IMPLIED

value CDATA #IMPLIED>

This element is used to evaluate the property state of the object in focus. The set of testable properties can be extended using the propery tester extension point. The test expression returns EvaluationResult.NOT_LOADED if teh property tester doing the actual testing isn't loaded yet.

- **property –** the name of an object's property to test.
- **args –** additional arguments passed to the property tester. Multiple arguments are seperated by commas. Each individual argument is converted into a Java base type using the same rules as defined for the value attribute of the test expression.
- **value –** the expected value of the property. Can be omitted if the property is a boolean property. The test expression is supposed to return EvaluationResult.TRUE if the property matches the value and EvaluationResult.FALSE otherwise. The value attribute is converted into a Java base type using the following rules:
  - the string "true" is converted into Boolean.TRUE
  - the string "false" is converted into Boolean.FALSE
  - if the string contains a dot then the interpreter tries to convert the value into a Float object. If this fails the string is treated as a java.lang.String
  - if the string only consists of numbers then the interpreter converts the value in an Integer object.
  - in all other cases the string is treated as a java.lang.String
  - the conversion of the string into a Boolean, Float, or Integer can be suppressed by surrounding the string with single quotes. For example, the attribute value="'true'" is converted into the string "true"

<!ELEMENT systemTest EMPTY>

<!ATTLIST systemTest

property CDATA #REQUIRED

value CDATA #REQUIRED>

Tests a system property by calling the System.getProperty method and compares the result with the value specified through the value attribute.

- **property** – the name of an system property to test.
- **value** – the expected value of the property. The value is interpreted as a string value.

<!ELEMENT equals EMPTY>

<!ATTLIST equals

value CDATA #REQUIRED>

This element is used to perform an equals check of the object in focus. The expression returns EvaluationResult.TRUE if the object is equal to the value provided by the attribute value. Otherwise EvaluationResult.FALSE is returned.

- **value** – the operatand of the equals tests. The value provided as a string is converted into a Java base type using the same rules as for the value attribute of the test expression.

<!ELEMENT count EMPTY>

<!ATTLIST count

value CDATA #REQUIRED>

This element is used to test the number of elements in a collection.

- **value** – an expression to specify the number of elements in a list. Following wildcard characters can be used:
  *
  any number of elements
  ?
  no elements or one element
  +
  one or more elements
  !
  no elements
  integer value
  the list must contain the exact number of elements

<!ELEMENT with (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST with

variable CDATA #REQUIRED>

This element changes the object to be inspected for all its child element to the object referneced by the given variable. If the variable can not be resolved then the expression will throw a ExpressionException when evaluating it. The children of a with expression are combined using the and operator.

- **variable –** the name of the variable to be used for further inspection. It is up to the evaluator of an extension point to provide the variable in the variable pool.

<!ELEMENT resolve (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST resolve

variable CDATA #REQUIRED

args CDATA #IMPLIED>

This element changes the object to be inspected for all its child element to the object referneced by the given variable. If the variable can not be resolved then the expression will throw a ExpressionException when evaluating it. The children of a with expression are combined using the and operator.

- **variable –** the name of the variable to be resolved. This variable is then used as the object in focus for child element evaluation. It is up to the evaluator of an extension point to provide a corresponding variable resolver (see IVariableResolver) through the evaluation context passed to the root expression element when evaluating the expression.
- **args –** additional arguments passed to the variable resolver. Multiple arguments are seperated by commas. Each individual argument is converted into a Java base type using the same rules as defined for the value attribute of the test expression.

<!ELEMENT adapt (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST adapt

type CDATA #REQUIRED>

This element is used to adapt the object in focus to the type specified by the attribute type. The expression returns not loaded if either the adapter or the type referenced isn't loaded yet. It throws a ExpressionException during evaluation if the type name doesn't exist at all. The children of an adapt expression are combined using the and operator.

- **type** – the type to which the object in focus is to be adapted.

<!ELEMENT iterate (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST iterate

operator (or|and) >

This element is used to iterate over a variable that is of type java.util.Collection. If the object in focus is not of type java.util.Collection then an ExpressionException will be thrown while evaluating the expression.

- **operator** – either "and" or "or". The operator defines how the child elements will be combined. If not specified, "and" will be used.

Examples:

The following is an example of a console pattern match listener extension point:

<extension point=

"org.eclipse.ui.console.consolePatternMatchListener"

>

<consolePatternMatchListener class=

"com.example.ExampleConsolePatternMatcher"

id=

"com.example.ExampleConsolePatternMatcher"

regex=

".*foo.*"

>

<enablement>

<test property=

"org.eclipse.ui.console.consoleTypeTest"

value=

"exampleConsole"

/>

</enablement>

</consolePatternMatchListener>

</extension>

In the above example, the contributed console pattern matcher will be used for consoles with a type of "exampleConsole."

API Information:

Value of the attribute **class** must be a fully qualified name of a Java class that implements the interface **org.eclipse.ui.console.IPatternMatchListenerDelegate**.

Supplied Implementation:

The console plug−in provides a console type property tester for enablement expressions that tests the value of `IConsole.getType()`. The property tester's identifier is `org.eclipse.ui.console.consoleTypeTest`.

Help Content Producer

Identifier:

org.eclipse.help.contentProducer

Since:

3.0

Description:

For providing dynamic, generated at run time, help content.

Configuration Markup:

<!ELEMENT extension (contentProducer?)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

<!ELEMENT contentProducer (producer)>

<!ATTLIST contentProducer

producer CDATA #IMPLIED>

- **producer –** the implementation class for the help content producer. This class must implement the `org.eclipse.help.IHelpContentProducer` interface. This attribute may be omitted, and the nested `producer` element may be provided instead.

<!ELEMENT producer (parameter*)>

<!ATTLIST producer

class CDATA #REQUIRED>

- **class –** the implementation class for the help content producer. This class must implement the `org.eclipse.help.IHelpContentProducer` interface.

<!ELEMENT parameter EMPTY>

<!ATTLIST parameter

name CDATA #REQUIRED

value CDATA #REQUIRED>

- **name** – name of a parameter passed to the implementation class
- **value** – value of a parameter passed to the implementation class

Examples:

The following is a sample usage of the browser extension point:

<extension point=

"org.eclipse.help.contentProducer"

id=

"org.eclipse.myPlugin.myDynamicHelpProducer"

name=

"My Dynamic Help Content"

>

<contentProducer producer=

"org.eclipse.myPlugin.myPackage.Myproducer"

/>

</extension>

API Information:

The supplied content producer class must implement the `org.eclipse.help.IHelpContentProducer` interface. The producer is responsible for providing content for dynamic help resources from a plug−in. The method of content producer is called by help for every help resource obtained from the plug−in.

Supplied Implementation:

None. If a documentation plug−in does not provide help content producer or a call to it results in null, help system searches doc.zip and file system in the plug−in install location for a static document and displays its content.

Contexts

Identifier:

org.eclipse.help.contexts

Description:

For defining context−sensitive help for an individual plug−in.

Configuration Markup:

<!ELEMENT extension (contexts*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

<!ELEMENT contexts EMPTY>

<!ATTLIST contexts

file CDATA #REQUIRED

plugin CDATA #IMPLIED>

- **file** − the name of the manifest file which contains the context−sensitive help documentation for this plug−in.

  Configuration Markup for what goes into the contexts manifest file:

  ```
  <!ELEMENT contexts (context)* ) >

  <!ELEMENT context (description?,topic*) >
  <!ATTLIST context id ID #REQUIRED >

  <!ELEMENT description (#PCDATA)>

  <!ELEMENT topic EMPTY >
  <!ATTLIST topic label CDATA #REQUIRED >
  <!ATTLIST topic href CDATA #IMPLIED >
  ```

  The contexts manifest files provide all the information needed when context−sensitive help is requested by the user. The id is passed by the platform to the help system to identify the currently active context.The context definitions with matching IDs are then retrieved. The IDs in the manifest

file, must not contain the period character, since the IDs are uniquely identified by pluginID.contextID string when in memory. The IContext object is then created by help system that contains descriptions and topics from all context definitions for a given ID in a plug−in. The description is to be displayed to the user, and related topics might be useful to the user for understanding the current context. The related topic are html files packaged in doc.zip, together with topics that are part of on line help.

- **plugin** – Plugin to which its context definitions are extended with extra information.

If a plugin defines some context id's, one can extend the description or related links of a context by declaring another context with the same id.

Examples:

The following is an example of using the contexts extension point:
(in file `plugin.xml`)

<extension point=

"org.eclipse.help.contexts"

>

<contexts file=

"xyzContexts.xml"

/>

</extension>

(in file `xyzContexts.xml`)

```
    <contexts>
      <context  id="generalContextId">
        <description> This is a sample F1 help string.</description>
        <topic href="contexts/RelatedContext1.html"  label="Help Related
Topic 1"/>
        <topic href="contexts/RelatedContext2.html"  label="Help Related
Topic 2"/>
      </context>
    </contexts>
```

Externalizing Strings The Context XML files can be translated and the resulting copy (with translated descriptions labels) should be placed in nl/<language>/<country> or nl/<language> directory. The <language> and <country> stand for two letter language and country codes as used in locale codes. For example, Traditional Chinese translations should be placed in the nl/zh/TW directory. The

nl/<language>/<country> directory has a higher priority than nl/<language>. Only if no file is found in the nl/<language>/<country>, the file residing in nl/<language> will be used. The the root directory of a plugin will be searched last.

The related topics contained in doc.zip can be localized by creating a doc.zip file with translated version of documents, and placing doc.zip in
nl/<language>/<country> or nl/<language> directory. The help system will look for the files under this directories before defaulting to plugin directory.

API Information:

No code is required to use this extension point. All that is needed is to supply the appropriate manifest file(s) mentioned in the plugin.xml file.

Supplied Implementation:

The optional default implementation of the help system UI supplied with the Eclipse platform fully supports the `contexts` extension point.

Browser

Identifier:

org.eclipse.help.base.browser

Since:

3.0 (originally added in release 2.0 as org.eclipse.help.ui.browser)

Description:

For providing web browsers capable of displaying html documents at a given URL.

Note: since 3.1, help system uses workbench browser support (see `org.eclipse.ui.browserSupport`). The extension point described in this document remains to be used for standalone help only. Any custom browser adapters that need to work in the workbench mode need to be moved to the workbench extension point.

Configuration Markup:

<!ELEMENT extension (browser*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

<!ELEMENT browser (factoryclass)>

<!ATTLIST browser

id CDATA #REQUIRED

factoryclass CDATA #IMPLIED

name CDATA #REQUIRED>

- **id –** the unique ID of the browser.
- **factoryclass –** the implementation class for the browser factory. This class must implement the `org.eclipse.help.browser.IBrowserFactory` interface. This attribute may be omitted, and the nested `factoryclass` element may be provided instead.
- **name –** the name of the browser (translatable).

<!ELEMENT factoryclass (parameter*)>

<!ATTLIST factoryclass

class CDATA #REQUIRED>

- **class** – the implementation class for the browser factory. This class must implement the `org.eclipse.help.browser.IBrowserFactory` interface.

<!ELEMENT parameter EMPTY>

<!ATTLIST parameter

name CDATA #REQUIRED

value CDATA #REQUIRED>

- **name** – name of a parameter passed to the implementation class
- **value** – value of a parameter passed to the implementation class

Examples:

The following is a sample usage of the browser extension point:

<extension point=

"org.eclipse.help.base.browser"

>

<browser id=

"org.eclipse.myPlugin.myBrowserID"

factoryClass=

"org.eclipse.myPlugin.myPackage.MyFactoryClass"

name=

"My Browser"

>

</browser>

Since:

</extension>

API Information:

The supplied factory class must implement the `org.eclipse.help.browser.IBrowserFactory` interface. Methods in that interface determine whether the factory is available on the given system, i.e. is capable of supplying browser instances, and create browser instances that implement IBrowser interface.

Supplied Implementation:

The `org.eclipse.help.base` and `org.eclipse.help.ui` plug−ins contain implementation of browsers on common platforms. Other plug−ins can provide different implementations. In the preferences, the user can select the default browser from among available browsers.

Lucene Analyzer

Identifier:

org.eclipse.help.base.luceneAnalyzer

Since:

3.0 (originally added in release 2.0 as org.eclipse.help.luceneAnalyzer)

Description:

This extension point is used to register text analyzers for use by help when indexing and searching documentation.

Help exploits capabilities of the Lucene search engine, that allows indexing of token streams (streams of words). Analyzers create tokens from the character stream. They examine text content and provide tokens for use with the index. The text stream can be tokenized in many unique ways. A trivial analyzer can tokenize streams at white space, a different one can perform filtering of tokens, based on the application needs. Since the documentation is mostly human−readable text, it is desired that analyzers used by the help system perform language and grammar aware tokenization and normalization of indexed text. For some languages, the quality of search increases significantly if stop word removal and stemming is performed on the indexed text.

The analyzer contributed to this extension point will override the one provided by the Eclipse help system for a given locale.

Configuration Markup:

<!ELEMENT extension (analyzer*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

<!ELEMENT analyzer EMPTY>

<!ATTLIST analyzer

locale CDATA #REQUIRED

class CDATA #REQUIRED>

- **locale** − a string identifying locale for which the supplied analyzer is to bue sued. If two letters, language is provided, and the analyzer will be available to all locales of that language.

- **class** – a fully qualified name of the Java class extending `org.apache.lucene.analysis.Analyzer`.

Examples:

Following is an example of Lucene Analyzer configuration:

<extension id=

"com.xyx.XYZ"

point=

"org.eclipse.help.base.luceneAnalyzer"

>

<analyzer locale=

"ll_CC"

class=

"com.xyz.ll_CCAnalyzer"

/>

</extension>

API Information:

The value of the `locale` attribute must represent either a five– or two–charcter locale string. If the analyzer is configured for a language by specifying two–letter language designation, the analyzer is going to be used for all locales of this language. If the analyzer is configured that matchs a five–character locale, it is going to be used instead.

The value of the `class` attribute must represent a class that extends `org.apache.lucene.analysis.Analyzer`. It is recommended that this analyzer performs lowercase filtering for languages where it is possible to increase number of search hits by making search case–sensitive.

Supplied Implementation:

The Eclipse help system provides analyzers for all languages. For English and German, the analyzers perform stop word filtering, lowercase filtering, and stemming. For all the other languages the supplied analyzer only performs lowercase filtering.

Ant Properties

Identifier:

org.eclipse.ant.core.antProperties

Since:

3.0

Description:

Allows plug−ins to define Ant properties for use in Ant build files.

Configuration Markup:

<!ELEMENT extension (antProperty*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point −** a fully qualified identifier of the target extension point
- **id −** an optional identifier of the extension instance
- **name −** an optional name of the extension instance

<!ELEMENT antProperty EMPTY>

<!ATTLIST antProperty

name CDATA #REQUIRED

value CDATA #IMPLIED

class CDATA #IMPLIED

headless (true | false)

eclipseRuntime (true | false) >

- **name −** The name of the property.
- **value −** The value of the property. If a value is provided here, the "class" attribute is ignored.
- **class −** If there is no 'value' attribute provided, this class is called to return the dynamic value for the Ant property. If `null` is returned, the value is not set.

- **headless** – indicates whether this property is suitable for use in a "headless" Ant environment. If running headless and the attribute is "false", the property will not be set and any specified `org.eclipse.ant.core.IAntPropertyProvider` will not be instantiated. The implied value is `true`, when not specified.
- **eclipseRuntime** – indicates whether this property should only be considered when run in the same VM as Eclipse. The implied value is `true`, when not specified.

Examples:

The following is an example of an Ant properties extension point:

<extension point=

"org.eclipse.ant.core.antProperties"

>

<antProperty name=

"eclipse.home"

class=

"org.eclipse.ant.internal.core.AntPropertyValueProvider"

/>

<antProperty name=

"eclipse.running"

value=

"true"

/>

</extension>

API Information:

The class named in the `class` property must implement the `org.eclipse.ant.core.IAntPropertyProvider` interface.

Supplied Implementation:

The platform uses this mechanism to set the Ant property eclipse.home to the Eclipse installation directory and to set the eclipse.running property.

Ant Tasks

Identifier:

org.eclipse.ant.core.antTasks

Description:

Allows plug−ins to define arbitrary Ant tasks for use by the Ant infrastructure. The standard Ant infrastructure allows for the addition of arbitrary tasks. Unfortunately, it is unlikely that the Ant Core plug−in would have the classes required by these tasks on its classpath (or that of any of its prerequisites). To address this, clients should define an extension which plugs into this extension−point and maps a task name onto a class. The Ant plug−in can then request that the declaring plug−in load the specified class.

Configuration Markup:

<!ELEMENT extension (antTask*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point −** a fully qualified identifier of the target extension point
- **id −** an optional identifier of the extension instance
- **name −** an optional name of the extension instance

<!ELEMENT antTask EMPTY>

<!ATTLIST antTask

name CDATA #REQUIRED

class CDATA #REQUIRED

library CDATA #REQUIRED

headless (true | false)

eclipseRuntime (true | false) >

- **name −** name of the task to be defined
- **class −** the fully qualified name of a Java class implementing the task. Generally this class must be a subclass of `org.apache.tools.ant.Task`.
- **library −** a path relative to the plug−in install location for the library containing the task.

- **headless** – indicates whether this task is suitable for use in a "headless" Ant environment. If running headless and the attribute is "false", the task will not be loaded or defined. As well, the plugin class loader will not be added as a parent classloader to the Ant classloader. The implied value is true, when not specified.
- **eclipseRuntime** – indicates whether this task requires an Eclipse runtime (i.e. must be run in the same VM as Eclipse). The implied value is true, when not specified.

Examples:

The following is an example of an Ant tasks extension point:

<extension point=

"org.eclipse.ant.core.antTasks"

>

<antTask name=

"coolTask"

class=

"com.example.CoolTask"

library=

"lib/antSupport.jar"

/>

</extension>

Supplied Implementation:

The platform itself supplies a number of tasks including eclipse.incrementalBuild and eclipse.refreshLocal.

Ant Types

Identifier:

org.eclipse.ant.core.antTypes

Description:

Allows plug−ins to define arbitrary Ant datatypes for use by the Ant infrastructure. The standard Ant infrastructure allows for the addition of arbitrary datatypes. Unfortunately, it is unlikely that the Ant Core plug−in would have the classes required by these datatypes on its classpath (or that of any of its prerequisites). To address this, clients should define an extension which plugs into this extension−point and maps a datatype name onto a class. The Ant plug−in can then request that the declaring plug−in load the specified class.

Configuration Markup:

<!ELEMENT extension (antType*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point −** a fully qualified identifier of the target extension point
- **id −** an optional identifier of the extension instance
- **name −** an optional name of the extension instance

<!ELEMENT antType EMPTY>

<!ATTLIST antType

name CDATA #REQUIRED

class CDATA #REQUIRED

library CDATA #REQUIRED

headless (true | false)

eclipseRuntime (true | false) >

- **name −** name of the type to be defined
- **class −** the fully qualified name of a Java class implementing the datatype. Generally this class must be a subclass of `org.apache.tools.ant.types.DataType`.
- **library −** a path relative to the plug−in install location for the library containing the type.

- **headless** – indicates whether this type is suitable for use in a "headless" Ant environment. If running headless and the attribute is `false`, the type will not be loaded or defined. As well, the plugin class loader will not be added as a parent classloader to the Ant classloader. The implied value is `true`, when not specified.
- **eclipseRuntime** – indicates whether this type requires an Eclipse runtime (i.e. must be run in the same VM as Eclipse). The implied value is `true`, when not specified.

Examples:

The following is an example of an Ant types extension point:

<extension point=

"org.eclipse.ant.core.antTypes"

>

<antType name=

"coolType"

class=

"com.example.CoolType"

library=

"lib/antSupport.jar"

/>

</extension>

Extra Ant Classpath Entries

Identifier:

org.eclipse.ant.core.extraClasspathEntries

Description:

Allows plug−ins to define arbitrary JARs for use by the Ant infrastructure. These JARs are put into the Ant classpath at runtime. Besides the JAR, the plug−in classloader of the plug−in providing the JAR is also added to the classpath.

Configuration Markup:

<!ELEMENT extension (extraClasspathEntry*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point −** a fully qualified identifier of the target extension point
- **id −** an optional identifier of the extension instance
- **name −** an optional name of the extension instance

<!ELEMENT extraClasspathEntry EMPTY>

<!ATTLIST extraClasspathEntry

library CDATA #REQUIRED

headless (true | false)

eclipseRuntime (true | false) >

- **library −** a path relative to the plug−in install location for the library.
- **headless −** indicates whether this extra classpath entry is suitable for use in a "headless" Ant environment. If running headless and the attribute is `false`, this entry will not be added to the Ant classpath. As well, the plugin class loader will not be added as a parent classloader to the Ant classloader. The implied value is `true`, when not specified.
- **eclipseRuntime −** indicates whether this extra classpath entry should only be considered for builds run in the same VM as Eclipse. The implied value is true, when not specified.

Examples:

The following is an example of an extra classpath entries extension point:

<extension point=

"org.eclipse.ant.core.extraClasspathEntries"

>

<extraClasspathEntry library=

"myExtraLibrary.jar"

/>

</extension>

Supplied Implementation:

The platform itself supplies an Ant support jar (antsupportlib.jar).

ContentMerge Viewers

Identifier:

org.eclipse.compare.contentMergeViewers

Description:

This extension point allows a plug−in to register compare/merge viewers for specific content types. The viewer is expected to extend `org.eclipse.jface.viewers.Viewer`. However, since viewers don't have a default constructor, the extension point must implement the factory interface for viewers `org.eclipse.compare.IViewerCreator`.

Configuration Markup:

<!ELEMENT extension (viewer* , contentTypeBinding*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT viewer EMPTY>

<!ATTLIST viewer

id CDATA #REQUIRED

class CDATA #REQUIRED

extensions CDATA #IMPLIED>

- **id –** a unique identifier that can be used to reference the viewer
- **class –** a fully qualified name of a class that implements a factory for the content merge viewer and implements `org.eclipse.compare.IViewerCreator`
- **extensions –** a comma separated list of file extensions e.g. "java, gif"

<!ELEMENT contentTypeBinding EMPTY>

<!ATTLIST contentTypeBinding

contentTypeId CDATA #REQUIRED

contentMergeViewerId CDATA #REQUIRED>

- **contentTypeId –**
- **contentMergeViewerId –**

Examples:

The following is an example of a compare/merge viewer for text files (extension "txt"):

<extension point =

"org.eclipse.compare.contentMergeViewers"

>

<viewer id=

"org.eclipse.compare.contentmergeviewer.TextMergeViewer"

class=

"org.eclipse.compare.internal.TextMergeViewerCreator"

extensions=

"txt"

/>

</extension>

API Information:

The contributed class must implement `org.eclipse.compare.IViewerCreator`

Supplied Implementation:

The Compare UI plugin defines content viewers for text, binary contents, and images.

Content Viewers

Identifier:

org.eclipse.compare.contentViewers

Description:

This extension point allows a plug−in to register viewers for specific content types. These viewers are used in the `EditionSelectionDialog` when presenting an edition of a resource or a subsection thereof. The viewer is expected to extend `org.eclipse.jface.viewers.Viewer`. However since viewers don't have a default constructor the extension point must implement the factory interface for viewers `org.eclipse.compare.IViewerCreator`.

Configuration Markup:

<!ELEMENT extension (viewer* , contentTypeBinding*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT viewer EMPTY>

<!ATTLIST viewer

id CDATA #REQUIRED

class CDATA #REQUIRED

extensions CDATA #IMPLIED>

- **id –** a unique identifier that can be used to reference the viewer
- **class –** a fully qualified name of a class that implements a factory for the content viewer and implements `org.eclipse.compare.IViewerCreator`
- **extensions –** a comma separated list of file extensions e.g. "java, gif"

<!ELEMENT contentTypeBinding EMPTY>

<!ATTLIST contentTypeBinding

contentTypeId CDATA #REQUIRED

contentViewerId CDATA #REQUIRED>

- **contentTypeId –**
- **contentViewerId –**

Examples:

The following is an example of a viewer for text files (extension "txt"):

<extension point =

"org.eclipse.compare.contentViewers"

>

<viewer id=

"org.eclipse.compare.internal.TextViewer"

class=

"org.eclipse.compare.internal.TextViewerCreator"

extensions=

"txt"

/>

</extension>

API Information:

The contributed class must implement `org.eclipse.compare.IViewerCreator`

Supplied Implementation:

The Compare UI plugin defines content viewers for text and images.

Stream Merger

Identifier:

org.eclipse.compare.streamMergers

Since:

3.0

Description:

This extension point allows a plug−in to register a stream merger for specific content types. The stream merger is expected to perform a three−way merge on three input streams and writes the result to an output stream. The extension point must implement the interface `org.eclipse.compare.IStreamMerger`.

Configuration Markup:

<!ELEMENT extension (streamMerger* , contentTypeBinding*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point −** a fully qualified identifier of the target extension point
- **id −** an optional identifier of the extension instance
- **name −** an optional name of the extension instance

<!ELEMENT streamMerger EMPTY>

<!ATTLIST streamMerger

id CDATA #REQUIRED

class CDATA #REQUIRED

extensions CDATA #IMPLIED>

- **id −** a unique identifier that can be used to reference the stream merger
- **class −** a fully qualified name of a class that implements `org.eclipse.compare.IStreamMerger`
- **extensions −** a comma separated list of file extensions e.g. "java, properties"

<!ELEMENT contentTypeBinding EMPTY>

<!ATTLIST contentTypeBinding

contentTypeId CDATA #REQUIRED

streamMergerId CDATA #REQUIRED>

- **contentTypeId –**
- **streamMergerId –**

Examples:

The following is an example of a stream merger for property files (extension "properties"):

<extension point =

"org.eclipse.compare.streamMergers"

>

<streamMerger id=

"org.eclipse.compare.internal.merge.TextStreamMerger"

class=

"org.eclipse.compare.internal.merge.TextStreamMerger"

extensions=

"properties"

/>

</extension>

API Information:

The contributed class must implement `org.eclipse.compare.IStreamMerger`

Supplied Implementation:

The Compare UI plugin defines a stream merger for line oriented text files.

Structure Creators

Identifier:

org.eclipse.compare.structureCreators

Description:

This extension point allows a plug−in to register a structure creator for specific content types. The structure creator is expected to create a tree of `IStructureComparator`s for a given content. This tree is used as the input for the structural compare. The extension point must implement the interface `org.eclipse.compare.structuremergeviewer.IStructureCreator`.

Configuration Markup:

<!ELEMENT extension (structureCreator* , contentTypeBinding*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT structureCreator EMPTY>

<!ATTLIST structureCreator

id CDATA #REQUIRED

class CDATA #REQUIRED

extensions CDATA #IMPLIED>

- **id –** a unique identifier that can be used to reference the structure creator
- **class –** a fully qualified name of a class that implements `org.eclipse.compare.structuremergeviewer.IStructureCreator`
- **extensions –** a comma separated list of file extensions e.g. "java, properties"

<!ELEMENT contentTypeBinding EMPTY>

<!ATTLIST contentTypeBinding

contentTypeId CDATA #REQUIRED

structureCreatorId CDATA #REQUIRED>

- **contentTypeId –**
- **structureCreatorId –**

Examples:

The following is an example of a structure creator for java files (extension "java"):

<extension point =

"org.eclipse.compare.structureCreators"

>

<structureCreator id=

"org.eclipse.compare.JavaStructureCreator"

class=

"org.eclipse.compare.JavaStructureCreator"

extensions=

"java"

/>

</extension>

API Information:

The contributed class must implement
`org.eclipse.compare.structuremergeviewer.IStructureCreator`

Supplied Implementation:

The Compare UI plugin defines a structure creator for zip archives.

StructureMerge Viewers

Identifier:

org.eclipse.compare.structureMergeViewers

Description:

This extension point allows a plug−in to register compare/merge viewers for structural content types. The viewer is expected to extend `org.eclipse.jface.viewers.Viewer`. However, since viewers don't have a default constructor, the extension point must implement the factory interface for viewers `org.eclipse.compare.IViewerCreator`.

Configuration Markup:

<!ELEMENT extension (viewer* , contentTypeBinding*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT viewer EMPTY>

<!ATTLIST viewer

id CDATA #REQUIRED

class CDATA #REQUIRED

extensions CDATA #REQUIRED>

- **id –** a unique identifier that can be used to reference the viewer
- **class –** a fully qualified name of a class that implements a factory for the structure merge viewer and implements `org.eclipse.compare.IViewerCreator`
- **extensions –** a comma separated list of file extensions e.g. "zip, jar"

<!ELEMENT contentTypeBinding EMPTY>

<!ATTLIST contentTypeBinding

contentTypeId CDATA #REQUIRED

structureMergeViewerId CDATA #REQUIRED>

- **contentTypeId –**
- **structureMergeViewerId –**

Examples:

The following is an example of compare/merge viewer for zip files (extension "zip"):

<extension point =

"org.eclipse.compare.structureMergeViewers"

>

<viewer id=

"org.eclipse.compare.ZipCompareViewer"

class=

"org.eclipse.compare.ZipCompareViewerCreator"

extensions=

"zip"

/>

</extension>

API Information:

The contributed class must implement `org.eclipse.compare.IViewerCreator`

Supplied Implementation:

The Compare UI plugin defines a structure compare viewer for zip archives.

Property Testers

Identifier:

org.eclipse.core.expressions.propertyTesters

Since:

3.0

Description:

This extension point allows to add properties to an already existing type. Those properties can then be used inside the expression language's test expression element.

Configuration Markup:

<!ELEMENT extension (propertyTester*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT propertyTester EMPTY>

<!ATTLIST propertyTester

id CDATA #REQUIRED

type CDATA #REQUIRED

namespace CDATA #REQUIRED

properties CDATA #REQUIRED

class CDATA #REQUIRED>

- **id –** unique identifier for the property tester
- **type –** the type to be extended by this property tester
- **namespace –** a unique id determining the name space the properties are added to
- **properties –** a comma separated list of properties provided by this property tester

- **class** – the name of the class that implements the testing methods. The class must be public and extend `org.eclipse.core.expressions.PropertyTester` with a public 0–argument constructor.

Examples:

The following is an example of a property tester contribution:

<extension point=

"org.eclipse.core.expressions.propertyTesters"

>

<propertyTester id=

"org.eclipse.jdt.ui.IResourceTester"

type=

"org.eclipse.core.resources.IResource"

namespace=

"org.eclipse.jdt.ui"

properties=

"canDelete"

class=

"org.eclipse.jdt.ui.internal.ResourceTester"

>

</propertyTester>

</extension>

API Information:

The contributed class must extend `org.eclipse.core.expressions.PropertyTester`

Since:

Dynamic String Substitution Variables

Identifier:

org.eclipse.core.variables.dynamicVariables

Since:

3.0

Description:

This extension point provides a mechanism for defining dynamic variables used in string substitution. The value of a dynamic variable is resolved at the time a string substitution is performed, with an optional argument.

Configuration Markup:

<!ELEMENT extension (variable*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT variable EMPTY>

<!ATTLIST variable

name CDATA #REQUIRED

resolver CDATA #REQUIRED

description CDATA #REQUIRED

supportsArgument (true | false) >

- **name –** specifies a unique name for this variable.
- **resolver –** specifies a Java class which implements `org.eclipse.core.variables.IDynamicVariableResolver`, which is used to determine the value of the variable
- **description –** specifies a human–readable description of this variable

- **supportsArgument** – Whether this variable supports an argument. When unspecified, the implied value is `true`.

Examples:

The following is a definition of a dynamic variable that resolves to the name of the selected resource:

<extension point=

"org.eclipse.core.variables.dynamicVariables"

>

<variable name=

"resource_name"

expanderClass=

"com.example.ResourceNameExpander"

description=

"The name of the selected resource"

>

</variable>

</extension>

API Information:

Value of the attribute **resolver** must be a fully qualified name of a Java class that implements the interface **org.eclipse.core.variables.IDynamicVariableResolver**.

Value Variables

Identifier:

org.eclipse.core.variables.valueVariables

Since:

3.0

Description:

This extension point provides a mechanism for defining variables used for string substitution. A value variable has a static value.

Configuration Markup:

<!ELEMENT extension (variable*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance
- **name –** an optional name of the extension instance

<!ELEMENT variable EMPTY>

<!ATTLIST variable

name CDATA #REQUIRED

initialValue CDATA #IMPLIED

initializerClass CDATA #IMPLIED

description CDATA #IMPLIED>

- **name –** specifies a unique name for this variable.
- **initialValue –** specifies the initial value for this variable. When specified, an `initializerClass` attribute must not be specified.
- **initializerClass –** specifies the fully qualified name of the Java class that implements `org.eclipse.core.variables.IValueVariableInitializer`. When specified, an `initialValue` attribute must not be specified.

- **description** − specifies a human−readable description of this variable.

Examples:

The following is an example of a value variable contribution with an initial value:

```
<extension point=
"org.eclipse.core.variables.valueVariables"
>
<variable name=
"FOO_HOME"
initialValue=
"/usr/local/foo"
>
</variable>
</extension>
```

In the example above, the specified variable is created with the initial value "/usr/local/foo". The following is an example of a value variable contribution with an initializer class:

```
<extension point=
"org.eclipse.core.variables.valueVariables"
>
<variable name=
"FOO_HOME"
initializerClass=
"com.example.FooLocator"
>
```

</variable>

</extension>

In the example above, the variable FOO_HOME is created and the class "com.example.FooLocator" will be used to initialize the value the first time it's requested.

API Information:

Value of the attribute **initializerClass** must be a fully qualified name of a Java class that implements the interface **org.eclipse.core.variables.IValueVariableInitializer**.

Copy Participants

Identifier:

org.eclipse.ltk.core.refactoring.copyParticipants

Since:

3.1

Description:

This extension point is used to define refactoring copy participants. The reader of the expression provides the following predefined variables which can be referenced via the <with variable="..."> expression element:

- Object element: the element to be copied
- List<String> affectedNatures: a list containing the natures of the projects affected by the refactoring
- String processorId: the id of the refactoring processor that will own the participant.

The default variable used during expression evaluation is bound to the element variable.

Configuration Markup:

<!ELEMENT extension (copyParticipant*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point** – a fully qualified identifier of the target extension point
- **id** – an optional identifier of the extension instance
- **name** – an optional name of the extension instance

<!ELEMENT copyParticipant (enablement)>

<!ATTLIST copyParticipant

id CDATA #REQUIRED

name CDATA #REQUIRED

class CDATA #REQUIRED

processOnCancel CDATA #IMPLIED>

- **id** – unique identifier for the copy participant.
- **name** – a human reabable name of the copy participant
- **class** – the name of the class that provides the participant implementation.
- **processOnCancel** – if true the change created by the participant will be executed even if the overall change got canceled.

<!ELEMENT enablement (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

A generic root element. The element can be used inside an extension point to define its enablement expression. The children of an enablement expression are combined using the and operator.

<!ELEMENT not (not | and | or | instanceof | test | systemTest | equals | count | with | resolve | adapt | iterate)>

This element represent a NOT operation on the result of evaluating it's sub−element expression.

<!ELEMENT and (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

This element represent an AND operation on the result of evaluating all it's sub−elements expressions.

<!ELEMENT or (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

This element represent an OR operation on the result of evaluating all it's sub−element expressions.

<!ELEMENT instanceof EMPTY>

<!ATTLIST instanceof

value CDATA #REQUIRED>

This element is used to perform an instanceof check of the object in focus. The expression returns EvaluationResult.TRUE if the object's type is a sub type of the type specified by the attribute value. Otherwise EvaluationResult.FALSE is returned.

Since:

- **value –** a fully qualified name of a class or interface.

<!ELEMENT test EMPTY>

<!ATTLIST test

property CDATA #REQUIRED

args CDATA #IMPLIED

value CDATA #IMPLIED>

This element is used to evaluate the property state of the object in focus. The set of testable properties can be extended using the propery tester extension point. The test expression returns EvaluationResult.NOT_LOADED if teh property tester doing the actual testing isn't loaded yet.

- **property –** the name of an object's property to test.
- **args –** additional arguments passed to the property tester. Multiple arguments are seperated by commas. Each individual argument is converted into a Java base type using the same rules as defined for the value attribute of the test expression.
- **value –** the expected value of the property. Can be omitted if the property is a boolean property. The test expression is supposed to return EvaluationResult.TRUE if the property matches the value and EvaluationResult.FALSE otherwise. The value attribute is converted into a Java base type using the following rules:
  - the string "true" is converted into Boolean.TRUE
  - the string "false" is converted into Boolean.FALSE
  - if the string contains a dot then the interpreter tries to convert the value into a Float object. If this fails the string is treated as a java.lang.String
  - if the string only consists of numbers then the interpreter converts the value in an Integer object.
  - in all other cases the string is treated as a java.lang.String
  - the conversion of the string into a Boolean, Float, or Integer can be suppressed by surrounding the string with single quotes. For example, the attribute value="'true'" is converted into the string "true"

<!ELEMENT systemTest EMPTY>

<!ATTLIST systemTest

property CDATA #REQUIRED

value CDATA #REQUIRED>

Tests a system property by calling the System.getProperty method and compares the result with the value specified through the value attribute.

Since:

- **property** – the name of an system property to test.
- **value** – the expected value of the property. The value is interpreted as a string value.

<!ELEMENT equals EMPTY>

<!ATTLIST equals

value CDATA #REQUIRED>

This element is used to perform an equals check of the object in focus. The expression returns EvaluationResult.TRUE if the object is equal to the value provided by the attribute value. Otherwise EvaluationResult.FALSE is returned.

- **value** – the operatand of the equals tests. The value provided as a string is converted into a Java base type using the same rules as for the value attribute of the test expression.

<!ELEMENT count EMPTY>

<!ATTLIST count

value CDATA #REQUIRED>

This element is used to test the number of elements in a collection.

- **value** – an expression to specify the number of elements in a list. Following wildcard characters can be used:
  *
  : any number of elements
  ?
  : no elements or one element
  +
  : one or more elements
  !
  : no elements
  integer value
  : the list must contain the exact number of elements

<!ELEMENT with (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

Since:

<!ATTLIST with

variable CDATA #REQUIRED>

This element changes the object to be inspected for all its child element to the object referneced by the given variable. If the variable can not be resolved then the expression will throw a ExpressionException when evaluating it. The children of a with expression are combined using the and operator.

- **variable –** the name of the variable to be used for further inspection. It is up to the evaluator of an extension point to provide the variable in the variable pool.

<!ELEMENT resolve (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST resolve

variable CDATA #REQUIRED

args CDATA #IMPLIED>

This element changes the object to be inspected for all its child element to the object referneced by the given variable. If the variable can not be resolved then the expression will throw a ExpressionException when evaluating it. The children of a with expression are combined using the and operator.

- **variable –** the name of the variable to be resolved. This variable is then used as the object in focus for child element evaluation. It is up to the evaluator of an extension point to provide a corresponding variable resolver (see IVariableResolver) through the evaluation context passed to the root expression element when evaluating the expression.
- **args –** additional arguments passed to the variable resolver. Multiple arguments are seperated by commas. Each individual argument is converted into a Java base type using the same rules as defined for the value attribute of the test expression.

<!ELEMENT adapt (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST adapt

type CDATA #REQUIRED>

This element is used to adapt the object in focus to the type specified by the attribute type. The expression returns not loaded if either the adapter or the type referenced isn't loaded yet. It throws a ExpressionException during evaluation if the type name doesn't exist at all. The children of an adapt expression are combined using the and operator.

- **type** – the type to which the object in focus is to be adapted.

<!ELEMENT iterate (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST iterate

operator (or|and) >

This element is used to iterate over a variable that is of type java.util.Collection. If the object in focus is not of type java.util.Collection then an ExpressionException will be thrown while evaluating the expression.

- **operator** – either "and" or "or". The operator defines how the child elements will be combined. If not specified, "and" will be used.

Examples:

The example below defines a participant for copying files. The participant is enabled if one of the project affected by the refactoring has a Java nature.

<copyParticipant id=

"org.eclipse.samples.copyParticipant"

name=

"%CopyFileParticipant.name"

class=

"org.eclipse.samples.CopyFileParticipant"

>

<enablement>

<with variable=

"affectedNatures"

>

<iterate operator=

"or"

>

<equals value=

"org.eclipse.jdt.core.javanature"

/>

</iterate>

</with>

<with variable=

"element"

>

<instanceof value=

"org.eclipse.core.resources.IFile"

/>

</with>

</enablement>

</copyParticipant>

API Information:

The contributed class must extend
`org.eclipse.ltk.core.refactoring.participants.CopyParticipant`

Create Participants

Identifier:

org.eclipse.ltk.core.refactoring.createParticipants

Since:

3.0

Description:

This extension point is used to define refactoring create participants. The reader of the expression provides the following predefined variables which can be referenced via the <with variable="..."> expression element:

- Object element: the element to be create or a corresponding descriptor
- List<String> affectedNatures: a list containing the natures of the projects affected by the refactoring
- String processorId: the id of the refactoring processor that will own the participant.

The default variable used during expression evaluation is bound to the element variable.

Configuration Markup:

<!ELEMENT extension (createParticipant*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point** – a fully qualified identifier of the target extension point
- **id** – an optional identifier of the extension instance
- **name** – an optional name of the extension instance

<!ELEMENT createParticipant (enablement)>

<!ATTLIST createParticipant

id CDATA #REQUIRED

name CDATA #REQUIRED

class CDATA #REQUIRED>

- **id** – unique identifier for the create participant.
- **name** – a human reabable name of the create participant

- **class** – the name of the class that provides the participant implementation.

<!ELEMENT enablement (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

A generic root element. The element can be used inside an extension point to define its enablement expression. The children of an enablement expression are combined using the and operator.

<!ELEMENT not (not | and | or | instanceof | test | systemTest | equals | count | with | resolve | adapt | iterate)>

This element represent a NOT operation on the result of evaluating it's sub−element expression.

<!ELEMENT and (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

This element represent an AND operation on the result of evaluating all it's sub−elements expressions.

<!ELEMENT or (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

This element represent an OR operation on the result of evaluating all it's sub−element expressions.

<!ELEMENT instanceof EMPTY>

<!ATTLIST instanceof

value CDATA #REQUIRED>

This element is used to perform an instanceof check of the object in focus. The expression returns EvaluationResult.TRUE if the object's type is a sub type of the type specified by the attribute value. Otherwise EvaluationResult.FALSE is returned.

- **value** – a fully qualified name of a class or interface.

<!ELEMENT test EMPTY>

<!ATTLIST test

property CDATA #REQUIRED

args CDATA #IMPLIED

value CDATA #IMPLIED>

This element is used to evaluate the property state of the object in focus. The set of testable properties can be extended using the propery tester extension point. The test expression returns EvaluationResult.NOT_LOADED if teh property tester doing the actual testing isn't loaded yet.

- **property** – the name of an object's property to test.
- **args** – additional arguments passed to the property tester. Multiple arguments are seperated by commas. Each individual argument is converted into a Java base type using the same rules as defined for the value attribute of the test expression.
- **value** – the expected value of the property. Can be omitted if the property is a boolean property. The test expression is supposed to return EvaluationResult.TRUE if the property matches the value and EvaluationResult.FALSE otherwise. The value attribute is converted into a Java base type using the following rules:
  - the string "true" is converted into Boolean.TRUE
  - the string "false" is converted into Boolean.FALSE
  - if the string contains a dot then the interpreter tries to convert the value into a Float object. If this fails the string is treated as a java.lang.String
  - if the string only consists of numbers then the interpreter converts the value in an Integer object.
  - in all other cases the string is treated as a java.lang.String
  - the conversion of the string into a Boolean, Float, or Integer can be suppressed by surrounding the string with single quotes. For example, the attribute value="'true'" is converted into the string "true"

<!ELEMENT systemTest EMPTY>

<!ATTLIST systemTest

property CDATA #REQUIRED

value CDATA #REQUIRED>

Tests a system property by calling the System.getProperty method and compares the result with the value specified through the value attribute.

- **property** – the name of an system property to test.

- **value** – the expected value of the property. The value is interpreted as a string value.

<!ELEMENT equals EMPTY>

<!ATTLIST equals

value CDATA #REQUIRED>

This element is used to perform an equals check of the object in focus. The expression returns EvaluationResult.TRUE if the object is equal to the value provided by the attribute value. Otherwise EvaluationResult.FALSE is returned.

- **value** – the operatand of the equals tests. The value provided as a string is converted into a Java base type using the same rules as for the value attribute of the test expression.

<!ELEMENT count EMPTY>

<!ATTLIST count

value CDATA #REQUIRED>

This element is used to test the number of elements in a collection.

- **value** – an expression to specify the number of elements in a list. Following wildcard characters can be used:
  *
  any number of elements
  ?
  no elements or one element
  +
  one or more elements
  !
  no elements
  integer value
  the list must contain the exact number of elements

<!ELEMENT with (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST with

variable CDATA #REQUIRED>

This element changes the object to be inspected for all its child element to the object referneced by the given variable. If the variable can not be resolved then the expression will throw a ExpressionException when evaluating it. The children of a with expression are combined using the and operator.

- **variable –** the name of the variable to be used for further inspection. It is up to the evaluator of an extension point to provide the variable in the variable pool.

<!ELEMENT resolve (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST resolve

variable CDATA #REQUIRED

args CDATA #IMPLIED>

This element changes the object to be inspected for all its child element to the object referneced by the given variable. If the variable can not be resolved then the expression will throw a ExpressionException when evaluating it. The children of a with expression are combined using the and operator.

- **variable –** the name of the variable to be resolved. This variable is then used as the object in focus for child element evaluation. It is up to the evaluator of an extension point to provide a corresponding variable resolver (see IVariableResolver) through the evaluation context passed to the root expression element when evaluating the expression.
- **args –** additional arguments passed to the variable resolver. Multiple arguments are seperated by commas. Each individual argument is converted into a Java base type using the same rules as defined for the value attribute of the test expression.

<!ELEMENT adapt (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST adapt

type CDATA #REQUIRED>

This element is used to adapt the object in focus to the type specified by the attribute type. The expression returns not loaded if either the adapter or the type referenced isn't loaded yet. It throws a ExpressionException during evaluation if the type name doesn't exist at all. The children of an adapt expression are combined using the and operator.

- **type** – the type to which the object in focus is to be adapted.

<!ELEMENT iterate (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST iterate

operator (orland) >

This element is used to iterate over a variable that is of type java.util.Collection. If the object in focus is not of type java.util.Collection then an ExpressionException will be thrown while evaluating the expression.

- **operator** – either "and" or "or". The operator defines how the child elements will be combined. If not specified, "and" will be used.

Examples:

The example below defines a create participant. The participant is enabled if one of the project affected by the refactoring has a Java nature and when the element to be created is a folder.

<createParticipant id=

"org.myCompany.createParticipant"

name=

"%CreateParticipant.name"

class=

"org.myCompany.CreateParticipant"

>

<enablement>

<with variable=

"affectedNatures"

>

<iterate operator=

"or"

>

<equals value=

"org.eclipse.jdt.core.javanature"

/>

</iterate>

</with>

<with variable=

"element"

>

<instanceof value=

"org.eclipse.core.resources.IFolder"

/>

</with>

</enablement>

</createParticipant>

API Information:

The contributed class must extend
`org.eclipse.ltk.core.refactoring.participants.CreateParticipant`

Delete Participants

Identifier:

org.eclipse.ltk.core.refactoring.deleteParticipants

Since:

3.0

Description:

This extension point is used to define refactoring delete participants. The reader of the expression provides the following predefined variables which can be referenced via the <with variable="..."> expression element:

- Object element: the element to be deleted
- List<String> affectedNatures: a list containing the natures of the projects affected by the refactoring
- String processorId: the id of the refactoring processor that will own the participant.

The default variable used during expression evaluation is bound to the element variable.

Configuration Markup:

<!ELEMENT extension (deleteParticipant*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point** – a fully qualified identifier of the target extension point
- **id** – an optional identifier of the extension instance
- **name** – an optional name of the extension instance

<!ELEMENT deleteParticipant (enablement)>

<!ATTLIST deleteParticipant

id CDATA #REQUIRED

name CDATA #REQUIRED

class CDATA #REQUIRED>

- **id** – unique identifier for the delete participant.
- **name** – a human reabable name of the delete participant

- **class** – the name of the class that provides the participant implementation.

<!ELEMENT enablement (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

A generic root element. The element can be used inside an extension point to define its enablement expression. The children of an enablement expression are combined using the and operator.

<!ELEMENT not (not | and | or | instanceof | test | systemTest | equals | count | with | resolve | adapt | iterate)>

This element represent a NOT operation on the result of evaluating it's sub−element expression.

<!ELEMENT and (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

This element represent an AND operation on the result of evaluating all it's sub−elements expressions.

<!ELEMENT or (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

This element represent an OR operation on the result of evaluating all it's sub−element expressions.

<!ELEMENT instanceof EMPTY>

<!ATTLIST instanceof

value CDATA #REQUIRED>

This element is used to perform an instanceof check of the object in focus. The expression returns EvaluationResult.TRUE if the object's type is a sub type of the type specified by the attribute value. Otherwise EvaluationResult.FALSE is returned.

- **value** – a fully qualified name of a class or interface.

<!ELEMENT test EMPTY>

<!ATTLIST test

property CDATA #REQUIRED

args CDATA #IMPLIED

value CDATA #IMPLIED>

This element is used to evaluate the property state of the object in focus. The set of testable properties can be extended using the propery tester extension point. The test expression returns EvaluationResult.NOT_LOADED if teh property tester doing the actual testing isn't loaded yet.

- **property** – the name of an object's property to test.
- **args** – additional arguments passed to the property tester. Multiple arguments are seperated by commas. Each individual argument is converted into a Java base type using the same rules as defined for the value attribute of the test expression.
- **value** – the expected value of the property. Can be omitted if the property is a boolean property. The test expression is supposed to return EvaluationResult.TRUE if the property matches the value and EvaluationResult.FALSE otherwise. The value attribute is converted into a Java base type using the following rules:
  - the string "true" is converted into Boolean.TRUE
  - the string "false" is converted into Boolean.FALSE
  - if the string contains a dot then the interpreter tries to convert the value into a Float object. If this fails the string is treated as a java.lang.String
  - if the string only consists of numbers then the interpreter converts the value in an Integer object.
  - in all other cases the string is treated as a java.lang.String
  - the conversion of the string into a Boolean, Float, or Integer can be suppressed by surrounding the string with single quotes. For example, the attribute value="'true'" is converted into the string "true"

<!ELEMENT systemTest EMPTY>

<!ATTLIST systemTest

property CDATA #REQUIRED

value CDATA #REQUIRED>

Tests a system property by calling the System.getProperty method and compares the result with the value specified through the value attribute.

- **property** – the name of an system property to test.

- **value** – the expected value of the property. The value is interpreted as a string value.

<!ELEMENT equals EMPTY>

<!ATTLIST equals

value CDATA #REQUIRED>

This element is used to perform an equals check of the object in focus. The expression returns EvaluationResult.TRUE if the object is equal to the value provided by the attribute value. Otherwise EvaluationResult.FALSE is returned.

- **value** – the operatand of the equals tests. The value provided as a string is converted into a Java base type using the same rules as for the value attribute of the test expression.

<!ELEMENT count EMPTY>

<!ATTLIST count

value CDATA #REQUIRED>

This element is used to test the number of elements in a collection.

- **value** – an expression to specify the number of elements in a list. Following wildcard characters can be used:
  *
  any number of elements
  ?
  no elements or one element
  +
  one or more elements
  !
  no elements
  integer value
  the list must contain the exact number of elements

<!ELEMENT with (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST with

variable CDATA #REQUIRED>

This element changes the object to be inspected for all its child element to the object referneced by the given variable. If the variable can not be resolved then the expression will throw a ExpressionException when evaluating it. The children of a with expression are combined using the and operator.

- **variable –** the name of the variable to be used for further inspection. It is up to the evaluator of an extension point to provide the variable in the variable pool.

<!ELEMENT resolve (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST resolve

variable CDATA #REQUIRED

args CDATA #IMPLIED>

This element changes the object to be inspected for all its child element to the object referneced by the given variable. If the variable can not be resolved then the expression will throw a ExpressionException when evaluating it. The children of a with expression are combined using the and operator.

- **variable –** the name of the variable to be resolved. This variable is then used as the object in focus for child element evaluation. It is up to the evaluator of an extension point to provide a corresponding variable resolver (see IVariableResolver) through the evaluation context passed to the root expression element when evaluating the expression.
- **args –** additional arguments passed to the variable resolver. Multiple arguments are seperated by commas. Each individual argument is converted into a Java base type using the same rules as defined for the value attribute of the test expression.

<!ELEMENT adapt (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST adapt

type CDATA #REQUIRED>

This element is used to adapt the object in focus to the type specified by the attribute type. The expression returns not loaded if either the adapter or the type referenced isn't loaded yet. It throws a ExpressionException during evaluation if the type name doesn't exist at all. The children of an adapt expression are combined using the and operator.

- **type** – the type to which the object in focus is to be adapted.

<!ELEMENT iterate (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST iterate

operator (orland) >

This element is used to iterate over a variable that is of type java.util.Collection. If the object in focus is not of type java.util.Collection then an ExpressionException will be thrown while evaluating the expression.

- **operator** – either "and" or "or". The operator defines how the child elements will be combined. If not specified, "and" will be used.

Examples:

The example below defines a delete participant. The participant is enabled if one of the project affected by the refactoring has a Java nature and when the element to be deleted is of type ICompilationUnit.

<deleteParticipant id=

"org.myCompany.deleteParticipant"

name=

"%DeleteParticipant.name"

class=

"org.myCompany.Participant"

>

<enablement>

<with variable=

"affectedNatures"

>

<iterate operator=

"or"

>

<equals value=

"org.eclipse.jdt.core.javanature"

/>

</iterate>

</with>

<with variable=

"element"

>

<instanceof value=

"org.eclipse.jdt.core.ICompilationUnit"

/>

</with>

</enablement>

</deleteParticipant>

API Information:

The contributed class must extend
`org.eclipse.ltk.core.refactoring.participants.DeleteParticipant`

Move Participants

Identifier:

org.eclipse.ltk.core.refactoring.moveParticipants

Since:

3.0

Description:

This extension point is used to define refactoring move participants. The reader of the expression provides the following predefined variables which can be referenced via the <with variable="..."> expression element:

- Object element: the element to be moved
- List<String> affectedNatures: a list containing the natures of the projects affected by the refactoring
- String processorId: the id of the refactoring processor that will own the participant.

The default variable used during expression evaluation is bound to the element variable.

Configuration Markup:

<!ELEMENT extension (moveParticipant*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point** – a fully qualified identifier of the target extension point
- **id** – an optional identifier of the extension instance
- **name** – an optional name of the extension instance

<!ELEMENT moveParticipant (enablement)>

<!ATTLIST moveParticipant

id CDATA #REQUIRED

name CDATA #REQUIRED

class CDATA #REQUIRED>

- **id** – unique identifier for the move participant.
- **name** – a human reabable name of the move participant

- **class** – the name of the class that provides the participant implementation.

<!ELEMENT enablement (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

A generic root element. The element can be used inside an extension point to define its enablement expression. The children of an enablement expression are combined using the and operator.

<!ELEMENT not (not | and | or | instanceof | test | systemTest | equals | count | with | resolve | adapt | iterate)>

This element represent a NOT operation on the result of evaluating it's sub−element expression.

<!ELEMENT and (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

This element represent an AND operation on the result of evaluating all it's sub−elements expressions.

<!ELEMENT or (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

This element represent an OR operation on the result of evaluating all it's sub−element expressions.

<!ELEMENT instanceof EMPTY>

<!ATTLIST instanceof

value CDATA #REQUIRED>

This element is used to perform an instanceof check of the object in focus. The expression returns EvaluationResult.TRUE if the object's type is a sub type of the type specified by the attribute value. Otherwise EvaluationResult.FALSE is returned.

- **value** – a fully qualified name of a class or interface.

<!ELEMENT test EMPTY>

<!ATTLIST test

property CDATA #REQUIRED

args CDATA #IMPLIED

value CDATA #IMPLIED>

This element is used to evaluate the property state of the object in focus. The set of testable properties can be extended using the propery tester extension point. The test expression returns EvaluationResult.NOT_LOADED if teh property tester doing the actual testing isn't loaded yet.

- **property** – the name of an object's property to test.
- **args** – additional arguments passed to the property tester. Multiple arguments are seperated by commas. Each individual argument is converted into a Java base type using the same rules as defined for the value attribute of the test expression.
- **value** – the expected value of the property. Can be omitted if the property is a boolean property. The test expression is supposed to return EvaluationResult.TRUE if the property matches the value and EvaluationResult.FALSE otherwise. The value attribute is converted into a Java base type using the following rules:
  - the string "true" is converted into Boolean.TRUE
  - the string "false" is converted into Boolean.FALSE
  - if the string contains a dot then the interpreter tries to convert the value into a Float object. If this fails the string is treated as a java.lang.String
  - if the string only consists of numbers then the interpreter converts the value in an Integer object.
  - in all other cases the string is treated as a java.lang.String
  - the conversion of the string into a Boolean, Float, or Integer can be suppressed by surrounding the string with single quotes. For example, the attribute value="'true'" is converted into the string "true"

<!ELEMENT systemTest EMPTY>

<!ATTLIST systemTest

property CDATA #REQUIRED

value CDATA #REQUIRED>

Tests a system property by calling the System.getProperty method and compares the result with the value specified through the value attribute.

- **property** – the name of an system property to test.

- **value** – the expected value of the property. The value is interpreted as a string value.

<!ELEMENT equals EMPTY>

<!ATTLIST equals

value CDATA #REQUIRED>

This element is used to perform an equals check of the object in focus. The expression returns EvaluationResult.TRUE if the object is equal to the value provided by the attribute value. Otherwise EvaluationResult.FALSE is returned.

- **value** – the operatand of the equals tests. The value provided as a string is converted into a Java base type using the same rules as for the value attribute of the test expression.

<!ELEMENT count EMPTY>

<!ATTLIST count

value CDATA #REQUIRED>

This element is used to test the number of elements in a collection.

- **value** – an expression to specify the number of elements in a list. Following wildcard characters can be used:
  *
  : any number of elements
  ?
  : no elements or one element
  +
  : one or more elements
  !
  : no elements
  integer value
  : the list must contain the exact number of elements

<!ELEMENT with (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST with

variable CDATA #REQUIRED>

This element changes the object to be inspected for all its child element to the object referneced by the given variable. If the variable can not be resolved then the expression will throw a ExpressionException when evaluating it. The children of a with expression are combined using the and operator.

- **variable –** the name of the variable to be used for further inspection. It is up to the evaluator of an extension point to provide the variable in the variable pool.

<!ELEMENT resolve (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST resolve

variable CDATA #REQUIRED

args CDATA #IMPLIED>

This element changes the object to be inspected for all its child element to the object referneced by the given variable. If the variable can not be resolved then the expression will throw a ExpressionException when evaluating it. The children of a with expression are combined using the and operator.

- **variable –** the name of the variable to be resolved. This variable is then used as the object in focus for child element evaluation. It is up to the evaluator of an extension point to provide a corresponding variable resolver (see IVariableResolver) through the evaluation context passed to the root expression element when evaluating the expression.
- **args –** additional arguments passed to the variable resolver. Multiple arguments are seperated by commas. Each individual argument is converted into a Java base type using the same rules as defined for the value attribute of the test expression.

<!ELEMENT adapt (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST adapt

type CDATA #REQUIRED>

This element is used to adapt the object in focus to the type specified by the attribute type. The expression returns not loaded if either the adapter or the type referenced isn't loaded yet. It throws a ExpressionException during evaluation if the type name doesn't exist at all. The children of an adapt expression are combined using the and operator.

- **type** – the type to which the object in focus is to be adapted.

<!ELEMENT iterate (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST iterate

operator (orland) >

This element is used to iterate over a variable that is of type java.util.Collection. If the object in focus is not of type java.util.Collection then an ExpressionException will be thrown while evaluating the expression.

- **operator** – either "and" or "or". The operator defines how the child elements will be combined. If not specified, "and" will be used.

Examples:

The example below defines a type move participant. The participant is enabled if one of the project affected by the refactoring has a Java nature and when the type to be moved is a JUnit test.

<moveParticipant id=

"org.eclipse.jdt.junit.moveTypeParticipant"

name=

"%MoveTypeParticipant.name"

class=

"org.eclipse.jdt.internal.junit.ui.TypeMoveParticipant"

>

<enablement>

<with variable=

"affectedNatures"

>

<iterate operator=

"or"

>

<equals value=

"org.eclipse.jdt.core.javanature"

/>

</iterate>

</with>

<with variable=

"element"

>

<instanceof value=

"org.eclipse.jdt.core.IType"

/>

<test property=

"org.eclipse.jdt.junit.isTest"

/>

</with>

</enablement>

</moveParticipant>

API Information:

The contributed class must extend
`org.eclipse.ltk.core.refactoring.participants.MoveParticipant`

Rename Participants

Identifier:

org.eclipse.ltk.core.refactoring.renameParticipants

Since:

3.0

Description:

This extension point is used to define refactoring rename participants. The reader of the expression provides the following predefined variables which can be referenced via the <with variable="..."> expression element:

- Object element: the element to be renamed
- List<String> affectedNatures: a list containing the natures of the projects affected by the refactoring
- String processorId: the id of the refactoring processor that will own the participant.

The default variable used during expression evaluation is bound to the element variable.

Configuration Markup:

<!ELEMENT extension (renameParticipant*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point** – a fully qualified identifier of the target extension point
- **id** – an optional identifier of the extension instance
- **name** – an optional name of the extension instance

<!ELEMENT renameParticipant (enablement)>

<!ATTLIST renameParticipant

id CDATA #REQUIRED

name CDATA #REQUIRED

class CDATA #REQUIRED>

- **id** – unique identifier for the rename participant.
- **name** – a human reabable name of the rename participant

- **class** – the name of the class that provides the participant implementation.

<!ELEMENT enablement (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

A generic root element. The element can be used inside an extension point to define its enablement expression. The children of an enablement expression are combined using the and operator.

<!ELEMENT not (not | and | or | instanceof | test | systemTest | equals | count | with | resolve | adapt | iterate)>

This element represent a NOT operation on the result of evaluating it's sub−element expression.

<!ELEMENT and (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

This element represent an AND operation on the result of evaluating all it's sub−elements expressions.

<!ELEMENT or (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

This element represent an OR operation on the result of evaluating all it's sub−element expressions.

<!ELEMENT instanceof EMPTY>

<!ATTLIST instanceof

value CDATA #REQUIRED>

This element is used to perform an instanceof check of the object in focus. The expression returns EvaluationResult.TRUE if the object's type is a sub type of the type specified by the attribute value. Otherwise EvaluationResult.FALSE is returned.

- **value** – a fully qualified name of a class or interface.

<!ELEMENT test EMPTY>

<!ATTLIST test

property CDATA #REQUIRED

args CDATA #IMPLIED

value CDATA #IMPLIED>

This element is used to evaluate the property state of the object in focus. The set of testable properties can be extended using the propery tester extension point. The test expression returns EvaluationResult.NOT_LOADED if teh property tester doing the actual testing isn't loaded yet.

- **property** – the name of an object's property to test.
- **args** – additional arguments passed to the property tester. Multiple arguments are seperated by commas. Each individual argument is converted into a Java base type using the same rules as defined for the value attribute of the test expression.
- **value** – the expected value of the property. Can be omitted if the property is a boolean property. The test expression is supposed to return EvaluationResult.TRUE if the property matches the value and EvaluationResult.FALSE otherwise. The value attribute is converted into a Java base type using the following rules:
  - the string "true" is converted into Boolean.TRUE
  - the string "false" is converted into Boolean.FALSE
  - if the string contains a dot then the interpreter tries to convert the value into a Float object. If this fails the string is treated as a java.lang.String
  - if the string only consists of numbers then the interpreter converts the value in an Integer object.
  - in all other cases the string is treated as a java.lang.String
  - the conversion of the string into a Boolean, Float, or Integer can be suppressed by surrounding the string with single quotes. For example, the attribute value="'true'" is converted into the string "true"

<!ELEMENT systemTest EMPTY>

<!ATTLIST systemTest

property CDATA #REQUIRED

value CDATA #REQUIRED>

Tests a system property by calling the System.getProperty method and compares the result with the value specified through the value attribute.

- **property** – the name of an system property to test.

- **value** – the expected value of the property. The value is interpreted as a string value.

<!ELEMENT equals EMPTY>

<!ATTLIST equals

value CDATA #REQUIRED>

This element is used to perform an equals check of the object in focus. The expression returns EvaluationResult.TRUE if the object is equal to the value provided by the attribute value. Otherwise EvaluationResult.FALSE is returned.

- **value** – the operatand of the equals tests. The value provided as a string is converted into a Java base type using the same rules as for the value attribute of the test expression.

<!ELEMENT count EMPTY>

<!ATTLIST count

value CDATA #REQUIRED>

This element is used to test the number of elements in a collection.

- **value** – an expression to specify the number of elements in a list. Following wildcard characters can be used:
  *
  : any number of elements
  ?
  : no elements or one element
  +
  : one or more elements
  !
  : no elements
  integer value
  : the list must contain the exact number of elements

<!ELEMENT with (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST with

variable CDATA #REQUIRED>

This element changes the object to be inspected for all its child element to the object referneced by the given variable. If the variable can not be resolved then the expression will throw a ExpressionException when evaluating it. The children of a with expression are combined using the and operator.

- **variable –** the name of the variable to be used for further inspection. It is up to the evaluator of an extension point to provide the variable in the variable pool.

<!ELEMENT resolve (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST resolve

variable CDATA #REQUIRED

args CDATA #IMPLIED>

This element changes the object to be inspected for all its child element to the object referneced by the given variable. If the variable can not be resolved then the expression will throw a ExpressionException when evaluating it. The children of a with expression are combined using the and operator.

- **variable –** the name of the variable to be resolved. This variable is then used as the object in focus for child element evaluation. It is up to the evaluator of an extension point to provide a corresponding variable resolver (see IVariableResolver) through the evaluation context passed to the root expression element when evaluating the expression.
- **args –** additional arguments passed to the variable resolver. Multiple arguments are seperated by commas. Each individual argument is converted into a Java base type using the same rules as defined for the value attribute of the test expression.

<!ELEMENT adapt (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST adapt

type CDATA #REQUIRED>

This element is used to adapt the object in focus to the type specified by the attribute type. The expression returns not loaded if either the adapter or the type referenced isn't loaded yet. It throws a ExpressionException during evaluation if the type name doesn't exist at all. The children of an adapt expression are combined using the and operator.

- **type** – the type to which the object in focus is to be adapted.

<!ELEMENT iterate (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST iterate

operator (or|and) >

This element is used to iterate over a variable that is of type java.util.Collection. If the object in focus is not of type java.util.Collection then an ExpressionException will be thrown while evaluating the expression.

- **operator** – either "and" or "or". The operator defines how the child elements will be combined. If not specified, "and" will be used.

Examples:

The example below defines a rename participant that participates in a type rename. The participant is enabled if one of the project affected by the refactoring has a Java nature and when the type to be renamed is a JUnit test.

<renameParticipant id=

"org.eclipse.jdt.junit.renameTypeParticipant"

name=

"%RenameTypeParticipant.name"

class=

"org.eclipse.jdt.internal.junit.ui.TypeRenameParticipant"

>

<enablement>

<with variable=

"affectedNatures"

>

<iterate operator=

"or"

>

<equals value=

"org.eclipse.jdt.core.javanature"

/>

</iterate>

</with>

<with variable=

"element"

>

<instanceof value=

"org.eclipse.jdt.core.IType"

/>

<test property=

"org.eclipse.jdt.junit.isTest"

/>

</with>

</enablement>

</renameParticipant>

API Information:

The contributed class must extend
`org.eclipse.ltk.core.refactoring.participants.RenameParticipant`

Refactoring Change Preview Viewers

Identifier:

org.eclipse.ltk.ui.refactoring.changePreviewViewers

Since:

3.0

Description:

This extension point is used to define a special viewer capable to present change objects. The reader of the extension point provides the following predefined variables which can be accessed during expression evaluation using the <with variable="..."/> tag:

change
the change object to present a preview for

The default variable used during expression evaluation is bound to the change variable.

Configuration Markup:

<!ELEMENT extension (changePreviewViewer*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point** – a fully qualified identifier of the target extension point.
- **id** – an optional identifier of the extension instance.
- **name** – an optional name of the extension instance.

<!ELEMENT changePreviewViewer (enablement)>

<!ATTLIST changePreviewViewer

id CDATA #REQUIRED

class CDATA #REQUIRED>

- **id** – unique identifier for the change preview viewer.
- **class** – the name of the class that provides the implementation.

<!ELEMENT enablement (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

A generic root element. The element can be used inside an extension point to define its enablement expression. The children of an enablement expression are combined using the and operator.

<!ELEMENT not (not | and | or | instanceof | test | systemTest | equals | count | with | resolve | adapt | iterate)>

This element represent a NOT operation on the result of evaluating it's sub−element expression.

<!ELEMENT and (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

This element represent an AND operation on the result of evaluating all it's sub−elements expressions.

<!ELEMENT or (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

This element represent an OR operation on the result of evaluating all it's sub−element expressions.

<!ELEMENT instanceof EMPTY>

<!ATTLIST instanceof

value CDATA #REQUIRED>

This element is used to perform an instanceof check of the object in focus. The expression returns EvaluationResult.TRUE if the object's type is a sub type of the type specified by the attribute value. Otherwise EvaluationResult.FALSE is returned.

- **value** − a fully qualified name of a class or interface.

<!ELEMENT test EMPTY>

<!ATTLIST test

property CDATA #REQUIRED

args CDATA #IMPLIED

value CDATA #IMPLIED>

This element is used to evaluate the property state of the object in focus. The set of testable properties can be extended using the propery tester extension point. The test expression returns EvaluationResult.NOT_LOADED if teh property tester doing the actual testing isn't loaded yet.

- **property** – the name of an object's property to test.
- **args** – additional arguments passed to the property tester. Multiple arguments are seperated by commas. Each individual argument is converted into a Java base type using the same rules as defined for the value attribute of the test expression.
- **value** – the expected value of the property. Can be omitted if the property is a boolean property. The test expression is supposed to return EvaluationResult.TRUE if the property matches the value and EvaluationResult.FALSE otherwise. The value attribute is converted into a Java base type using the following rules:
  - the string "true" is converted into Boolean.TRUE
  - the string "false" is converted into Boolean.FALSE
  - if the string contains a dot then the interpreter tries to convert the value into a Float object. If this fails the string is treated as a java.lang.String
  - if the string only consists of numbers then the interpreter converts the value in an Integer object.
  - in all other cases the string is treated as a java.lang.String
  - the conversion of the string into a Boolean, Float, or Integer can be suppressed by surrounding the string with single quotes. For example, the attribute value="'true'" is converted into the string "true"

<!ELEMENT systemTest EMPTY>

<!ATTLIST systemTest

property CDATA #REQUIRED

value CDATA #REQUIRED>

Tests a system property by calling the System.getProperty method and compares the result with the value specified through the value attribute.

- **property** – the name of an system property to test.
- **value** – the expected value of the property. The value is interpreted as a string value.

<!ELEMENT equals EMPTY>

<!ATTLIST equals

value CDATA #REQUIRED>

This element is used to perform an equals check of the object in focus. The expression returns EvaluationResult.TRUE if the object is equal to the value provided by the attribute value. Otherwise EvaluationResult.FALSE is returned.

- **value** – the operatand of the equals tests. The value provided as a string is converted into a Java base type using the same rules as for the value attribute of the test expression.

<!ELEMENT count EMPTY>

<!ATTLIST count

value CDATA #REQUIRED>

This element is used to test the number of elements in a collection.

- **value** – an expression to specify the number of elements in a list. Following wildcard characters can be used:
  *
  : any number of elements
  ?
  : no elements or one element
  +
  : one or more elements
  !
  : no elements
  integer value
  : the list must contain the exact number of elements

<!ELEMENT with (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST with

variable CDATA #REQUIRED>

This element changes the object to be inspected for all its child element to the object referneced by the given variable. If the variable can not be resolved then the expression will throw a ExpressionException when

evaluating it. The children of a with expression are combined using the and operator.

- **variable –** the name of the variable to be used for further inspection. It is up to the evaluator of an extension point to provide the variable in the variable pool.

<!ELEMENT resolve (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST resolve

variable CDATA #REQUIRED

args CDATA #IMPLIED>

This element changes the object to be inspected for all its child element to the object referneced by the given variable. If the variable can not be resolved then the expression will throw a ExpressionException when evaluating it. The children of a with expression are combined using the and operator.

- **variable –** the name of the variable to be resolved. This variable is then used as the object in focus for child element evaluation. It is up to the evaluator of an extension point to provide a corresponding variable resolver (see IVariableResolver) through the evaluation context passed to the root expression element when evaluating the expression.
- **args –** additional arguments passed to the variable resolver. Multiple arguments are seperated by commas. Each individual argument is converted into a Java base type using the same rules as defined for the value attribute of the test expression.

<!ELEMENT adapt (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST adapt

type CDATA #REQUIRED>

This element is used to adapt the object in focus to the type specified by the attribute type. The expression returns not loaded if either the adapter or the type referenced isn't loaded yet. It throws a ExpressionException during evaluation if the type name doesn't exist at all. The children of an adapt expression are combined using the and operator.

- **type –** the type to which the object in focus is to be adapted.

<!ELEMENT iterate (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST iterate

operator (or|and) >

This element is used to iterate over a variable that is of type java.util.Collection. If the object in focus is not of type java.util.Collection then an ExpressionException will be thrown while evaluating the expression.

- **operator –** either "and" or "or". The operator defines how the child elements will be combined. If not specified, "and" will be used.

Examples:

The example below contributes a preview viewer for text change objects.

<extension point=

"org.eclipse.ltk.ui.refactoring.changePreviewViewers"

>

<changePreviewViewer class=

"org.eclipse.ltk.internal.ui.refactoring.TextChangePreviewViewer"

id=

"org.eclipse.ltk.internal.ui.refactoring.textChangePreviewViewer"

>

<enablement>

<instanceof value=

"org.eclipse.ltk.core.refactoring.TextChange"

/>

</enablement>

</changePreviewViewer>

Since:

</extension>

API Information:

The contributed class must extend
`org.eclipse.ltk.ui.refactoring.IChangePreviewViewer`

Refactoring Status Context Viewers

Identifier:

org.eclipse.ltk.ui.refactoring.statusContextViewers

Since:

3.0

Description:

This extension point is used to define a special viewer capable to present the context of a refactoring status entry to the user. The reader of the extension point provides the following predefined variables which can be accessed during expression evaluation using the <with variable="..."/> tag:

context
: the context object managed by the refactoring status entry that is to be presented in the user interface.

Variables can be accessed using the <with variable="..."/> expression. The default variable used during expression evaluation is bound to the context variable.

Configuration Markup:

<!ELEMENT extension (statusContextViewer*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point.
- **id –** an optional identifier of the extension instance.
- **name –** an optional name of the extension instance.

<!ELEMENT statusContextViewer (enablement)>

<!ATTLIST statusContextViewer

id CDATA #REQUIRED

class CDATA #REQUIRED>

- **id –** unique identifier for the status context viewer.
- **class –** the name of the class that provides the implementation.

<!ELEMENT enablement (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

A generic root element. The element can be used inside an extension point to define its enablement expression. The children of an enablement expression are combined using the and operator.

<!ELEMENT not (not | and | or | instanceof | test | systemTest | equals | count | with | resolve | adapt | iterate)>

This element represent a NOT operation on the result of evaluating it's sub−element expression.

<!ELEMENT and (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

This element represent an AND operation on the result of evaluating all it's sub−elements expressions.

<!ELEMENT or (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

This element represent an OR operation on the result of evaluating all it's sub−element expressions.

<!ELEMENT instanceof EMPTY>

<!ATTLIST instanceof

value CDATA #REQUIRED>

This element is used to perform an instanceof check of the object in focus. The expression returns EvaluationResult.TRUE if the object's type is a sub type of the type specified by the attribute value. Otherwise EvaluationResult.FALSE is returned.

- **value** − a fully qualified name of a class or interface.

<!ELEMENT test EMPTY>

<!ATTLIST test

property CDATA #REQUIRED

args CDATA #IMPLIED

value CDATA #IMPLIED>

This element is used to evaluate the property state of the object in focus. The set of testable properties can be extended using the propery tester extension point. The test expression returns EvaluationResult.NOT_LOADED if teh property tester doing the actual testing isn't loaded yet.

- **property** – the name of an object's property to test.
- **args** – additional arguments passed to the property tester. Multiple arguments are seperated by commas. Each individual argument is converted into a Java base type using the same rules as defined for the value attribute of the test expression.
- **value** – the expected value of the property. Can be omitted if the property is a boolean property. The test expression is supposed to return EvaluationResult.TRUE if the property matches the value and EvaluationResult.FALSE otherwise. The value attribute is converted into a Java base type using the following rules:
  - the string "true" is converted into Boolean.TRUE
  - the string "false" is converted into Boolean.FALSE
  - if the string contains a dot then the interpreter tries to convert the value into a Float object. If this fails the string is treated as a java.lang.String
  - if the string only consists of numbers then the interpreter converts the value in an Integer object.
  - in all other cases the string is treated as a java.lang.String
  - the conversion of the string into a Boolean, Float, or Integer can be suppressed by surrounding the string with single quotes. For example, the attribute value="'true'" is converted into the string "true"

<!ELEMENT systemTest EMPTY>

<!ATTLIST systemTest

property CDATA #REQUIRED

value CDATA #REQUIRED>

Tests a system property by calling the System.getProperty method and compares the result with the value specified through the value attribute.

- **property** – the name of an system property to test.
- **value** – the expected value of the property. The value is interpreted as a string value.

Since:

<!ELEMENT equals EMPTY>

<!ATTLIST equals

value CDATA #REQUIRED>

This element is used to perform an equals check of the object in focus. The expression returns EvaluationResult.TRUE if the object is equal to the value provided by the attribute value. Otherwise EvaluationResult.FALSE is returned.

- **value** – the operatand of the equals tests. The value provided as a string is converted into a Java base type using the same rules as for the value attribute of the test expression.

<!ELEMENT count EMPTY>

<!ATTLIST count

value CDATA #REQUIRED>

This element is used to test the number of elements in a collection.

- **value** – an expression to specify the number of elements in a list. Following wildcard characters can be used:
  *
  : any number of elements
  ?
  : no elements or one element
  +
  : one or more elements
  !
  : no elements
  integer value
  : the list must contain the exact number of elements

<!ELEMENT with (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST with

variable CDATA #REQUIRED>

This element changes the object to be inspected for all its child element to the object referneced by the given variable. If the variable can not be resolved then the expression will throw a ExpressionException when evaluating it. The children of a with expression are combined using the and operator.

- **variable –** the name of the variable to be used for further inspection. It is up to the evaluator of an extension point to provide the variable in the variable pool.

<!ELEMENT resolve (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST resolve

variable CDATA #REQUIRED

args CDATA #IMPLIED>

This element changes the object to be inspected for all its child element to the object referneced by the given variable. If the variable can not be resolved then the expression will throw a ExpressionException when evaluating it. The children of a with expression are combined using the and operator.

- **variable –** the name of the variable to be resolved. This variable is then used as the object in focus for child element evaluation. It is up to the evaluator of an extension point to provide a corresponding variable resolver (see IVariableResolver) through the evaluation context passed to the root expression element when evaluating the expression.
- **args –** additional arguments passed to the variable resolver. Multiple arguments are seperated by commas. Each individual argument is converted into a Java base type using the same rules as defined for the value attribute of the test expression.

<!ELEMENT adapt (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST adapt

type CDATA #REQUIRED>

This element is used to adapt the object in focus to the type specified by the attribute type. The expression returns not loaded if either the adapter or the type referenced isn't loaded yet. It throws a ExpressionException during evaluation if the type name doesn't exist at all. The children of an adapt expression are combined using the and operator.

- **type –** the type to which the object in focus is to be adapted.

<!ELEMENT iterate (not , and , or , instanceof , test , systemTest , equals , count , with , resolve , adapt , iterate)*>

<!ATTLIST iterate

operator (or|and) >

This element is used to iterate over a variable that is of type java.util.Collection. If the object in focus is not of type java.util.Collection then an ExpressionException will be thrown while evaluating the expression.

- **operator –** either "and" or "or". The operator defines how the child elements will be combined. If not specified, "and" will be used.

Examples:

The example below contributes a status context viewer for FileStatusContext objects.

<extension point=

"org.eclipse.ltk.ui.refactoring.statusContextViewers"

>

<statusContextViewer class=

"org.eclipse.ltk.internal.ui.refactoring.FileStatusContextViewer"

id=

"org.eclipse.ltk.internal.ui.refactoring.fileStatusContextViewer"

>

<enablement>

<instanceof value=

"org.eclipse.ltk.core.refactoring.FileStatusContext"

/>

</enablement>

</statusContextViewer>

</extension>

API Information:

The contributed class must extend
`org.eclipse.ltk.ui.refactoring.IStatusContextViewer`

Search Pages

Identifier:

org.eclipse.search.searchPages

Description:

This extension point allows a plug−in to register search pages for specialized searches. When the search action is performed on a resource, the search infrastructure locates the most specific registered search page for it.

Configuration Markup:

<!ELEMENT extension (page*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point −** a fully qualified identifier of the target extension point
- **id −** an optional identifier of the extension instance
- **name −** an optional name of the extension instance

<!ELEMENT page EMPTY>

<!ATTLIST page

id CDATA #REQUIRED

label CDATA #REQUIRED

class CDATA #REQUIRED

icon CDATA #IMPLIED

sizeHint CDATA #IMPLIED

tabPosition CDATA #IMPLIED

extensions CDATA #IMPLIED

searchViewHelpContextId CDATA #IMPLIED

showScopeSection (true | false)

enabled (true | false)

canSearchEnclosingProjects (true | false) >

- **id –** a unique name that will be used to identify this search page
- **label –** a translatable label that will be used in the search page tab
- **class –** a name of the class which implements `org.eclipse.search.ui.ISearchPage`. We recommend subclassing `org.eclipse.jface.dialogs.DialogPage`.
- **icon –** a relative name of the image that will be used for all resources that match the specified extensions. If omitted, the search page's tab will only contain a label.
- **sizeHint –** a hint for the initial size of the page. This is a string containing the width and height separated by comma (e.g. "50, 60"). In the case this hint is omitted the page will be no larger than the other pages.
- **tabPosition –** an integer defining where the page is placed in the page book. The value does not define the absolute position but the position relative to the other pages. The ordering of pages is as follows:
    1. if neither page defines the tab position then they are ordered alphabetically according to their labels
    2. if both pages have the tab position specified then the page with the lower value comes first. If the values are equal then the pages are treated as if the values would not exist (see 1.)
    3. if only one page has the value specified then this page comes first
- **extensions –** a comma separated list with file extensions on which the search page can operate. Each extension must also include a weight (0 meaning lowest weight) which enables the search infrastructure to find the best fitting page. The weight is separated from the extension by a colon. If a search page can search all possible resources then "*" can be used.
- **searchViewHelpContextId –** an optional help context ID of the Search view displaying results of this page. If this attribute is missing then the default search help context ID (`org.eclipse.search.search_view_context`) is used.
- **showScopeSection –** If this attribute is missing or set to "false", then the scope section is not shown in the Search dialog. To see the scope section, this attribute has to be set to "true". Plug–ins which add their own Search page and want to see the scope section have to add this to their plugin.xml.
- **enabled –** If this attribute is missing or set to "false", then the page is not initially shown in the Search dialog. The page can be activated by the user via the "Customize..." button on the Search dialog.
- **canSearchEnclosingProjects –** If this attribute is missing or set to "false", the "Enclosing Projects" search scope is not shown in the search dialog's scope part. If the attribute "showScopeSection" is missing or set to "false", this attribute will be ignored.

Examples:

The following is an example of a search page extension definition:

<extension point=

"org.eclipse.search.searchPages"

>

<page id=

"org.eclipse.search.ui.text.TextSearchPage"

label=

"Text Search"

icon=

"icons/full/obj16/tsearch_pref.gif"

sizeHint=

"250,160"

tabPosition=

"1"

extensions=

"*:1"

showScopeSection=

"true"

class=

"org.eclipse.search.ui.text.TextSearchPage"

>

</page>

</extension>

API Information:

The contributed class must implement `org.eclipse.search.ui.ISearchPage`

Supplied Implementation:

The search infrastructure provides a search page for full−text search.

http://www.eclipse.org/legal/epl−v10.html

Result Sorters

Identifier:

org.eclipse.search.searchResultSorters

Description:

This extension point allows a plug−in to contribute search result sorters to the (old) search result view's Sort context menu. This extension point is deprecated since 3.0. The search result views are now contributed by clients and sorters are directly managed by these pages.

Configuration Markup:

<!ELEMENT extension (sorter*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point −** a fully qualified identifier of the target extension point
- **id −** an optional identifier of the extension instance
- **name −** an optional name of the extension instance

<!ELEMENT sorter EMPTY>

<!ATTLIST sorter

id CDATA #REQUIRED

pageId CDATA #REQUIRED

label CDATA #REQUIRED

tooltip CDATA #IMPLIED

icon CDATA #IMPLIED

class CDATA #REQUIRED>

- **id −** a unique name that will be used to identify this search result sorter.
- **pageId −** the ID of a registered search page for which the sorter will be activated. "*" activates the sorter for all pages.
- **label −** a translatable label that will be used as the menu item's label.

- **tooltip** – a translatable text that will be used as the menu item's tool tip . If omitted, the menu item will have no tool tip.
- **icon** – a relative name of the image that will be shown in the context menu along with the label. If omitted, the menu entry will only contain a label.
- **class** – a name of the class that extends `org.eclipse.jface.viewers.ViewerSorter`

Examples:

The following is an example of a search page extension definition:

<extension point=

"org.eclipse.search.searchResultSorters"

>

<sorter id=

"org.eclipse.search.internal.ui.FileNameSorter"

pageId=

"*"

label=

"%FileNameSorter.label"

tooltip=

"%FilenNameSorter.tooltip"

icon=

"icons/full/ecl16/search_sort.gif"

class=

"org.eclipse.search.internal.ui.FileNameSorter"

>

</sorter>

</extension>

API Information:

The contributed class must implement `org.eclipse.jface.viewers.ViewerSorter`

Supplied Implementation:

The search infrastructure provides a sorter that sorts the matches by the resource name.

Search Result View Pages

Identifier:

org.eclipse.search.searchResultViewPages

Since:

3.0

Description:

This extension point allows clients to plug pages into the search result view.

Configuration Markup:

<!ELEMENT extension (viewPage)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED>

- **point –** a fully qualified identifier of the target extension point
- **id –** an optional identifier of the extension instance

<!ELEMENT viewPage EMPTY>

<!ATTLIST viewPage

searchResultClass CDATA #REQUIRED

class CDATA #REQUIRED

id CDATA #REQUIRED>

- **searchResultClass –** The fully qualified class name of the `ISearchResult` implementation this search result page supposed to show.
- **class –** the fully qualified class name implementing this search result page. The class must implement `org.eclipse.search.ui.ISearchResultPage`.
- **id –** the id, typically the same as the fully qualified class name

Examples:

As an example, here is the markup for the file search result page.

<extension id=

"FileSearchPage"

point=

"org.eclipse.search.searchResultViewPages"

>

<viewPage id=

"org.eclipse.search.text.FileSearchResultPage"

searchResultClass=

"org.eclipse.search.internal.ui.text.FileSearchResult"

class=

"org.eclipse.search.internal.ui.text.FileSearchPage"

>

</viewPage>

</extension>

API Information:

The contributed class must implement `org.eclipse.search.ui.ISearchResultPage`

Supplied Implementation:

The search plugin provide a search result page for file searches.

Configuration Duplication Maps

Identifier:

org.eclipse.ui.externaltools.configurationDuplicationMaps

Since:

3.0

Description:

This is an internal extension point to declare the launch configuration type that should be created when duplicating an existing configuration as a project builder. Clients are not intended to use this extension point.

Configuration Markup:

<!ELEMENT extension (configurationMap*)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

<!ELEMENT configurationMap EMPTY>

<!ATTLIST configurationMap

sourceType CDATA #REQUIRED

builderType CDATA #REQUIRED>

- **sourceType –** specifies the identifier of the launch configuration type for which this mapping is provided
- **builderType –** specifies the identifier of the launch configuration type which should be created when a configuration of type "sourceType" is imported to be a project builder

Examples:

The following example specifies that when the user chooses to import a launch configuration of the type "org.eclipse.ui.externaltools.ProgramLaunchConfigurationType", a new launch configuration of the type"org.eclipse.ui.externaltools.ProgramBuilderLaunchConfigurationType" should be created.

<extension point=

"org.eclipse.ui.externaltools.configurationDuplicationMaps"

>

<configurationMap sourceType=

"org.eclipse.ui.externaltools.ProgramLaunchConfigurationType"

builderType=

"org.eclipse.ui.externaltools.ProgramBuilderLaunchConfigurationType"

>

</configurationMap>

</extension>

Feature Type Factory

Identifier:

org.eclipse.update.core.featureTypes

Description:

The platform update mechanism supports pluggable feature type implementations. A new feature type can be registered in order to support alternate packaging and verification schemes.

The `featureTypes` extension point allows alternate feature implementations to be registered using a symbolic type identifier. Whenever the type is referenced using this identifier, the supplied factory is used to create the correct concrete feature implementation.

Configuration Markup:

<!ELEMENT extension (feature−factory+)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point** − must be specified as **org.eclipse.update.core.featureTypes**
- **id** − must be specified. Identifies the new feature type
- **name** − optional displayable label for the new feature type

<!ELEMENT feature−factory EMPTY>

<!ATTLIST feature−factory

class CDATA #REQUIRED>

- **class** − fully qualified name of the factory class for the identified feature type

Examples:

The following is an example of new feature type registration:

<extension id=

"custom"

point=

"org.eclipse.update.core.featureTypes"

name=

"Custom packaged feature"

>

<feature−factory class=

"com.xyz.update.CustomFeatureFactory"

>

</feature−factory>

</extension>

API Information:

Registered factory classes must implement **`org.eclipse.update.core.IFeatureFactory`**

Supplied Implementation:

The platform supplies two standard implementations of feature types. One representing the default packaged feature type, and the other representing an installed feature type.

Global Install Handlers

Identifier:

org.eclipse.update.core.installHandlers

Description:

Extension point for registering global install handlers. Global install handlers can be referenced by features (using the `<feature> <install-handler>` tags) without having to include a copy of the handler code as part of the downloadable feature.

Configuration Markup:

<!ELEMENT extension (install−handler+)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point −** must be specified as **org.eclipse.update.core.installHandlers**
- **id −** must be specified. Identifies the new install handler
- **name −** optional displayable label for the new install handler

<!ELEMENT install−handler EMPTY>

<!ATTLIST install−handler

class CDATA #REQUIRED>

- **class −** fully qualified name of the handler implementation class for the identified install handler

Examples:

The following is an example of new global install handler registration:

<extension id=

"custom"

point=

"org.eclipse.update.core.installHandlers"

name=

"Custom install handler"

>

<install−handler class=

"com.xyz.update.CustomInstallHandler"

>

</install−handler>

</extension>

API Information:

Registered install handler classes must implement `org.eclipse.update.core.IInstallHandler` interface. Implementers should extend base class `org.eclipse.update.core.BaseInstallHandler`.

Supplied Implementation:

The platform supplies a simple install handler that is registered as `org.eclipse.update.core.DefaultInstallHandler`. If used, it will copy any non−plug−in data entries provided with the feature into the feature installation directory.

Site Type Factory

Identifier:

org.eclipse.update.core.siteTypes

Description:

The platform update mechanism supports pluggable site type implementations. A new site type can be registered in order to support alternate site layout schemes.

The `siteTypes` extension point allows alternate site implementations to be registered using a symbolic type identifier. Whenever the type is referenced using this identifier, the supplied factory is used to create the correct concrete site implementation.

Configuration Markup:

<!ELEMENT extension (site−factory+)>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

- **point** − must be specified as **org.eclipse.update.core.siteTypes**
- **id** − must be specified. Identifies the new site type
- **name** − optional displayable label for the new site type

<!ELEMENT site−factory EMPTY>

<!ATTLIST site−factory

class CDATA #REQUIRED>

- **class** − fully qualified name of the factory class for the identified site type

Examples:

The following is an example of new site type registration.

<extension id=

"custom"

point=

"org.eclipse.update.core.siteTypes"

name=

"Custom site"

>

<site−factory class=

"com.xyz.update.CustomSiteFactory"

>

</site−factory>

</extension>

API Information:

Registered factory classes must implement **`org.eclipse.update.core.ISiteFactory`**

Supplied Implementation:

The platform supplies two standard implementations of site types. One representing the default update server type, and the other representing the local file system site.

Runtime overview

The Eclipse runtime defines the plug−ins (**org.eclipse.osgi** and **org.eclipse.core.runtime**) on which all other plug−ins depend. The runtime is responsible for defining a structure for plug−ins and the implementation detail (bundles and classloaders) behind them. The runtime is also responsible for finding and executing the main Eclipse application and for maintaining a registry of plug−ins, their extensions, and extension points.

The runtime also provides an assortment of utilities, such as logging, debug trace options, adapters, a preference store, and a concurrency infrastructure. Of course, as a minimal kernel, the runtime is only interesting once plug−ins that make use of it and perform some kind of task are created. Like Atlas, the runtime plug−in stoically resides at the bottom of the plug−in heap, holding the Eclipse universe aloft on its steady shoulders.

The runtime plug−in model

The platform runtime engine is started when a user starts an application developed with Eclipse. The runtime implements the basic **plug−in** model and infrastructure used by the platform. It keeps track of all of the installed plug−ins and the function that they provide.

A plug−in is a structured component that contributes code (or documentation or both) to the system and describes it in a structured way. Plug−ins can define **extension points**, well−defined function points that can be extended by other plug−ins. When a plug−in contributes an implementation for an extension point, we say that it adds an **extension** to the platform. These extensions and extension points are declared in the plug−ins's manifest (**plugin.xml**) file.

Using a common extension model provides a structured way for plug−ins to describe the ways they can be extended, and for client plug−ins to describe the extensions they supply. Defining an extension point is much like defining any other API. The only difference is that the extension point is declared using XML instead of a code signature. Likewise, a client plug−in uses XML to describe its specific extension to the system.

A general goal of the runtime is that the end user should not pay a memory or performance penalty for plug−ins that are installed, but not used. The declarative nature of the platform extension model allows the runtime engine to determine what extension points and extensions are supplied by a plug−in without ever running it. Thus, many plug−ins can be installed, but none will be activated until a function provided by a plug−in has been requested according to the user's activity. This is an important feature in providing a scalable, robust platform.

Plug−ins and bundles

The mechanics for supporting plug−ins are implemented using the OSGi framework. From this standpoint, a plug−in is the same thing as an OSGi **bundle**. The bundle and its associated classes specify and implement the process for Java class−loading, prequisite management, and the bundle's life−cycle. For the rest of this discussion, we use the terms **plug−in** and **bundle** interchangeably, unless discussing a particular class in the framework.

Plugin

The **Plugin** class represents a plug−in that is running in the platform. It is a convenient place to centralize the life−cycle aspects and overall semantics of a plug−in. A plug−in can implement specialized function for the

start and **stop** aspects of its life−cycle. Each life−cycle method includes a reference to a **BundleContext** which can supply additional information.

The **start** portion of the life−cycle is worth particular discussion. We've seen already that information about a plug−in can be obtained from the plug−in's manifest file without ever running any of the plug−in's code. Typically, some user action in the workbench causes a chain of events that requires the starting of a plug−in. From an implementation point of view, a plug−in is never started until a class contained the in the plug−in needs to be loaded.

The **start** method has been a convenient place to implement initialization and registration behavior for a plug−in. However, it is important to realize that your plug−in can be started in many different circumstances. Something as simple as obtaining an icon to decorate an object can cause one of your plug−in's classes to be loaded, thus starting your plug−in. Over−eager initialization can cause your plug−in's code and data to be loaded long before it is necessary. Therefore, it's important to look closely at your plug−in's initialization tasks and consider alternatives to performing initialization at start−up.

- **Registration** activities such as registering listeners or starting background threads are appropriate during plug−in start−up if they can be performed quickly. However, it is advisable to trigger these actions as part of accessing the plug−in's data if the registration activities have side−effects such as initializing large data structures or performing unrelated operations.
- **Initialization** of data is best done lazily, when the data is first accessed, rather than automatically in the start−up code. This ensures that large data structures are not built until they are truly necessary.

Bundle Context

Life−cycle management is where the OSGi "bundle" terminology and the platform's "plug−in" terminology meet. When your plug−in is started, it is given a reference to a **BundleContext** from which it can obtain information related to the plug−in. The **BundleContext** can also be used to find out about other bundles/plug−ins in the system.

BundleContext.getBundles() can be used to obtain an array of all bundles in the system. Listeners for **BundleEvent** can be registered so that your plug−in is aware when another bundle has a change in its life−cycle status. See the javadoc for **BundleContext** and **BundleEvent** for more information.

> *Prior to 3.0, a plugin registry (**IPluginRegistry**) was provided to supply similar information. For example, it could be queried for the plug−in descriptors of all plug−ins in the system. This registry is now deprecated and **BundleContext** should be used for this purpose. The platform registry is now used exclusively for information about extensions and extension points.*

Bundle Activator

The **BundleActivator** interface defines the start and stop behavior implemented in **Plugin**. Although the **Plugin** class is a convenient place to implement this function, a plug−in developer has complete freedom to implement the interface for **BundleActivator** in any class appropriate for the plug−in's design. In fact, your plug−in need not implement this interface at all if it does not have specific life−cycle management needs.

Bundles

Underneath every plug−in lies an OSGi bundle managed by the framework. The **Bundle** is the OSGi unit of modularity. Fundamentally, a bundle is just a collection of files (resources and code) installed in the platform.

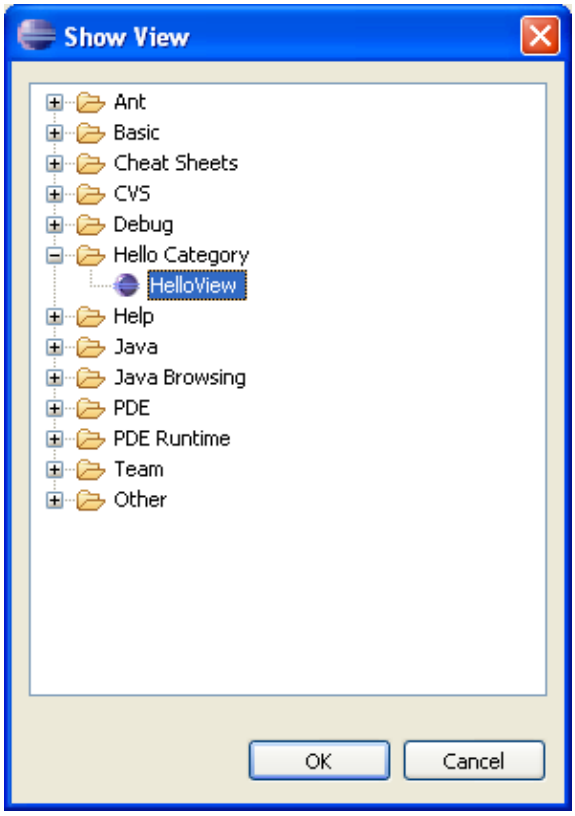
Each bundle has its own Java class loader, and includes protocol for starting, stopping, and uninstalling itself. From the Eclipse platform point of view, **Bundle** is merely an implementation class. Plug−in developers do not extend the bundle class, but use **Plugin** or other **BundleActivator** implementations to represent the plug−in.

Extension points and the registry

While the "bundle" aspects of a plug−in may be interesting to the runtime plug−in and runtime tools, it is much more common that a plug−in is concerned with what extension points have been defined by plug−ins and what extensions are contributed by plug−ins. This information is provided by the platform extension registry, **IExtensionRegistry**.

Why might a plug−in want to know what extensions are present? A concrete example will help show the need for this information and the protocol for getting it.

Recall the workbench **Show View** dialog which shows all of the available views that have been installed in the platform.



We know that the category names and view names of all contributed views are specified in the **plugin.xml** file for any plug−in that contributes an extension for **org.eclipse.ui.views**. But how does the workbench find out this information? From the platform extension registry. The following code is a simplified snippet based on the workbench implementation of the **Show View** dialog:

```
...
IExtensionRegistry registry = Platform.getExtensionRegistry();
IExtensionPoint point = registry.getExtensionPoint("org.eclipse.ui.views");
if (point == null) return;
IExtension[] extensions = point.getExtensions();
```

```
for (int i = 0; i < extensions.length; i++)
        readExtension(extensions[i]);  //get the information about each extension
...
```

We see above that the registry can be obtained from the **Platform** class. The protocol in **IExtensionRegistry** is used to find the extension point named **org.eclipse.ui.views**. Information in the registry about particular extension points or extensions can be found using protocol defined in **IExtensionRegistry**, **IExtensionPoint**, and **IExtension**. The javadoc for these classes provides detailed information about the registry protocol.

Once the extension definition of interest has been found, protocol in **IConfigurationElement** can be used to examine the individual attributes of an extension.

Contributing content types

Providing a new content type

The platform defines some fundamental content types, such as plain text and XML. These content types are defined the same way as those contributed by any other plug−ins. We will look at how the platform defines some of its content types in order to better understand the content type framework.

Plug−ins define content types by contributing an extension for the extension point **org.eclipse.core.runtime.contentTypes**. In this extension, a plug−in specifies a simple id and name for the content type (the full id is always the simple id prefixed by the current namespace). The following snippet shows a trimmed down version of the `org.eclipse.core.runtime.text` content type contribution:

```
<extension point="org.eclipse.core.runtime.contentTypes">
        <content-type
                id="text"
                name="%textContentTypeName">
                file-extensions="txt">
                <describer class="org.eclipse.core.internal.content.TextContentDescriber"
        </content-type>
        ...
```

The `file-extensions` attribute defines what file extensions are associated with the content type (in this example, ".txt"). The `file-names` attribute (not used in this case) allows associating full names. Both attributes are taken into account by the platform when performing content type detection and description (if the client provides a file name).

The `describer` element is used to define a *content describer* for the content type.

Detecting and describing content

A content type should provide a content describer if there are any identifiable characteristics that allow automatic content type detection, or any interesting properties in data belonging to the content type. In the case of `org.eclipse.core.runtime.text`, it is not possible to figure out the content type by just looking at the contents. However, text streams might be prepended by a *byte order mark*, which is a property clients might be interested in knowing about, so this warrants a content describer.

The describer is an implementation of **IContentDescriber** or **ITextContentDescriber**. The latter is a specialization of the former that must be implemented by describers of text−oriented content types. Regardless the nature of the content type, the describer has two responsibilities: helping determining whether

its content type is appropriate for a given data stream, and extracting interesting properties from a data stream that supposedly belongs to its content type.

The method **describe(stream, description)** is called whenever the platform is trying to determine the content type for a particular data stream or describe its contents. The description is `null` when only detection is requested. Otherwise, the describer should try to fill the content description with any properties that could be found *by reading the stream*, and only those. The content type markup should be used to declare any properties that have default values (for example, `org.eclipse.core.runtime.xml` declares UTF−8 as the default charset).

When performing its duty, the content describer is expected to execute as quickly as possible. The less the data stream has to be read, the better. Also, *it is expected that the content describer implementation be declared in a package that is exempt from plug−in activation (see the Eclipse−AutoStart bundle manifest header). Since all describers are instantiated when the content type framework is initialized, failure in complying with this requirement causes premature activation, which must be avoided. Future implementations of the platform might refuse to instantiate describers if doing so would trigger activation of the corresponding plug−in.*

Extending an existing content type

Content types are hierarchical in nature. This allows new content types to leverage the attributes or behavior of more general content types. For example, a content type for XML data is considered a child of the text content type:

```
<content-type
        id="xml"
        name="%xmlContentTypeName"
        base-type="org.eclipse.core.runtime.text"
        file-extensions="xml">
        <describer class="org.eclipse.core.internal.content.XMLContentDescriber"/>
        <property name="charset" default="UTF-8"/>
</content-type>
```

A XML file is deemed a kind of text file, so any features applicable to the latter should be applicable to the former as well.

Note that the XML content type overrides several content type attributes originally defined in the Text content type such as the file associations and the describer implementation. Also, this content type declares a default property value for `charset` property. That means that during content description for a data stream considered as belonging to the XML content type, if the describer does not fill in the charset property, the platform will set it to be "UTF−8".

As another example, the `org.eclipse.ant.core.antBuildFile` content type (for Ant Build Scripts) extends the XML content type:

```
<content-type
        id="antBuildFile"
        name="%antBuildFileContentType.name"
        base-type="org.eclipse.core.runtime.xml"
        file-names="build.xml"
        file-extensions="macrodef,ent,xml">
        <describer
                class="org.eclipse.ant.internal.core.contentDescriber.AntBuildfileContentDescribe
        </describer>
```

```
</content-type>
```

Note that the default value for the charset property is inherited. It is possible to cancel an inherited property or describer by redeclaring them with the empty string as value.

Additional file associations

New file associations can be added to existing content types. For instance, the Resources plug−in associates the `org.eclipse.core.runtime.xml` to ".project" files:

```
<extension point="org.eclipse.core.runtime.contentTypes">
        <file-association content-type="org.eclipse.core.runtime.xml" file-names=".project"/>
        ...
```

Content type aliasing

Due to the extensible nature of Eclipse, a content type a plug−in rely on may not be available in a given product configuration. This can be worked around by using content type aliasing. A *content type alias* is a placeholder for another preferred content type whose availability is not guaranteed. For instance, the Runtime declares an alias (`org.eclipse.core.runtime.properties`) for the Java properties content type provided by the Java Development Tooling (`org.eclipse.jdt.core.javaProperties`):

```
<!-- a placeholder for setups where JDT's official type is not available -->
<content-type
        id="properties"
        name="%propertiesContentTypeName"
        base-type="org.eclipse.core.runtime.text"
        alias-for="org.eclipse.jdt.core.javaProperties"
        file-extensions="properties">
        <property name="charset" default="ISO-8859-1"/>
</content-type>
```

This provides plug−ins with a placeholder they can refer to regardless the preferred content type is available or not. If it is, the alias content type is supressed from the content type catalog and any references to it are interpreted as references to the target content type. If it is not, the alias will be used as an ordinary content type.

OSGi Bundle Manifest Headers

Version 3.1 – Last revised June 20, 2005

A bundle can carry descriptive information about itself in the manifest file named META–INF/MANIFEST.MF. The OSGi R4 Framework specification defines a set of manifest headers such as Export–Package and Bundle–Classpath, which bundle developers use to supply descriptive information about a bundle. The Eclipse OSGi Framework implements the complete OSGi R4 Framework specification and all of the Core Framework services. The OSGi R4 Core Framework services include the following:

- Package Admin Service Specification
- URL Handlers Service Specification
- Start Level Service Specification
- Conditional Permission Admin Specification
- Permission Admin Service Specification

There are a number of optional services defined in the OSGi R4 specification. The optional services are not included with the Eclipse OSGi Framework implementation. For information on OSGi R4 manifest headers and services refer to the OSGi R4 specification.

Eclipse Bundle Manifest Headers

The Eclipse OSGi Framework supports a number of additional bundle manifest headers and directives. A bundle developer may use these additional headers and directives to take advantage of some additional features of the Eclipse OSGi Framework which are not specified as part of a standard OSGi R4 Framework.

Additional Export–Package Directives

The Eclipse OSGi Framework supports additional directives on the Export–Package header. These directives are used to specify the access restriction rules of an exported package. See osgi.resolverMode to configure the Eclipse OSGi Framework to enforce the access restriction rules at runtime.

The x–internal Directive

The x–internal directive can be used in an Export–Package header to specify whether the package is an internal package. The Plug–in Development Environment will discourage other bundles from using an internal package. If the x–internal directive is not specified then a default value of 'false' is used. The x–internal directive must use the following syntax:

```
x-internal ::= ( 'true' | 'false' )
```

The following is an example of the x–internal directive:

```
Export-Package: org.eclipse.foo.internal; x-internal:=true
```

The x–friends Directive

The x–friends directive can be used in an Export–Package header to specify a list of bundles which are allowed access to the package. The Plug–in Development Environment will discourage other bundles from

using the package. The x−friends directive must use the following syntax:

```
x-friends ::= '"' ( target-bundle ) ( ',' target-bundle ) * '"'
target-bundle ::= a bundle symbolic name
```

The following is an example of the x−friends directive:

```
Export-Package: org.eclipse.foo.formyfriends; x-friends:="org.eclispe.foo.friend1, org.eclipse.fo
```

The example specifies that only the bundles org.eclispe.foo.friend1 and org.eclipse.foo.friend2 should be encouraged to use the org.eclipse.foo.formyfriends package. The x−internal package takes priority over the x−friends directive. If the x−internal directive specifies 'true' then The Plug−in Development Environment will discourage all bundles from using the package even if they are specified as a friend.

The Eclipse−AutoStart Header

The Eclipse−AutoStart header is used to specify if a bundle should be started automatically before the first class or resource is accessed from that bundle. This feature allows Eclipse to activate bundles on demand the first time they are needed. Using this model Eclipse can startup with as few active bundles as possible. The Eclipse−AutoStart header must use the following syntax:

```
Eclipse-AutoStart ::= ( 'true' | 'false' ) ( ';' 'exceptions' '=' '"' exceptions-list '"' ) ?
exceptions-list ::= a comma ',' separated list of packages
```

The 'exceptions' attribute is used to specify a list of packages which must not cause the bundle to be activated when classes or resources are loaded from them. If the Eclipse−AutoStart header is not defined in the bundle manifest then a default value of 'false' is used. The following is an example of the Eclipse−AutoStart header:

```
Eclipse-AutoStart: true; exceptions="org.eclipse.foo1, org.eclipse.foo2"
```

The example specifies that this bundle must be activated for any classes or resources that are loaded from this bundle, except the classes and resources in the packages 'org.eclipse.foo1' and 'org.eclipse.foo2'.

The Eclipse−PlatformFilter Header

The Eclipse−PlatformFilter is used to specify a platform filter for a bundle. A platform filter must evaluate to true in a running platform in order for a bundle to be allowed to resolve. The Eclipse−PlatformFilter header must use the following syntax:

```
Eclipse-PlatformFilter ::= a valid LDAP filter string
```

The Framework supports filtering on the following system properties:

- **osgi.nl** – the platform language setting.
- **osgi.os** – the platform operating system.
- **osgi.arch** – the platform architecture.
- **osgi.ws** – the platform windowing system.

The following is an example of the Eclipse−PlatformFilter header:

```
Eclipse-PlatformFilter: (& (osgi.ws=win32) (osgi.os=win32) (osgi.arch=x86))
```

This example specifies that this bundle can only be resolved if the platform properties are osgi.ws=win32 and osgi.os=win32 and osgi.arch=x86. In other words a platform running on an x86 architecture, using a win32 operating system and the win32 windowing system.

The Eclipse−BuddyPolicy Header

The Eclipse−BuddyPolicy header is used to specify the buddy classloading policies for a bundle. The Eclipse−BuddyPolicy header must use the following syntax:

```
Eclipse-BuddyPolicy ::= ( policy-name ) ( ',' policy-name ) *
policy-name ::= ( 'dependent' | 'global' | 'registered' |
                  'app' | 'ext' | 'boot' | 'parent' )
```

The following is an example of the Eclipse−BuddyPolicy header:

```
Eclipse-BuddyPolicy: dependent
```

The Eclipse−RegisterBuddy Header

The Eclipse−RegisterBuddy header is used to specify a list of bundles for which this bundle is a registered buddy. The Eclipse−RegisterBuddy header must use the following syntax:

```
Eclipse-RegisterBuddy ::= ( target-bundle ) ( ',' target-bundle ) *
target-bundle ::= a bundle symbolic name
```

The following is an example of the Eclipse−RegisterBuddy header:

```
Eclipse-RegisterBuddy: org.eclipse.foo.bundle1, org.eclipse.foo.bundle2
```

The Eclipse−ExtensibleAPI Header

The Eclipse−ExtensibleAPI is used to specify whether a host bundle allows fragment bundles to add addtional API to the host. This header should be used if a host bundle wants to allow fragments to add additional packages to the API of the host. If this header is not specified then a default value of 'false' is used. The Eclipse−ExtensibleAPI header must use the following syntax:

```
Eclipse-ExtensibleAPI ::= ( 'true' | 'false' )
```

The following is an example of the Eclipse−ExtensibleAPI header:

```
Eclipse-ExtensibleAPI: true
```

The Plugin−Class Header

The Plugin−Class header is only used to support plugins developed for the Eclipse 2.1 platform. This header is used to specify a class name that will be used to activate a plugin using the old Eclipse 2.1 activation model. New bundles developed for the Eclipse 3.0 or greater platform should not use this header. The following is an example of the Plugin−Class header:

```
Plugin-Class: org.eclipse.foo.FooPlugin
```

Concurrency infrastructure

One of the major challenges of a complex system is to remain responsive while tasks are being performed. This challenge is even greater in an extensible system, when components that weren't designed to run together are sharing the same resources. The **org.eclipse.core.runtime.jobs** package addresses this challenge by providing infrastructure for scheduling, executing, and managing concurrently running operations. This infrastructure is based on the use of *jobs* to represent a unit of work that can run asynchronously.

Jobs

The **Job** class represents a unit of asynchronous work running concurrently with other jobs. To perform a task, a plug–in creates a job and then *schedules* it. Once a job is scheduled, it is added to a job queue managed by the platform. The platform uses a background scheduling thread to manage all of the pending jobs. As a running job completes, it is removed from the queue and the platform decides which job to run next. When a job becomes active, the platform invokes its **run()** method. Jobs are best demonstrated with a simple example:

```
class TrivialJob extends Job {
   public TrivialJob() {
      super("Trivial Job");
   }
   public IStatus run(IProgressMonitor monitor) {
      System.out.println("This is a job");
      return Status.OK_STATUS;
   }
}
```

The job is created and scheduled in the following snippet:

```
TrivialJob job = new TrivialJob();
System.out.println("About to schedule a job");
job.schedule();
System.out.println("Finished scheduling a job");
```

The output of this program is timing dependent. That is, there is no way to be sure when the job's **run** method will execute in relation to the thread that created the job and scheduled it. The output will either be:

```
About to schedule a job
This is a job
Finished scheduling a job
```

or:

```
About to schedule a job
Finished scheduling a job
This is a job
```

If you want to be certain that a job has completed before continuing, you can use the **join()** method. This method will block the caller until the job has completed, or until the calling thread is interrupted. Let's rewrite our snippet from above in a more deterministic manner:

```
TrivialJob job = new TrivialJob();
System.out.println("About to schedule a job");
job.schedule();
job.join();
```

```
if (job.getResult().isOk())
   System.out.println("Job completed with success");
else
   System.out.println("Job did not complete successfully");
```

Assuming the **join()** call is not interrupted, this method is guaranteed to return the following result:

```
About to schedule a job
This is a job
Job completed with success
```

Of course, it is generally not useful to join a job immediately after scheduling it, since you obtain no concurrency by doing so. In this case you might as well do the work from the job's run method directly in the calling thread. We'll look at some examples later on where the use of join makes more sense.

The last snippet also makes use of the job **result**. The result is the **IStatus** object that is returned from the job's **run()** method. You can use this result to pass any necessary objects back from the job's run method. The result can also be used to indicate failure (by returning an **IStatus** with severity **IStatus.ERROR**), or cancellation (**IStatus.CANCEL**).

Common job operations

We've seen how to schedule a job and wait for it complete, but there are many other interesting things you can to do jobs. If you schedule a job but then decide it is no longer needed, the job can be stopped using the **cancel()** method. If the job has not yet started running when canceled, the job is immediately discarded and will not run. If, on the other hand, the job has already started running, it is up to the job whether it wants to respond to the cancellation. When you are trying to cancel a job, waiting for it using the **join()** method comes in handy. Here is a common idiom for canceling a job, and waiting until the job is finished before proceeding:

```
if (!job.cancel())
   job.join();
```

If the cancellation does not take effect immediately, then **cancel()** will return false and the caller will use **join()** to wait for the job to successfully cancel.

Slightly less drastic than cancellation is the **sleep()** method. Again, if the job has not yet started running, this method will cause the job to be put on hold indefinitely. The job will still be remembered by the platform, and a **wakeUp()** call will cause the job to be added to the wait queue where it will eventually be executed.

Job states

A job goes through several states during its lifetime. Not only can it be manipulated through API such as **cancel()** and **sleep()**, but its state also changes as the platform runs and completes the job. Jobs can move through the following states:

- **WAITING** indicates that the job been scheduled to run, but is not running yet.
- **RUNNING** indicates that the job is running.
- **SLEEPING** indicates that the job is sleeping due to a sleep request or because it was scheduled to run after a certain delay.
- **NONE** indicates that the job is not waiting, running, or sleeping. A job is in this state when it has been created but is not yet scheduled. It is also in this state after it is finished running or when it has been canceled.

A job can only be put to sleep if it is currently **WAITING**. Waking up a sleeping job will put it back in the **WAITING** state. Canceling a job will return it to the **NONE** state.

If your plug−in needs to know the state of a particular job, it can register a *job change listener* that is notified as the job moves through its life−cycle. This is useful for showing progress or otherwise reporting on a job.

Job change listeners

The **Job** method **addJobChangeListener** can be used to register a listener on a particular job. **IJobChangeListener** defines protocol for responding to the state changes in a job:

- **aboutToRun** is sent when the job is about to be run.
- **awake** is sent when a previously sleeping job is now waiting to be run.
- **done** is sent when a job finishes execution.
- **running** is sent when a job starts running.
- **scheduled** is sent when a job is scheduled and waiting in the queue of jobs.
- **sleeping** is sent when a waiting job is put to sleep.

In all of these cases, the listener is provided with an **IJobChangeEvent** that specifies the job undergoing the state change and status on its completion (if it is done).

> *Note: Jobs also define the **getState()** method for obtaining the (relatively) current state of a job. However, this result is not always reliable since jobs run in a different thread and may change state again by the time the call returns. Job change listeners are the recommended mechanism for discovering state changes in a job.*

The job manager

IJobManager defines protocol for working with all of the jobs in the system. Plug−ins that show progress or otherwise work with the job infrastructure can use **IJobManager** to perform tasks such as suspending all jobs in the system, finding out which job is running, or receiving progress feedback about a particular job. The platform's job manager can be obtained using **Platform** API:

```
IJobManager jobMan = Platform.getJobManager();
```

Plug−ins interested in the state of all jobs in the system can register a job change listener on the job manager rather than registering listeners on many individual jobs.

Job families

It is sometimes easier for a plug−in to work with a group of related jobs as a single unit. This can be accomplished using *job families*. A job declares that it belongs to a certain family by overriding the **belongsTo** method:

```
public static final String MY_FAMILY = "myJobFamily";
...
class FamilyJob extends Job {
   ...
   public boolean belongsTo(Object family) {
      return family == MY_FAMILY;
   }
}
```

IJobManager protocol can be used to cancel, join, sleep, or find all jobs in a family:

```
IJobManager jobMan = Platform.getJobManager();
jobMan.cancel(MY_FAMILY);
jobMan.join(MY_FAMILY, null);
```

Since job families are represented using arbitrary objects, you can store interesting state in the job family itself, and jobs can dynamically build family objects as needed. It is important to use family objects that are fairly unique, to avoid accidental interaction with the families created by other plug−ins.

Families are also a convenient way of locating groups of jobs. The method **IJobManager.find(Object family)** can be used to locate instances of all running, waiting, and sleeping jobs at any given time.

Reporting progress

Long running jobs (those lasting more than a second) should report progress to the **IProgressMonitor** that is passed to the job's **run** method. The workbench progress view will show all progress messages and units of completed work given to this monitor.

The supplied progress monitor should also be used to check for cancellation requests made from the progress view. When a user (or plug−in using job API) attempts to cancel a job, the **IProgressMonitor** method **isCanceled()** will return `true`. It is the job's responsibility to frequently check the cancellation status of a job and respond to a cancellation by exiting the **run** method as soon as possible once it detects a cancellation. The following **run** method reports progress and responds to job cancellation:

```
public IStatus run(IProgressMonitor monitor) {
   final int ticks = 6000;
   monitor.beginTask("Doing some work", ticks);
   try {
      for (int i = 0; i < ticks; i++) {
         if (monitor.isCanceled())
            return Status.CANCEL_STATUS;
         monitor.subTask("Processing tick #" + i);
         //... do some work ...
         monitor.worked(1);
      }
   } finally {
      monitor.done();
   }
   return Status.OK_STATUS;
}
```

The **beginTask** method is used to name the task in the corresponding progress view and to establish the total amount of work to be done so that the view can compute progress. The **subTask** messages will appear as a child in the progress tree as work is done. The progress view will calculate and display a percent completion based on the amount of work reported in the **worked** calls.

Progress monitors and the UI

As you can see, the **IProgressMonitor** class is designed with corresponding UI support in mind. The platform's UI plug−in provides support so that the workbench can show progress for jobs that are running. You can set up your jobs with this in mind, so that you can control how they are presented.

See Workbench Concurrency Support for a detailed look at the APIs available for showing progress for jobs.

System jobs

What if your job is a low−level implementation detail that you don't want to show to users? You can flag your job as a *system job*. A system job is just like any other job, except the corresponding UI support will not set up a progress view or show any other UI affordances associated with running a job. If your job is not either directly initiated by a user, or a periodic task that can be configured by a user, then your job should be a system job. The protocol for setting a system job is simple:

```
class TrivialJob extends Job {
   public TrivialJob() {
      super("Trivial Job");
      setSystem(true);
   }
   ...
}
```

The **setSystem** call must be made before the job is scheduled. An exception will be triggered if you attempt this call on a job that is currently waiting, sleeping, or running.

User jobs

If your job is a long running operation that is initiated by a user, then you should flag your job as a *user job*. A user job will appear in a modal progress dialog that provides a button for moving the dialog into the background. The workbench defines a user preference that controls whether these dialogs are ever modal. By defining your job as a user job, your progress feedback will automatically conform with the user preference for progress viewing. The protocol for setting a user job is similar:

```
class TrivialJob extends Job {
   public TrivialJob() {
      super("Trivial Job");
      setUser(true);
   }
   ...
}
```

The **setUser** call must also be made before the job is scheduled.

Progress groups

Progress groups are another mechanism that can be used to influence the way that a job is shown in the UI. When it is more appropriate to show the aggregate progress of several related jobs in the UI, a special **IProgressMonitor** that represents a group of related jobs can be created. This monitor is created using **IJobManager** protocol. The following snippet shows how to create a progress group and associate it with a job.

```
...
IJobManager jobMan = Platform.getJobManager();
myGroup = jobMan.createProgressGroup();
job.setProgressGroup(myGroup, 600);  // specify the units of work the job needs to show.
job.schedule()
...
```

The group facility allows plug−ins to break tasks into multiple jobs if needed, but to report them to the user as if they are a single task. The progress group monitor will handle the details for computing the percentage

completion relative to all of the jobs in the group.

A job must be placed into the progress group before it is scheduled. After a job finishes running, its reference to the progress group is lost. If the job is to be scheduled again, it must be set into the group once again before it is scheduled.

Workbench concurrency support

We've seen that the JFace UI framework provides basic support for showing task progress in a dialog (see Long running operations for details). In Concurrency infrastructure, we reviewed the platform runtime support for concurrency and long running operations. Now we will look at how the platform UI enhances this infrastructure in the **org.eclipse.ui.progress** package. This package supplies the UI for showing job progress in the workbench and defines additional support for jobs that run in the UI thread.

First, let's look at the different kinds of background operations that may be running and how they are shown in the workbench UI:

- *User initiated* jobs are those that the user has triggered. The workbench will automatically show user jobs in a modal progress dialog with a button to allow the user to run the operation in the background and continue working. A global preference is used to indicate whether user jobs should always run in the background. User jobs are distinguished as such in the Job API using (Job#setUser). Examples of user jobs include building, checking out a project, synchronizing with the repository, exporting a plug−in, and searching.
- *Automatically triggered* jobs have a meaning for users but are not initiated by the user. These jobs are shown in the progress view and in the status line, but a modal progress dialog won't be shown when they are run. Examples include autobuild and scheduled synchronization.
- *System operations* are not triggered by the user and can be considered a platform implementation detail. These jobs are created by setting the system flag using(Job#setSystem). Examples of system jobs include jobs that lazily populate widgets or compute decorations and annotations for views.

Given an environment where several things may be happening at the same time, a user needs the following:

- Indication that a long running operation has started.

  User jobs are shown to the user in a progress dialog giving immediate feedback, whereas automatically triggered jobs are shown in the status line and progress view. Jobs that affect a part should be scheduled or registered with the part so that the workbench can provide hints to the user that something is running that affects the part.

- Indication that an operation has ended.

  The user can easily know when a user job ends because the progress dialog closes. For non−user jobs, there are a couple of feedback mechanisms available. If the job is scheduled or registered with a part then the part's progress hint will show when it is complete. If a job returns an error, an error indicator will appear in the bottom right of the status line showing a hint that an error has occured.

- Indication of interesting new results, or new information, without stealing focus by using a dialog.

  A user job can directly show the results to the user when the operation completes. For non−user jobs, it is recommended to use something other than a dialog to show results, so that the user is not interrupted. For example, a view could be opened when the job starts and the results shown in this view without disrupting the user's workflow. In addition, job properties can be added to the job to indicate that it should be kept in the progress view and that it provides an action that will show the results. In this case, a warning indication will appear in the bottom right corner of the status line when a job remains in the progress view and has results to show the user.

- A general feeling of being in control of what is running, with the ability to monitor and cancel background operations.

  User jobs provide the best control for the user since they are easily cancelled and provide strong indication of blocking or conccurent operations running via the **Details** tab of the progress dialog. Note that the enhanced progress dialog that provides the **Details** area is only shown when plug−ins use IProgressService#busyCursorWhile or IProgressService#runInUI. In addition, the progress view provides access to jobs that are running.

- Consistent reporting of progress by all installed plug−ins.

  The advantage of using the progress service API is that users get a consistent progress experience.

Progress service

The workbench progress service (IProgressService) is the primary interface to the workbench progress support. It can be obtained from the workbench and then used to show progress for both background operations and operations that run in the UI thread. The main purpose of this class is to provide one−stop shopping for running operations, removing the need for plug−in developers to decide what mechanism should be used for showing progress in a given situation. Another advantage is that the progress dialog shown with these methods provides good support for indicating when an operation is blocked by another and gives the user control to resolve the conflict. Where possible, long running operations should be run using IProgressService#busyCursorWhile:

```
IProgressService progressService = PlatformUI.getWorkbench().getProgressService();
progressService.busyCursorWhile(new IRunnableWithProgress(){
   public void run(IProgressMonitor monitor) {
      //do non-UI work
   }
});
```

This method will initially put up a busy cursor, and replace it with a progress dialog if the operation lasts longer than a specified time threshhold. The advantage of this method over using a progress dialog is that the progress dialog won't be shown if the operation is short running . If your operation must update the UI, you

can always use Display.asyncExec or Display.syncExec to run the code that modifies the UI.

If an operation must be run in its entirety in the UI thread, then IProgressService#runInUI should be used. This method will also display a progress dialog if the operation is blocked and give the user control.

```
progressService.runInUI(
   PlatformUI.getWorkbench().getProgressService(),
   new IRunnableWithProgress() {
      public void run(IProgressMonitor monitor) {
         //do UI work
      }
   },
   Platform.getWorkspace().getRoot());
```

The third parameter can be null, or a scheduling rule for the operation. In this example, we are specifying the workspace root which will essentially lock the workspace while this UI operation is run.

You can also register an icon for a job family with the progress service so that the progress view can show the icon next to the running job. Here is an example that shows how the auto−build job family is associated with its icon:

```
IProgressService service = PlatformUI.getWorkbench().getProgressService();
ImageDescriptor newImage = IDEInternalWorkbenchImages.getImageDescriptor(
   IDEInternalWorkbenchImages.IMG_ETOOL_BUILD_EXEC);
service.registerIconForFamily(newImage, ResourcesPlugin.FAMILY_MANUAL_BUILD);
service.registerIconForFamily(newImage, ResourcesPlugin.FAMILY_AUTO_BUILD);
```

Showing that a part is busy

IWorkbenchSiteProgressService includes API for scheduling jobs that change the appearance of a workbench part while the job is running. If your plug−in is running background operations that affect the state of a part, you can schedule the job via the part and the user will get feedback that the part is busy. Here is an example:

```
IWorkbenchSiteProgressService siteService =
   (IWorkbenchSiteProgressService)view.getSite().getAdapter(IWorkbenchSiteProgressService.clas
siteService.schedule(job, 0 /* now */, true /* use the half-busy cursor in the part */);
```

Progress Properties for Jobs

The workbench defines progress−related properties for jobs in IProgressConstants . These can be used to control how a job is shown in the progress view. These can be used to tell the progress view to keep (IProgressConstants#KEEP_PROPERTY) your job in the view after it has finished, or only keep one (IProgressConstants#KEEPONE_PROPERTY) job at a time in the view. You can also associate an action (IProgressConstants#ACTION_PROPERTY) with a job. When a job has an associated action, the progress view shows a hyperlink so that a user can run the action. You can also find out if a user job is currently being shown in a progress dialog (IProgressConstants#PROPERTY_IN_DIALOG). A hint is provided in the bottom right of the status line when an action is available. The following example uses these properties:

```
Job job = new Job("Do Work") {
   public IStatus run(IProgressMonitor monitor) {
      // do some work.
      // Keep the finished job in the progress view only if it is not running in the progress
      Boolean inDialog = (Boolean)getProperty(IProgressConstants.PROPERTY_IN_DIALOG);
```

```
        if(!inDialog.booleanValue())
            setProperty(IProgressConstants.KEEP_PROPERTY, Boolean.TRUE);
    }
};
job.setProperty(IProgressConstants.ICON_PROPERTY, Plugin.getImageDescriptor(WORK_IMAGE));
IAction gotoAction = new Action("Results") {
    public void run() {
        // show the results
    }
};
job.setProperty(IProgressConstants.ACTION_PROPERTY, gotoAction);
job.setUser(true);
job.schedule();
```

Workbench jobs

Where possible, long running operations should be performed outside of the UI thread. However, this cannot always be avoided when the operation's purpose is to update the UI. SWT threading issues explains how this can be done using the SWT **Display**. The workbench defines a special job, **UIJob**, whose run method runs inside an SWT `asyncExec`. Subclasses of **UIJob** should implement the method **runInUIThread** instead of the **run** method.

WorkbenchJob extends **UIJob** so that the job can only be scheduled or run when the workbench is running. As always, you should avoid excessive work in the UI thread because the UI will not refresh for the duration of the UI Job.

Long−running operations

The **org.eclipse.jface.operations** package defines interfaces for long−running operations that require progress indicators or allow user cancellation of the operation. These interfaces are used in the implementation of the workbench progress dialogs and views.

In general, plug−ins should use the workbench support provided in IProgressService for running long operations, so that all plug−ins will have a consistent presentation of progress. See Workbench Concurrency Support for a complete discussion of the available support for progress dialogs and views. The remainder of this discussion highlights the details of the JFace operations infrastructure which is used by the workbench.

Runnables and progress

The platform runtime defines a common interface, **IProgressMonitor**, which is used to report progress to the user while long running operations are in progress. The client can provide a monitor as a parameter in many platform API methods when it is important to show progress to the user.

JFace defines more specific interfaces for objects that implement the user interface for a progress monitor.

IRunnableWithProgress is the interface for a long−running operation. The **run** method for this interface has an **IProgressMonitor** parameter that is used to report progress and check for user cancelation.

IRunnableContext is the interface for the different places in the UI where progress can be reported. Classes that implement this interface may choose to use different techniques for showing progress and running the operation. For example, **ProgressMonitorDialog** implements this interface by showing a progress dialog. **IWorkbenchWindow** implements this interface by showing progress in the workbench window's status line.

WizardDialog implements this interface to show long running operations inside the wizard status line.

> *Note: The workbench UI provides additional support for operations in **WorkspaceModifyOperation**. This class simplifies the implementation of long−running operations that modify the workspace. It maps between **IRunnableWithProgress** and **IWorkspaceRunnable**. See the javadoc for further detail.*

Modal operations

The **ModalContext** class is provided to run an operation that is modal from the client code's perspective. It is used inside the different implementations of **IRunnableContext**. If your plug−in needs to wait on the completion of a long−running operation before continuing execution, **ModalContext** can be used to accomplish this while still keeping the user interface responsive.

When you run an operation in a modal context, you can choose to fork the operation in a different thread. If **fork** is false, the operation will be run in the calling thread. If **fork** is true, the operation will be run in a new thread, the calling thread will be blocked, and the UI event loop will be run until the operation terminates.

For more information on the UI event loop, see Threading issues for clients.

Job scheduling

Our examples so far have demonstrated simple job creation, scheduling, and progress reporting. The job scheduling mechanism is actually more powerful than we've shown so far. You can have more fine−grained control over the way your job is scheduled by using priorities, delays, and custom scheduling conditions.

Job priorities

A job *priority* can be used to establish the importance of a job relative to other jobs in the system. Setting the priority of a job won't affect a job that is already running, but it will affect how a waiting job is scheduled relative to other jobs. The priority of a job can be one of several pre−defined priority constants:

- **INTERACTIVE** jobs generally have priority over other jobs. They should be short−running or low on processor usage, so that they don't block other INTERACTIVE jobs from running.
- **SHORT** jobs typically complete within a second, but may take a little longer. They run in the background and have priority over all jobs except INTERACTIVE jobs.
- **LONG** jobs are for longer running background jobs. They run only after INTERACTIVE and SHORT jobs have been run.
- **BUILD** jobs are for jobs associated with building tasks. They are a lower priority than LONG. BUILD jobs only run when all LONG jobs are complete.
- **DECORATE** jobs are the lowest priority in the system. They are used for tasks that provide information that may help supplement the UI, but that the user is not generally waiting for.

The default priority for a job is LONG. The following snippet creates the trivial job we used earlier, but sets the priority to DECORATE to indicate that it is the lowest level priority:

```
TrivialJob job = new TrivialJob();
job.setPriority(Job.DECORATE);
job.schedule();
```

Scheduling with a delay

Another technique for controlling how a job is scheduled is to use a scheduling delay. A scheduling delay can be specified when the job is scheduled. The job will be delayed for the specified number of milliseconds before it is scheduled.

```
TrivialJob job = new TrivialJob();
job.schedule(1000);  // wait one second before scheduling
```

Rescheduling a job

Scheduling a job that is already waiting or is sleeping has no effect. However, scheduling a job that is already running will cause it to be rescheduled after it is finished. This is a convenient mechanism for repetitive jobs such as background polling loops. If the job is rescheduled multiple times while it is running, it will only be rescheduled once with the most recently supplied delay. The following snippet defines a job that reschedules itself to run 10 seconds after it finishes the current iteration.

```
class RepetitiveTrivialJob extends Job {
   public RepetitiveTrivialJob() {
      super("Repetitive Trivial Job");
   }
   public IStatus run(IProgressMonitor monitor) {
      System.out.println("Running the job.");
      // reschedule after 10 seconds
      schedule(10000);
      return Status.OK_STATUS;
   }
}
```

Custom scheduling conditions

Additional protocol in the **Job** class allows a job to check for preconditions just before it is scheduled or run. This is best demonstrated by example:

```
class JobWithPreconditions extends Job {
        ...
        public boolean shouldSchedule() {
                return super.shouldSchedule() && checkJobPreconditions();
        }
        public boolean shouldRun() {
                return super.shouldRun() && checkJobPreconditions();
        }
        ...
}
```

The **shouldSchedule** method is called just before the job manager places the job in the queue. This allows the job to cancel itself if basic preconditions for scheduling are not met. The job should return `false` it is inappropriate to schedule it. Likewise, the **shouldRun** method is called just before the job manager runs the job. Any additional conditions that must be met before the job is run must be checked at this time.

Locks

It's possible that multiple jobs in the system need to access and manipulate the same object. **ILock** defines protocol for granting exclusive access to a shared object. When a job needs access to the shared object, it *acquires* a lock for that object. When it is finished manipulating the object, it *releases* the lock.

A lock is typically created when the shared object is created or first accessed by a plug−in. That is, code that has a reference to the shared object also has a reference to its lock. We'll start by creating a lock, **myLock**, that will be used to control access to **myObject**:

```
...
myObject = initializeImportantObject();
IJobManager jobMan = Platform.getJobManager();
myLock = jobMan.newLock();
...
```

A robust implementation of **ILock** is provided by the platform. The job manager provides instances of this lock for use by clients. These locks are aware of each other and can avoid circular deadlock.(We'll explain more about that statement in a moment.)

Whenever code in a job requires access to **myObject**, it must first acquire the lock on it. The following snippet shows a common idiom for working with a lock:

```
...
// I need to manipulate myObject, so I get its lock first.
try {
myLock.acquire();
        updateState(myObject);  // manipulate the object
} finally {
lock.release();
}
...
```

The **acquire()** method will not return until the calling job can be granted exclusive access to the lock. In other words, if some other job has already acquired the lock, then this code will be blocked until the lock is available. Note that the code that acquires the lock and manipulates **myObject** is wrapped in a `try` block, so that the lock can be released if any exceptions occur while working with the object.

Seems simple enough, right? Fortunately, locks are pretty straightforward to use. They are also reentrant, which means you don't have to worry about your job acquiring the same lock multiple times. Each lock keeps a count of the number of acquires and releases for a particular thread, and will only release from a job when the number of releases equals the number of acquires.

Deadlock

Earlier we noted that locks provided by the job manager are aware of each other and can avoid circular deadlock. To understand how deadlock occurs, let's look at a simple scenario. Suppose "Job A" acquires "Lock A" and subsequently tries to acquire "Lock B." Meanwhile, "Lock B" is held by "Job B" which is now blocked waiting on "Lock A." This kind of deadlock indicates an underlying design problem with the use of the locks between the jobs. While this simple case can be avoided easily enough, the chances of accidentally introducing deadlock increase as the number of jobs and locks used in your design increase.

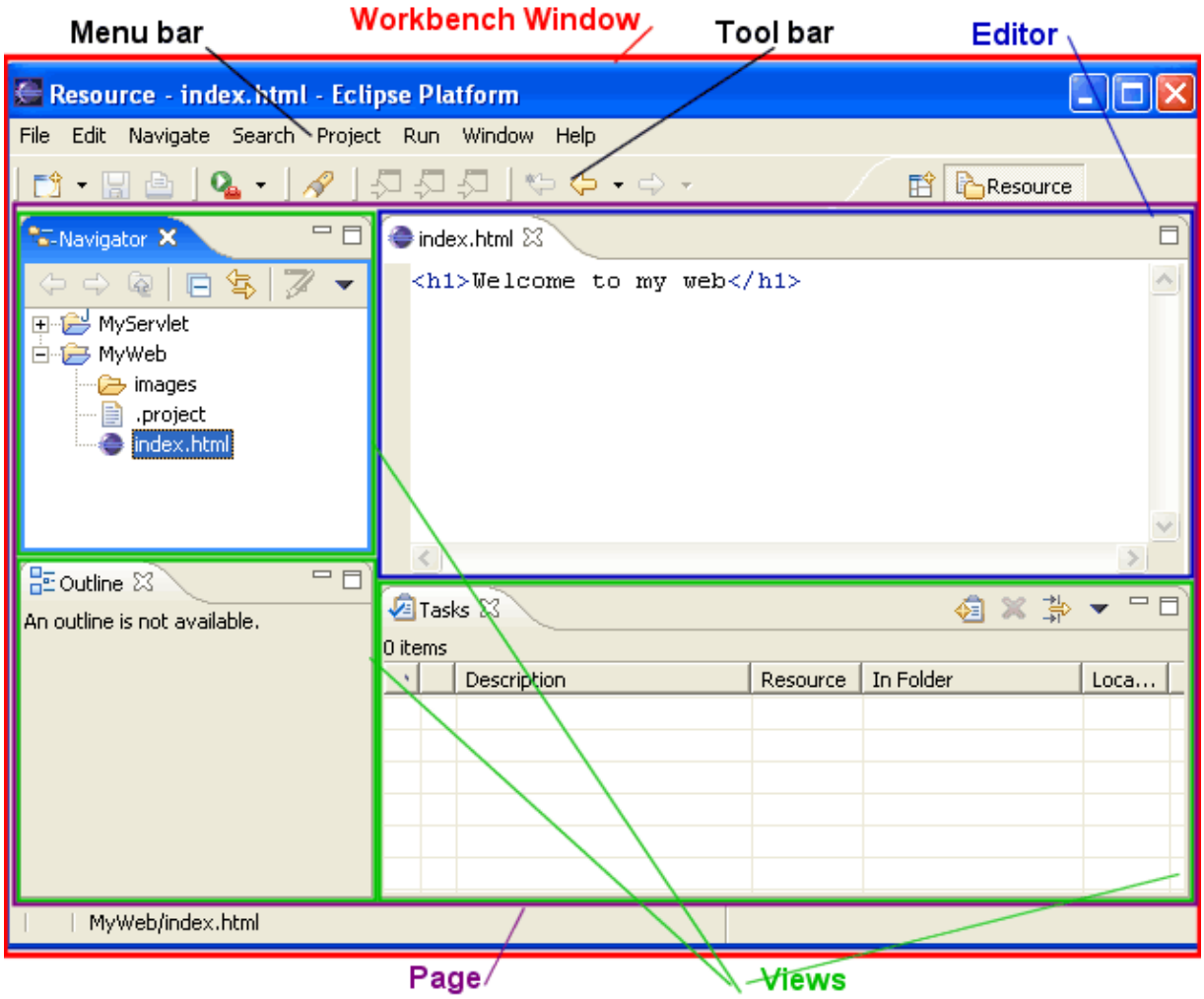
Fortunately, the platform will help you in identifying deadlocks. When the job manager detects a deadlock condition, it prints diagnostic information to the log describing the deadlock condition. Then it breaks the deadlock by temporarily granting access to the locks owned by a blocked job to other jobs that are waiting on them. It is important to carefully test any implementation involving multiple locks and fix any deadlock conditions that are reported by the platform.

Workbench under the covers

The workbench provides an extensive set of classes and interfaces for building complex user interfaces. Fortunately you don't need to understand all of them to do something simple. We'll start by looking at some concepts that are exposed in the workbench user interface and their corresponding structure under the covers.

Workbench

We've been using the term **workbench** loosely to refer to "that window that opens when you start the platform." Let's drill down a little and look at some of the visual components that make up the workbench.



For the rest of this discussion, when we use the term workbench, we will be referring to the workbench window (**IWorkbenchWindow**). The workbench window is the top−level window in a workbench. It is the frame that holds the menu bar, tool bar, status line, short cut bar, and pages. In general, you don't need to program to the workbench window. You just want to know that it's there.

> *Note: You can open multiple workbench windows; however each workbench window is a self−contained world of editors and views, so we'll just focus on a single workbench window.*

From the user's point of view, a workbench contains views and editors. There are a few other classes used to implement the workbench window.

Page

Inside the workbench window, you'll find one page (**IWorkbenchPage**) that in turn contains parts. Pages are an implementation mechanism for grouping parts. You typically don't need to program to the page, but you'll see it in the context of programming and debugging.

Perspectives

Perspectives provide an additional layer of organization inside the workbench page. A perspective defines an appropriate collection of views, their layout, and applicable actions for a given user task. Users can switch between perspectives as they move across tasks. From an implementation point of view, the user's active perspective controls which views are shown on the workbench page and their positions and sizes. Editors are not affected by a change in perspective.

Views and editors

Views and editors are where we move beyond implementation details into some common plug−in programming. When you add a visual component to the workbench, you must decide whether you want to implement a view or an editor. How do you decide this?

- A **view** is typically used to navigate a hierarchy of information, open an editor, or display properties for the active editor. For example, the **navigator** view allows you to navigate the workspace hierarchy. The **properties** and **outline** views show information about an object in the active editor. Any modifications that can be made in a view (such as changing a property value) are saved immediately.
- An **editor** is typically used to edit or browse a document or input object. Modifications made in an editor follow an open−save−close model, much like an external file system editor. The platform text editor and Java editor are examples of workbench editors.

In either case, you will be building your view or editor according to a common lifecycle.

- You implement a **createPartControl** method to create the SWT widgets that represent your visual component. You must determine which widgets to use and allocate any related UI resources needed to display your view or editor.
- When your view or editor is given focus, you'll receive a **setFocus** notification so that you can set the focus to the correct widget.
- When the view or editor is closed, you will receive a **dispose** message to signify that the view or editor is being closed. At this point the controls allocated in **createPartControl** have already been disposed for you, but you must dispose of any graphics resources (such as cursors, icons, or fonts) that you allocated for the view or editor.

Throughout this lifecycle, events will fire from the containing workbench page to notify interested parties about the opening, activation, deactivation, and closing of the views and editors.

Seem simple? It can be. That's the beauty of workbench views and editors. They're just widget holders, and can be as simple or complex as you need them to be. We saw the simplest of views earlier when we built a hello world view. Let's look at it again now that we've explained more about what's going on.

```
package org.eclipse.examples.helloworld;

import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.SWT;
import org.eclipse.ui.part.ViewPart;

public class HelloWorldView extends ViewPart {
   Label label;
   public HelloWorldView() {
   }
   public void createPartControl(Composite parent) {
      label = new Label(parent, SWT.WRAP);
      label.setText("Hello World");
   }
   public void setFocus() {
      // set focus to my widget.  For a label, this doesn't
      // make much sense, but for more complex sets of widgets
      // you would decide which one gets the focus.
   }
}
```

Notice that we didn't have to implement a **dispose()** method since we didn't do anything but create a label in the **createPartControl(parent)** method. If we had allocated any UI resources, such as images or fonts, we would have disposed of them here. Since we extended the **ViewPart** class, we inherit the "do nothing" implementation of **dispose()**.

org.eclipse.ui.views

A view is a workbench part that can navigate a hierarchy of information or display properties for an object. Only one instance of any given view is open in a workbench page. When the user makes selections or other changes in a view, those changes are immediately reflected in the workbench. Views are often provided to support a corresponding editor. For example, an **outline** view shows a structured view of the information in an editor. A **properties** view shows the properties of an object that is currently being edited.

The extension point **org.eclipse.ui.views** allows plug−ins to add views to the workbench. Plug−ins that contribute a view must register the view in their **plugin.xml** file and provide configuration information about the view, such as its implementation class, the category (or group) of views to which it belongs, and the name and icon that should be used to describe the view in menus and labels.

The interface for views is defined in **IViewPart**, but plug−ins can choose to extend the **ViewPart** class rather than implement an **IViewPart** from scratch.

We implemented a minimal view extension in the hello world example. Now we'll look at one that is aware of other workbench views and responds to user navigation and selection changes in the workbench. First, let's take a look at the declaration of the extension in the **plugin.xml**.

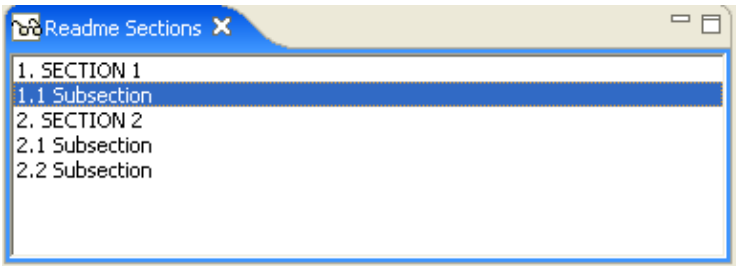
```
<extension
   point="org.eclipse.ui.views">
        <category
           id="org.eclipse.ui.examples.readmetool"
           name="%Views.category">
        </category>
        <view
           id="org.eclipse.ui.examples.readmetool.views.SectionsView"
           name="%Views.ReadmeSections"
```

```
            icon="icons/view16/sections.png"
            category="org.eclipse.ui.examples.readmetool"
            class="org.eclipse.ui.examples.readmetool.ReadmeSectionsView">
         </view>
</extension>
```

This should look pretty familiar. We see that a new view, **ReadmeSectionsView**, is contributed to the workbench. The **view id**, **name**, and **category** are specified as we've seen before. An **icon** is also provided for the view, using a path relative to the plug–in's installation directory.

Let's look at the **ReadmeSectionsView**. You can show any view in the workbench by choosing **Window–>Show View–>Other...** and selecting the view from the **Show View** list.

When we show the **ReadmeSectionsView**, a view with a list in it pops up. The list is empty unless we click on a file with an extension of **.readme**, in which case the list is populated with sections from the readme file.



How does the plug–in recognize the readme file and how did it know about selection changes? If we can track down the answers to these questions, we are well on our way to understanding how to build integrated workbench plug–ins.

We'll start with the familiar **createPartControl** method. As we saw in the Hello World example, this is where the widgets that represent a view are created. We'll ignore some of the code to get started.

```
public void createPartControl(Composite parent) {
   viewer = new ListViewer(parent);
   ...
   // add myself as a global selection listener
   getSite().getPage().addSelectionListener(this);

   // prime the selection
   selectionChanged(null, getSite().getPage().getSelection());
}
```

The view creates and stores a ListViewer and registers itself as a selection listener on its page. It obtains the page from an **IViewSite**, which contains information about the view's context, such as its workbench window, its containing page, and its plug–in. When we are notified of a selection change, what happens? The following code is executed:

```
public void selectionChanged(IWorkbenchPart part, ISelection sel) {
   //if the selection is a readme file, get its sections.
   AdaptableList input = ReadmeModelFactory.getInstance().getSections(sel);
   viewer.setInput(input);
}
```

It looks like the **ReadmeModelFactory** class is responsible for turning the selection into readme sections and these sections are input for the viewer that we created in the **createPartControl** method.

But how did the viewer populate its list widgets? For now, let's assume that once the viewer was told its input element, it knew how to populate its list widget with the information – it is a ListViewer, after all. If you must know right now what this viewer is all about, go to Viewers.

We still do not know how readme files are detected or where the file's section information comes from. A quick look at the **ReadmeModelFactory** sheds some light.

```
public AdaptableList getSections(ISelection sel) {
   // If sel is not a structured selection just return.
   if (!(sel instanceof IStructuredSelection))
      return null;
   IStructuredSelection structured = (IStructuredSelection)sel;

   //if the selection is a readme file, get its sections.
   Object object = structured.getFirstElement();
   if (object instanceof IFile) {
      IFile file = (IFile) object;
      String extension = file.getFileExtension();
      if (extension != null && extension.equals(IReadmeConstants.EXTENSION)) {
         return getSections(file);
      }
   }

   //the selected object is not a readme file
   return null;
}
```

We check the selection to see if it is a structured (multiple) selection. (The concept of a structured selection comes from JFace viewers.) For the first object in the selection, we check to see whether it is a file (**IFile**) resource. If it is, we check its extension to see if it matches the "**.readme**" extension. Once we know we have a readme file, we can use other methods to parse the sections. You can browse the rest of **ReadmeModelFactory**, **MarkElement**, and **DefaultSectionsParser** for the details about the file parsing.

We've covered a lot of common workbench concepts by studying this extension. Now we'll move on to some other workbench extensions and examine how your plug–in can further contribute to the workbench UI.

Viewers

Why would you ever want to use a viewer when we have already seen that workbench UI contributions like views, editors, wizards, and dialogs can be implemented directly with SWT widgets?

Viewers allow you to create widgets while still using your model objects. If you use an SWT widget directly, you have to convert your objects into the strings and images expected by SWT. Viewers act as adapters on SWT widgets, handling the common code for handling widget events that you would otherwise have to implement yourself.

We first saw a viewer in the readme tool's view contribution, inside the **ReadmeSectionsView**.

```
public void createPartControl(Composite parent) {
   viewer = new ListViewer(parent);
   ...
}
```

Note: Viewers can be used to provide the implementation for both workbench views and

*editors. The term viewer does not imply that they are only useful for implementing views. For example, the **TextViewer** is used in the implementation in many of the workbench and plug−in editors.*

Standard viewers

JFace provides viewers for most of the non−trivial widgets in SWT. Viewers are most commonly used for list, tree, table, and text widgets.

Each viewer has an associated SWT widget. This widget can be created implicitly by supplying the parent **Composite** in a convenience viewer constructor, or explicitly by creating it first and supplying it to the viewer in its constructor.

List−oriented viewers

Lists, trees, and tables share many common capabilities from a user's point of view, such as population with objects, selection, sorting, and filtering.

These viewers keep a list of domain objects (called **elements**) and display them in their corresponding SWT widget. A list viewer knows how to get a text label from any element in the list. It obtains the label from an **ILabelProvider** which can be set on the viewer. List viewers know how to map from the widget callbacks back into the world of elements known by the viewer client.

Clients that use a plain SWT widget have to operate at the SWT level − where items are strings and events often relate to an index within the list of strings. Viewers provide higher level semantics. Clients are notified of selections and changes to the list using the elements they provided to the viewer. The viewer handles all the grunt work for mapping indexes back to elements, adjusting for a filtered view of the objects, and re−sorting when necessary.

Filtering and sorting capability is handled by designating a viewer sorter (**ViewerSorter**) and/or viewer filter (**ViewerFilter**) for the viewer. (These can be specified for tree and table viewers in addition to list viewers.) The client need only provide a class that can compare or filter the objects in the list. The viewer handles the details of populating the list according to the specified order and filter, and maintaining the order and filter as elements are added and removed.

Viewers are not intended to be extended by clients. To customize a viewer, you can configure it with your own content and label providers.

A **ListViewer** maps elements in a list to an SWT **List** control.

A **TreeViewer** displays hierarchical objects in an SWT **Tree** widget. It handles the details for expanding and collapsing items. There are several different kinds of tree viewers for different SWT tree controls (plain tree, table tree, checkbox tree).

A **TableViewer** is very similar to a list viewer, but adds the ability to view multiple columns of information for each element in the table. Table viewers significantly extend the function of the SWT table widget by introducing the concept of editing a cell. Special cell editors can be used to allow the user to edit a table cell using a combo box, dialog, or text widget. The table viewer handles the creation and placement of these widgets when needed for user editing. This is done using the **CellEditor** classes, such as **TextCellEditor** and **CheckboxCellEditor**. A virtual table, only populated when viewed, the table viewer only runs a designated

number of results regardless of what is actually created. The database "lazily" requests JIT and will only query a predetermined number at a time.

Text viewer

Text widgets have many common semantics such as double click behavior, undo, coloring, and navigating by index or line. A **TextViewer** is an adapter for an SWT **StyledText** widget. Text viewers provide a document model to the client and manage the conversion of the document to the styled text information provided by the text widget.

Text viewers are covered in more detail in Workbench Editors.

Viewer architecture

To understand a viewer, you must become familiar with the relationship between a viewer's input element, its contents, its selection, and the information actually displayed in the widget that it is manipulating.

Input elements

An **input element** is the main object that the viewer is displaying (or editing). From the viewer's point of view, an input element can be any object at all. It does not assume any particular interface is implemented by the input element. (We'll see why in a moment when we look at content providers.)

A viewer must be able to handle a change of input element. If a new input element is set into a viewer, it must repopulate its widget according to the new element, and disassociate itself from the previous input element. The semantics for registering as a listener on an input element and populating the widget based on the element are different for each kind of viewer.

Content viewers

A **content viewer** is a viewer that has a well defined protocol for obtaining information from its input element. Content viewers use two specialized helper classes, the **IContentProvider** and **ILabelProvider**, to populate their widget and display information about the input element.

IContentProvider provides basic lifecycle protocol for associating a content provider with an input element and handling a change of input element. More specialized content providers are implemented for different kinds of viewers. The most common content provider is **IStructuredContentProvider**, which can provide a list of objects given an input element. It is used in list−like viewers, such as lists, tables, or trees. In general, the content provider knows how to map between the input element and the expected viewer content.

ILabelProvider goes a step further. Given the content of a viewer (derived from the input element and content provider), it can produce the specific UI elements, such as names and icons, that are needed to display the content in the viewer. Label providers can aid in saving icon resources since they can ensure the same instance of the icon is used for all like types in a viewer.

> *Note: Instances of particular content and label providers are not intended to be shared across multiple viewers. Even if all your viewers use the same type of content or label provider, each viewer should be initialized with its own instance of the provider class. The provider life cycle protocol is designed for a 1−to−1 relationship between a provider and its viewer.*

Input elements, content providers, and label providers allow viewers to hide most of the implementation details for populating widgets. Clients of a viewer need only worry about populating a viewer with the right kind of input and content provider. The label provider must know how to derive the UI information from the viewer content.

A label provider can show more than just text and an image. JFace provides several classes and interfaces to support the most popular extra functionality. The following classes are supported by the TableViewer, AbstractTreeViewer and TableTreeViewer.

- **IColorProvider** . If an label provider implements IColorProvider it will set the foreground and background color of an item in the viewer when it returns a color. It is recommended that unless system colors are used that these colors are cached to minimize the amount of system resources used.
- **IFontProvider** . If an label provider implements IFontProvider it will set the font of an item in the viewer when it returns a color. Fonts should also be cached. **Control#getFont()** should be called as little as possible as this will create a new instance of Font whenever it is called.
- **ILabelDecorator** An ILabelDecorator is an object that can take an image or text and add adornments to them.
- **DecoratingLabelProvider** The DecoratingLabelProvider is a compound object that takes both a label provider and an ILabelDecorator. This allows a label provider to hook into a decorator mechanism such as the one provided in the Workbench.
- **IViewerLabelProvider** The IViewerLabelProvider is a label provider that allows for building of labels by an external object such as a decorator. The DecoratingLabelProvider is an IViewerLabelProvider.
- **IDelayedLabelDecorator** The IDelayedLabelDecorator is an ILabelDecorator that supports decoration that is delayed (such as an IDecoratorManager that decorates in a Thread). IDecoratorManagers are IDelayedLabelDecorators. You can get the Workbench **IDecoratorManager** IDecoratorManager by calling **IWorkbench#getDecoratorManager()** .
- **IColorDecorator** An IColorDecorator is an object that can support decorating foreground and background colors. This is supported internally in the Workbench DecoratorManager.
- **IFontDecorator** An IFontDecorator is an object that can support decorating fonts. This is supported internally in the Workbench DecoratorManager.

It is possible to affect the colors of a view in two ways as of Eclipse 3.1 – either with its own label provider or with a decorator that sets colors and fonts. Generally it is better to use the color and font support in label providers as decorators affect every view that show a particular type. If you do use a color or font decorator make sure it's values can be set in the Colors and Fonts preference page.

Viewers and the workbench

The flexibility provided by viewers, content providers, and label providers can be demonstrated by looking at how the workbench uses them.

The **WorkbenchContentProvider** is a structured content provider that obtains contents from an input element by asking for its children. The concept of adapters is used again in order to implement generic function. When asked for the list of elements from its input element, the **WorkbenchContentProvider** obtains an **IWorkbenchAdapter** for the input element. If an **IWorkbenchAdapter** has been registered for the input element, then the content provider can safely assume that the element can be queried for its children. **WorkbenchContentProvider** also does the work needed to keep its viewer up to date when the workspace changes.

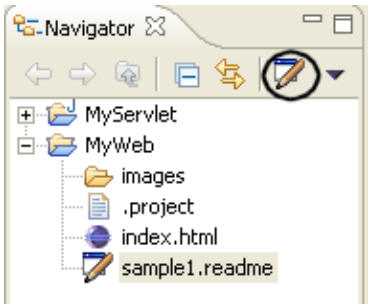
The **WorkbenchLabelProvider** is a label provider that obtains an **IWorkbenchAdapter** from an object in order to find its text and image. The concept of a label provider is particularly helpful for workbench objects because it allows a single label provider to cache images that are commonly used in a viewer. For example, once the **WorkbenchLabelProvider** obtains an image to use for an **IProject**, it can cache that image and use it for all **IProject** objects shown in the viewer.

By defining a common adapter, **IWorkbenchAdapter**, and registering it for many of the platform types, we make it possible for these types to be represented correctly in many of the common viewers and the workbench views that contain them.

org.eclipse.ui.viewActions

It is common for plug−ins to contribute behavior to views that already exist in the workbench. This is done through the **org.eclipse.ui.viewActions** extension point. This extension point allows plug−ins to contribute menu items, submenus and tool bar entries to an existing view's local pull−down menu and local tool bar.

You may have noticed an item in the navigator's local tool bar that becomes enabled whenever a readme file is selected. This item also appears in the navigator's local pull−down menu. These actions appear because the readme tool plug−in contributes them using the **viewActions** extension.



The relevant **plugin.xml** contribution is below.

```
<extension
    point = "org.eclipse.ui.viewActions">
      <viewContribution
         id="org.eclipse.ui.examples.readmetool.vc1"
         targetID="org.eclipse.ui.views.ResourceNavigator">
           <action id="org.eclipse.ui.examples.readmetool.va1"
              label="%PopupMenu.ResourceNav.label"
              menubarPath="additions"
              toolbarPath="additions"
              icon="icons/obj16/editor.png"
              tooltip="%PopupMenu.ResourceNav.tooltip"
              helpContextId="org.eclipse.ui.examples.readmetool.view_action_context"
              class="org.eclipse.ui.examples.readmetool.ViewActionDelegate"
              enablesFor="1">
                <selection class="org.eclipse.core.resources.IFile" name="*.readme"/>
           </action>
      </viewContribution>
 </extension>
```

A view contribution with a unique id is specified. The view to which we are adding the action is specified in the **targetID**. We are contributing to the resource navigator's menu. We specify the label and the menu bar and tool bar locations for the new action. (For a complete discussion of menu and toolbar locations, see Menu and toolbar paths).

We also specify the conditions under which the action should be enabled. You can see that this action will be enabled when there is one selection (**enablesFor="1"**) of type **IFile (class="org.eclipse.core.resources.IFile")**, whose name has "**.readme**" in the file extension (**name="*.readme"**). Sure enough, that's exactly what happens when you click around in the resource navigator.

The information in the **plugin.xml** is all that's needed to add items to menus and tool bars since plug−in code will only run when the action is actually selected from the menu or toolbar. To provide the action behavior, the implementation class specified in the **plugin.xml** must implement the **IViewActionDelegate** interface.

In this example, the readme plug−in supplies **ViewActionDelegate** to implement the action. If you browse this class you will see that it includes methods for remembering its view, handling selection changes, and invoking its action. When invoked the action itself simply launches a dialog that announces it was executed.

```
public void run(org.eclipse.jface.action.IAction action) {
        MessageDialog.openInformation(view.getSite().getShell(),
                MessageUtil.getString("Readme_Editor"),
                MessageUtil.getString("View_Action_executed"));
}
```

Although this action is simple, we can imagine how using selections and more functional dialogs could make this action do something more interesting.

org.eclipse.ui.editors

An editor is a workbench part that allows a user to edit an object (often a file). Editors operate in a manner similar to file system editing tools, except that they are tightly integrated into the platform workbench UI. An editor is always associated with an input object (**IEditorInput**). You can think of the input object as the document or file that is being edited. Changes made in an editor are not committed until the user saves them.

Only one editor can be open for any particular editor input in a workbench page. For example, if the user is editing **readme.txt** in the workbench, opening it again in the same perspective will activate the same editor. (You can open another editor on the same file from a different workbench window or perspective). Unlike views, however, the same editor type, such as a text editor, may be open many times within one workbench page for different inputs.

The workbench extension point **org.eclipse.ui.editors** is used by plug−ins to add editors to the workbench. Plug−ins that contribute an editor must register the editor extension in their **plugin.xml** file, along with configuration information for the editor. Some of the editor information, such as the implementation **class** and the **name** and the **icon** to be used in the workbench menus and labels, is similar to the view information. In addition, editor extensions specify the file extensions or file name patterns of the file types that the editor understands. Editors can also define a **contributorClass**, which is a class that adds actions to workbench menus and tool bars when the editor is active.

The interface for editors is defined in **IEditorPart**, but plug−ins can choose to extend the **EditorPart** class rather than implement an **IEditorPart** from scratch.

*Note: An editor extension can also be configured to launch an external program or to call pre−existing java code. In this discussion, we are focusing on those editors that are actually tightly integrated with the workbench and are implemented using **IEditorPart**.*

The readme tool provides a custom editor primarily for the purpose of contributing its own content outliner page to the workbench outline view.

The configuration for the editor extension is defined as follows.

```
<extension
    point = "org.eclipse.ui.editors">
        <editor
           id = "org.eclipse.ui.examples.readmetool.ReadmeEditor"
           name="%Editors.ReadmeEditor"
           icon="icons/obj16/editor.png"
           class="org.eclipse.ui.examples.readmetool.ReadmeEditor"
           extensions="readme"
           contributorClass="org.eclipse.ui.examples.readmetool.ReadmeEditorActionBarContributor"
        </editor>
</extension>
```

We see the familiar configuration markup for **id**, **name**, **icon**, and **class**. The **extensions** attribute describes the file types that the editor understands. (You could also specify **filenames** if you need to be more specific.) The **class** implements the editor, and the **contributorClass** is responsible for providing editor−related actions. Let's look at the contributor in more detail.

Editor action contributors

The contributor class adds editor−related actions to the workbench menu and toolbar. It must implement the **IEditorActionBarContributor** interface. The contributor is separate from the editor itself since any given workbench page can have multiple editors of the same type. A single contributor is shared by all the editors of a specific type, rather than having each instance of an editor create actions and images.

In **ReadmeEditorActionBarContributor**, we contribute three actions, "Editor Action1," "Editor Action2," and "Editor Action3." These are set up in the constructor.

```
   public ReadmeEditorActionBarContributor() {
        ...
        action1 = new EditorAction(MessageUtil.getString("Editor_Action1"));
        action1.setToolTipText(MessageUtil.getString("Readme_Editor_Action1"));
        action1.setDisabledImageDescriptor(ReadmeImages.EDITOR_ACTION1_IMAGE_DISABLE);
        action1.setImageDescriptor(ReadmeImages.EDITOR_ACTION1_IMAGE_ENABLE);
        ...
        action2 = new RetargetAction(IReadmeConstants.RETARGET2, MessageUtil.getString("Editor_Ac
        action2.setToolTipText(MessageUtil.getString("Readme_Editor_Action2"));
        action2.setDisabledImageDescriptor(ReadmeImages.EDITOR_ACTION2_IMAGE_DISABLE);
        action2.setImageDescriptor(ReadmeImages.EDITOR_ACTION2_IMAGE_ENABLE);
        ...
        action3 = new LabelRetargetAction(IReadmeConstants.LABELRETARGET3, MessageUtil.getString(
        action3.setDisabledImageDescriptor(ReadmeImages.EDITOR_ACTION3_IMAGE_DISABLE);
        action3.setImageDescriptor(ReadmeImages.EDITOR_ACTION3_IMAGE_ENABLE);
        ...
   }
```

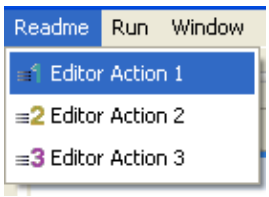
The names and icons for the actions are set up in the code rather than in the **plugin.xml**. (We'll ignore the differences in the action classes for now until we look at retargetable actions.)

Note how similar the action information is to the **viewActions** information we saw in the markup for the view action. The actions are set up in code since we have to manage the sharing of the actions among different instances of the same editor. When the actions are created in the constructor, they are independent of any particular instance of the editor.

When an editor becomes active and it has actions that need to be installed in the workbench menus and tool bar, the **setActiveEditor** message is sent to the contributor. The contributor connects the editor actions to a specific editor.

```
public void setActiveEditor(IEditorPart editor) {
    ...
    action1.setActiveEditor(editor);
    ...
}
```

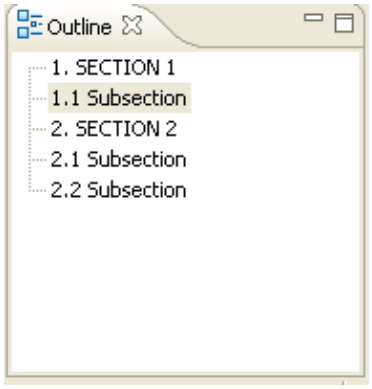
As you can see, the actions show up in the workbench menu and tool bar when a readme editor is active.



These menu and tool bar items are only shown when the editor is active. The location for the menu and tool bar items can be specified as described in Menu and toolbar paths.

Editors and content outliners

The readme editor itself, **ReadmeEditor**, is not very complicated. It extends the **TextEditor** class so that it can contribute a customized content outliner page to the outline view when a readme file is being edited. It does not change any behavior inside the text editor.



Editors often have corresponding content outliners that provide a structured view of the editor's contents and assist the user in navigating through the contents of the editor. See Content outliners for more detail.

We'll look at the implementation of text editors in Text editors and platform text.

Contributing new retargetable actions

The workbench is not the only plug−in that can create retargetable actions. Your plug−in can define its own retargetable action, so that views and editors within your plug−in can share the same menu actions. There are two ways to contribute retargetable actions from your plug−in:

- An **editor** can define a retargetable action for which the editor and related views can hook handlers. The action is only available in the menu bar when the editor is open.
- An **action set** can define a retargetable action for which editors and views can hook handlers. The action will be visible as long as the action set is visible, but it will only be enabled if the active part has hooked a handler for the action.

This mechanism is useful for providing tight integration between editors and related views. For example, a content outline view can implement a handler for an action defined by its associated editor.

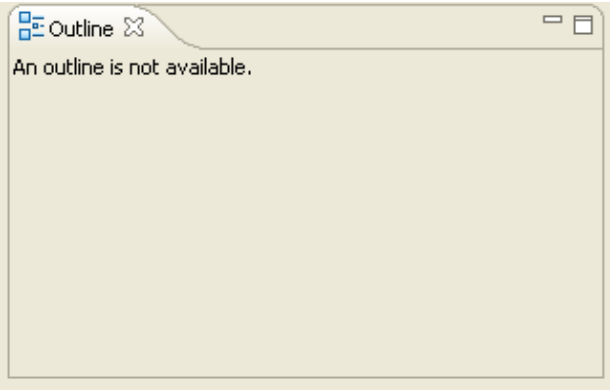
The readme tool example demonstrates both of these techniques.

Content outliners

Editors often have corresponding **content outliners** that provide a structured view of the editor contents and assist the user in navigating through the contents of the editor.

The workbench provides a standard **Outline** view for this purpose. The workbench user controls when this view is visible using the **Window > Show View** menu.

Since the generic **TextEditor** doesn't know anything about the structure of its text, it cannot provide behavior for an interesting outline view. Therefore, the default **Outline** view, shown below, doesn't do much.



Editors in the text framework can supply their own content outliner page to the outline view. The outliner for an editor is specified when the workbench requests an adapter of type **IContentOutlinePage**.

```
public Object getAdapter(Class required) {
        if (IContentOutlinePage.class.equals(required)) {
                if (fOutlinePage == null) {
                  JavaContentOutlinePage fOutlinePage= new JavaContentOutlinePage(getDocumentProvider(), this);
                        if (getEditorInput() != null)
                                fOutlinePage.setInput(getEditorInput());
```

```
                }
                return fOutlinePage;
        }
        return super.getAdapter(required);
}
```

A content outliner page must implement **IContentOutlinePage**. This interface combines the ability to notify selection change listeners (**ISelectionProvider**) with the behavior of being a page in a view (**IPage**). Content outliners are typically implemented using JFace viewers. The default implementation of a content outliner (**ContentOutlinePage**) uses a JFace tree viewer to display a hierarchical representation of the outline. This representation is suitable for many structured outliners, including **JavaContentOutlinePage**.

Let's take a look at the implementation of the page. When the outline page is created by the editor in the snippet above, its input element is set to the editor's input element. This input can often be passed directly to the outline page's viewer, as is done below.

```
public void createControl(Composite parent) {

        super.createControl(parent);

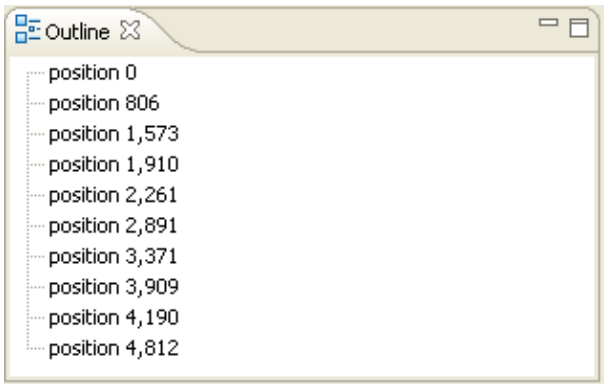
        TreeViewer viewer= getTreeViewer();
        viewer.setContentProvider(new ContentProvider());
        viewer.setLabelProvider(new LabelProvider());
        viewer.addSelectionChangedListener(this);

        if (fInput != null)
                viewer.setInput(fInput);
}
```

The tree viewer creation is inherited from **ContentOutlinePage**. The standard label provider is used. The content provider is provided inside **JavaContentOutlinePage** and is responsible for parsing the editor input into individual segments whenever it changes.

```
                public void inputChanged(Viewer viewer, Object oldInput, Object newInput) {
                        ...
                        if (newInput != null) {
                                IDocument document= fDocumentProvider.getDocument(newInput);
                                if (document != null) {
                                        document.addPositionCategory(SEGMENTS);
                                        document.addPositionUpdater(fPositionUpdater);
                                        parse(document);
                                }
                        }
                }
```

The text is parsed into ranges, called segments, within the document. These segments are displayed by name in the outline view.



When the selection changes, the selected segment is retrieved. Its offsets are used to set the highlight range in the editor.

```
public void selectionChanged(SelectionChangedEvent event) {

        super.selectionChanged(event);

        ISelection selection= event.getSelection();
        if (selection.isEmpty())
                fTextEditor.resetHighlightRange();
        else {
                Segment segment= (Segment) ((IStructuredSelection) selection).getFirstElement();
                int start= segment.position.getOffset();
                int length= segment.position.getLength();
                try {
                        fTextEditor.setHighlightRange(start, length, true);
                } catch (IllegalArgumentException x) {
                        fTextEditor.resetHighlightRange();
                }
        }
}
```

Text editors and platform text

The platform text facility is used to implement the default text editor for the workbench. The interface for text editing is defined in **ITextEditor** as a text specific extension of **IEditorPart**.

The implementation of **ITextEditor** in the platform is structured in layers. **AbstractTextEditor** is the core class of the framework for extending the editor to support source code style editing of text. This framework is defined in **org.eclipse.ui.texteditor**.

The concrete implementation class **TextEditor** defines the behavior for the standard platform text editor. It is defined in the package **org.eclipse.ui.editors.text**.

The text editor framework provides a model−independent editor that supports the following features:

- presentation and user modification of text
- standard text editing operations such as cut/copy/paste, find/replace
- support for context and pulldown menus
- visual presentation of text annotations in rulers or as squigglies in the text
- automatic update of annotations as the user edits text
- presentation of additional information such as line numbers

- syntax highlighting
- content assist
- text outlining pages that show the hierarchical structure of the text
- context sensitive behavior
- hover support over rulers and text
- key binding contexts
- preference handling

We will explore how these features can be implemented in an editor by studying the **org.eclipse.ui.examples.javaeditor** example. This example shows how complex features like text coloring, hover help, and automatic indenting can be implemented.

In discussing these features we will be moving between the abstract framework, the platform editor **TextEditor**, and the example's subclass, **JavaEditor**.

org.eclipse.ui.editorActions

We've just seen how editors can contribute their own actions to the workbench menus and tool bar when they become active. The **org.eclipse.ui.editorActions** extension point allows a plug−in to add to the workbench menus and tool bar when another plug−in's editor becomes active.

In the readme example, the plug−in uses the **editorActions** extension point to contribute additional actions to the menu contributed by the readme editor. The definition in our **plugin.xml** should look pretty familiar by now.

```
<extension
    point = "org.eclipse.ui.editorActions">
      <editorContribution
         id="org.eclipse.ui.examples.readmetool.ec1"
         targetID="org.eclipse.ui.examples.readmetool.ReadmeEditor">
           <action id="org.eclipse.ui.examples.readmetool.ea1"
              label="%Editors.Action.label"
              toolbarPath="ReadmeEditor"
              icon="icons/obj16/editor.png"
              tooltip="%Editors.Action.tooltip"
              class="org.eclipse.ui.examples.readmetool.EditorActionDelegate"
              />
      </editorContribution>
 </extension>
```

Similar to a view action, the extension must specify the **targetID** of the editor to which it is contributing actions. The action itself is very similar to a view action (**id**, **label**, **icon**, **toolbarPath**, ...), except that the specified class must implement **IEditorActionDelegate**.

Note that a menu bar path is not specified in this markup. Therefore, the action will appear in the workbench tool bar when the editor is active, but not in the workbench menu bar. (See Menu and toolbar paths for a discussion of toolbar and menu paths.)

Sure enough, when the editor is active, we see our editor action on the tool bar next to the actions that were contributed by the editor itself.

The readme tool supplies **EditorActionDelegate** to implement the action. This class is very similar to the view action delegate we saw earlier.

```
public void run(IAction action) {
        MessageDialog.openInformation(editor.getSite().getShell(),
                MessageUtil.getString("Readme_Editor"),
                MessageUtil.getString("Editor_Action_executed"));
}
```

org.eclipse.ui.popupMenus

The **org.eclipse.ui.popupMenus** extension point allows a plug–in to contribute to the popup menus of other views and editors.

You can contribute an action to a specific popup menu by its id (**viewerContribution**), or by associating it with a particular object type (**objectContribution**).

- An **objectContribution** will cause the menu item to appear in popup menus for views or editors where objects of the specified type are selected.
- A **viewerContribution** will cause the menu item to appear in the popup menu of a view or editor specified by id in the markup.

The readme tool defines both. Let's look at the object contribution first.

```
<extension point = "org.eclipse.ui.popupMenus">
     <objectContribution
        id="org.eclipse.ui.examples.readmetool"
        objectClass="org.eclipse.core.resources.IFile"
            nameFilter="*.readme">
            <action id="org.eclipse.ui.examples.readmetool.action1"
               label="%PopupMenus.action"
               icon="icons/ctool16/openbrwsr.png"
               menubarPath="additions"
               helpContextId="org.eclipse.ui.examples.readmetool.open_browser_action_context"
               class="org.eclipse.ui.examples.readmetool.PopupMenuActionDelegate"
               enablesFor="1">
            </action>
         </objectContribution>
         ...
```

Object contribution

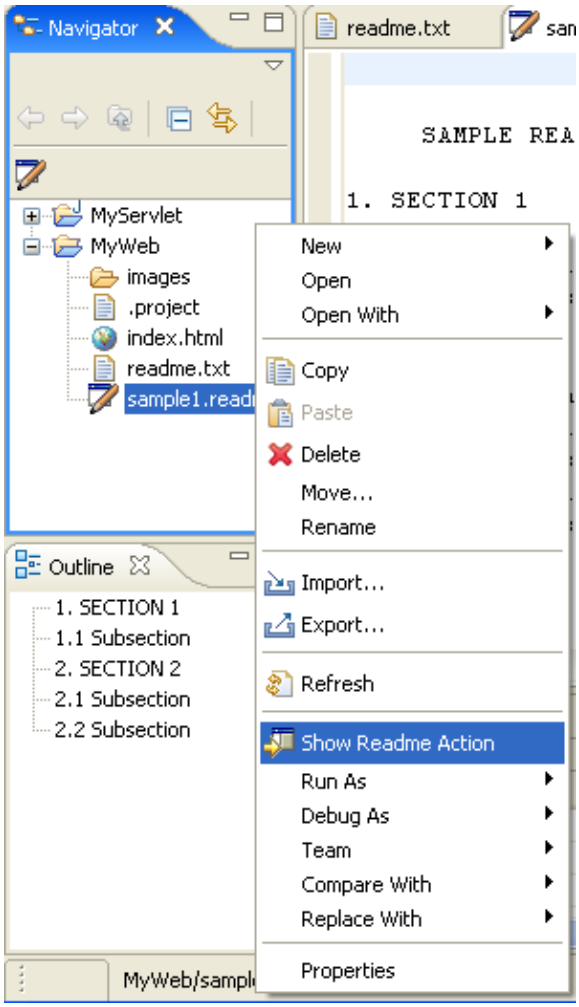
The action "Show Readme Action" is contributed for the object class **IFile**. This means that any view containing **IFile** objects will show the contribution if **IFile** objects are selected. We see that the selection criteria is restricted further with a name filter (**nameFilter="*.readme"**) and for single selections (**enablesFor="1"**). As we've discussed before, the registration of this menu does not run any code from our plug–in until the menu item is actually selected.

When the menu item is selected, the workbench will run the specified class. Since the popup is declared as an **objectContribution**, the supplied class must implement **IObjectActionDelegate**.

The action is implemented in **PopupMenuActionDelegate**.

```
public void run(IAction action) {
   MessageDialog.openInformation(
      this.part.getSite().getShell(),
      "Readme Example",
      "Popup Menu Action executed");
}
```

We can see the popup menu contribution when we select a readme file from the resource navigator.



Viewer contribution

A viewer contribution is used to contribute to a specific view or editor's popup menu by using its id. Here is the readme tool's viewer contribution:

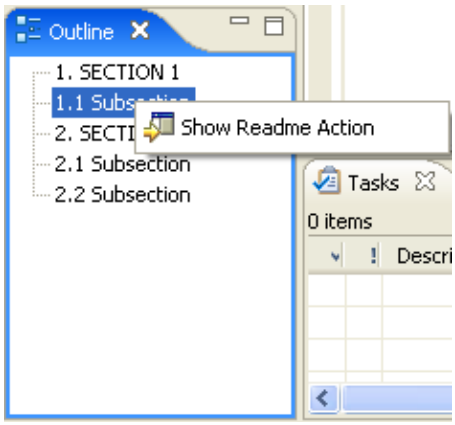
```
...
<viewerContribution
  id="org.eclipse.ui.examples.readmetool2"
  targetID="org.eclipse.ui.examples.readmetool.outline">
      <action id="org.eclipse.ui.examples.readmetool.action1"
         label="%PopupMenus.action"
         icon="icons/ctool16/openbrwsr.png"
         menubarPath="additions"
```

```
                helpContextId="org.eclipse.ui.examples.readmetool.open_browser_action_context"
                class="org.eclipse.ui.examples.readmetool.ViewActionDelegate">
            </action>
        </viewerContribution>
</extension>
```

*Note: The name **viewerContribution** is somewhat misleading, as it does not relate to JFace viewers. A better name would be **popupMenuContribution**.*

When the extension is a **viewerContribution**, the supplied class must implement the **IEditorActionDelegate** or **IViewActionDelegate** interface, depending on whether the action is contributed to an editor's or view's popup menu.

The **targetID** specifies the view whose popup menu will be altered. In this case, we are adding an action to one of the readme tool views, the outliner. The action itself is similar to others that we've seen. We specify the **id**, **label**, and **icon** of the action, and the **path** within the popup for our contribution. The action will be shown only in the readme outline view's popup menu.

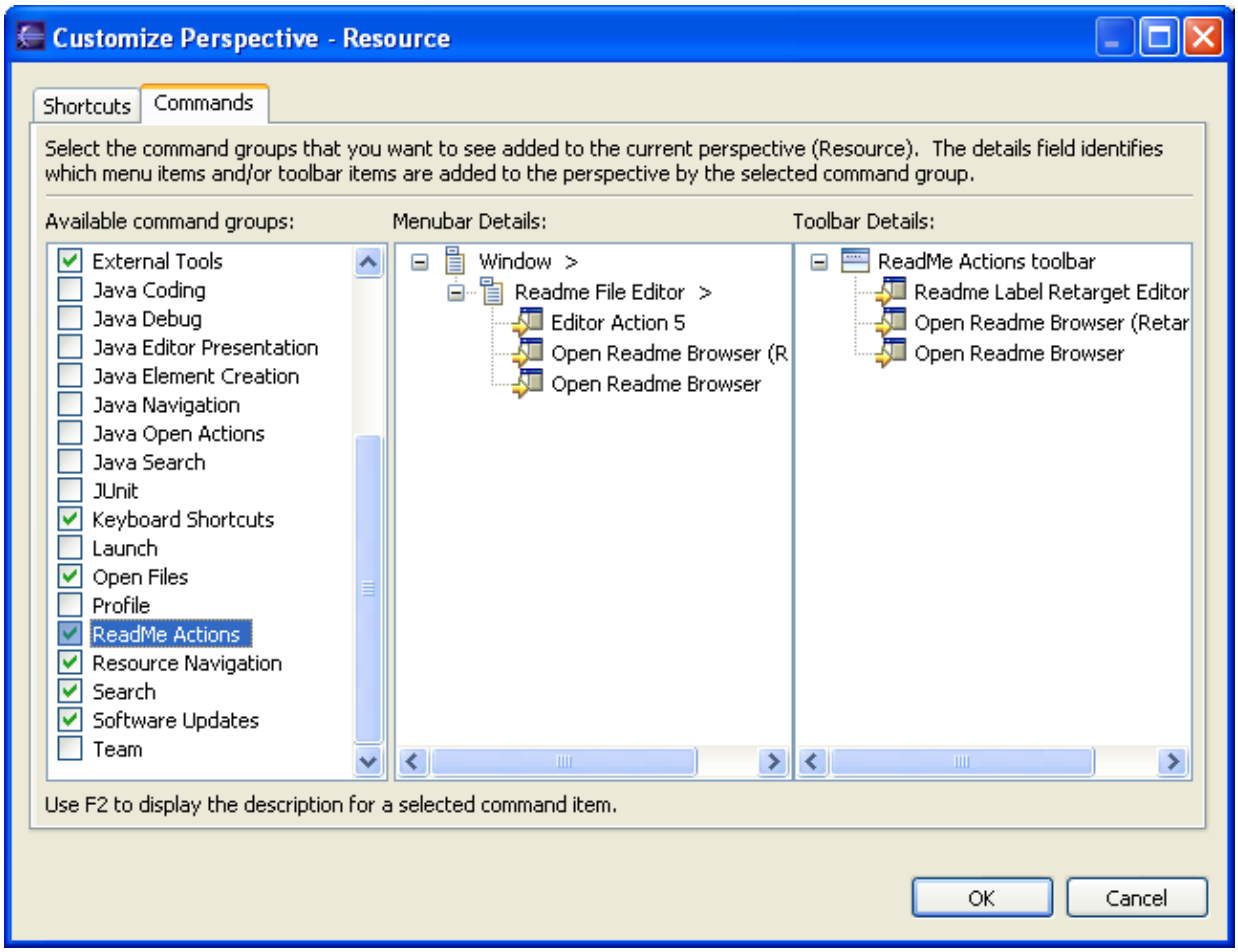


The interfaces required to contribute a **viewerContribution** to the **popupMenus** extension point are the same as those required by the **viewActions** and **editorActions** extension points. If you want to contribute the same action to the popup menu and the local menu of a view or editor, you can use the same class for both extensions.

org.eclipse.ui.actionSets

Your plug−in can contribute menus, menu items, and tool bar items to the workbench menus and toolbar by using the **org.eclipse.ui.actionSets** extension point. In order to reduce the clutter that would be caused by having every plug−in's menu contributions shown at once, the contributions are grouped into action sets which can be made visible by user preference.

You can see which action sets have been contributed to your workbench by choosing **Window−>Customize Perspective...** from the workbench menu. This option will show you a dialog that lists action sets as groups of commands. A checkmark by a command group means that the menu and tool bar actions are visible in the workbench. You can select the name of the command group to see the list of available menu and toolbar actions to the right. The figure below shows the list of command groups available in our workbench. (Your workbench may look different depending on which plug−ins you have installed and which perspective is active.)



The readme tool uses an action set to contribute several different "Open Readme Browser" actions to the workbench menu. (We contributed a similar action to the popup menu of the resource navigator.) The markup follows:

```
<extension point = "org.eclipse.ui.actionSets">
          <actionSet id="org_eclipse_ui_examples_readmetool_actionSet"
                  label="%ActionSet.name"
                  visible="true">
                  <menu id="org_eclipse_ui_examples_readmetool"
                          label="%ActionSet.menu"
                          path="window/additions">
                          <separator name="slot1"/>
                          <separator name="slot2"/>
                          <separator name="slot3"/>
                  </menu>
                  <action id="org_eclipse_ui_examples_readmetool_readmeAction"
                          menubarPath="window/org_eclipse_ui_examples_readmetool/slot1"
                          toolbarPath="readme"
                          label="%ReadmeAction.label"
                          tooltip="%ReadmeAction.tooltip"
                          helpContextId="org.eclipse.ui.examples.readmetool.open_browser_action_
                          icon="icons/ctool16/openbrwsr.png"
                          class="org.eclipse.ui.examples.readmetool.
                          enablesFor="1">
                          <selection class="org.eclipse.core.resources.IFile"
                                  name="*.readme">
                          </selection>
```

```
                </action>
                ...
            </actionSet>
    </extension>
```

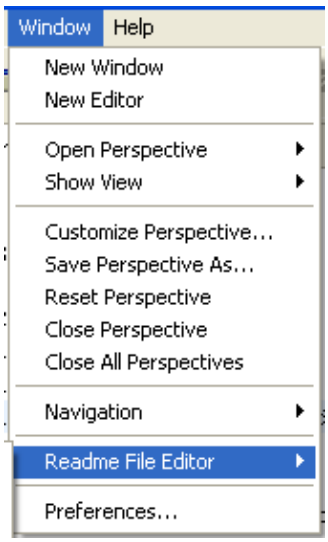
Wow, there's a lot going on here! Let's take it a step at a time, looking only at the first action for now.

First, the action set is declared and given a **label**. The label "ReadMe Actions" (defined for **%ActionSet.name** key in the plug–in's properties file) is used to display the action set in the dialog shown above. Since we set **visible** to true, the workbench will initially have the action set checked in the action set list and the actions will be visible.

The rest of the action set declaration is concerned with defining the menu in which the actions appears and the actions themselves.

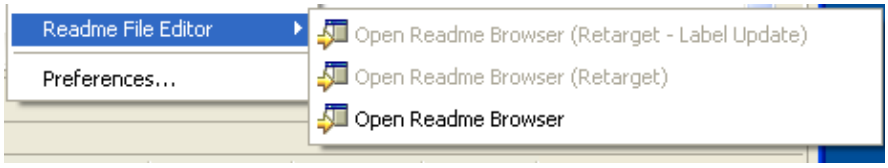
We define a menu whose **label** appears in the workbench menus. The menu's **path** tells the workbench to place the new menu in the **additions** slot of the **window** menu. (For a discussion of menu paths and slots, see Menu and toolbar paths.) We also define some slots in our new menu so that actions can be inserted at specific locations in our menu.

This markup alone is enough to cause the menu to appear in the workbench **Window** menu.

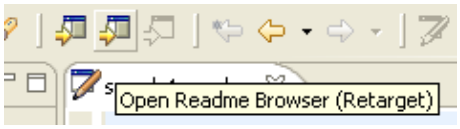


Next, we define the actions themselves.

The action definition (**id**, **label**, **icon**, **class**) is similar to the other actions we've seen in views, editors, and popups. We'll focus here on what's different: where does the action go? We use **menubarPath** and **toolbarPath** to indicate its location. First, we define the **menubarPath** to add the action to a slot in the menu that we just defined (**"window/org_eclipse_ui_examples_readmetool/slot1"**).



Then, we define a new **toolbarPath** to insert our actions in the workbench tool bar. Since we've defined a new tool path, **"readme"**, the workbench will decide where it goes relative to other plug–in's toolbar contributions.



What happens when the action is selected by the user? The action is implemented by the class specified in the **class** attribute. The action **class** must implement **IWorkbenchWindowActionDelegate**, or **IWorkbenchWindowPulldownDelegate** if the action will be shown as a pull–down tool item in the tool bar. Since we are not creating a pull–down tool item, we provide **WindowActionDelegate**. This class is similar to **ObjectActionDelegate**. It launches the readme sections dialog when the user chooses the action. (We'll discuss the sections dialog in Application dialogs.)

The action also supplies enabling conditions for its menu item and tool bar item. The menu and tool bar items will only be enabled when a single (**enablesFor="1"**) readme file (**selectionClass ="org.eclipse.core.resources.IFile" name="*.readme"**) is selected. This action's menu and toolbar item appear and are enabled by virtue of the markup in the **plugin.xml** file. None of the plug–in code will execute until the user chooses the action and the workbench runs the action **class**.

We'll look at the other two actions later in the context of retargetable actions.

Application dialogs

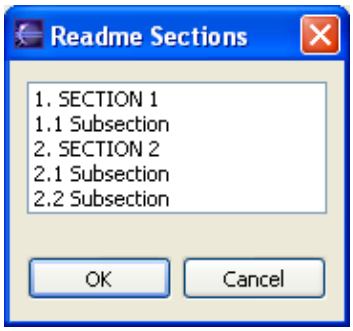
When a standard dialog is too simple for your plug–in, you can build your own dialog using the **Dialog** class. Earlier, we saw how the readme tool contributed an "Open Readme Browser" action in an action set. This action set is shown in the workbench tool bar and **Window–>Readme File Editor** menu.

Now we are ready to look at the implementation of this action in the readme tool's **WindowActionDelegate**.

```
public void run(IAction action) {
   SectionsDialog dialog = new SectionsDialog(window.getShell(),
      ReadmeModelFactory.getInstance().getSections(selection));
   dialog.open();
}
```

The window action delegate for the action set uses the current selection in the resource navigator view (the **.readme** file) to get a list of sections in the readme file. This list and the workbench window's shell are passed to the **SectionsDialog**.

When the user selects the action, the **SectionsDialog** is opened.



The **SectionsDialog** is implemented in the readme tool plug–in by subclassing the **Dialog** class in the **org.eclipse.jface.dialogs package.**

The **Dialog** class provides basic support for building a dialog shell window, creating the common dialog buttons, and launching the dialog. The subclasses are responsible for handling the content of the dialog itself:

- **createDialogArea** creates the SWT controls that represent the dialog contents. This is similar to creating the controls for a view or editor.

  The **SectionsDialog** creates an SWT list to display the list of sections. It uses a JFace viewer to populate the list. (We'll look at JFace viewers in Viewers.) Note that our dialog does not have to create any of the buttons for the dialog since this is done by our superclass.

```
protected Control createDialogArea(Composite parent) {
   Composite composite = (Composite)super.createDialogArea(parent);
   List list = new List(composite, SWT.BORDER);
   ...
   ListViewer viewer = new ListViewer(list);
   ...
   return composite;
}
```

- **configureShell** is overridden to set an appropriate title for the shell window.

```
protected void configureShell(Shell newShell) {
   super.configureShell(newShell);
   newShell.setText(MessageUtil.getString("Readme Sections"));
   ...
}
```

- **okButtonPressed** is overridden to perform whatever action is necessary when the user presses the OK button. (You can also override **cancelButtonPressed** or **buttonPressed(int)** depending on the design of your dialog.)

SectionsDialog does not implement an **okButtonPressed** method. It inherits the "do–nothing" implementation from **Dialog**. This is not typical. Your dialog usually performs some processing in response to one of the dialog buttons being pressed.

Dialogs can be as simple or as complicated as necessary. When you implement a dialog, most of your dialog code is concerned with creating the SWT controls that represent its content area and handling any events necessary while the dialog is up. Once a button is pressed by the user, the dialog can query the state of the

various controls (or viewers) that make up the dialog to determine what to do.

The plug−in class

So far, we've been looking at the different extensions that are provided by the readme tool. Let's look at the general definition of the readme tool plug−in.

Plug−in definition

The readme tool plug−in is defined at the top of the **plugin.xml** file.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin
   id="org.eclipse.ui.examples.readmetool"
   name="%Plugin.name"
   version="2.1.0"
   provider-name="%Plugin.providerName"
   class="org.eclipse.ui.examples.readmetool.ReadmePlugin">

   <runtime>
      <library name="readmetool.jar"/>
   </runtime>
   <requires>
      <import plugin="org.eclipse.ui"/>
      <import plugin="org.eclipse.core.resources"/>
      <import plugin="org.eclipse.core.runtime.compatibility"/>
      <import plugin="org.eclipse.ui.views"/>
      <import plugin="org.eclipse.ui.ide"/>
      <import plugin="org.eclipse.jface.text"/>
      <import plugin="org.eclipse.text"/>
      <import plugin="org.eclipse.ui.workbench.texteditor"/>
      <import plugin="org.eclipse.ui.editors"/>
   </requires>
   ...
```

The plug−in definition includes the **name**, **id, version**, and **provider name** of the plug−in. We saw most of these parameters before in our hello world plug−in. The readme tool also defines a specialized plug−in class, **ReadmePlugin**.

The name of the jar file is also provided. File names specified in a **plugin.xml** file are relative to the plug−in's directory, so the readme tool's jar file should be located directly in the plug−in's directory.

The **requires** element informs the platform of the readme tool's dependencies. The workbench UI plug−ins are listed as required plug−ins, along with the various core, jface, and text plug−ins.

AbstractUIPlugin

The **ReadmePlugin** class represents the readme tool plug−in and manages the life cycle of the plug−in. As we saw in the Hello World example, you don't have to specify a plug−in class. The platform will provide one for you. In this case, our plug−in needs to initialize UI related data when it starts up. The platform class **AbstractUIPlugin** provides a structure for managing UI resources and is extended by **ReadmePlugin**.

AbstractUIPlugin uses the generic startup and shutdown methods to manage images, dialog settings, and a preference store during the lifetime of the plug−in. We'll look at the specifics of the **ReadmePlugin** class when we work with dialogs and preferences.

Preference pages

Once a plug−in has contributed extensions to the workbench user interface, it is common for the plug−in to allow the user to control some of the behavior of the plug−in through user preferences.

The platform UI provides support for storing plug−in preferences and showing them to the user on pages in the workbench **Preferences** dialog. Plug−in preferences are key/value pairs, where the key describes the name of the preference, and the value is one of several different types. (See Runtime preferences for a detailed description of the runtime preferences infrastructure.)

How does a plug−in contribute a page for showing its preferences? We will use the readme tool example to see how it contributes a preference page to the workbench and then look at some of the underlying support for building preference pages.

Contributing a preference page

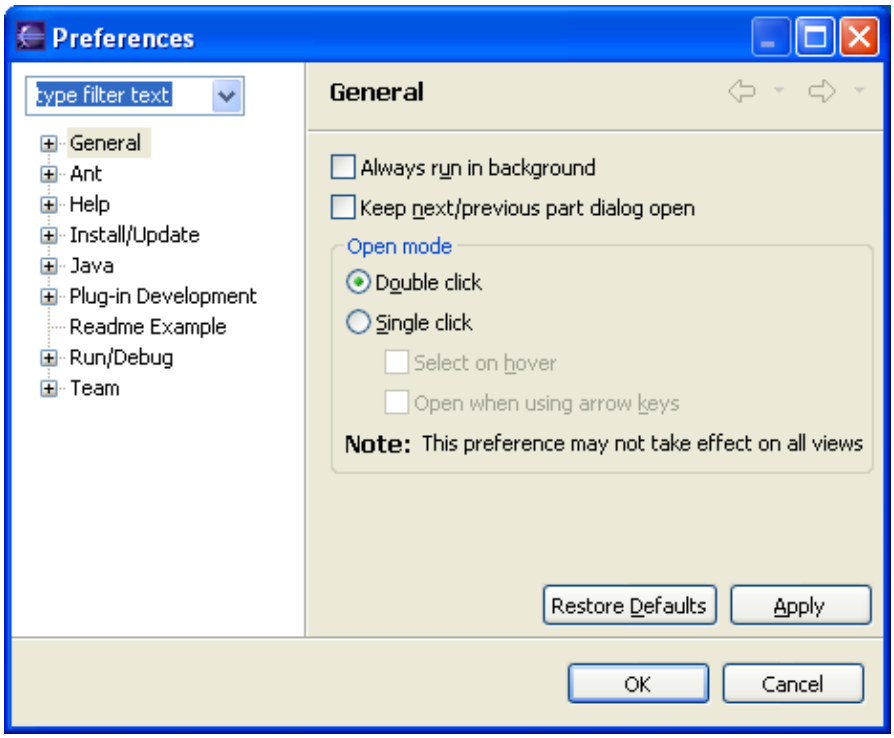
The **org.eclipse.ui.preferencePages** extension point allows you to contribute pages to the general preferences (**Window−>Preferences**) dialog. The preferences dialog presents a hierarchical list of user preference entries. Each entry displays a corresponding preference page when selected.

The readme tool uses this extension point to add the Readme Example preferences page.

```
<extension
   point = "org.eclipse.ui.preferencePages">
    <page id="org.eclipse.ui.examples.readmetool.Page1"
         class="org.eclipse.ui.examples.readmetool.ReadmePreferencePage"
         name="%PreferencePage.name">
    </page>
</extension>
```

This markup defines a preference page named "Readme Example" which is implemented by the class **ReadmePreferencePage**. The class must implement the **IWorkbenchPreferencePage** interface.

The workbench uses the core runtime's preference mechanisms to access all nodes in the preference tree and their corresponding pages. This list can be initialized from information in the preferences service without running any plug−in code.



The "Readme Example" preference is added to the top level of the preference tree on the left. Why? Because a preference page contribution will be added as a root of the tree unless a **category** attribute is specified. (The name **category** is somewhat misleading. Perhaps a better name is **path**.) The **category** attribute specifies the id (or a sequence of ids from the root) of the parent page. For example, the following markup would create a second readme tool preference page, "Readme Example Child Page," as a child of the original page.

```
<extension
   point = "org.eclipse.ui.preferencePages">
   <page
      id="org.eclipse.ui.examples.readmetool.Page1"
      class="org.eclipse.ui.examples.readmetool.ReadmePreferencePage"
      name="%PreferencePage.name">
   </page>
   <page
      id="org.eclipse.ui.examples.readmetool.Page2"
      class="org.eclipse.ui.examples.readmetool.ReadmePreferencePage2"
      name="Readme Example Child Page"
      category="org.eclipse.ui.examples.readmetool.Page1>
   </page>
</extension>
```

Once the user selects the entry for a preference page in the tree on the left, the workbench will create and display a preference page using the **class** specified in the extension definition. This action is what activates the plug–in (if it wasn't already activated due to another user operation).

Implementing a preference page

Defining the page

The JFace plug–in provides a framework for implementing wizards, preference pages, and dialogs. The implementation for these dialogs follows a common pattern. The contents of a page or dialog is defined by

implementing a **createContents** method that creates the SWT controls representing the page content. This method should also add listeners for any events of interest. The page is responsible for creating and returning the composite that will parent all of the controls in the page. The following snippet shows the highlights:

```
protected Control createContents(Composite parent)
{
        ...
        //composite_textField << parent
        Composite composite_textField = createComposite(parent, 2);
        Label label_textField = createLabel(composite_textField, MessageUtil.getString("Text_Fiel
        textField = createTextField(composite_textField);
        pushButton_textField = createPushButton(composite_textField, MessageUtil.getString("Chang

        //composite_tab << parent
        Composite composite_tab = createComposite(parent, 2);
        Label label1 = createLabel(composite_tab, MessageUtil.getString("Radio_Button_Options"));

        //
        tabForward(composite_tab);
        //radio button composite << tab composite
        Composite composite_radioButton = createComposite(composite_tab, 1);
        radioButton1 = createRadioButton(composite_radioButton, MessageUtil.getString("Radio_butt
        radioButton2 = createRadioButton(composite_radioButton, MessageUtil.getString("Radio_butt
        radioButton3 = createRadioButton(composite_radioButton, MessageUtil.getString("Radio_butt

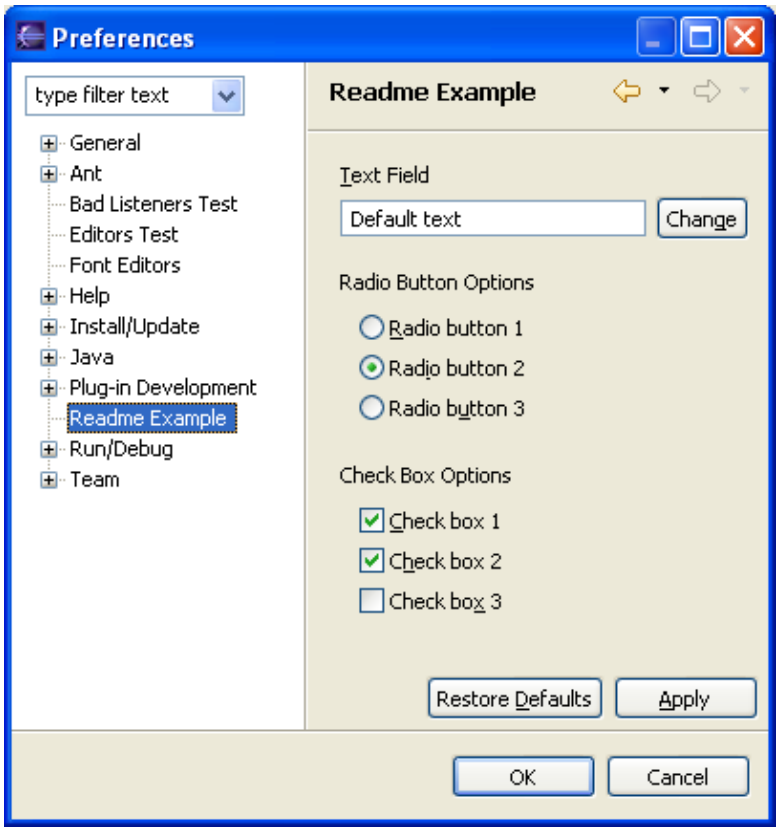

        //composite_tab2 << parent
        Composite composite_tab2 = createComposite(parent, 2);
        Label label2 = createLabel(composite_tab2, MessageUtil.getString("Check_Box_Options")); /

        //
        tabForward(composite_tab2);
        //composite_checkBox << composite_tab2
        Composite composite_checkBox = createComposite(composite_tab2, 1);
        checkBox1 = createCheckBox(composite_checkBox, MessageUtil.getString("Check_box_1"));
        checkBox2 = createCheckBox(composite_checkBox, MessageUtil.getString("Check_box_2"));
        checkBox3 = createCheckBox(composite_checkBox, MessageUtil.getString("Check_box_3"));

        initializeValues();

        return new Composite(parent, SWT.NULL);
}
```

Most of the code in this method is concerned with creating and laying out the controls, so we won't dissect it here. Here is what the corresponding page looks like:



The other primary responsibility of a preference page is to react to the **performOk** message. Typically, this method updates and stores the user preferences and, if necessary, updates any other plug–in objects to reflect the change in preferences. The **performDefaults** method is used to restore preferences to their default state when the user presses the **Restore Defaults** button.

You may override **performApply** if you have additional processing when the user selects **Apply**. The default implementation is to call **performOk**.

Preference pages should override the **doGetPreferenceStore()** method to return a preference store for storing their values.

Plug–in preference store

Preference stores are a convenience mechanism for accessing and storing preference values in a plug–in class. They provide plug–in level access to preferences that are actually stored using the runtime preferences service. **AbstractUIPlugin** defines a plug–in wide preference store that is maintained during the lifetime of the plug–in. Your plug–in can add entries to this preference store and update the values as the user changes the settings in your preferences page. Since preference stores use the platform preferences service, they will take care of saving preference values at the appropriate scope and location, and initializing the preference store using the appropriate mechanisms.

The following code in the **ReadmePreferencePage** obtains the preference store for the **ReadmePlugin**.

```
protected IPreferenceStore doGetPreferenceStore() {
   return ReadmePlugin.getDefault().getPreferenceStore();
}
```

Because **ReadmePlugin** extends the **AbstractUIPlugin** class, it automatically inherits a preference store. This preference store is initialized using the platform preferences service. The only thing the **ReadmePlugin** has to do is implement a method that initializes the preference controls to their default values. These values are used the first time the preference page is shown or when the user presses the **Defaults** button in the preferences page.

```
protected void initializeDefaultPreferences(IPreferenceStore store) {
        // These settings will show up when Preference dialog
        // opens up for the first time.
        store.setDefault(IReadmeConstants.PRE_CHECK1, true);
        store.setDefault(IReadmeConstants.PRE_CHECK2, true);
        store.setDefault(IReadmeConstants.PRE_CHECK3, false);
        store.setDefault(IReadmeConstants.PRE_RADIO_CHOICE, 2);
        store.setDefault(IReadmeConstants.PRE_TEXT, MessageUtil.getString("Default_text")); //$NO
}
```

Note: If there are no preferences saved anywhere for a plug−in, the plug−in will get an empty preference store.

Retrieving and saving preferences

Once you've associated your plug−in's preference store with your preference page, you can implement the logic for retrieving and saving the preferences.

Preference pages are responsible for initializing the values of their controls using the preferences settings from the preference store. This process is similar to initializing dialog control values from dialog settings. The **ReadmePreferencePage** initializes all of its controls in a single method, **initializeValues**, which is called from its **createContents** method.

```
private void initializeValues() {
        IPreferenceStore store = getPreferenceStore();
        checkBox1.setSelection(store.getBoolean(IReadmeConstants.PRE_CHECK1));
        checkBox2.setSelection(store.getBoolean(IReadmeConstants.PRE_CHECK2));
        checkBox3.setSelection(store.getBoolean(IReadmeConstants.PRE_CHECK3));
        ...
}
```

When the **OK** (or **Apply**) button is pressed, the current values of the controls on the preference page should be stored back into the preference store. The **ReadmePreferencePage** implements this logic in a separate method, **storeValues**.

```
private void storeValues() {
        IPreferenceStore store = getPreferenceStore();
        store.setValue(IReadmeConstants.PRE_CHECK1, checkBox1.getSelection());
        store.setValue(IReadmeConstants.PRE_CHECK2, checkBox2.getSelection());
        store.setValue(IReadmeConstants.PRE_CHECK3, checkBox3.getSelection());
        ...
}
```

When the user presses the **Defaults** button, the platform will restore all preference store values to the default values specified in the plug−in class. However, your preference page is responsible for reflecting these default values in the controls on the preference page. The **ReadmePreferencePage** implements this in **initializeDefaults**.

```
   private void initializeDefaults() {
```

```
        IPreferenceStore store = getPreferenceStore();
        checkBox1.setSelection(store.getDefaultBoolean(IReadmeConstants.PRE_CHECK1));
        checkBox2.setSelection(store.getDefaultBoolean(IReadmeConstants.PRE_CHECK2));
        checkBox3.setSelection(store.getDefaultBoolean(IReadmeConstants.PRE_CHECK3));
        ...
    }
```

Field editors

The implementation of a preference page is primarily SWT code. SWT code is used to create the preference page controls, set the values of the controls, and retrieve the values of the controls. The **org.eclipse.jface.preference** package provides helper classes, called **field editors**, that create the widgets and implement the value setting and retrieval code for the most common preference types. The platform provides field editors for displaying and updating many value types, including booleans, colors, strings, integers, fonts, and file names.

FieldEditorPreferencePage implements a page that uses these field editors to display and store the preference values on the page. Instead of creating SWT controls to fill its contents, a **FieldEditorPreferencePage** subclass creates field editors to display the contents. All of the fields on the page must be implemented as field editors. The following is a snippet from the debug UI preferences page:

```
protected void createFieldEditors() {
        addField(new BooleanFieldEditor(IDebugUIConstants.PREF_BUILD_BEFORE_LAUNCH,
                DebugPreferencesMessages.getString("DebugPreferencePage.auto_build_before_launch"
                SWT.NONE, getFieldEditorParent()));
        ...
        String[][] perspectiveNamesAndIds = getPerspectiveNamesAndIds();
        addField(new ComboFieldEditor(IDebugUIConstants.PREF_SHOW_DEBUG_PERSPECTIVE_DEFAULT,
                DebugPreferencesMessages.getString("DebugPreferencePage.Default_perspective_for_D
                perspectiveNamesAndIds,
                getFieldEditorParent()));
        ...
}
```

Each field editor is assigned the name of its corresponding preference key and the text label for the SWT control that it will create. The kind of control created depends on the type of field editor. For example, a boolean field editor creates a checkbox.

Since the preference page is associated with a preference store (specified in the **doGetPreferenceStore** method), the code for storing the current values, for initializing the control values from the preference store, and for restoring the controls to their default values can all be implemented in the **FieldEditorPreferencePage**.

The **FieldEditorPreferencePage** will use a grid layout with one column as the default layout for field editor widgets. For special layout requirements, you can override the **createContents** method.

Dialogs and wizards

We've seen how to extend the workbench UI by adding views, editors, and actions to the workbench. We've contributed a preference page for controlling the behavior of our plug−in. Now we can tie it all together by launching our own dialogs in response to these actions.

The JFace UI framework provides several standard dialogs and a framework for building your own dialogs and wizards. We'll look at the different kinds of dialogs and wizards and how to build them.

We'll also cover some simple workbench extensions for contributing wizards.

Standard dialogs

The package **org.eclipse.jface.dialogs** defines the basic support for dialogs. This package provides standard dialogs for displaying user messages and obtaining simple input from the user.

- **MessageDialog** displays a message to the user. You can set the dialog title, image, button text, and message in the constructor for this dialog.
- **ErrorDialog** displays information about an error. You can set the dialog title and message for the dialog. You can also supply an IStatus object which the dialog will use to obtain an error message.
- **InputDialog** allows the user to enter text. You can set the dialog title, default text value, and supply an object that will validate the text input.
- **ProgressMonitorDialog** shows progress to the user during the running of a long operation.

The standard dialogs are designed so that you can completely specify the dialog in its constructor. We saw a **MessageDialog** in action in the readme tool's view action:

```
MessageDialog.openInformation(
   view.getSite().getShell(),"Readme Editor","View Action executed");
```

Dialog settings

The **org.eclipse.jface.dialogs** package provides a utility class, **DialogSettings**, for storing and retrieving keyed values. You can use this class to save and retrieve primitive data types and string values that you associate with key names. The settings are loaded and saved using an XML file.

AbstractUIPlugin provides support for plug−in wide dialog settings stored in an XML file in your plug−in's directory. If a dialog settings file is not found in your plug−in directory, an empty **DialogSettings** will be created for you. When the plug−in is shut down, any settings that were added to it will be saved in an XML file and retrieved the next time the plug−in is started up.

You can access your dialog settings anywhere in your plug−in code. The following snippet shows how you could obtain the dialog settings for the readme tool.

```
IDialogSettings settings = ReadmePlugin.getDefault().getDialogSettings();
```

Values are stored and retrieved using get and put methods. The get methods are named after the type of primitive that is being accessed. You can store and retrieve boolean, long, double, float, int, array, and string values. The following snippet shows how we could use dialog settings to initialize control values in a dialog.

```
protected Control createDialogArea(Composite parent) {
   IDialogSettings settings = ReadmePlugin.getDefault().getDialogSettings();
   checkbox = new Button(parent,SWT.CHECK);
   checkbox.setText("Generate sample section titles");
   // initialize the checkbox according to the dialog settings
   checkbox.setSelection(settings.getBoolean("GenSections"));
}
```

The value of the setting can be stored later when the ok button is pressed.

```
protected void okPressed() {
   IDialogSettings settings = ReadmePlugin.getDefault().getDialogSettings();
   // store the value of the generate sections checkbox
   settings.put("GenSections", checkbox.getSelection());
   super.okPressed();
}
```

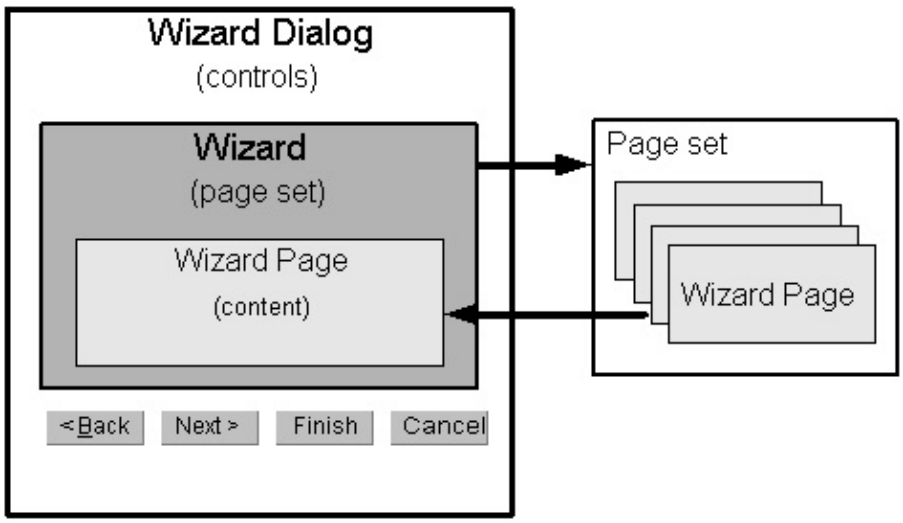
Wizards

Wizards are used to guide the user through a sequenced set of tasks. Your plug−in can contribute wizards at predefined extension points in the workbench. It can also create and launch its own wizards.

When you contribute to a workbench wizard extension point, the actions that launch the wizard are already set up by the workbench. You need only supply the wizard that will be used.

If you need to launch other wizards that are not already defined in workbench wizard extension points, you must launch them yourself. You can launch your own wizards by adding an action to a view, editor, popup, or an action set.

A wizard is composed of several different underlying parts.



Wizard dialog

The wizard dialog (**WizardDialog**) is the top level dialog in a wizard. It defines the standard wizard buttons

and manages a set of pages that are provided to it.

When you contribute to a workbench wizard extension, you do not have to create a wizard dialog. One is created on your behalf by the workbench, and your wizard is set into it.

The wizard dialog performs the enabling and disabling of the **Next**, **Back**, and **Finish** buttons based on information it obtains from the wizard and the current wizard page.

Wizard

The wizard (**IWizard**) controls the overall appearance and behavior of the wizard, such as title bar text, image, and the availability of a help button. Wizards often use a corresponding **DialogSettings** to obtain (and store) the default values for the settings of controls on the wizard pages.

The **Wizard** class implements many of the details for standard wizard behavior. You typically extend this class to implement behavior specific to your wizard. The primary responsibilities of your wizard will include:

- Creating and adding your pages to your wizard
- Implementing the behavior that should occur when the user presses the **Finish** button.

Wizard page

The wizard page (**IWizardPage**) defines the controls that are used to show the content of the wizard page. It responds to events in its content areas and determines when the page is completed.

Your wizard page typically extends the **WizardPage** class. The primary responsibilities of your wizard page will include:

- creating the SWT controls that represent the page
- determing when the user has supplied enough information to complete the page (that is, when the user can move to the next page.)

org.eclipse.ui.newWizards

You can add a wizard to the **File > New >** menu options in the workbench using the **org.eclipse.ui.newWizards** extension point. The readme tool example uses this extension point definition to add the Readme File wizard:

```
<extension
      point = "org.eclipse.ui.newWizards">
        <category
           id = "org.eclipse.ui.examples.readmetool.new"
           parentCategory="org.eclipse.ui.Examples"
           name="%NewWizard.category">
        </category>
        <wizard
           id = "org.eclipse.ui.examples.readmetool.wizards.new.file"
           name = "%NewWizard.name"
           class="org.eclipse.ui.examples.readmetool.ReadmeCreationWizard"
           category="org.eclipse.ui.Examples/org.eclipse.ui.examples.readmetool.new"
           icon="icons/obj16/newreadme_wiz.png">
           <description>%NewWizard.desc</description>
           <selection class="org.eclipse.core.resources.IResource"/>
```

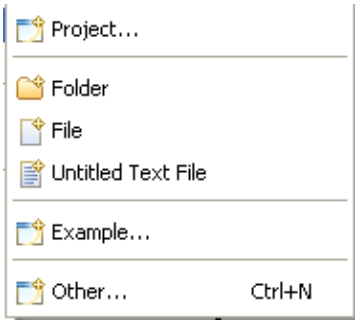
```
        </wizard>
</extension>
```

The **category** describes the grouping for the wizard. An optional **parentCategory** establishes the new category as a child of an existing category.

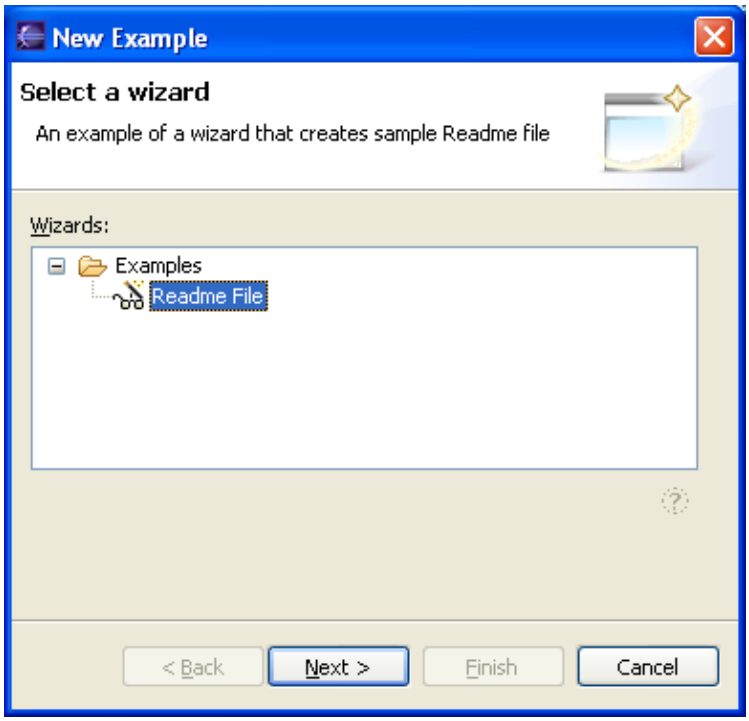
Top level categories will appear in the **File > New** menu. In this example, the **parentCategory** is set to an "Examples" category. Where did the parent category come from? The **org.eclipse.ui** plug−in defines a standard examples category in its markup:

```
<extension
        point="org.eclipse.ui.newWizards">
    <category
          name="%NewWizards.Category.Examples"
          id="org.eclipse.ui.Examples">
    </category>
    ...
```

This category appears in the **File > New** menu.



The readme tool's category **name** defines the label that is used for the next layer of grouping underneath the parent category. These categories are shown as the second level in the tree shown in the **New Example** wizard. The wizard's **name** and **icon** are shown underneath when you expand the category. The **description** of the selected wizard is shown at the top of the wizard when you select it.



This information about the wizard appears solely because of the markup in the **plugin.xml** file. None of the plug–in code runs until the user chooses the **Next** button. Once this happens, the workbench will instantiate the wizard **class** specified in the markup and pass it an expected selection **class**.

The class identified in this extension (**ReadmeCreationWizard**) must implement the **INewWizard** interface. Most wizards do so by extending the platform **Wizard** class although this is an implementation mechanism and not required by the extension point.

The wizard itself does little but create the pages inside of it. Let's look at the implementation of the page first, and then come back to the wizard.

Pages

The workbench provides base wizard page classes that support the type of processing performed for each wizard extension point. You can use these pages, or extend them to add additional processing.

The goal of the **ReadmeCreationWizard** is to create a new file, add the required content to the file, and as an option, open an editor on the file. Our page needs to define the controls that let the user specify what content goes in the file and whether an editor should be launched.

We create the wizard page, **ReadmeCreationPage**, by extending **WizardNewFileCreationPage**. The controls for a wizard page are defined in a fashion similar to the definition of the controls for a view or an editor. The page implements a **createControl** method, creating the necessary SWT widgets as children of the supplied **Composite**. Since the superclass already adds widgets that support new file processing, we need only extend the **createControl** method in our wizard page to add the additional checkboxes that control generation of sections and opening of the editor.

```
public void createControl(Composite parent) {
   // inherit default container and name specification widgets
   super.createControl(parent);
```

```
    Composite composite = (Composite)getControl();
    ...
    // sample section generation group
    Group group = new Group(composite,SWT.NONE);
    group.setLayout(new GridLayout());
    group.setText(MessageUtil.getString("Automatic_sample_section_generation"));
    group.setLayoutData(new GridData(GridData.GRAB_HORIZONTAL |
       GridData.HORIZONTAL_ALIGN_FILL));
    ...
    // sample section generation checkboxes
    sectionCheckbox = new Button(group,SWT.CHECK);
    sectionCheckbox.setText(MessageUtil.getString("Generate_sample_section_titles"));
    sectionCheckbox.setSelection(true);
    sectionCheckbox.addListener(SWT.Selection,this);

    subsectionCheckbox = new Button(group,SWT.CHECK);
    subsectionCheckbox.setText(MessageUtil.getString("Generate_sample_subsection_titles"));
    subsectionCheckbox.setSelection(true);
    subsectionCheckbox.addListener(SWT.Selection,this);
    ...
    // open file for editing checkbox
    openFileCheckbox = new Button(composite,SWT.CHECK);
    openFileCheckbox.setText(MessageUtil.getString("Open_file_for_editing_when_done"));
    openFileCheckbox.setSelection(true);
    ...
}
```

You should be able to follow this code if you understand the concepts in Standard Widget Toolkit.

The basic patterns for implementing a page include:

- Add listeners to any controls that affect dynamic behavior of the page. For example, if selecting an item in a list or checking a box affects the state of other controls of the page, add a listener so you can change the state of the page.
- Populate the controls with data based on the current selection when the wizard was launched. Some of the data may depend on the values in other controls. Some of the controls may use dialog settings to initialize their values.
- Use **setPageComplete(true)** when enough information is provided by the user to exit the page (and move to the next page or finish the wizard.)

The **ReadmeCreationPage** class inherits a lot of this behavior from the **WizardNewFileCreationPage**. Browse the implementation of these classes for further information.

Now that we understand what a page does, let's look again at the wizard.

Wizard

The wizard is responsible for creating the pages and providing the "finish" logic.

The basic patterns for implementing a wizard include:

- Implement the **init** method to set up local variables for context information such as the workbench and the current selection.

```
public void init(IWorkbench workbench,IStructuredSelection selection) {
   this.workbench = workbench;
```

```
        this.selection = selection;
        setWindowTitle(MessageUtil.getString("New_Readme_File"));
        setDefaultPageImageDescriptor(ReadmeImages.README_WIZARD_BANNER);
    }
```

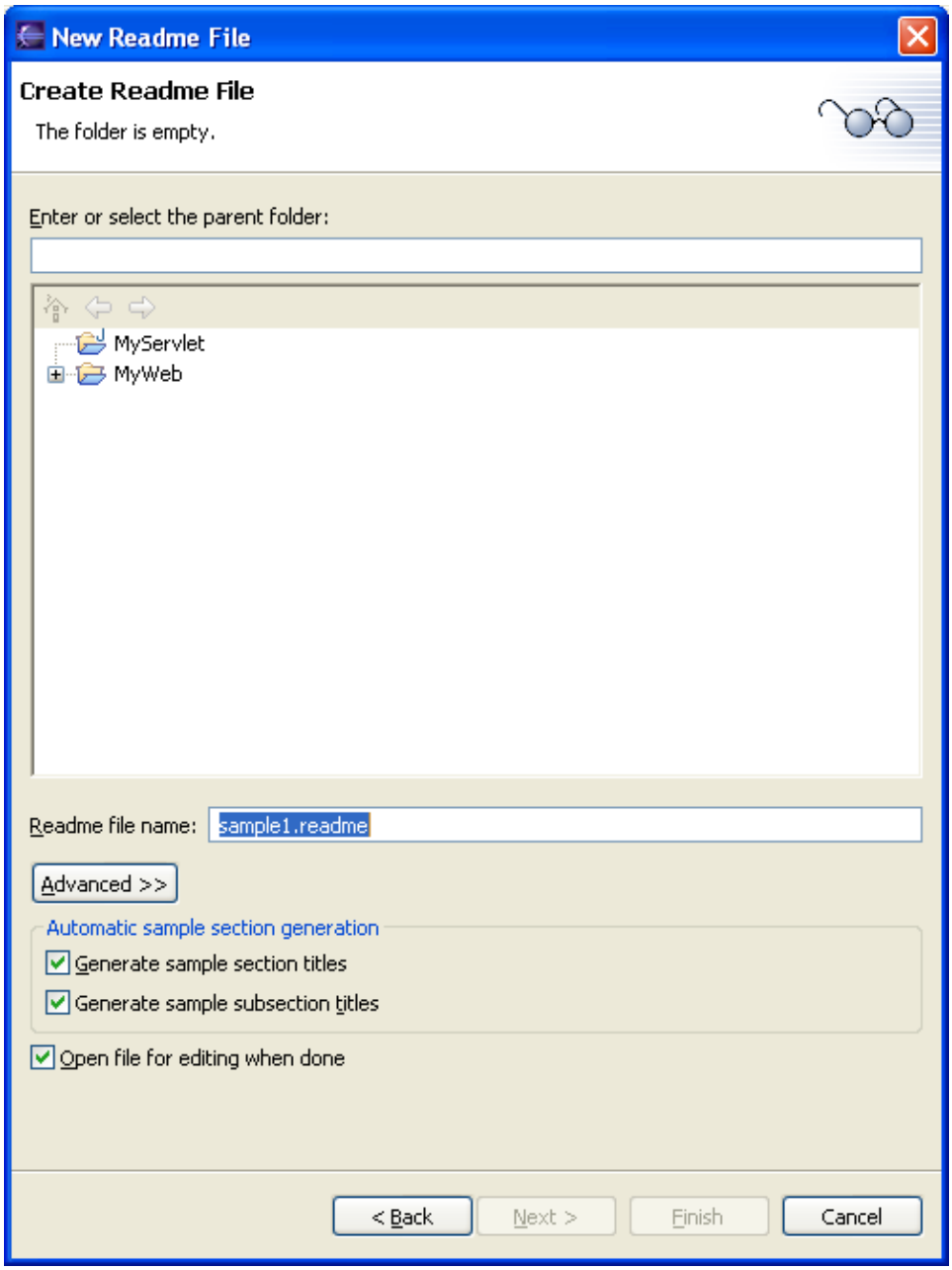
- Implement **addPages** by creating instances of the pages.

```
    public void addPages() {
        mainPage = new ReadmeCreationPage(workbench, selection);
        addPage(mainPage);
    }
```

- Implement **performFinish** to finish the task.
  Multi−page wizards typically handle the finish logic in the wizard itself, since each page will contribute information that determines how the task is implemented. Single page wizards can implement the logic in the wizard or ask the page to finish the job. The approach you take largely depends on where your important state is kept. In the case of the readme wizard, we are going to ask our page to handle the finish processing.

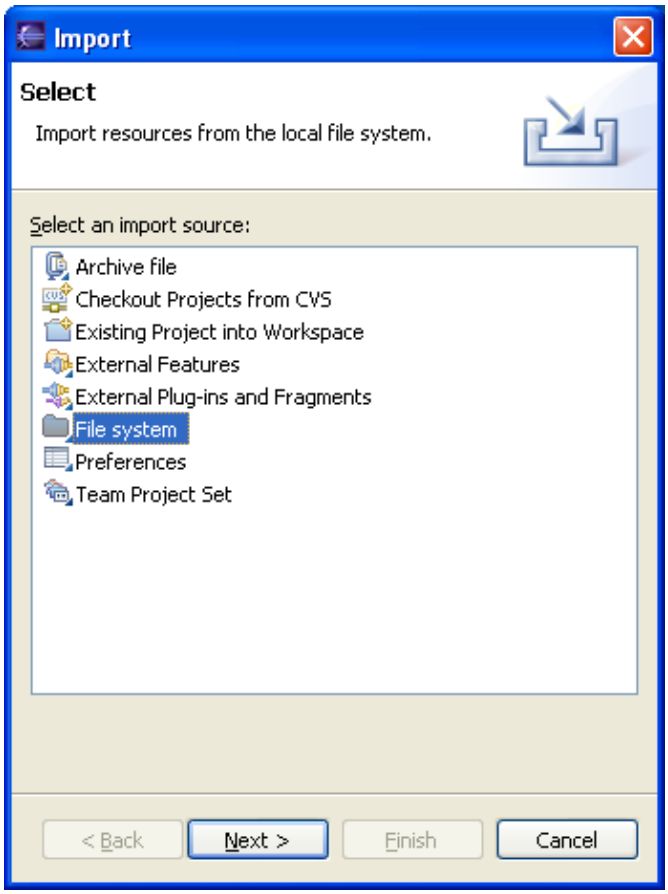
```
    public boolean performFinish() {
        return mainPage.finish();
    }
```

The completed wizard looks like this:



org.eclipse.ui.importWizards

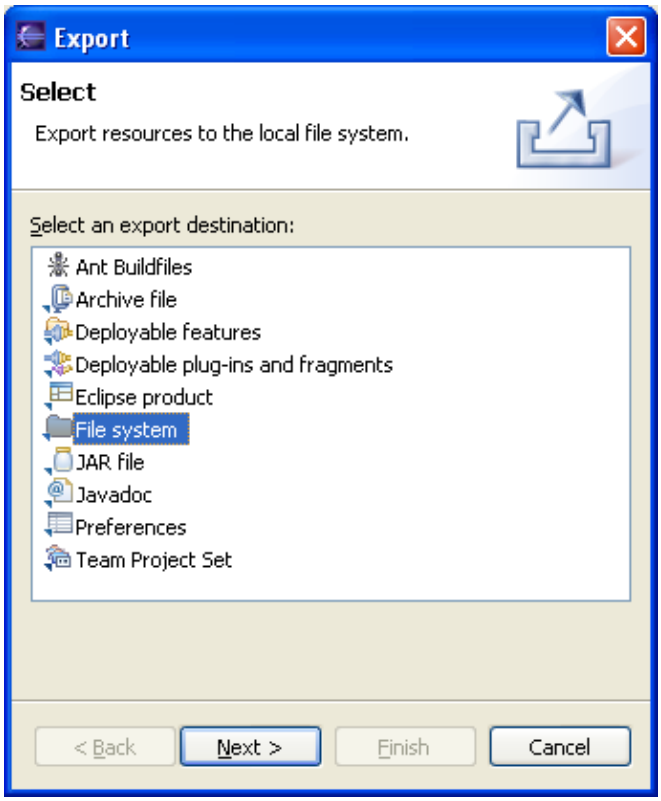
You can add a wizard to the **File > Import** menu option in the workbench using the **org.eclipse.ui.importWizards** extension point. The process for defining the extension and implementing the wizard is similar to **org.eclipse.ui.newWizards**. The primary difference in the markup is that import wizards do not define or assign categories for the wizards themselves. The wizards appear uncategorized in a wizard dialog.



The wizard supplied in the class parameter of the markup must implement **IImportWizard**. Its pages are typically extended from **WizardImportPage**.

org.eclipse.ui.exportWizards

You can add a wizard to the **File > Export** menu option in the workbench using the **org.eclipse.ui.exportWizards** extension point. The process for defining the extension and implementing the wizard is similar to **org.eclipse.ui.newWizards**. The primary difference in the markup is that export wizards do not define or assign categories for the wizards themselves. The wizards appear uncategorized in a wizard dialog.



The wizard supplied in the class parameter of the markup must implement **IExportWizard**. Its pages are typically extended from **WizardExportPage**.

Wizard dialogs

The previous example supplied a wizard for a specified extension point. Another, perhaps more common, case is that you want to launch your own plug−in's wizard from some action that you have defined. (In Workbench menu contributions, we discuss the ways you can contribute actions to the workbench.)

Wizards are displayed in the UI by placing them in a containing dialog. This detail is handled for you when you contribute to a wizard extension. When you are launching your own wizard, you must display it yourself by wrapping it in a **WizardDialog**.

For example, the **ReadmeCreationWizard** could be launched independently by creating a wizard dialog and associating it with the **ReadmeCreationWizard**. The following code snippet shows how this could be done from some action delegate. (The method assumes that we know the workbench and the selection.)

```
public void run(IAction action) {
   // Create the wizard
   ReadmeCreationWizard wizard = new ReadmeCreationWizard();
   wizard.init(getWorkbench(), selection);

   // Create the wizard dialog
   WizardDialog dialog = new WizardDialog
      (getWorkbench().getActiveWorkbenchWindow().getShell(),wizard);
   // Open the wizard dialog
   dialog.open();
}
```

If you need to embed a wizard anywhere else in your plug−in's user interface, the interface **IWizardContainer** defines the necessary protocol for hosting a wizard.

Multi−page wizards

If your wizard implements a complex task, you may want to use more than one page to obtain information from the user.

In general, the implementation pattern is the same as for a single page wizard.

- Create a **WizardPage** subclass for each page in your wizard. Each wizard page should use **setPageComplete(true)** when it has enough information.
- Create a **Wizard** subclass which adds each page to the wizard.
- Implement a **performFinish** method to perform the finish logic.

When you design a wizard, it's good practice to put all the required information on the first page if possible. This way, the user does not have to traverse the entire set of pages in order to finish the task. Optional information can go on subsequent pages.

When a page requires input from the user before it can be considered complete, use **setPageComplete(false)** to signify that it is not complete. As the page receives events from its controls, it rechecks to see if the page is complete. Once the required input is provided, **setPageComplete(true)** signals completion.

The **Wizard** class handles the logic required to enable and disable the **Finish** button according to the completion state of the pages. The **Finish** button is only enabled for a wizard when each of its pages have set its completion state to true.

Validation and page control

The classes **WizardNewFileCreationPage** and **CreateReadme1** show a common pattern for implementing page validation.

WizardNewFileCreationPage defines a common event handler for all SWT events which validates the page. This means the page will be validated whenever an event is received from a widget to which the page added a listener.

```
public void handleEvent(Event event) {
   setPageComplete(validatePage());
}
```

Once the **ReadmeCreationPage** creates its controls, it sets the state of the page using **validatePage**.

```
public void createControl(Composite parent) {
   super.createControl(parent);
   // create controls, add listeners, and layout the page
   ...
   // sample section generation checkboxes
   sectionCheckbox = new Button(group,SWT.CHECK);
   sectionCheckbox.setText(MessageUtil.getString("Generate_sample_section_titles"));
   sectionCheckbox.setSelection(true);
   sectionCheckbox.addListener(SWT.Selection,this);

   subsectionCheckbox = new Button(group,SWT.CHECK);
```

```
    subsectionCheckbox.setText(MessageUtil.getString("Generate_sample_subsection_titles"));
    subsectionCheckbox.setSelection(true);
    subsectionCheckbox.addListener(SWT.Selection,this);
    ...
    setPageComplete(validatePage());
}
```

Using this pattern, a wizard page can put all of its page validation code in one method, **validatePage**(). This method determines the initial state of the page and recalculates the state any time it receives an event from a widget on its page.

Since we added a listener to the section checkbox, we will recompute the valid state of the page whenever that checkbox receives a selection event. Note that the page's **handleEvent** method must call super to ensure that the inherited page validation behavior occurs in addition to any specific event handling for this page.

```
public void handleEvent(Event e) {
    Widget source = e.widget;
    if (source == sectionCheckbox) {
        if (!sectionCheckbox.getSelection())
            subsectionCheckbox.setSelection(false);
        subsectionCheckbox.setEnabled(sectionCheckbox.getSelection());
    }
    super.handleEvent(e);
}
```

Actions and contributions

The action classes allow you to define user commands independently from their presentation in the UI. This gives you the flexibility to change the presentation of an action in your plug–in without changing the code that actually performs the command once it has been chosen. The contribution classes are used to manage the actual UI items representing the commands. You don't program to the contribution classes, but you will see them in some of the workbench and JFace API.

Actions

An action (**IAction**) represents a command that can be triggered by the end user. Actions are typically associated with buttons, menu items, and items in tool bars.

Although actions do not place themselves in the UI, they do have UI oriented properties, such as tool tip text, label text, and an image. This allows other classes to construct widgets for the presentation of the action.

When the user triggers the action in the UI, the action's run method is invoked to do the actual work. A common pattern in the run method is to query the workbench selections and manipulate the objects that are selected. Another common pattern is to launch a wizard or dialog when an action is chosen.

You should not directly implement the **IAction** interface. Instead, you should subclass the **Action** class. Browse the subclasses of this class to see many of the common patterns for actions. The code below implements the "About" action. It is one of the simpler actions in the workbench.

```
public void run() {
    new AboutDialog(workbenchWindow.getShell()).open();
}
```

Earlier we saw the workbench interfaces **IViewActionDelegate** and **IEditorActionDelegate**. These interfaces are used when contributing view actions or editor actions to the workbench. The workbench action delegates are initialized with a reference to their associated view or editor. With this knowledge, they can navigate to the workbench page or window, accessing selections or any other information needed to perform the action.

You will implement your own action classes whenever you want to define a command in your plug−in. If you are contributing actions to other views and editors, you will implement action delegates.

Contribution items

A contribution item (**IContributionItem**) represents the UI portion of an action. More specifically, it represents an item that is contributed to a shared UI resource such as a menu or tool bar.

Contribution items know how to fill a specific SWT widget with the appropriate SWT item that represents the contribution.

You don't have to worry about creating a contribution item when you are contributing actions to the workbench UI. This is done on your behalf when the workbench creates UI items for the actions that you have defined.

Contribution managers

A contribution manager (**IContributionManager**) represents a collection of contribution items that will be presented in the UI. You can add and insert contribution items using named contribution ids to place the items in the appropriate order. You can also find items by id and remove individual items.

Each implementation of **IContributionManager** knows how to fill a specific SWT widget with its items. JFace provides contribution managers for menus (**IMenuManager**), tool bars (**IToolBarManager**), and status lines (**IStatusLineManager**).

As a plug−in developer, you do not need to implement these interfaces, but you will see references to some of these managers in API methods.

User interface resources

The **org.eclipse.jface.resource** package defines classes that help plug−ins manage UI resources such as fonts and icons.

Many of the workbench extension points allow plug−ins to supply icons that can be used to show their contributions in the workbench. Since GUI operating systems support a limited number of images or fonts in memory at once, a plug−in's UI resources must be carefully managed and sometimes shared between widgets.

We've already seen several references to icons in the readme tool plug−in. Some of its icons are specified in the **plugin.xml** markup.

```
<extension
   point="org.eclipse.ui.views">
        <category
           id="org.eclipse.ui.examples.readmetool"
           name="%Views.category">
```

```
        </category>
        <view
           id="org.eclipse.ui.examples.readmetool.views.SectionsView"
           name="%Views.ReadmeSections"
   icon="icons/view16/sections.png"
           category="org.eclipse.ui.examples.readmetool"
           class="org.eclipse.ui.examples.readmetool.ReadmeSectionsView">
        </view>
</extension>
```

We've also seen code that describes images on the fly. The following is from the readme tool's **ReadmeEditorActionBarContributor**.

```
public ReadmeEditorActionBarContributor() {
        ...
        action1 = new EditorAction(MessageUtil.getString("Editor_Action1"));
        action1.setToolTipText(MessageUtil.getString("Readme_Editor_Action1"));
        action1.setDisabledImageDescriptor(ReadmeImages.EDITOR_ACTION1_IMAGE_DISABLE);
        action1.setImageDescriptor(ReadmeImages.EDITOR_ACTION1_IMAGE_ENABLE);
        ...
```

JFace provides the basic support classes that allow plug−ins to manage their icons and fonts without worrying about when the corresponding platform graphics objects are created and destroyed. These support classes are used directly by plug−ins as shown above, or indirectly when the workbench uses these classes to obtain images that are described in extension point markup.

Image descriptors and the registry

The SWT **Image** class represents an image from the operating system's perspective. Because most GUI operating systems have a limit on the number of images that can be open at once, plug−ins should be very careful when creating them, and ensure that they also dispose of them properly when finished using them. By using the JFace **ImageDescriptor** and **ImageRegistry** classes instead of the SWT image, plug−ins can generally avoid creating, managing, and disposing these images directly.

Image descriptor

The **ImageDescriptor** class can be used as a lightweight description of an image. It specifies everything that is needed to create an image, such as the URL or filename where the image can be obtained. **ImageDescriptors** do not allocate an actual platform image unless specifically requested using the **createImage()** method.

Image descriptors are the best strategy when your code is structured such that it defines all the icons in one place and allocates them as they are needed. Image descriptors can be created at any time without concern for OS resources, making it convenient to create them all in initialization code.

Image registry

The **ImageRegistry** class is used to keep a list of named images. Clients can add image descriptors or SWT images directly to the list. When an image is requested by name from the registry, the registry will return the image if it has been created, or create one from the descriptor. This allows clients of the registry to share images.

Images that are added to or retrieved from the registry must not be disposed by any client. The registry is

responsible for disposing of the image since the images are shared by multiple clients. The registry will dispose of the images when the platform GUI system shuts down.

Plug−in patterns for using images

Specifying the image in the plugin.xml

Where possible, specify the icon for your plug−in's UI objects in the **plugin.xml** file. Many of the workbench extension points include configuration parameters for an icon file. By defining your icons in your extension contribution in the plugin.xml, you leave the image management strategy up the platform. Since the icons are typically kept in your plug−in's directory, this allows you to specify the icons and manage the files all in one place.

The other patterns should only be considered when you can't specify the icon as part of your extension contribution.

Explicit creation

Explicitly creating an image is the best strategy when the image is infrequently used and not shared. The image can be created directly in SWT and disposed after it is used.

Images can also be created explicitly using an **ImageDescriptor** and invoking the **createImage()** method. As in the first case, the dispose() method for the image must be invoked after the image is no longer needed. For example, if a dialog creates an image when it is opened, it should dispose the image when it is closed.

Image registry

When an image is used frequently in a plug−in and shared across many different objects in the UI, it is useful to register the image descriptor with an **ImageRegistry**. The images in the registry will be shared with any object that queries an image by the same name. You must not dispose any images in the registry since they are shared by other objects.

Adding an image to the image registry is the best strategy when the image is used frequently, perhaps through the lifetime of the plug−in, and is shared by many objects. The disadvantage of using the registry is that images in the registry are not disposed until the GUI system shuts down. Since there is a limit on the number of platform (SWT) images that can be open at one time, plug−ins should be careful not to register too many icons in a registry.

The class **AbstractUIPlugin** includes protocol for creating a plug−in wide image registry.

Label providers

When an icon is used frequently to display items in a particular viewer, it can be shared among similar items in the viewer using a label provider. Since a label provider is responsible for returning an image for any object in a viewer, it can control the creation of the image and any sharing of images across objects in the viewer.

The label provider can use any of the previously discussed techniques to produce an image. If you browse the various implementations of **getImage()** in the **LabelProvider** subclasses, you will see a variety of approaches including caching a single icon for objects and maintaining a table of images by type. Images created by a label provider must be disposed in the provider's **dispose()** method, which is called when the

viewer is disposed.

Using a label provider is a good compromise between explicit creation and the image registry. It promotes sharing of icons like the image registry, yet still maintains control over the creation and disposal of the actual image.

Plug−in wide image class

When fine−tuning a plug−in, it is common to experiment with all of these different image creation patterns. It can be useful to isolate the decision making regarding image creation in a separate class and instruct all clients to use the class to obtain all images. This way, the creation sequence can be tuned to reflect the actual performance characteristics of the plug−in.

ResourceManager

The **ResourceManager** class is used to keep a mapping of ImageDescriptors to Images so that an Image can be reused be referring to it via it's descriptor.When an image is requested by descriptor from the registry, the registry will return the image if it has been created, or create one from the descriptor. This allows clients of the registry to share images.

The top level ResourceManager is a ***DeviceResourceManager*** which is created on a Display. The ResourceManager defined by ***JFaceResources*.*getResources()*** is a DeviceResourceManager and can be used as the top level ResourceManager. If you need a ResourceManager with a shorter lifecycle than the DeviceResourceManager you can create a ***LocalResourceManager*** as a child and dispose of it when you are done with it.

A DeviceResourceManager will be disposed when the Display used to create it is disposed so no special management code is required.

Images that are added to or retrieved from the manager must not be disposed by any client. The manager is responsible for disposing of the image since the images are shared by multiple clients. The registry will dispose of the images when the ResourceManager that holds onto them is disposed.

Font registry

Fonts are another limited resource in platform operating systems. The creation and disposal issues are the same for fonts as for images, requiring similar speed/space tradeoffs. In general, fonts are allocated in SWT by requesting a font with a platform dependent font name.

The **FontRegistry** class keeps a table of fonts by their name. It manages the allocation and disposal of the font.

In general, plug−ins should avoid allocating any fonts or describing fonts with platform specific names. Although the font registry is used internally in JFace, it is typically not used by plug−ins. The **JFaceResources** class should be used to access common fonts.

It is very common to allow users to specify their preferences for the application's fonts in a preference page. In these cases, the **FontFieldEditor** should be used to obtain the font name from the user, and a **FontRegistry** may be used to keep the font. The **FontFieldEditor** is only used in preference pages.

JFaceResources

The class **JFaceResources** controls access to common platform fonts and images. It maintains an internal font and image registry so that clients can share named fonts and images.

There are many techniques used in the workbench and other plug−ins to share images where required. The **JFaceResources** image registry is not widely used across the workbench and plug−in code.

Use of fonts is much simpler. The workbench and most plug−ins use the **JFaceResources** class to request fonts by logical name. Methods such as **getDialogFont()** and **getDefaultFont()** are provided so that plug−ins can use the expected fonts in their UI.

Widgets

SWT includes many rich features, but a basic knowledge of the system's core − *widgets*, *layouts*, and *events* − is all that is needed to implement useful and robust applications.

Widget application structure

When you are contributing UI elements using platform workbench extensions, the mechanics of starting up SWT are handled for you by the workbench.

If you are writing an SWT application from scratch outside of the workbench, you must understand more about SWT's application structure.

A typical stand−alone SWT application has the following structure:

- Create a ***Display*** which represents an SWT session.
- Create one or more ***Shell***s which serve as the main window(s) for the application.
- Create any other widgets that are needed inside the shell.
- Initialize the sizes and other necessary state for the widgets. Register listeners for widget events that need to be handled.
- Open the shell window.
- Run the event dispatching loop until an exit condition occurs, which is typically when the main shell window is closed by the user.
- Dispose the display.

The following code snippet is adapted from the ***org.eclipse.swt.examples.helloworld.HelloWorld2*** application. Since the application only displays the string "Hello World," it does not need to register for any widget events.

```
public static void main (String [] args) {
   Display display = new Display ();
   Shell shell = new Shell (display);
   Label label = new Label (shell, SWT.CENTER);
   label.setText ("Hello_world");
   label.setBounds (shell.getClientArea ());
   shell.open ();
   while (!shell.isDisposed ()) {
      if (!display.readAndDispatch ()) display.sleep ();
   }
   display.dispose ();
```

```
}
```

Display

The ***Display*** represents the connection between SWT and the underlying platform's GUI system. Displays are primarily used to manage the platform event loop and control communication between the UI thread and other threads. (See Threading issues for clients for a complete discussion of UI threading issues.)

For most applications you can follow the pattern that is used above. You must create a display before creating any windows, and you must dispose of the display when your shell is closed. You don't need to think about the display much more unless you are designing a multi−threaded application.

Shell

A ***Shell*** is a "window" managed by the OS platform window manager. Top level shells are those that are created as a child of the display. These windows are the windows that users move, resize, minimize, and maximize while using the application. Secondary shells are those that are created as a child of another shell. These windows are typically used as dialog windows or other transient windows that only exist in the context of another window.

Parents and children

All widgets that are not top level shells must have a parent. Top level shells do not have a parent, but they are created in association with a particular ***Display***. You can access this display using ***getDisplay()***. All other widgets are created as descendants (direct or indirect) of top level shells.

Composite widgets are widgets that can have children.

When you see an application window, you can think of it as a widget tree, or hierarchy, whose root is the shell. Depending on the complexity of the application, there may be a single child of the shell, several children, or nested layers of composites with children.

Style bits

Some widget properties must be set at the time a widget is created and cannot be subsequently changed. For example, a list may be single or multi−selection, and may or may not have scroll bars.

These properties, called *styles*, are set in the constructor. All widget constructors take an ***int*** argument that specifies the bitwise ***OR*** of all desired styles. In some cases, a particular style is considered a hint, which means that it may not be available on all platforms, but will be gracefully ignored on platforms that do not support it.

The style constants are located in the ***SWT*** class as public static fields. A list of applicable constants for each widget class is contained in the API Reference for ***SWT***.

Resource disposal

The platforms underneath SWT require explicit allocation and freeing of OS resources. In keeping with the SWT design philosophy of reflecting the platform application structure in the widget toolkit, SWT requires that you explicitly free any OS resources that you have allocated. In SWT, the ***Widget.dispose()*** method is used to free resources associated with a particular toolkit object.

The rule of thumb is that if you create the object, you must dispose of it. Here are some specific ground rules that further explain this philosophy:

- If you create a graphic object or widget using a constructor, you must explicitely dispose of it when you are finished using it.
- When a ***Composite***, is disposed, the composite and all of its child widgets are recursively disposed. In this case, you do not need to dispose of the widgets themselves. However, you must free any graphics resources allocated in conjunction with those widgets.
- If you get a graphic object or widget without using a constructor (e.g. Control.getBackground()), do not dispose of it since you did not allocate it.
- If you pass a reference to your widget or graphic object to another object, you must take care not to dispose of it while it is still being used. (Similar to the rule described in Plug−in patterns for using images.)
- If you create a graphic object for use during the lifetime of one of your widgets, you must dispose of the graphic object when the widget is disposed. This can be done by registering a dispose listener for your widget and freeing the graphic object when the ***dispose*** event is received.

There is one exception to these rules. Simple data objects, such as ***Rectangle*** and ***Point***, do not use operating system resources. They do not have a ***dispose()*** method and you do not have to free them. If in doubt, check the javadoc for a particular class.

See Managing operating resources for further discussion of this topic.

SWT standalone example – Hello World

The Hello World examples are a set of introductory examples that show how to get started on creating an application with SWT. They cover the creation of a shell, the use of event listeners, using layouts, processing events in an event loop, and drawing with a Graphics Context.

Hello World 1

This example demonstrates how to open a Shell and process the events.

Hello World 2

This example builds on HelloWorld1 and demonstrates how to display a Label inside of the Shell.

Hello World 3

This example builds on HelloWorld2 and demonstrates how to use a listener mechanism to resize the Label when the Shell size changes.

Hello World 4

This example builds on HelloWorld2 and demonstrates how to use a Layout to resize the Label when the Shell size changes.

Hello World 5

This example builds on HelloWorld1 and demonstrates how to draw directly on an SWT Control using a Graphics Context.

Running the example

Follow the SWT standalone examples setup instructions to install and run the example from your workspace.

The "Main" class is `org.eclipse.swt.examples.helloworld.HelloWorld[1-5]`.

This example can also be run using the Example Launcher. Select one of the ***Hello World*** items from the ***Standalone*** category and click ***Run***.

SWT standalone examples setup

Adding SWT to your workspace

1. Download SWT for standalone applications. A standalone version of SWT is available on the same download page as the Eclipse SDK. Look for the section titled ***SWT Binary and Source***. Do not extract the archive file, just save it to disk.
2. Select ***Import...*** from the ***File*** menu.
3. Select ***Existing Projects into Workspace*** and click on the ***Next*** button.
4. Select ***Select archive file:*** and use the ***Browse*** button to locate the SWT standalone archive you have previously downloaded.
5. Click on the ***Finish*** button.

Importing example source

1. Download and install the Eclipse Example Plug−ins. The Eclipse Example Plug−ins are available on the same download page as the Eclipse SDK. Look for the section titled ***Example Plug−ins***. You can install the examples in the same location as you installed Eclipse or you can choose a different location. If you install the examples in the same location as Eclipse, the example views and editors will show up in your Eclipse environment (e.g. ***Windows > Show View ... > Other ... > SWT Examples*** If the SWT example views do not appear right away in your menu, restart eclipse with the ***−clean*** argument). This is fine but it is not necessary.
2. Select ***New > Project ...*** from the ***File*** menu.
3. Select ***Java Project*** and click on the ***Next*** button.
4. Give the java project a name such as "SWT Examples".
5. Select ***Create project from existing source*** and click on the ***Browse ...*** button to locate the following directory:
   `eclipse/plugins/org.eclipse.sdk.examples.source_3.1.0/src/org.eclipse.sw`
6. Click on the ***Next*** button.
7. Click on the ***Projects*** tab and click on the ***Add*** button.
8. Place a check beside ***org.eclipse.swt*** and click on the ***OK*** button.
9. Click on the ***Finish*** button.

At this point your SWT examples should be compiled without any errors. Check the ***Problems*** view for errors. If you get an error like "`java.lang.Object` not found" it means you have not configured a JRE. Go to the

Window > Preferences ... dialog and select the ***Java > Installed JREs*** preference page. Ensure that a JRE is installed and that the path to the JRE is correct.

Running the Example

Now you can run the SWT standalone examples.

1. Open the Java perspective.
2. In the Packages view, select the main class that you want to run. For example, the main class for the Address Book example is `org.eclipse.swt.examples.addressbook.AddressBook`.
3. Select ***Run > Run As... > Java Application*** from the main menu.

Examples Overview

Consult the documentation of each individual example for the name of its main class and additional details. The following examples are included in the `swtexamples.jar`:

- Address Book
- Browser Example
- Clipboard
- Controls
- Custom Controls
- Drag and Drop Example
- File Viewer
- Hello World [1–5]
- Hover Help
- Image Analyzer
- Java Syntax Viewer
- Layouts
- Paint Example
- Text Editor

SWT standalone example – Address Book

The AddressBook example shows how a Table control can be used to present information in a tabular format. The application can save and load data from a file, sort the entries, and search for strings within the fields.

Running the example

Follow the SWT standalone examples setup instructions to install and run the example from your workspace.

The "Main" class is `org.eclipse.swt.examples.addressbook.AddressBook`.

This example can also be run using the Example Launcher. Select the ***Address Book*** item from the ***Standalone*** category and click ***Run***.

SWT Example Launcher

The Example Launcher is used to launch SWT examples, which can either be Workbench views or standalone applications.

- Workbench views are examples that are integrated into Eclipse. When the launcher starts a Workbench view, it is opened in the currently active perspective.
- Standalone applications are launched in a separate window.

For information on how to run the standalone examples without the SWT Example Launcher, refer to SWT standalone examples setup.

The SWT Workbench view examples can also be launched directly without using the SWT Example Launcher. SWT Workbench view examples can be found under the ***SWT Examples*** category of the ***Show Views*** dialog.

Running the Example Launcher

From Eclipse's ***Window*** menu, select ***Show View > Other***. In the ***Show View*** dialog, expand ***SWT Examples*** and select the ***SWT Example Launcher*** view. A view containing a list of examples will appear in your current perspective. When you select an example from the list a brief description of the example is displayed. Click on the ***Run*** button to launch the example.

SWT example – Browser

The Browser Example is a simple demonstration of the SWT Browser widget. It consists of a composite containing a Browser widget to render HTML and some additional widgets to implement actions commonly found on browsers (toolbar with back, forward, refresh and stop buttons, status bar etc.).

Running the example

Follow the SWT standalone examples setup instructions to install and run the example from your workspace.

The "Main" class is `org.eclipse.swt.examples.browserexample.BrowserExample`.

This example can also be run using the Example Launcher. Select the ***Web Browser*** item from the ***Workbench Views*** category and click ***Run***.

SWT standalone example – Clipboard

The Clipboard example shows the various SWT clipboard transfer types in use. The example can cut, copy and paste using Text, RTF, HTML and File transfer types.

Running the example

Follow the SWT standalone examples setup instructions to install and run the example from your workspace.

The "Main" class is `org.eclipse.swt.examples.clipboard.ClipboardExample`.

This example can also be run using the Example Launcher. Select the ***Clipboard*** item from the ***Standalone*** category and click ***Run***.

SWT example – Controls

The Controls Example is a simple demonstration of common SWT controls. It consists of a tab folder where each tab in the folder allows the user to interact with a different control. The user can change styles and settings and view how this affects each control.

Running the example

Follow the SWT standalone examples setup instructions to install and run the example from your workspace.

The "Main" class is `org.eclipse.swt.examples.controlexample.ControlExample`.

This example can also be run using the Example Launcher. Select the ***Controls*** item from the ***Workbench Views*** category and click ***Run***.

SWT example – Custom Controls

The Custom Controls example is a simple demonstration of emulated SWT controls. It consists of a tab folder where each tab in the folder allows the user to interact with a different emulated control. The user can change styles and settings and view how this affects each control.

Running the example

Follow the SWT standalone examples setup instructions to install and run the example from your workspace.

The "Main" class is
`org.eclipse.swt.examples.controlexample.CustomControlExample`.

This example can also be run using the Example Launcher. Select the ***Custom Controls*** item from the ***Workbench Views*** category and click ***Run***.

SWT standalone example – Drag and Drop

The Drag and Drop example shows the various SWT data transfer types in use. The example can drag and drop using Text, RTF, HTML and File transfer types.

Running the example

Follow the SWT standalone examples setup instructions to install and run the example from your workspace.

The "Main" class is `org.eclipse.swt.examples.dnd.DNDExample`.

This example can also be run using the Example Launcher. Select the ***Drag and Drop*** item from the ***Standalone*** category and click ***Run***.

SWT standalone example – File Viewer

The File Viewer example shows how a simple application can be implemented using SWT. This application provides the ability to navigate files and folders on the local file system and manipulate them using drag and drop. It uses alternate threads for long actions and demonstrates the use of the Tree, Table, and Toolbar widgets and the Program class.

Running the example

Follow the SWT standalone examples setup instructions to install and run the example from your workspace.

The "Main" class is `org.eclipse.swt.examples.fileviewer.FileViewer`.

This example can also be run using the Example Launcher. Select the ***File Viewer*** item from the ***Standalone*** category and click ***Run***.

SWT standalone example – Hover Help

The Hover Help example shows how to implement custom tooltips and hover help support on various SWT controls including Buttons, TableItems, ToolItems and TreeItems. To see the custom tooltips in action, hover over an item or button in the UI, and notice that images appear in the left−hand corner of the tooltip. To see the custom hover help in action, hover over an item or button in the UI until the tooltip is displayed. Then, without moving the mouse, press ***F1*** and a new Shell will be shown with the extended hover help information for the UI element.

Running the example

Follow the SWT standalone examples setup instructions to install and run the example from your workspace.

The "Main" class is `org.eclipse.swt.examples.hoverhelp.HoverHelp`.

This example can also be run using the Example Launcher. Select the ***Hover Help*** item from the ***Standalone*** category and click ***Run***.

SWT standalone example – Image Analyzer

The ImageAnalyzer example opens image files and displays their visual contents and an image data summary. The user can make adjustments to various elements of the image such as scaling and Alpha blending, and can save these changes to a file.

The ImageAnalyzer can load and display image files of type GIF, JPEG, BMP, ICO, and PNG. If a loaded file is an interlaced GIF or PNG, or a progressive JPEG, and ***Incremental Display*** is selected, then the ImageAnalyzer will display the image increments as they are loaded. If the file contains an animated GIF, then the ***Next***, ***Previous***, and ***Animate*** buttons become enabled, and can be used to cycle through and animate the images in the file. If a GIF defines a background color, as many animated GIFs do, then selecting ***Background*** will use the GIF's background color. If the image has transparency, which is possible with images of type GIF, PNG, and ICO, then selecting ***Display Mask*** will draw the image's transparency mask to the right of the image. You can change the background color of the ImageAnalyzer in order to see the transparency work. To turn off transparency, deselect ***Display Transparency***. After an image is loaded, it can

be scaled with the ***Scale*** combo, or have alpha transparency applied to it using the ***Alpha−K*** combo and ***Alpha*** menu. ***File > Reopen*** restores the scaling and alpha attributes to their default values and reloads the current image file. If the image has transparency, ***File > Save Mask As...*** can be used to save the image's transparency mask.

When SWT loads an image file, an instance of `org.eclipse.swt.graphics.ImageData` is created (though in the case of an ICO file or multi−image GIF an array of `ImageData` instances is created). The ImageAnalyzer displays all of the data stored in the `ImageData` instance(s) for the currently loaded image file, including the pixel data. Hovering over a pixel in the image display will show the RGB color data for that pixel. For certain images, particularly animated GIFs, additional data is stored in the `org.eclipse.swt.graphics.ImageLoader` instance that is used to load the image. The ImageAnalyzer displays this data as well.

Running the example

Follow the SWT standalone examples setup instructions to install and run the example from your workspace.

The "Main" class is `org.eclipse.swt.examples.imageanalyzer.ImageAnalyzer`.

This example can also be run using the Example Launcher. Select the ***Image Analyzer*** item from the ***Standalone*** category and click ***Run***.

SWT standalone example – Java Syntax Viewer

This example shows how to implement a user−defined line styler for the StyledText widget. The example provides a typical editor interface. To see the effect of the line styler, open a *.java file, and when the contents of the file are displayed in the editor area, notice that the keywords have been highlighted.

Running the example

Follow the SWT standalone examples setup instructions to install and run the example from your workspace.

The "Main" class is `org.eclipse.swt.examples.javaviewer.JavaViewer`.

This example can also be run using the Example Launcher. Select the ***Java Syntax Viewer*** item from the ***Standalone*** category and click ***Run***.

SWT example – Layouts

This example is a simple demonstration of common SWT layouts. It consists of a tab folder where each tab in the folder allows the user to interact with a different SWT layout. The user can insert widgets into a layout and set the values of the layout data using a property sheet. When the user has a suitable arrangement, the underlying code can be generated by clicking on the ***Code*** button.

Running the example

Follow the SWT standalone examples setup instructions to install and run the example from your workspace.

The "Main" class is `org.eclipse.swt.examples.layoutexample.LayoutExample`.

This example can also be run using the Example Launcher. Select the ***Layouts*** item from the ***Workbench Views*** category and click ***Run***.

SWT example – Paint Tool

This example demonstrates the use of SWT graphics operations in the form of a rudimentary bitmap painting program. The Paint Tool implementation also demonstrates a mechanism for managing timed GUI operations in the background that are triggered by user input.

Select a tool with which to draw in the drawing area. There are a number of tools to choose from on the toolbar. To change the color selection, click on a color in the palette below the drawing area; left−click to set the foreground color, right−click to set the background color.

Running the example

Follow the SWT standalone examples setup instructions to install and run the example from your workspace.

The "Main" class is `org.eclipse.swt.examples.paint.PaintExample`.

This example can also be run using the Example Launcher. Select the ***Paint*** item from the ***Workbench Views*** category and click ***Run***.

SWT standalone example – Text Editor

This example demonstrates how to use a StyledText widget to implement a text editor with formatting support. The example has a typical text editor interface. The ***Edit*** menu contains ***Cut***, ***Copy***, ***Paste*** and ***Set Font*** items. The toolbar provides a bold text toggle, three colour tools, and a reset button, all of which operate on the current selection.

Running the example

Follow the SWT standalone examples setup instructions to install and run the example from your workspace.

The "Main" class is `org.eclipse.swt.examples.texteditor.TextEditor`.

This example can also be run using the Example Launcher. Select the ***Text Editor*** item from the ***Standalone*** category and click ***Run***.

Controls

A ***Control*** is a widget that you can create and place anywhere you want in your widget parent/child tree. The SWT API reference and examples contains detailed information about the different kinds of controls and their usage. The ***org.eclipse.swt.widgets*** package defines the core set of widgets in SWT. The following table summarizes the concrete types of controls provided in this package.

Widget	Purpose
Browser	Control containing a native HTML renderer.
Button	Selectable control that issues notification when pressed and/or released.

Canvas	Composite control that provides a surface for drawing arbitrary graphics. Often used to implement custom controls.
Caret	An i−beam that is typically used as the insertion point for text.
Combo	Selectable control that allows the user to choose a string from a list of strings, or optionally type a new value into an editable text field.
Composite	Control that is capable of containing other widgets.
CoolBar	Composite control that allows users to dynamically reposition the cool items contained in the bar.
CoolItem	Selectable user interface object that represents a dynamically positionable area of a cool bar.
Group	Composite control that groups other widgets and surrounds them with an etched border and/or label.
Label	Non−selectable control that displays a string or an image.
Link	Selectable control that displays a text with links.
List	Selectable control that allows the user to choose a string or strings from a list of strings.
Menu	User interface object that contains menu items.
MenuItem	Selectable user interface object that represents an item in a menu.
ProgressBar	Non−selectable control that displays progress to the user, typically in the form of a bar graph.
Sash	Selectable control that allows the user to drag a rubber banded outline of the sash within the parent window. Used to allow users to resize child widgets by repositioning their dividing line.
Scale	Selectable control that represents a range of numeric values.
ScrollBar	Selectable control that represents a range of positive numeric values. Used in a Composite that has V_SCROLL and/or H_SCROLL styles.
Shell	Window that is managed by the OS window manager. Shells can be parented by a Display (top level shells) or by another shell (secondary shells).
Slider	Selectable control that represents a range of numeric values. A slider is distinguished from a scale by providing a draggable thumb that can adjust the current value along the range.
Spinner	Selectable control that allows the user to enter and modify numeric values.
TabFolder	Composite control that groups pages that can be selected by the user using labeled tabs.
TabItem	Selectable user interface object corresponding to a tab for a page in a tab folder.
Table	Selectable control that displays a list of table items that can be selected by the user. Items are presented in rows that display multiple columns representing different aspects of the items.
TableColumn	Selectable user interface object that represents a column in a table.
TableItem	Selectable user interface object that represents an item in a table.
Text	Editable control that allows the user to type text into it.
ToolBar	Composite control that supports the layout of selectable tool bar items.
ToolItem	Selectable user interface object that represents an item in a tool bar.
Tracker	User interface object that implements rubber banding rectangles.

Tray	Represents the system tray that is part of the task bar status area on some operating systems.
TrayItem	Selectable user interface object that represents an item in the operating system's system tray.
Tree	Selectable control that displays a hierarchical list of tree items that can be selected by the user.
TreeColumn	Selectable user interface object that represents a column in a tree.
TreeItem	Selectable user interface object that represents a hierarchy of tree items in a tree.

Events

Once we create a display and some widgets, and start up the application's message loop, where does the real work happen? It happens every time an event is read from the queue and dispatched to a widget. Most of the application logic is implemented as responses to user events.

The basic pattern is that you add a listener to some widget that you have created, and when the appropriate event occurs the listener code will be executed. This simple example is adapted from ***org.eclipse.swt.examples.helloworld.HelloWorld3***:

```
Display display = new Display ();
Shell shell = new Shell (display);
Label label = new Label (shell, SWT.CENTER);
...
shell.addControlListener (new ControlAdapter () {
   public void controlResized (ControlEvent e) {
      label.setBounds (shell.getClientArea ());
   }
});
```

For each type of listener, there is an interface that defines the listener (*Xyz**Listener***), a class that provides event information (*Xyz**Event***), and an API method to add the listener (***add**XyzListener***). If there is more than one method defined in the listener interface then there is an adapter (*Xyz**Adapter***) that implements the listener interface and provides empty methods. All of the events, listeners, and adapters are defined in the package ***org.eclipse.swt.events***.

The following tables summarize the events that are available and the widgets that support each event. Events can be split into two general categories: high level events which represent a logical operation on a control, and low level events which describe more specific user interactions. High level events may be represented by multiple low level events which may differ per platform. Low level events should generally only be used for custom widget implementations.

High level events

Event Type	Description
Activate, ***Decativate***	Generated when a Control is activated or deactivated.
Arm	A MenuItem is armed (highlighted and ready to be selected).
Close	A Shell is about to close as requested by the window manager.
DefaultSelection	The user selects an item by invoking a default selection action. For example, by hitting Enter or double clicking on a row in a Table.
Dispose	A widget is about to be disposed, either programmatically or by user.

DragDetect	The user has initiated a possible drag operation.
Expand, ***Collapse***	An item in a Tree is expanded or collapsed.
Help	The user has requested help for a widget. For example, this occurs when the F1 key is pressed under Windows.
Iconify, ***Deiconify***	A Shell has been minimized, maximized, or restored.
MenuDetect	The user has requested a context menu.
Modify	The widget's text has been modified.
Move, ***Resize***	A control has changed position or has been resized, either programmatically or by user.
Selection	The user selects an item in the control. For example, by single clicking on a row in a Table or by keyboard navigating through the items.
SetData	Data needs to be set on a TableItem when using a virtual table.
Show, ***Hide***	A control's visibility has changed.
Traverse	The user is trying to traverse out of the control using a keystroke. For example, the escape or tab keys are used for traversal.
Verify	A widget's text is about to be modified. This event gives the application a chance to alter the text or prevent the modification.

Low level events

Event Type	Description
FocusIn, ***FocusOut***	A control has gained or lost focus.
KeyDown, ***KeyUp***	The user has pressed or released a keyboard key when the control has keyboard focus.
MouseDown, ***MouseUp***, ***MouseDoubleClick***	The user has pressed, released, or double clicked the mouse over the control.
MouseMove	The user has moved the mouse above the control.
MouseEnter, ***MouseExit***, ***MouseHover***	The mouse has entered, exited, or hovered over the control.
MouseWheel	The mouse wheel has been rotated.
Paint	The control has been damaged and requires repainting.

Untyped events

In addition to the typed event system described above, SWT supports a low level, untyped widget event mechanism. The untyped mechanism relies on a constant to identify the event type and defines a generic listener that is supplied with this constant. This allows the listener to implement a "case style" listener. In the following snippet, we define a generic event handler and add several listeners to a shell.

```
Shell shell = new Shell ();
Listener listener = new Listener () {
   public void handleEvent (Event e) {
      switch (e.type) {
         case SWT.Resize:
            System.out.println ("Resize received");
            break;
         case SWT.Paint:
            System.out.println ("Paint received");
```

```
            break;
         default:
            System.out.println ("Unknown event received");
      }
   }
};
shell.addListener (SWT.Resize, listener);
shell.addListener (SWT.Paint, listener);
```

Custom widgets

You may want to extend SWT by implementing your own custom widget. SWT itself provides a package, ***org.eclipse.swt.custom***, which contains custom controls that are not in the core set of SWT controls but are needed to implement the platform workbench.

Control	Purpose
CBanner	CBanner is used in the workbench to layout the toolbar area and perspective switching toolbar.
CCombo	Similar to Combo, but is vertically resizable allowing it to fit inside table cells.
CLabel	Similar to Label, but supports shortening of text with an ellipsis. Also supports a gradient effect for the background color as seen in the active workbench view. Does not support wrapping.
CTabFolder	Similar to TabFolder, but supports additional configuration of the visual appearance of tabs (top or bottom) and borders.
CTabItem	Selectable user interface object corresponding to a tab for a page in a CTabFolder.
SashForm	Composite control that lays out its children in a row or column arrangement and uses a Sash to separate them so that the user can resize them.
ScrolledComposite	Composite control that scrolls its contents and optionally stretches its contents to fill the available space.
StyledText	Editable control that allows the user to type text. Ranges of text inside the control can have distinct colors and font styles.
ViewForm	ViewForm is used in the workbench to position and size a view's label/toolbar/menu local bar.

Implementing a custom widget

Once you've determined that you need a custom widget and have decided which platforms must be supported, you can consider several implementation techniques for your widget. These techniques can be mixed and matched depending on what is available in the underlying OS platform.

Native implementation

If your application requires a native widget that is not provided by SWT, you will need to implement it natively. This may be a platform widget, a third party widget, or any other widget in a platform shared library. A complete example of a native custom widget implementation can be found in Creating Your Own Widgets using SWT.

Combining existing widgets

Widgets can be combined to form widgets that are more sophisticated. For example, a Combo can be implemented using a text entry widget along with a button and a drop−down list. To implement a combined widget, you create a subclass of ***Composite*** and manage the children internally.

A simple example can be found in ***CCombo***.

Custom drawn implementation

In some cases, you don't have any native code or existing widgets that help you in the implementation of your new widget. This means you must draw the widget yourself in the handler for the Paint event. Although this technique can become quite complicated, it has the advantage of producing a completely portable implementation.

Custom drawn controls are implemented by subclassing the ***Canvas*** or ***Composite***. Subclass ***Canvas*** if your widget will not contain any child controls.

The internal implementation of a custom drawn widget usually involves these major tasks:

- Create any graphics objects needed in your constructor and store them in an instance variable. Register a listener for the ***dispose*** event on your canvas or composite so that you can free these objects when the widget is destroyed.
- Add a ***paintListener*** to your canvas or composite and paint the widget according to your design. For complex widgets, a lot of work goes into optimizing this process by calculating and repainting only what's absolutely necessary.
- Ensure that any API calls that affect the appearance of your widget trigger a repaint of the widget. In general, you should use ***redraw*** to damage your widget when you know you must repaint, rather than call your internal painting code directly. This gives the platform a chance to collapse the paint you want to generate with any other pending paints and helps streamline your code by funneling all painting through one place.
- If your widget defines events in its API, determine what low level ***Canvas*** or ***Composite*** events will trigger your widget's events. For example, if you have a clicked event, you will want to register a mouse event on your canvas and perform calculations (such as hit testing) to determine whether the mouse event in your canvas should trigger your widget event.

Many of the widgets implemented in the ***org.eclipse.swt.custom*** use this approach. A simple example can be found in ***CLabel***.

Further information on custom widgets can be found in Creating your own widgets using SWT.

Layouts

Often the best way to handle simple widget positioning is in a resize event listener. However, there are common patterns used by applications when placing widgets. These patterns can be structured as configurable layout algorithms that can be reused by many different applications.

SWT defines *layouts* that provide general purpose positioning and sizing of child widgets in a composite. Layouts are subclasses of the abstract class ***Layout***. The SWT standard layouts can be found in the ***org.eclipse.swt.layout*** package.

There are some general definitions used when resizing and positioning widgets:

- The *location* of a widget is its x,y coordinate location within its parent widget.
- The *preferred size* of a widget is the minimum size needed to show its content. This is computed differently for each kind of widget.
- The *clientArea* is the area in which a child can be placed without being clipped.
- The *trim* is the distance between a widget's client area and its actual border. Trim is occupied by the widget's borders or extra space at its edge. The size and appearance of the trim is widget and platform dependent.

These concepts are relevant for applications regardless of whether a layout is used. You can think of a layout as a convenient way to package resize functionality for reuse.

Some additional concepts are introduced by layouts:

- Some layouts support *spacing* between widgets in the layout.
- Some layouts support a *margin* between the edge of the layout and the widget adjacent to the edge.

See Understanding layouts in SWT for further discussion and pictures demonstrating these concepts.

The following code snippet shows the simple case of an application using a resize callback to size a label to the size of its parent shell:

```
Display display = new Display ();
Shell shell = new Shell (display);
Label label = new Label (shell, SWT.CENTER);
shell.addControlListener (new ControlAdapter () {
   public void controlResized (ControlEvent e) {
      label.setBounds (shell.getClientArea ());
   }
});
```

The next snippet uses a layout to achieve the same effect:

```
Display display = new Display ();
Shell shell = new Shell (display);
Label label = new Label (shell, SWT.CENTER);
shell.setLayout (new FillLayout ());
```

Even for this simple example, using a layout reduces the application code. For more complex layouts, the simplification is much greater.

The following table summarizes the standard layouts provided by SWT.

Layout	Purpose
FillLayout	Lays out controls in a single row or column, forcing them to be the same size.
FormLayout	Positions the children by using FormAttachments to optionally configure the left, top, right and bottom edges of each child.
GridLayout	Positions the children by rows and columns.
RowLayout	Places the children either in horizontal rows or vertical columns.

Custom layouts

Occasionally you may need to write your own custom ***Layout*** class. This is most appropriate when you have a complex layout that is used in many different places in your application. Note that unless you are writing a very generic layout that will be used by several ***Composite*** widgets, it is sometimes simpler and easier to calculate sizes and position children in a resize listener.

Layouts are responsible for implementing two methods:

- ***computeSize(...)*** calculates the width and height of a rectangle that encloses all of the composite's children once they have been sized and placed according to the layout algorithm. The hint parameters allow the width and/or height to be constrained. For example, a layout may choose to grow in one dimension if constrained in another.
- ***layout(...)*** positions and sizes the composite's children. A layout can choose to cache layout–related information, such as the preferred extent of each of the children. The ***flushCache*** parameter tells the ***Layout*** to flush cached data, which is necessary when other factors besides the size of the composite have changed, such as the creation or removal of children, or a change in the widget's font.

A third method, ***flushCache(...)***, can be optionally implemented to clear any cached data associated with a specific control. Often, the computeSize() method of a widget can be expensive, and so layouts can cache results to improve performance.

Further discussion of custom layouts can be found in Understanding layouts in SWT.

Error handling

SWT can trigger three types of exceptions: ***IllegalArgumentException***, ***SWTException***, and ***SWTError***. Applications should not have to catch any other kind of exception or error when calling SWT.

> *Note: If any other exception besides these three is thrown from SWT, it should be considered a bug in the SWT implementation.*

Where possible, exceptions are triggered consistently across platforms. However, some errors are specific to an SWT implementation on a particular platform.

IllegalArgumentException

The arguments passed in SWT API methods are checked for appropriate state and range before any other work is done. An ***IllegalArgumentException*** will be thrown when it is determined that an argument is invalid.

Code that causes an ***IllegalArgumentException*** on one platform will cause the same exception on a different platform.

SWTException

SWTException is thrown when a recoverable error occurs internally in SWT. The error code and message text provide a further description of the problem.

SWT remains in a known stable state after throwing the exception. For example, this exception is thrown when an SWT call is made from a non−UI thread.

SWTError

SWTError is thrown when an unrecoverable error occurs inside SWT.

SWT will throw this error when an underlying platform call fails, leaving SWT in an unknown state, or when SWT is known to have an unrecoverable error, such as running out of platform graphics resources.

Once an SWT error has occurred, there is little that an application can do to correct the problem. These errors should not be encountered during normal course of operation in an application, but high reliability applications should still catch and report the errors.

Graphics

SWT provides a graphics engine for drawing graphics and displaying images in widgets. You can get pretty far without ever programming to the graphics interface, since widgets handle the painting of icons, text, and other data for you. However, if your application displays custom graphics, or if you are implementing a custom drawn widget, then you will need to understand some basic drawing objects in SWT.

Graphics context

The graphics context, ***GC***, is the focal point for SWT graphics support. Its API describes all of the drawing capabilities in SWT.

A GC can be used for drawing on a control (the most common case), on an image, on a display, or to a printer. When drawing on a control, you use the GC supplied to you in the control's paint event. When drawing on an image, display, or printer, you must create a GC configured for it, and dispose of the GC when you are finished using it.

Once you've got a GC, you can set its attributes, such as color, line width, and font, which control the appearance of the graphics drawn in the GC.

The API Reference for ***GC*** describes the complete set of graphics functions.

Fonts

The ***Font*** and ***FontData*** classes are used when manipulating fonts in SWT.

FontData describes the characteristics of a font. You can create a FontData by specifying a font name, style, and size. FontData includes API for querying these attributes. Since FontData does not allocate any OS resources, you do not need to dispose of it.

The Font is the actual graphic object representing a font that is used in the drawing API. You create a Font for a ***Display*** by specifying the Display and the FontData of the font that you want. You can also query a Font for its FontData.

You must dispose of an allocated Font when you are finished using it.

Colors

Colors are similar to fonts. You create a ***Color*** for a Display by specifying the RGB values for the desired color. You must dispose of an allocated color when you are finished using it.

The Display method `getSystemColor(int)` allows you to query the predefined system colors for the OS platform. You should not free colors obtained using this technique.

The color model is discussed in detail in the article SWT color model.

Images

The ***Image***, ***ImageData***, and ***ImageLoader*** classes are used when manipulating Images in SWT.

ImageData describes the actual pixels in the image, using the ***PaletteData*** class to describe the utilized color values. ImageData is a device− and platform−independent description of an image.

ImageLoader loads and saves ImageData in different file formats. SWT currently supports loading and saving of image formats including *BMP* (Windows Bitmap), *ICO* (Windows Icon), *JPEG*, *GIF*, and *PNG*.

The Image is the actual graphic object representing the image that is used in the drawing API. You create an image for a particular Display. Images can be created in several ways:

- use an ImageData to initialize the image's contents
- copy an existing Image
- load an Image from a file

Regardless of how you create the Image, you are responsible for disposing it.

Graphics object lifecycle

Most of the graphics objects used for drawing in SWT allocate resources in the underlying OS and must be explicitly freed. The same rule discussed earlier applies here. If you create it using a constructor, you should free it. If you get access to it from somewhere else, do not free it.

Creation

Graphics objects such as graphics contexts, fonts, colors, and images are allocated in the OS as soon as the object is created. How you plan to use your graphics objects determines when you should create them.

For graphics objects that are used heavily throughout the application, you can create them at the time that you create your widgets. This is commonly done for colors and fonts. In other cases, it is more appropriate to create your graphics objects on the fly. For example, you might create a graphics context in one of your widget event handlers in order to perform some calculations.

If you are implementing a custom widget, you typically allocate graphics objects in the constructor if you always make use of them. You might allocate them on the fly if you do not always use them or if they are dependent upon the state of some attribute.

Painting

Once you have allocated your graphics objects, you are ready to paint. *You should always do your painting inside of a paint listener.* There are rare cases, particularly when implementing custom widgets, when you paint while responding to some other event. However this is generally discouraged. If you think you need to paint while handling some other event, you should first try to use the ***redraw()*** method, which will generate another paint event in the OS. Drawing outside of the paint method defeats platform optimizations and can cause bugs depending upon the number of pending paints in the event queue.

When you receive a paint event, you will be supplied with a ***GC*** pre−configured for drawing in the widget. *Do not free this* ***GC***! You did not create it.

Any other graphics objects must be allocated while handling the event (or beforehand). Below is a snippet based on the ***org.eclipse.swt.examples.HelloWorld5*** sample. The color red was previously allocated when creating the widget, so it can be used here.

```
shell.addPaintListener (new PaintListener () {
   public void paintControl (PaintEvent event) {
      GC gc = event.gc;
      gc.setForeground (red);
      Rectangle rect = event.widget.getClientArea ();
      gc.drawRectangle (rect.x + 10, rect.y + 10, rect.width - 20, rect.height - 20);
      gc.drawString (resHello.getString("Hello_world"), rect.x + 20, rect.y + 20);
   }
});
```

Disposal

Every graphics object that you allocate must be freed when you are finished using it.

The timing of the disposal depends upon when you created the object. If you create a graphics object while creating your widget, you should generally add a dispose listener onto the widget and dispose of the graphics when the widget is disposed. If you create an object on the fly while painting, you should dispose of it when finished painting.

The next code snippet shows a slightly modified version of our paint listener. In this example, it allocates and frees the color red while painting.

```
shell.addPaintListener (new PaintListener () {
   public void paintControl (PaintEvent event) {
      GC gc = event.gc;
      Color red = new Color (event.widget.getDisplay (), 0xFF, 0, 0);
      gc.setForeground (red);
      Rectangle rect = event.widget.getClientArea ();
      gc.drawRectangle (rect.x + 10, rect.y + 10, rect.width - 20, rect.height - 20);
      gc.drawString (resHello.getString ("Hello_world"), rect.x + 20, rect.y + 20);
      red.dispose ();
   }
});
```

UI Forms

UI Forms is an optional Rich Client plug−in based on SWT and JFace that provides the support for creating portable web−style user interfaces across all Eclipse UI categories. It provides a few carefully chosen custom widgets, layouts and support classes required to achieve the desired Web look. Being based on SWT, they are inherently portable across all the platforms where SWT is supported.

UI Forms break the established expectations on which classes of widgets can be expected in Eclipse workbench UI categories (editors, views, wizards, dialogs). An Eclipse form can appear in any and all of them, expanding the possibilities of the UI developers to use the most appropriate concept for the task regardless where they are.

Eclipse Forms add the following to make web−style user interfaces possible:

- A concept of a 'form' that is suitable for inclusion in the content areas such as views and editors
- A toolkit that manages colors, hyperlink groups and other aspects of a form, as well as serve as a factory for a number of SWT controls
- A new layout manager that lays out controls similar to HTML table layout algorithm
- A set of custom control designed to fit in the form (hyperlink, image hyperlink, scrollable composite, section)
- A multi−page editor where each page is a form (e.g. PDE manifest editors)

UI Forms controls

UI Forms are based on SWT and JFace. Consequently, standard SWT controls are routinely used. However, a few custom controls have been added in order to deliver a rich web−style user interface when combined with the standard SWT control set:

- Form
- Hyperlink
- ExpandableComposite and Section
- FormText

Form control

Form is a basic control used to host UI Forms. It provides for setting a title and scrolling the content similar to a Web browser. What makes forms appealing is the fact that the content is an SWT composite that can be used as you would use it in other contexts. For example, consider the following code snippet:

```
public class FormView extends ViewPart {
        private FormToolkit toolkit;
        private ScrolledForm form;
        /**
         * The constructor.
         */
        public FormView() {
        }
        /**
         * This is a callback that will allow us to create the viewer and
         * initialize it.
         */
        public void createPartControl(Composite parent) {
                toolkit = new FormToolkit(parent.getDisplay());
                form = toolkit.createScrolledForm(parent);
                form.setText("Hello, Eclipse Forms");
        }
        /**
         * Passing the focus request to the form.
         */
        public void setFocus() {
                form.setFocus();
        }
        /**
         * Disposes the toolkit
         */
        public void dispose() {
                toolkit.dispose();
                super.dispose();
        }
}
```

UI Forms manipulate SWT widgets in a number of ways in order to achieve the desired effect. For that reason, controls are typically created using the `FormToolkit`. Normally an instance of a `ScrolledForm` is created in order to get scrolling. When forms need to be nested, a `Form` instance provides everything except for scrolling of the form content.

Form content is rendered below the title. SWT widgets are created in the form using `Form.getBody()` as a parent.

Hyperlink control

Hyperlink is a custom widget created to complement the standard SWT widget set when used in the context of UI Forms. Hyperlink is a selectable text control that acts like a Web browser hyperlink:

```
Hyperlink link = toolkit.createHyperlink(form.getBody(), "Click here.",
                        SWT.WRAP);
        link.addHyperlinkListener(new HyperlinkAdapter() {
                public void linkActivated(HyperlinkEvent e) {
                        System.out.println("Link activated!");
                }
        });
        link.setText("A sample link");
```

Hyperlinks fire `HyperlinkEvent` objects when users interact with them. By adding a `HyperlinkListener`, clients can capture when the mouse enters and exits the link, as well as activates it (either by mouse click or by 'Enter' key).

Hyperlinks created by the form toolkit are automatically inserted into a **hyperlink group**. `HyperlinkGroup` manages common hyperlink properties like normal and hover foreground color, underline style etc. for all the links that belong to the group.

Since many hyperlinks are combined with a small image, UI Forms provide a subclass called `ImageHyperlink` that add the ability to combine text and image in one clickable control. This class can also be used when a hyperlink image (without text) is needed. If image is not set, `ImageHyperlink` behaves identically to `Hyperlink`.

Expandable composite and Section controls

`ExpandableComposite` acts similar to `Group` control with the ability to collapse a portion of a page a toggle control:

```
ExpandableComposite ec = toolkit.createExpandableComposite(form.getBody(),
                                ExpandableComposite.TREE_NODE|
                                ExpandableComposite.CLIENT_INDENT);
ec.setText("Expandable Composite title");
String ctext = "We will now create a somewhat long text so that "+
"we can use it as content for the expandable composite. "+
"Expandable composite is used to hide or show the text using the "+
"toggle control";
Label client = toolkit.createLabel(ec, ctext, SWT.WRAP);
ec.setClient(client);
ec.addExpansionListener(new ExpansionAdapter() {
        public void expansionStateChanged(ExpansionEvent e) {
                form.reflow(true);
        }
});
```

The `ExpandableComposite` control accepts a number of styles that affect its appearance and behavior. Style `TREE_NODE` will create the toggle control used in a tree widget for expanding and collapsing nodes, while `TWISTIE` will create a triangle–style toggle. Using `EXPANDED` will create the control in the initial expanded state. If style `COMPACT` is used, control will report width in the collapsed state enough to fit in the title line only (i.e. when collapsed, it will be as compact horizontally as possible). Finally, `CLIENT_INDENT` will indent the client to align with the title (otherwise, client will be aligned with the toggle control).

Expandable composite itself is responsible for rendering the toggle control and the title. The control to expand or collapse is set as a client. Note the requirement that the client is a direct child of the expandable composite.

Expandable composite fires `ExpansionEvent` objects when expansion state changes. Adding an expansion listener to the control is needed in order to reflow the form on state change. This is because expansion causes changes in expandable composite size, but the change will not take effect until the next time the parent is laid out (hence we need to force it).

`Section` is a subclass of the expandable composite that adds additional capabilities. It is typically used to partition a form into a number of sections, each with its own title and optional description. When `Section.TITLE_BAR` or `Section.SHORT_TITLE_BAR` styles are used, decoration around the title area further enhances the grouping.

Unlike `ExpandableComposite`, `Section` automatically handles reflows on expansion state change. Other interesting uses of the expansion state notification are lazy creation of the Section content that is delayed until the section is expaned.

FormText control

It is possible to achieve highly polished results using images, hyperlinks and text snippets mixed together in a form. However, when the mix of these elements is needed as part of one integral text, it is very hard to do. To remedy the problem, UI Forms offer a rudimentary text control that can do the following:

- Render plain wrapped text
- Render plain text but convert any segment that starts with **http://** into a hyperlink on the flyv
- Render text with XML tags

In all the modes, `FormText` control is capable of rendering either a string or an input stream.

Rendering normal text (label mode)

```
FormText rtext = toolkit.createFormText(form.getBody(), true);
String data = "Here is some plain text for the text to render.";
rtext.setText(data, false, false);
```

Second argument set to `false` means that we will treat input text as–is, and the third that we will not try to expand URLs if found.

Automatic conversion of URLs into hyperlinks

It is possible to still handle the text as normal but automatically convert segments with http:// protocol into hyperlinks:

```
FormText rtext = toolkit.createFormText(form.getBody(), true);
String data = "Here is some plain text for the text to render; "+
              "this text is at http://www.eclipse.org web site.";
rtext.setText(data,true,true);
```

Similar to the `Hyperlink` control, `FormText` accepts listeners that implement `HyperlinkListener`. These listeners will be notified about events related to the hyperlink segments within the control.

Parsing formatting markup

The most powerful use of the `FormText` control is when formatting tags are added to the text. The expected root tag is `form`. It can have one or more children that can either be `<p>` or `<li>`. Either of these can have normal text, text between `<b>` or `<span>` tags, images, links and `SWT` controls. Images are declared using `<img href="image key"/>` (no content), while links are expressed using `<a href="href">text</a>`.

Some of the tags mentioned above have additional attributes. Tag `<a>` can accept `nowrap="true"` to block the link from being wrapped into the new line. Tag `<p>` can have attribute `vspace="false"` (`true` by default) that adds additional space between paragraphs. Tag `<li>` has more attributes:

- **style –** can be `text`, `bullet` and `image` (default is `bullet`)
- **value –** not used for `bullet`; if style is `text`, the value will be rendered instead in place of a bullet; if style is `image`, value represents a key in the image table of an image to be rendered in place of a bullet
- **vspace –** the same as for the '`p`' tag.
- **indent –** the number of pixels to indent text
- **bindent –** the number of pixels to indent the bullet (this number is independent from 'indent' – be careful not to overlap them)

Tags that affect appearance of the normal text are `<b>` (works as expected), and `<span>`. The later allows you to change font and/or color of the text within the tag. Finally, soft line breaks can be added using `<br/>` tag (note that this is XML, so you cannot use open `<br>` as in HTML).

Since release 3.1, `FormText` can be used to mix SWT widgets inside text, hyperlinks and images. SWT controls are created as children of `FormText`, which makes `FormText` a layout manager of a sort, with instruction on where to place the control relative to text embedded directly in the XML.

One common theme that can be observed is that `FormText` is not responsible for loading images, fonts, resolving links or colors. This is not a browser and it is much better to separate concerns and simply assign images and colors managed elsewhere. Both links and images simply have '`href`' attribute to reference them. For links, the value of this attribute will be provided in the hyperlink event when listeners are notified. Images need to be registered with the text control using the matching 'href' key. This way, the control does not need to worry about loading the images – it has them in the hash table and can render them immediately.

Similar approach has been used for colors and fonts. Colors are already handled by the toolkit, so you can allocate as many as you want using a unique key and RGB values by calling `toolkit.getColors().createColor()`. What is left is to set all the colors referenced in the '`span`' tag so that the control will be able to use them during rendering.

UI Forms Layouts

UI Forms offer two new layouts over those provided by SWT. These layouts implement the common interface and can be used on any UI SWT composite, but are typically used in conjunction with UI Forms. The purpose of these layouts is to manage form layout in a way that is similar to Web browsers. Controls are laid out with respect to the form width. The goal is to respect form width as much as possible and compensate by growing the form vertically (and employing scroll bars where needed).

TableWrapLayout

TableWrapLayout is a grid−based layout very similar to versatile SWT's GridLayout. It differs from it in that it uses a layout algorithm that work more like HTML tables. It tries to respect the provided client area with and grow vertically to compensate.

There are many similarities between GridLayout and TableWrapLayout. Both organize children in grids. Both have layout data that instructs the layout how to treat each control. Both can accept hints on which control should grab excess space etc.

However, they fundamentally differ in the approach to the layout. `TableWrapLayout` starts with columns. It computes minimal, preferred and maximum widths of each column and uses this information to assign excess space. It also tries to be fair when dividing space across columns so that there is no excess wrapping of some controls.

It is possible to mix `GridLayout` and `TableWrapLayout` but the branch where `GridLayout` is used is the one where wrapping stops. This is quite acceptable if you don't want it to wrap (if the composite contains controls that cannot wrap anyway, like text, buttons, trees etc.). However, you should have an unbroken path from the form body to each text control that needs to wrap.

ColumnLayout

Another custom layout in UI Forms is a variation of the `RowLayout`. If we configure `RowLayout` to place children vertically (in columns), and to make all controls the same with within the column, we would get several columns (depending on the width of controls), but the last column would typically not be completely filled (depending on the number of controls). Again, if placed in a form, we would get all the controls in one column because `RowLayout` cannot do 'vertical' wrapping. If we use `GridLayout`, we must choose the number of columns up front and live with the choice.

There are situations in more complex forms where we want the number of columns to be adaptive. In other words, we would like the number to change depending on the width of the form – use more when possible, drop the number down as the width decreases. We would also like to fill the form area more–less equally (with all the columns roughly the same height). All this can be achieved with `ColumnLayout`.

Compared to `TableWrapLayout`, `ColumnLayout` is much simpler. Hardly any configuration is needed. The only choice you need to make is the range of columns you want to have (default is 1 to 3).

Color and font management

When using forms in a non−trivial way, it is important to share as much as possible to conserve resources. For this reason, color management should be separated from the toolkit when there are more than one form to handle.

Of course, it is possible to create one toolkit per form, but that is too wasteful if there are many forms. Instead:

- Create one toolkit for all the forms that have the same life cycle. For example, if creating a multi−page editor, create one toolkit per editor and dispose it when editor is disposed. All the pages in the editor should share this toolkit.
- Create one color manager (`FormColors`) per plug−in. When creating the toolkit, pass the color manager to the toolkit. The toolkit will know that the colors are shared and will not dispose them.
- Use platform support for fonts and if possible, use `JFaceResources` predefined fonts. Between default, 'banner' and 'header' fonts, you can accomplish a lot. Using many fonts is very confusing for the user, and if you do manage your own, you must ensure alternatives across platforms. The JFace fonts are guaranteed to work on all the platforms Eclipse ships on.
- Dispose the color manager on plug−in shutdown (don't assume that plug−in shutdown also means platform shutdown − Eclipse runtime can uninstall your plug−in dynamically while the platform is still running).
- Use form color manager to allocate all the colors needed by the forms.

Managed forms

Managed forms are wrappers that add life cycle management and notification to form members. Managed form is not a form by itself. It *has* a form and accepts registration of `IFormPart` element. For each `IFormPart`, it manages events like dirty state, saving, commit, focus, selection changes etc. In order to reach to the wrapped form widget, call the '`getfForm()`' method.

There is a similarity between managed forms and JFace viewers – the relationship between a form and a managed form is similar to the one between a `Table` widget and `TableViewer` in JFace, for example.

Not every control on the form needs to be a form part. It is better to group a number of controls and implement `IFormPart` interface for the group. Section is a natural group and Eclipse Form provides `SectionPart` implementation. It implements the interface and contains a `Section` instance (either created outside and passed into the constructor, or created in the part itself).

Master/Details block

Master/Details is a pattern used throughout the UI world. It consists of a list or a tree ('master') and a set of properties ('details') driven by the selection in the master. Eclipse Forms provide the implementation of the pattern as a useful building block with the following properties:

- While details part is created, master part factory method is abstract and must be implemented by the subclass
- Master and details parts are children of the sash form and the ratio of the form space allocated for each can be changed by moving the sash.
- Through the nature of the sash form, master and details parts can be organized horizontally or vertically in the form.

The idea of master/details block is to create a tree or a table section that fires the selection notification via the managed form. If the details part can handle the selected object, it should switch to the page for it and display properties. When building on top of the provided master/details block, subclasses should:

- Create the master part (the one that drives the details)
- Contribute actions to the form tool bar (consumes upper−right portion of the form in the title area)
- Register details pages, one for each distinct input that can arrive from the master part

Multi–page form editors

UI Forms provide a basic support for multi–page editors you can build on.

You should start building a UI Forms multi–page editor by extending `FormEditor`:

```
public class SimpleFormEditor extends FormEditor {

        public SimpleFormEditor() {
        }

        protected FormToolkit createToolkit(Display display) {
                // Create a toolkit that shares colors between editors.
                return new FormToolkit(ExamplesPlugin.getDefault().getFormColors(
                                display));
        }

        protected void addPages() {
                try {
                addPage(new FreeFormPage(this));
                addPage(new SecondPage(this));
                addPage(new ThirdPage(this));
                addPage(new MasterDetailsPage(this));
                addPage(new PageWithSubPages(this));
                }
                catch (PartInitException e) {
                        //
                }
        }

        public void doSave(IProgressMonitor monitor) {
        }

        public void doSaveAs() {
        }

        public boolean isSaveAsAllowed() {
                return false;
        }
```

A very simple way to get started is to create pages and add them as above. Each page need to implement `FormPage` and override `createFormContent(IManagedForm managedForm)` method. Obviously there is a managed form already created in the page, and you should create contents in the enclosed form, and also register any form part that needs to be part of the managed life cycle.

In addition to form pages, you can add one or more text editors as a raw source alternative to the GUI pages. For this, you should call '`addPage(IEditorPart, IEditorInput input)`' method in the superclass.

Recommended practices for Eclipse Forms multi–page editors

There are many ways you can go about writing a form–based multi–page editor. It mostly depends on the type of content you are editing and proficiency of your users. There are two ways you can approach it:

1. If the typical users are using the editor infrequently, raw source is hard to edit by hand or complex, your users are not very technical etc., you should make COMPLETE pages that are fully capable of editing every aspect of the content without the need to turn to the raw source. In this approach, source page is there only for occasional validation, rather than for regular work. In that respect, you can get away with a basic text editor. PDE extension point schema editor falls into this group.
2. If your users are more technical, have no problem editing the file by hand but would appreciate some help from time to time, consider providing a mixed experience – make a good source editor with all the add–ons like incremental outline, context assist, syntax highlighting etc. In turn, add complex value–add functionality in the form pages that are hard to achieve from source. We have learned from experience that it is very hard to convince seasoned users to switch from source editing if the value–add is marginal or debatable. However, function that was only available in GUI pages and was very high–quality was used readily.

Creating a high quality multi–page editor with mixed GUI and source pages has its challenges. Accepting that users will switch pages frequently requires a good model of the underlying content. The model should be directly tied to the underlying document(s) so that it is in sync both when users type in the text directly and when they change it structurally through the GUI pages (don't forget the indirect changes caused by other workbench actions while the editor is still up).

Menu and toolbar paths

We've seen many action contributions that specify the path for the location of their action. Let's take a close look at what these paths mean.

Menu paths

We'll look at menu paths first by looking at the workbench **Help** menu.

Named groups in the workbench

The locations for inserting new menus and menu items are defined using named groups. You can think of a named group as a slot or placeholder that allows you to insert your menu items at certain points in a menu bar or pulldown menu.

The workbench defines all of its group slot names in the classes **IWorkbenchActionConstants** and **IIDEActionConstants**. (Two different classes are used since resource–related menu items are factored out of the generic workbench). For each workbench menu, named groups are placed in the menu at locations where it is expected that plug–ins will insert new actions.

The following description of the help menu is adapted from the **IWorkbenchActionConstants** class definition.

```
Standard Help menu actions
Start group - HELP_START - "start"
End group - HELP_END - "end"
```

The standard workbench help menu defines a named group called "**start**," followed by a named group called "**end**,". Defining two groups gives plug–ins a little more control over where their contributed items will be positioned within the help menu. When you define a menu, you can define as many slots as you like. Adding more slots gives other plug–ins more control over where their contributions appear relative to existing

contributions.

Plug−ins that add a menu item to the help menu can use these group names to decide where their menu item will go. For example, the cheatsheet plug−in adds an action set containing the "Cheat Sheets..." menu to the workbench. Here's the markup from the **org.eclipse.ui.cheatsheets** plug−in's **plugin.xml**.

```
<extension
        point="org.eclipse.ui.actionSets">
        <actionSet
                label="%CHEAT_SHEETS"
                visible="true"
                id="org.eclipse.ui.cheatsheets.actionSet">
                <action
                        label="%CHEAT_SHEETS_MENU"
                        class="org.eclipse.ui.internal.cheatsheets.actions.CheatSheetHelpMenuActi
menubarPath="help/helpStart"
                        id="org.eclipse.ui.cheatsheets.actions.CheatSheetHelpMenuAction">
                </action>
        </actionSet>
</extension>
```

The new help action will be placed in the help menu, inside the **helpStart** group.

Fully qualified menu paths

A complete menu path is simply "menu name/group name." Most menu names for the workbench are defined in **IWorkbenchActionConstants**. (Resource−related menu names are defined in **IIDEActionConstants**.) If we look for the name of the help menu in this class, we'll find that the fully qualified path name for our help action is "**help/helpEnd**."

Some menus have nested submenus. This is where longer paths come into play. If the help menu had defined a submenu called "**submenu**" with a named group called "**submenuStart**," then the fully qualified menu path for an action in the new submenu would be "**help/submenu/submenuStart**."

Externalizing UI labels

The example above demonstrates a technique for externalizing strings that appear in the UI. Externalized strings are used to make translating the plug−in's UI to other languages simpler. We can externalize the strings in our **plugin.xml** files by replacing the string with a key (**%CHEAT_SHEETS_MENU**) and creating entries in the **plugin.properties** file of the form:

```
        CHEAT_SHEETS_MENU = Cheat Sheets...
```

The **plugin.properties** file can be translated for different languages and the **plugin.xml** will not need to be modified.

Adding new menus and groups

In many of the examples we've seen so far, the actions contributed by the sample plug−ins have been added to existing named groups within menus.

The **actionSets**, **viewActions**, **editorActions**, and **popupMenus** extension points also allow you to define new menus and groups within your contribution. This means that you can define new submenus or new

pull−down menus and contribute your actions to these new menus. In this case, the path for your new action will contain the name of your newly defined menu.

We saw this technique when the readme tool defined a new menu for its action set. Let's look at the markup one more time now that we've looked at menu paths in more detail.

```
<extension point = "org.eclipse.ui.actionSets">
<actionSet id="org_eclipse_ui_examples_readmetool_actionSet"
        label="%ActionSet.name"
        visible="true">
 menu id="org_eclipse_ui_examples_readmetool"
                label="%ActionSet.menu"
path="window/additions">
                <separator name="slot1"/>
                <separator name="slot2"/>
                <separator name="slot3"/>
        </menu>
        <action id="org_eclipse_ui_examples_readmetool_readmeAction"
menubarPath="window/org_eclipse_ui_examples_readmetool/slot1"
                toolbarPath="readme"
                label="%ReadmeAction.label"
                tooltip="%ReadmeAction.tooltip"
                helpContextId="org.eclipse.ui.examples.readmetool.open_browser_action_context"
                icon="icons/ctool16/openbrwsr.png"
                class="org.eclipse.ui.examples.readmetool.WindowActionDelegate"
                enablesFor="1">
                <selection class="org.eclipse.core.resources.IFile"
                             name="*.readme">
                </selection>
        </action>
        ...
```

We added a new menu called "**org_eclipse_ui_examples_readmetool**" whose label is defined in by the key "**%ActionSet.name**" in the properties file. Within this menu, we define three named groups: "**slot1**," "**slot2**," and "**slot3**." We add this new menu to the path "**window/additions**."

If we go back to **IWorkbenchActionConstants**, we see this definition of the window menu in the javadoc:

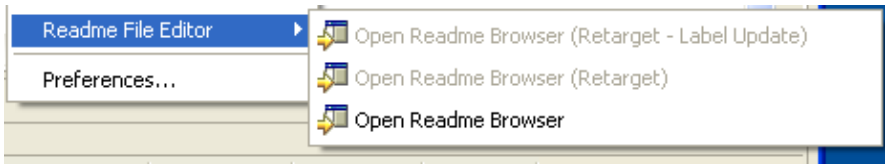
```
 * <h3>Standard Window menu actions</h3>
 * <ul>
 * <li>Extra Window-like action group (<code>WINDOW_EXT</code>)</li>
```

If we look further at the class definition, we will see these related definitions:

```
public static final String MENU_PREFIX = "";
...
public static final String M_WINDOW = MENU_PREFIX+"window";
...
public static final String MB_ADDITIONS = "additions";  // Group.
...
public static final String WINDOW_EXT = MB_ADDITIONS;   // Group.
```

From this information, we can piece together the path for adding something to the workbench "Window" menu. The menu itself is called "**window**" and it defines one slot called "**additions**." We use the path "**window/additions**" to add our new menu.

In the action set declaration, we add an action to our newly defined menu, using the path "**window/org_eclipse_ui_examples_readmetool/slot1**."



Other plug−ins could add to our menu by using this same path (or perhaps one of the other slots) to add one of their own menus.

In the readme tool example, we use the **separator** attribute to identify the group names. This will cause a separator line to appear between these groups when they contain items. We could instead use the **groupMarker** attribute if we want to define a named group without showing any separators in the menu to distinguish between the groups.

Tool bar paths

Tool bar paths work similarly to menu paths.

Named tool bars in the workbench

The workbench tool bar is composed of tool bars contributed by different plug−ins, including the workbench itself. Within any particular tool bar, there are named groups or slots that can be used for inserting new tool bar items.

The following description of the workbench tool bars is adapted from the **IWorkbenchActionConstants** class definition.

```
// Workbench toolbar ids
public static final String TOOLBAR_FILE = "org.eclipse.ui.workbench.file"
public static final String TOOLBAR_NAVIGATE = "org.eclipse.ui.workbench.navigate";

// Workbench toolbar group ids.  To add an item at the beginning of the group,
// use the GROUP id.  To add an item at the end of the group, use the EXT id.
public static final String PIN_GROUP = "pin.group";
public static final String HISTORY_GROUP = "history.group";
public static final String NEW_GROUP = "new.group";
public static final String SAVE_GROUP = "save.group";
public static final String BUILD_GROUP = "build.group";
```

In the simplest case, a plug−in can contribute a tool bar item in its own tool bar. For example, the readme tool actions contributed to the menu are also given a tool bar path:

```
<action id="org_eclipse_ui_examples_readmetool_readmeAction"
   menubarPath="window/org_eclipse_ui_examples_readmetool/slot1"
   toolbarPath="readme"
...
```

Since there is no reference to the workbench tool bar paths or groups, the readme actions appear in their own group on the tool bar. Specifying the following path would instead place the item in the file tool bar in the save group:

```
...
<action id="org_eclipse_ui_examples_readmetool_readmeAction"
   menubarPath="window/org_eclipse_ui_examples_readmetool/slot1"
   toolbarPath="org.eclipse.ui.workbench.file/save.group"
...
```

The paths defined in **IWorkbenchActionConstants** may be referenced in the tool bar paths of other plug−ins.

Adding to action sets of another plug−in

Suppose a plug−in wants its tool bar items better integrated with actions from a different plug−in? Let's look at how the external tools plug−in (**org.eclipse.ui.externaltools**) integrates its action with the debugger tool bar. The debugger (**org.eclipse.debug.ui**) defines its tool bar actions like this:

```
<extension
      point="org.eclipse.ui.actionSets">
   <actionSet
         label="%LaunchActionSet.label"
         visible="false"
         id="org.eclipse.debug.ui.launchActionSet">
   ...
   <action
         toolbarPath="debug"
         id="org.eclipse.debug.internal.ui.actions.RunDropDownAction"
         hoverIcon="icons/full/ctool16/run_exc.png"
         class="org.eclipse.debug.internal.ui.actions.RunToolbarAction"
         disabledIcon="icons/full/dtool16/run_exc.png"
         icon="icons/full/etool16/run_exc.png"
         helpContextId="run_action_context"
         label="%RunDropDownAction.label"
         pulldown="true">
   </action>
   ...
```

Just like the readme tool, the debugger plug−in defines its own tool bar path, which means its tool bar items will be inside their own tool bar on the workbench. What does the external tools plug−in do?

```
<extension point="org.eclipse.ui.actionSets">
        <actionSet
                id="org.eclipse.ui.externaltools.ExternalToolsSet"
                label="%ActionSet.externalTools"
                visible="true">
                ...
                <action
                        id="org.eclipse.ui.externaltools.ExternalToolMenuDelegateToolbar"
                        definitionId= "org.eclipse.ui.externaltools.ExternalToolMenuDelegateToolb
                        label="%Action.externalTools"
toolbarPath="org.eclipse.debug.ui.launchActionSet/debug"
                        disabledIcon="icons/full/dtool16/external_tools.png"
                        icon="icons/full/etool16/external_tools.png"
                        hoverIcon="icons/full/ctool16/external_tools.png"
                        tooltip="%Action.externalToolsTip"
                        pulldown="true"
                        class="org.eclipse.ui.externaltools.internal.menu.ExternalToolMenuDelegat
                </action>
        </actionSet>
</extension>
```

Note the use of the action set ID of the debugger in the tool bar path. Using an action set ID in the path denotes that the tool bar item should be placed in the tool bar used by the referenced action set. Within a toolbar group, items are ordered by action set id, so for our example, the external tools action will appear after the debugger actions.

When adding to an action set's tool bar, new groups can also be defined. If the external tools plug−in defined its **toolbarpath** as **"org.eclipse.debug.ui.launchActionSet/external"** a new group would be created for the action on the tool bar. As with menus, tool bar groups are delineated by separators.

Using paths from another plug−in

In general, it's not good practice to contribute to another plug−in's menu or tool bar by deriving the path name from the **plugin.xml** unless it has been marked specifically as being available for clients. It's possible that a future version of the plug−in could change the names of the paths. Two common ways to mark your plug−in's action set ids and paths as fair game are:

- annotate the XML with comments that explicitly mark the menu path or action set as usable by clients
- define a public interface (much like **IWorkbenchActionConstants**) which specifies exactly which menus, tool bar groups, and slots are considered fair game for use by other plug−ins

Action set part associations

Once your plug−in defines an action set, it can use the **org.eclipse.ui.actionSetPartAssociations** extension point to specify that an action set should be made visible when a particular view or editor is active.

Ultimately, the user controls the appearance of action sets using **Window−>Customize Perspectives...** in the workbench menu. If the user marks an action set visible, it will always be visible when the perspective is active, regardless of the active view or editor. Likewise, if the user marks the action set as hidden, it will always be hidden when the perspective is active. If the user does not change the state of an action set in this dialog, then the action set part associations are used to determine the visibility of the action set.

The markup for an action set part association is straightforward. The following example comes from the Java development tools (JDT) UI plug−in.

```
   <extension point="org.eclipse.ui.actionSetPartAssociations">
        <actionSetPartAssociation
targetID="org.eclipse.jdt.ui.CodingActionSet">
                <part id="org.eclipse.jdt.ui.PackageExplorer"/>
                <part id="org.eclipse.jdt.ui.TypeHierarchy" />
                <part id="org.eclipse.jdt.ui.CompilationUnitEditor"/>
                <part id="org.eclipse.jdt.ui.ClassFileEditor"/>
                <part id="org.eclipse.jdt.ui.ProjectsView"/>
                <part id="org.eclipse.jdt.ui.PackagesView"/>
                <part id="org.eclipse.jdt.ui.TypesView"/>
                <part id="org.eclipse.jdt.ui.MembersView"/>
        </actionSetPartAssociation>
</extension>
```

The **targetID** specifies the action set. (The **CodingActionSet** was previously defined in the JDT plug−in manifest.) One or more **part** attributes can be specified to indicate which views and editors will cause the action set to become visible in the menus and toolbar. The effect of this extension contribution is that the actions associated with writing Java code will only be visible when one of the specified views is active.

Boolean expressions and action filters

When a plug−in contributes an action to the workbench UI using one of the menu extension points, it can specify the conditions under which the menu item is visible and/or enabled in the menu. In addition to supplying simple enabling conditions, such as selection counts and selection classes, plug−ins can use **boolean expressions** for more flexibility in determining when an action should be visible or enabled.

Boolean enablement expressions

Boolean expressions can contain the boolean operators (NOT, AND, OR) combined with a predefined syntax for evaluating certain conditions. Many of these conditions test a particular object. The identity of the "object in focus" (the object being tested) depends upon the specific context of the enablement expression:

- **instanceof** tests whether the type of the object in focus is a subtype of the specified type name.
- **test** tests whether the value of a named property of the object in focus matches the specified value.
- **systemTest** tests whether the value of a named system property matches the specified value.
- **equals** tests whether the object in focus is equal to the specified value.
- **count** tests the number of elements in a list.
- **with** changes the object in focus to the object referenced by a supplied variable.
- **resolve** changes the object in focus to the object referenced by a supplied variable, supplying additional arguments with the variable.
- **adapt** adapts the object in focus to the type specified.
- **iterate** iterates over a variable that is a collection and combines the boolean value of each value using AND or OR.

When specifying a value to be tested against any of these expressions, the value is assumed to be a string except for when the following conversions are successful:

- the string "true" is converted into Boolean.TRUE
- the string "false" is converted into Boolean.FALSE
- if the string contains a dot, the interpreter tries to convert the value into a Float object
- if the string only consists of numbers, the interpreter converts the value into an Integer object
- the conversion into a Boolean, Float, or Integer can be suppressed by surrounding the string with single quotes.

A complete definition of enablement XML syntax can be found in the extension point reference documentation for any extension that defines an **enablement** element, such as **org.eclipse.ui.popupMenus**.

Prior to R3.0, these generalized boolean expressions were not available. The following predefined expressions were used to evaluate certain conditions without building a general expression. Note that any of these expressions could now be expressed with the more generalized syntax. The predefined expressions can still be used as follows:

- **objectClass −** true if each object in the selection subclasses or implements the class.
- **objectState −** true if the named attribute equals the specified value. **IActionFilter** assists in evaluating the expression. An action filter dynamically computes the enablement criteria for an action based on the target selection and the value of named attributes.
- **systemProperty −** true if the named system property equals the specified value.
- **pluginState −** specifies whether the specified plugin (by **id**) should be **installed** or **activated**

For example, the following snippets represent enablement expressions that could be used on a hypothetical action in an action set:

```
<action id="org.eclipse.examples.actionEnablement.class"
       label="Red Element"
       menubarPath="additions"
       class="org.eclipse.examples.actionEnablement.ObjectTestAction">
       <enablement>
         <and>
           <objectClass name="org.eclipse.examples.actionEnablement.TestElement"/>
           <objectState name="name" value="red"/>
         </and>
       </enablement>
</action>

<action id="org.eclipse.examples.actionEnablement.property"
       label="Property"
       menubarPath="additions"
       class="org.eclipse.examples.actionEnablement.PropertyTestAction">
       <enablement>
           <systemProperty name="MyTestProperty" value="puppy"/>
       </enablement>
</action>

<action id="org.eclipse.examples.actionEnablement.pluginState"
       label="Installed"
       menubarPath="additions"
       class="org.eclipse.examples.actionEnablement.PluginTestAction">
       <enablement>
           <pluginState id="x.y.z.anotherPlugin" value="installed"/>
       </enablement>
</action>
```

See the reference documentation of the extension points for more elaborate samples of these expressions and a complete description of the XML.

The following table lists extension points that contribute actions and summarizes how XML markup attributes and boolean expressions can be used to affect enablement.

Extension point name	Attributes affecting enablement	Boolean expressions
viewActions, **editorActions**, **actionSets**	**enablesFor –** specifies the selection count that must be met for the action to be enabled **selection class –** the class that the selected objects must subclass or implement in order for the action to be enabled **selection name –** a wild card filter that can be applied to the objects in the selection.	**visibility –** a boolean expression. Controls whether the menu item is visible in the menu. **enablement –** a boolean expression. Controls whether the menu item is enabled in the menu. The **enablesFor** attribute and the **selection class** and **name**, and must be satisfied before applying the enablement expression.
popupMenus	(For object contributions only.) **objectClass –** specifies the class that objects in the selection must subclass or implement	(For both object and viewer contributions) **visibility –** a boolean expression. Controls whether the menu item is visible in the

	(For both object and viewer contributions) **enablesFor –** specifies the selection count that must be met for the action to be enabled **selection class –** the class that the selected objects must subclass or implement to enable the action **selection name –** a wild card filter that can be applied to the objects in the selection.	menu. **enablement –** a boolean expression. Controls whether the menu item is enabled in the menu. The **enablesFor** attribute and the **selection class** and **name**, and must be satisfied before applying the enablement expression.

Using objectState with content types

The ability to define content types (see Content types) can be combined with boolean expressions to define very specific enablement or visibility conditions based on the content type of a resource. For example, the following snippet makes a popup menu item visible only if the selected file's content matches the plug–in's specialized content types.

```
<extension point="org.eclipse.ui.popupMenus">
   <objectContribution
      id="com.example.objectContributions"
      objectClass="org.eclipse.core.resources.IFile"
      nameFilter="*.xml">
         <visibility>
            <or>
               <objectState
                  name="contentTypeId"
                  value="com.example.employeeRecordContentType"/>
               <objectState
                  name="contentTypeId"
                  value="com.example.customerRecordContentType"/>
            </or>
         </visibility>
         <action id="com.example.action1"
         ...
```

The **contentTypeId** attribute can be used in an objectState expression to check the content type of the selected xml file. This allows a plug–in to apply very specific content checking before enabling or showing menu actions related to specific types of files. See Content types for more detail about the content type extension.

Retargetable actions

It is common for a plug–in's views and editors to implement actions that are semantically similar to existing workbench actions, such as clipboard cut/copy/paste, view refresh, or properties. The popup menu for views and editors can become quite cluttered if every view or editor has to define unique actions for these operations and include them in their menus.

To solve this problem, the workbench defines **retargetable** (also called **global**) actions that can be handled by any view or editor. When a view or editor is active, its handler will be run when the user chooses the action from the workbench menu or toolbar. This allows views and editors to share workbench menu space for

semantically similar actions.

IWorkbenchActionConstants documents all of the workbench actions and denotes retargetable actions as global. For example, here is the definition of the **Properties** action.

```
public static final String PROPERTIES = "properties"; // Global action.
```

The following table summarizes some of the more common retargetable actions that are implemented by views and editors:

	File menu	**Edit menu**	**Navigate menu**	**Project menu**
views	move rename refresh properties		go into go to resource sync with editor back forward up next previous	open close build rebuild
editors	revert print	find		
views and editors		cut copy paste delete select all undo redo		

Retargetable actions are created using **RetargetAction**. The following snippet is from **WorkbenchActionBuilder**.

```
 propertiesAction = createGlobalAction(IWorkbenchActionConstants.PROPERTIES, "file", false);
```

The **createGlobalAction** method shows us exactly how to make a **RetargetAction**.

```
private RetargetAction createGlobalAction(String id, String actionDefPrefix, boolean labelRetarge
        RetargetAction action;
        if (labelRetarget) {
                action = new LabelRetargetAction(id, WorkbenchMessages.getString("Workbench." + i
        }
        else {
                action = new RetargetAction(id, WorkbenchMessages.getString("Workbench." + id));
        }
        ...
        return action;
}
```

When creating a retargetable action, the workbench assigns the id for the action and the default label. Note that there are two styles of retarget actions. **RetargetAction** simply allows a view or editor to reimplement an action. **LabelRetargetAction** also allows views and editors to reset the label of the action. This is useful for

making the menu label more specific, such as relabeling an **Undo** action as **Undo Typing**.

Now we know how the retarget actions are defined by the workbench. Let's look next at how your view or editor can provide an implementation for a retargetable action. This is done by setting a global action handler.

Setting a global action handler

A plug–in contributes a retargetable action for a view or editor part by implementing an **IAction** and registering it as a global action handler with the part's action bars. This is usually done at the time that the part creates its actions and controls. The name of the retargeted action (as defined in **IWorkbenchActionConstants**) is used to specify which action the handler is intended for. The following shows how the workbench task list registers its handler for the **PROPERTIES** action.

```
public void createPartControl(Composite parent) {
        ...
makeActions();
        ...

        // Add global action handlers.
        ...
        getViewSite().getActionBars().
                setGlobalActionHandler(
                IWorkbenchActionConstants.PROPERTIES,
                propertiesAction);
        ...
```

The properties action is created in the local method **makeActions**:

```
void makeActions() {
        ...
        // properties
        propertiesAction = new TaskPropertiesAction(this, "properties");
        propertiesAction.setText(TaskListMessages.getString("Properties.text"));
        propertiesAction.setToolTipText(TaskListMessages.getString("Properties.tooltip"));
        propertiesAction.setEnabled(false);
}
```

That's all that is needed. Your action will be run when the user chooses the action from the workbench menu bar or tool bar and your view or editor is active. The workbench handles the details of ensuring that the retargeted action is always associated with the currently active view or editor.

Retargetable editor actions

Recall that the readme tool defines its own editor which contributes actions to the workbench menu bar using its **ReadmeEditorActionBarContributor**.

```
<extension
    point = "org.eclipse.ui.editors">
        <editor
           id = "org.eclipse.ui.examples.readmetool.ReadmeEditor"
           name="%Editors.ReadmeEditor"
           icon="icons/obj16/editor.png"
           class="org.eclipse.ui.examples.readmetool.ReadmeEditor"
           extensions="readme"
           contributorClass="org.eclipse.ui.examples.readmetool.ReadmeEditorActionBarContributor"
        </editor>
</extension>
```

Let's look closer at what happens in the contributor class.

```
public ReadmeEditorActionBarContributor() {
        ...
        actionRetargetAction(IReadmeConstants.RETARGET2, MessageUtil.getString("Editor_Action2"))
        action2.setToolTipText(MessageUtil.getString("Readme_Editor_Action2"));
        action2.setDisabledImageDescriptor(ReadmeImages.EDITOR_ACTION2_IMAGE_DISABLE);
        action2.setImageDescriptor(ReadmeImages.EDITOR_ACTION2_IMAGE_ENABLE);
        ...
        actionLabelRetargetAction(IReadmeConstants.LABELRETARGET3, MessageUtil.getString("Editor_
        action3.setDisabledImageDescriptor(ReadmeImages.EDITOR_ACTION3_IMAGE_DISABLE);
        action3.setImageDescriptor(ReadmeImages.EDITOR_ACTION3_IMAGE_ENABLE);
        ...
handler2 = new EditorAction(MessageUtil.getString("Editor_Action2"));
        ...
handler3 = new EditorAction(MessageUtil.getString("Editor_Action3"));
        ...
```

When the contributor is created, it creates two retargetable actions (one that allows label update and one that does not). Creation of the actions uses the same technique that the workbench uses. It also creates two handlers that will be used for the actions when the editor is the active part.

But where are the handlers for the actions registered? Setting the global handlers is done a little differently when your editor defines the retargeted actions. Why? Because your contributor is in charge of tracking the active view and hooking different handlers as different views or the editor itself becomes active. (The workbench does this for you when you set a handler for one of its global actions). Here's how the **ReadmeEditorActionBarContributor** sets things up:

```
public void init(IActionBars bars, IWorkbenchPage page) {
        super.init(bars, page);
        bars.setGlobalActionHandler(IReadmeConstants.RETARGET2, handler2);
        bars.setGlobalActionHandler(IReadmeConstants.LABELRETARGET3, handler3);
        ...
```

First, the contributor registers its handlers for the retargeted actions. This ensures that the contributor's actions will be run when the editor itself is active. The next step is to register each **RetargetAction** as a part listener on the page.

```
        ...
        // Hook retarget actions as page listeners
        page.addPartListener(action2);
        page.addPartListener(action3);
        IWorkbenchPart activePart = page.getActivePart();
        if (activePart != null) {
                action2.partActivated(activePart);
                action3.partActivated(activePart);
        }
}
```

Adding each **RetargetAction** as a part listener means that it will be notified when the active part changes. The action can get the correct global handler from the newly activated part. (See the implementation of **RetargetAction** for all the details.) Note that to start, the action is seeded with the currently active part.

When the editor contributor is disposed, it should unhook the retargetable actions as page listeners.

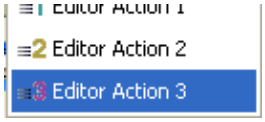
```
public void dispose() {
        // Remove retarget actions as page listeners
```

```
        getPage().removePartListener(action2);
        getPage().removePartListener(action3);
}
```

Finally, recall that action bar contributors are shared among instances of the same editor class. For this reason, the handlers must be notified when the active editor changes so that they can connect to the proper editor instance.

```
public void setActiveEditor(IEditorPart editor) {
        ...
        handler2.setActiveEditor(editor);
        handler3.setActiveEditor(editor);
        ...
}
```

That completes the setup on the editor side. When the editor is open and active, the handlers (and their labels) as defined by the **ReadmeEditorActionBarContributor** will appear in the workbench menu bar.

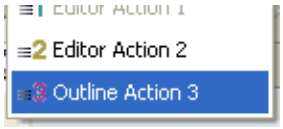


Now that the editor's contributions are in place, what does a view do to register a handler? The code on the client side is similar to registering a handler for a workbench action, except that the action id is the one defined by the plug–in's editor. The **ReadmeContentOutlinePage** registers a handler for these actions.

```
public void createControl(Composite parent) {
        super.createControl(parent);
        ...
        getSite().getActionBars().setGlobalActionHandler(
                IReadmeConstants.RETARGET2,
                new OutlineAction(MessageUtil.getString("Outline_Action2")));

        OutlineAction action = new OutlineAction(MessageUtil.getString("Outline_Action3"));
action.setToolTipText(MessageUtil.getString("Readme_Outline_Action3"));
        getSite().getActionBars().setGlobalActionHandler(
                IReadmeConstants.LABELRETARGET3,
                action);
        ...
```

Note that the outliner sets tool tip text and a label on the second action, since it allows relabeling. When the readme outliner view is made active, its handlers (and their labels) will now appear in the workbench menu bar.



Note that the relabeled action shows the new label.

Retargetable action set actions

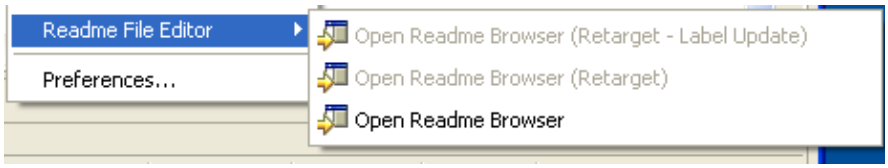
The readme tool action set also defines retargetable actions. The action remains visible as long as the readme action set is visible, but it is only enabled when a view or editor that implements the action is active. When

using action sets to define retargetable actions, the actions are created in the action set markup rather than in code. The following is from the readme tool's action set definition:

```
<extension point = "org.eclipse.ui.actionSets">
          <actionSet id="org_eclipse_ui_examples_readmetool_actionSet"
                  label="%ActionSet.name"
                  visible="true">
...
<action id="org_eclipse_ui_examples_readmetool_readmeRetargetAction"
   menubarPath="window/org_eclipse_ui_examples_readmetool/slot1"
   toolbarPath="readme"
   label="%ReadmeRetargetAction.label"
   tooltip="%ReadmeRetargetAction.tooltip"
   helpContextId="org.eclipse.ui.examples.readmetool.open_browser_action_context"
   icon="icons/ctool16/openbrwsr.png"
   retarget="true">
</action>
<action id="org_eclipse_ui_examples_readmetool_readmeRelabelRetargetAction"
   menubarPath="window/org_eclipse_ui_examples_readmetool/slot1"
   toolbarPath="readme"
   label="%ReadmeRelabelRetargetAction.label"
   tooltip="%ReadmeRelabelRetargetAction.tooltip"
   helpContextId="org.eclipse.ui.examples.readmetool.open_browser_action_context"
   icon="icons/ctool16/openbrwsr.png"
   retarget="true"
   allowLabelUpdate="true">
</action>
...
```

Retargeted actions are specified by using the **retarget="true"** attribute. This will cause a **RetargetAction** to be created in the action set. Note that the retargetable actions do not specify an implementing **class** since it is up to each view or editor in the plug–in to set up a handler that implements each action. If the **allowLabelUpdate** is true, then a **LabelRetargetAction** will be created instead.

The retargeted actions will be visible in the window menu when the readme action set is visible. However, they will not be enabled if the readme tool's editor or outline view are not active.



What do the editor and view have to do? Again, the client side is similar to registering a handler for the workbench or an editor's retargetable action. The action id specified in the markup must be used when registering a global action handler.

The **ReadmeEditorActionBarContributor** takes care of this for the editor. First, it defines the handlers for the actions.

```
public ReadmeEditorActionBarContributor() {
        ...
        handler4 = new EditorAction(MessageUtil.getString("Editor_Action4"));
        handler5 = new EditorAction(MessageUtil.getString("Editor_Action5"));
        handler5.setToolTipText(MessageUtil.getString("Readme_Editor_Action5"));
```

```
        ...
}
```

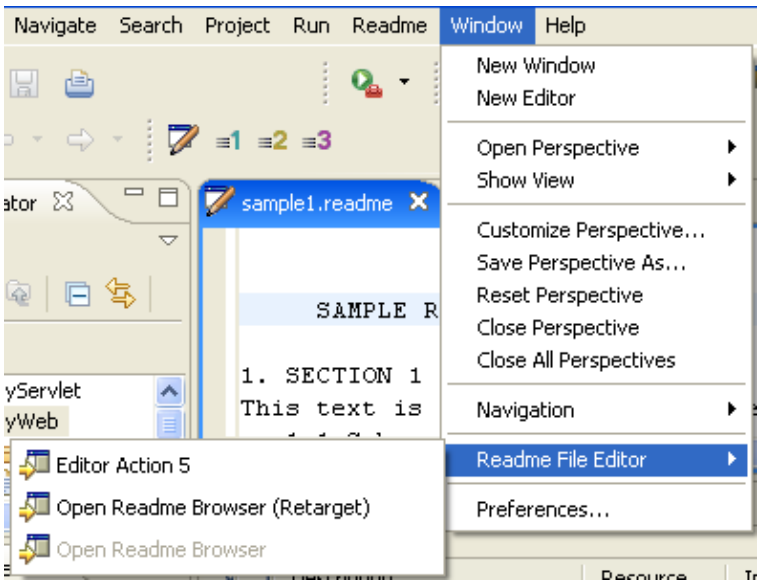
The handlers are registered at the same time that the handlers for the editor retargetable actions were registered.

```
public void init(IActionBars bars, IWorkbenchPage page) {
        ...
        bars.setGlobalActionHandler(IReadmeConstants.ACTION_SET_RETARGET4, handler4);
        bars.setGlobalActionHandler(IReadmeConstants.ACTION_SET_LABELRETARGET5, handler5);
        ...
}
```

Recall that action bar contributors are shared among different instances of the same editor. This means the handlers must be notified if the active editor for the **ReadmeEditorActionBarContributor** changes.

```
public void setActiveEditor(IEditorPart editor) {
        ...
        handler4.setActiveEditor(editor);
        handler5.setActiveEditor(editor);
        ...
}
```

That's it for the editor. We should see these actions enable when the editor is activated.



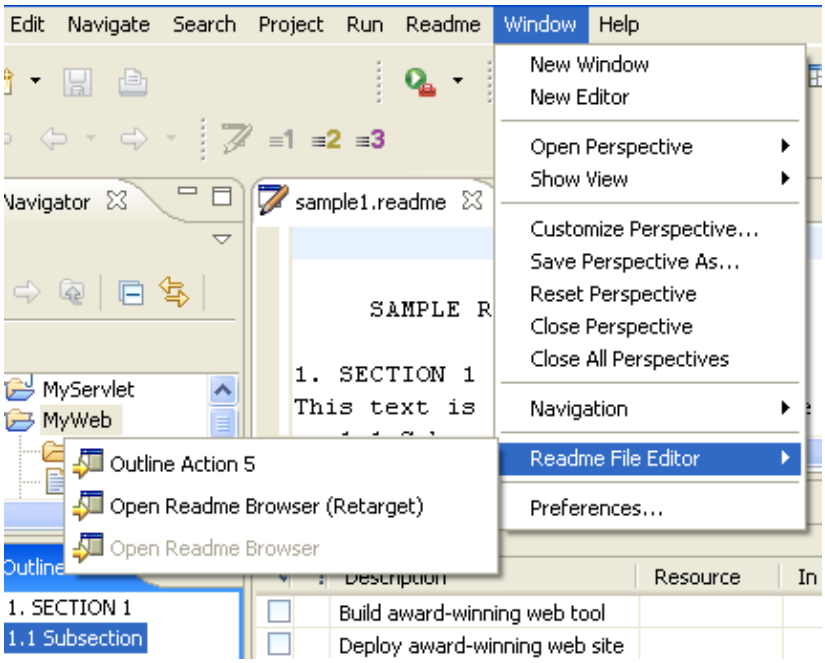
Note that the label for the first retargetable action ("Editor Action 4") was not used since the action set XML markup did not set **allowLabelUpdate**.

The **ReadmeContentOutlinePage** defines its handlers in the same place it defined handlers for the editor's retargetable actions:

```
public void createControl(Composite parent) {
        ...
        action = new OutlineAction(MessageUtil.getString("Outline_Action4"));
        getSite().getActionBars().setGlobalActionHandler(
                IReadmeConstants.ACTION_SET_RETARGET4,
                action);
```

```
        action = new OutlineAction(MessageUtil.getString("Outline_Action5"));
        action.setToolTipText(MessageUtil.getString("Readme_Outline_Action5"));
        getSite().getActionBars().setGlobalActionHandler(
                IReadmeConstants.ACTION_SET_LABELRETARGET5,
                action);
}
```

We should see its relabeled action when the content outliner is active.



Undoable operations

We've looked at many different ways to contribute actions to the workbench, but we haven't focused on the implementation of an action's `run()` method. The mechanics of the method depend on the specific action in question, but structuring the code as an **undoable operation** allows the action to participate in the platform undo and redo support.

The platform provides an **undoable operations framework** in the package **org.eclipse.core.commands.operations**. By implementing the code inside a `run()` method to create an **IUndoableOperation**, the operation can be made available for undo and redo. Converting an action to use operations is straightforward, apart from implementing the undo and redo behavior itself.

Writing an undoable operation

We'll start by looking at a very simple example. Recall the simple **ViewActionDelegate** provided in the readme example plug–in. When invoked, the action simply launches a dialog that announces it was executed.

```
public void run(org.eclipse.jface.action.IAction action) {
        MessageDialog.openInformation(view.getSite().getShell(),
                MessageUtil.getString("Readme_Editor"),
                MessageUtil.getString("View_Action_executed"));
```

```
}
```

Using operations, the run method is responsible for creating an operation that does the work formerly done in the run method, and requesting that an **operations history** execute the operation, so that it can be remembered for undo and redo.

```
public void run(org.eclipse.jface.action.IAction action) {
        IUndoableOperation operation = new ReadmeOperation(
                view.getSite().getShell());
        ...
        operationHistory.execute(operation, null, null);
}
```

The operation encapsulates the old behavior from the run method, as well as the undo and redo for the operation.

```
class ReadmeOperation extends AbstractOperation {
        Shell shell;
        public ReadmeOperation(Shell shell) {
                super("Readme Operation");
                this.shell = shell;
        }
        public IStatus execute(IProgressMonitor monitor, IAdaptable info) {
                MessageDialog.openInformation(shell,
                        MessageUtil.getString("Readme_Editor"),
                        MessageUtil.getString("View_Action_executed"));
                return Status.OK_STATUS;
        }
        public IStatus undo(IProgressMonitor monitor) {
                MessageDialog.openInformation(shell,
                        MessageUtil.getString("Readme_Editor"),
                        "Undoing view action"));
                return Status.OK_STATUS;
        }
        public IStatus redo(IProgressMonitor monitor) {
                MessageDialog.openInformation(shell,
                        MessageUtil.getString("Readme_Editor"),
                        "Redoing view action"));
                return Status.OK_STATUS;
        }
}
```

For simple actions, it may be possible to move all of the nuts and bolt work into the operation class. In this case, it may be appropriate to collapse the former action classes into a single action class that is parameterized. The action would simply execute the supplied operation when it is time to run. This is largely an application design decision.

When an action launches a wizard, then the operation is typically created as part of the wizard's `performFinish()` method or a wizard page's `finish()` method. Converting the `finish` method to use operations is similar to converting a `run` method. The method is responsible for creating and executing an operation that does the work previously done inline.

Operation history

So far we've used an **operations history** without really explaining it. Let's look again at the code that creates our example operation.

```
public void run(org.eclipse.jface.action.IAction action) {
        IUndoableOperation operation = new ReadmeOperation(
                view.getSite().getShell());
        ...
operationHistory.execute(operation, null, null);
}
```

What is the **operation history** all about? **IOperationHistory** defines the interface for the object that keeps track of all of the undoable operations. When an operation history executes an operation, it first executes the operation, and then adds it to the undo history. Clients that wish to undo and redo operations do so by using **IOperationHistory** protocol.

The operation history used by an application can be retrieved in several ways. The simplest way is to use the **OperationHistoryFactory**.

```
IOperationHistory operationHistory = OperationHistoryFactory.getOperationHistory();
```

The workbench can also be used to retrieve the operations history. The workbench configures the default operation history and also provides protocol to access it. The following snippet demonstrates how to obtain the operation history from the workbench.

```
IWorkbench workbench = view.getSite().getWorkbenchWindow().getWorkbench();
IOperationHistory operationHistory = workbench.getOperationSupport().getOperationHistory();
```

Once an operation history is obtained, it can be used to query the undo or redo history, find out which operation is the next in line for undo or redo, or to undo or redo particular operations. Clients can add an **IOperationHistoryListener** in order to receive notifications about changes to the history. Other protocol allows clients to set limits on the history or notify listeners about changes to a particular operation. Before we look at the protocol in detail, we need to understand the **undo context**.

Undo contexts

When an operation is created, it is assigned an **undo context** that describes the user context in which the original operation was performed. The undo context typically depends on the view or editor that originated the undoable operation. For example, changes made inside an editor are often local to that editor. In this case, the editor should create its own own undo context and assign that context to operations it adds to the history. In this way, all of the operations performed in the editor are considered local and semi−private. Editors or views that operate on a shared model often use an undo context that is related to the model that they are manipulating. By using a more general undo context, operations performed by one view or editor may be available for undo in another view or editor that operates on the same model.

Undo contexts are relatively simple in behavior; the protocol for **IUndoContext** is fairly minimal. The main role of a context is to "tag" a particular operation as belonging in that undo context, in order to distinguish it from operations created in different undo contexts. This allows the operation history to keep track of the global history of all undoable operations that have been executed, while views and editors can filter the history for a specific point of view using the undo context.

Undo contexts can be created by the plug−in that is creating the undoable operations, or accessed through API. For example, the workbench provides access to an undo context that can be used for workbench−wide operations. However they are obtained, undo contexts should be assigned when an operation is created. The following snippet shows how the readme plug−in's **ViewActionDelegate** could assign a workbench−wide context to its operations.

```
public void run(org.eclipse.jface.action.IAction action) {
        IUndoableOperation operation = new ReadmeOperation(
                view.getSite().getShell());
        IWorkbench workbench = view.getSite().getWorkbenchWindow().getWorkbench();
        IOperationHistory operationHistory = workbench.getOperationSupport().getOperationHistory(
        IUndoContext undoContext = workbench.getOperationSupport().getUndoContext();
        operation.addContext(undoContext);
        operationHistory.execute(operation, null, null);
}
```

Why use undo contexts at all? Why not use separate operation histories for separate views and editors? Using separate operation histories assumes that any particular view or editor maintains its own private undo history, and that undo has no global meaning in the application. This may be appropriate for some applications, and in these cases each view or editor should create its own separate undo context. Other applications may wish to implement a global undo that applies to all user operations, regardless of the view or editor where they originated. In this case, the workbench context should be used by all plug−ins that add operations to the history.

In more complicated applications, the undo is neither strictly local or strictly global. Instead, there is some cross−over between undo contexts. This can be achieved by assigning multiple contexts to an operation. For example, an IDE workbench view may manipulate the entire workspace and consider the workspace its undo context. An editor that is open on a particular resource in the workspace may consider its operations mostly local. However, operations performed inside the editor may in fact affect both the particular resource and the workspace at large. (A good example of this case is the JDT refactoring support, which allows structural changes to a Java element to occur while editing the source file). In these cases, it is useful to be able to add both undo contexts to the operation so that the undo can be performed from the editor itself, as well as those views that manipulate the workspace.

Now that we understand what an undo context does, we can look again at the protocol for **IOperationHistory**. The following snippet is used to perform an undo on the some context:

```
IOperationHistory operationHistory = workbench.getOperationSupport().getOperationHistory();
try {
        IStatus status = operationHistory.undo(myContext, progressMonitor, someInfo);
} catch (ExecutionException e) {
        // handle the exception
}
```

The history will obtain the most recently performed operation that has the given context and ask it to undo itself. Other protocol can be used to get the entire undo or redo history for a context, or to find the operation that will be undone or redone in a partcular context. The following snippet obtains the label for the operation that will be undone in a particular context.

```
IOperationHistory operationHistory = workbench.getOperationSupport().getOperationHistory();
String label = history.getUndoOperation(myContext).getLabel();
```

The global undo context, **IOperationHistory.GLOBAL_UNDO_CONTEXT**, may be used to refer to the global undo history. That is, to all of the operations in the history regardless of their specific context. The following snippet obtains the global undo history.

```
IOperationHistory operationHistory = workbench.getOperationSupport().getOperationHistory();
IUndoableOperation [] undoHistory = operationHistory.getUndoHistory(IOperationHistory.GLOBAL_UNDO
```

Whenever an operation is executed, undone, or redone using operation history protocol, clients can provide a progress monitor and any additional UI info that may be needed for performing the operation. This information is passed to the operation itself. In our original example, the readme action constructed an operation with a shell parameter that could be used to open the dialog. Instead of storing the shell in the operation, a better approach is to pass parameters to the execute, undo, and redo methods that provide any UI information needed to run the operation. These parameters will be passed on to the operation itself.

```
public void run(org.eclipse.jface.action.IAction action) {
        IUndoableOperation operation = new ReadmeOperation();
        ...
operationHistory.execute(operation, null, infoAdapter);
}
```

The **infoAdapter** is an **IAdaptable** that minimally can provide the **Shell** that can be used when launching dialogs. Our example operation would use this parameter as follows:

```
public IStatus execute(IProgressMonitor monitor, IAdaptable info) {
        if (info != null) {
                Shell shell = (Shell)info.getAdapter(Shell.class);
                if (shell != null) {
                        MessageDialog.openInformation(shell,
                                MessageUtil.getString("Readme_Editor"),
                                MessageUtil.getString("View_Action_executed"));
                        return Status.OK_STATUS;
                }
        }
        // do something else...
}
```

Undo and redo action handlers

The platform provides standard undo and redo **retargetable action handlers** that can be configured by views and editors to provide undo and redo support for their particular context. When the action handler is created, a context is assigned to it so that the operations history is filtered in a way appropriate for that particular view. The action handlers take care of updating the undo and redo labels to show the current operation in question, providing the appropriate progress monitor and UI info to the operation history, and optionally pruning the history when the current operation is invalid. An action group that creates the action handlers and assigns them to the global undo and redo actions is provided for convenience.

```
new UndoRedoActionGroup(this.getSite(), undoContext, true);
```

The last parameter is a boolean indicating whether the undo and redo histories for the specified context should be disposed when the operation currently available for undo or redo is not valid. The setting for this parameter is related to the undo context provided and the validation strategy used by operations with that context.

Application undo models

Earlier we looked at how undo contexts can be used to implement different kinds of application undo models. The ability to assign one or more contexts to operations allows applications to implement undo strategies that are strictly local to each view or editor, strictly global across all plug−ins, or some model in between. Another design decision involving undo and redo is whether any operation can be undone or redone at any time, or whether the model is strictly linear, with only the most recent operation being considered for undo or redo.

IOperationHistory defines protocol that allows flexible undo models, leaving it up to individual implementations to determine what is allowed. The undo and redo protocol we've seen so far assumes that there is only one implied operation available for undo or redo in a particular undo context. Additional protocol is provided to allow clients to execute a specific operation, regardless of its position in the history. The operation history can be configured so that the model appropriate for an application can be implemented. This is done with an interface that is used to pre−approve any undo or redo request before the operation is undone or redone.

Operation approvers

IOperationApprover defines the protocol for approving undo and redo of a particular operation. An operation approver is installed on an operation history. Specific operation approvers may in turn check all operations for their validity, check operations of only certain contexts, or prompt the user when unexpected conditions are found in an operation. The following snippet shows how an application could configure the operation history to enforce a linear undo model for all operations.

```
IOperationHistory history = OperationHistoryFactory.getOperationHistory();

// set an approver on the history that will disallow any undo that is not the most recent operati
history.addOperationApprover(new LinearUndoEnforcer());
```

In this case, an operation approver provided by the framework, **LinearUndoEnforcer**, is installed on the history to prevent the undo or redo of any operation that is not the most recently done or undone operation in all of its undo contexts.

Another operation approver, **LinearUndoViolationUserApprover**, detects the same condition and prompts the user as to whether the operation should be allowed to continue. This operation approver can be installed on a particular workbench part.

```
IOperationHistory history = OperationHistoryFactory.getOperationHistory();

// set an approver on this part that will prompt the user when the operation is not the most rece
IOperationApprover approver = new LinearUndoViolationUserApprover(myUndoContext, myWorkbenchPart)
history.addOperationApprover(approver);
```

Plug−in developers are free to develop and install their own operation approvers for implementing application−specific undo models and approval strategies.

Undo and the IDE Workbench

We've seen code snippets that use workbench protocol for accessing the operations history and the workbench undo context. This is achieved using **IWorkbenchOperationSupport**, which can be obtained from the workbench. The notion of a workbench−wide undo context is fairly general. It is up to the workbench application to determine what specific scope is implied by the workbench undo context, and which views or editors use the workbench context when providing undo support.

In the case of the Eclipse IDE workbench, the workbench undo context should be assigned to any operation that affects the IDE workspace at large. This context is used by views that manipulate the workspace, such as the Resource Navigator. The IDE workbench installs an adapter on the workspace for **IUndoContext** that returns the workbench undo context. This model−based registration allows plug−ins that manipulate the workspace to obtain the appropriate undo context, even if they are headless and do not reference any

workbench classes.

```
// get the operation history
IOperationHistory history = OperationHistoryFactory.getOperationHistory();

// obtain the appropriate undo context for my model
IUndoContext workspaceContext = (IUndoContext)ResourcesPlugin.getWorkspace().getAdapter(IUndoCont
if (workspaceContext != null) {
        // create an operation and assign it the context
}
```

Other plug−ins are encouraged to use this same technique for registering model−based undo contexts.

Perspectives

We've already seen some ways the workbench allows the user to control the appearance of plug−in functionality. Views can be hidden or shown using the **Window >Show View** menu. Action sets can be hidden or shown using the **Window >Customize Perspective...** menu. These features help the user organize the workbench.

Perspectives provide an additional layer of organization inside a workbench window. Users can switch between perspectives as they move across tasks. A perspective defines a collection of views, a layout for the views, and the visible action sets that should be used when the user first opens the perspective.

Perspectives are implemented using **IPerspectiveFactory.** Implementors of **IPerspectiveFactory** are expected to configure an **IPageLayout** with information that describes the perspective and its perspective page layout.

Workbench part layout

One of the main jobs of an **IPageLayout** is to describe the placement of the editor and the views in the workbench window. Note that these layouts are different than the **Layout** class in SWT. Although **IPageLayout** and **Layout** solve a similar problem (sizing and positioning widgets within a larger area), you do not have to understand SWT layouts in order to supply a perspective page layout.

A perspective page layout is initialized with one area for displaying an editor. The perspective factory is responsible for adding additional views relative to the editor. Views are added to the layout relative to (top, bottom, left, right) another part. Placeholders (empty space) can also be added for items that are not initially shown.

To organize related views and reduce clutter, you can use **IFolderLayout** to group views into tabbed folders. For example, the Resource perspective places the resource navigator inside a folder at the top left corner of the workbench. Placeholders are commonly used with folder layouts. The Resource perspective defines a placeholder for the bookmarks view in the same folder as the resource navigator. If the user shows the bookmarks view, it will appear in the same folder with the navigator, with a tab for each view.

IPageLayout also allows you to define the available actions and shortcuts inside a perspective.

- **addActionSet** is used to add action sets to a perspective.
- **addNewWizardShortcut** adds a new entry to the **File >New** menu for a perspective.

- **addShowViewShortcut** adds the names of views that should appear in the **Window >Show View** menu when the perspective is active.
- **addPerspectiveShortcut** adds the names of perspectives that should appear in the **Window >Open Perspective** menu when the perspective is active.

Linking views and editors with "show–in"

Another valuable service provided by perspectives and the **IPageLayout** is to aid in navigation between an editor and its related views. We typically think of views as helping the user find the objects to work with in editors. However, the converse operation is also useful: a user working with an object in an editor may need to navigate to that object inside a view. This can be accomplished using the workbench **Navigate > Show In** menu. This command allows the user to jump to one of any number of related views in the context of the currently edited (or selected) object. For example, a user editing a file may want to jump over to that file in the resource navigator.

The plug–in architecture of the workbench allows developers to contribute views and editors in different plug–ins that are not even aware of each other. By implementing support for "show in," your view or editor can support convenient navigation to or from the views and editors contributed by other plug–ins.

This navigation allows users to move quickly between views and to easily open a view that is not usually shown in a particular perspective. For example, a user working in the Java perspective can use **Navigate > Show In** to view the currently edited Java file in the Navigator view.

Show–in source

If you want to allow users to use **Navigate > Show In** from your editor or view to jump to another view, you must implement **IShowInSource**. Your part can supply its **IShowInSource** directly using protocol (**getShowInSource**()) or as an adapter. **IShowInSource** allows your part to supply a context (**ShowInContext**) which is used by the target to decide how to show the source. The show in context for an editor is typically its input element. For a view, the context is typically its selection. Both a selection and an input element are provided in a **ShowInContext** to give the target flexibility in determining how to show the source.

A default context for editors is provided, so that your editor can participate in "show–in" without any special coding. For editors, the input element and selection are used to create an appropriate context.

For views, **IShowInSource** must be implemented by the view in order to offer **Navigate > Show In** functionality.

Show–in target

You must implement **IShowInTarget** if you want your view to be a valid target for a "show in" operation. The target is responsible for showing a given context in a manner appropriate for its presentation. For example, the Navigator view expands its tree to select and reveal a resource specified in the context.

A target should check the selection in the **ShowInContext** first in deciding what to show, since this is the more specific information. It should show the input element only if no selection is indicated.

Presenting appropriate targets

How is the list of available targets determined? You can specify the available targets for your perspective in its **IPageLayout**. Recall that a "show in" navigation may open a view that is not already present in the perspective. Using **IPageLayout.addShowInPart**, you can specify a valid "show in" target by id. In this way, the valid targets can be established without unnecessarily creating any views.

org.eclipse.ui.perspectives

The platform itself defines one perspective, the **Resource** perspective. Other platform plug−ins, such as the help system and the Java tooling, define additional perspectives. Your plug−in can define its own perspective by contributing to the **org.eclipse.ui.perspectives** extension point.

The specification of the perspective in the **plugin.xml** is straightforward. The following markup is used by the workbench in defining its own resource perspective.

```
<extension
         point="org.eclipse.ui.perspectives">
      <perspective
            name="%Perspective.resourcePerspective"
            icon="icons/full/cview16/resource_persp.png"
            class="org.eclipse.ui.internal.ResourcePerspective"
            id="org.eclipse.ui.resourcePerspective">
      </perspective>
   </extension>
```

A plug−in must supply an **id** and **name** for the perspective, along with the name of the **class** that implements the perspective. An **icon** can also be specified. The perspective class should implement **IPerspectiveFactory**.

org.eclipse.ui.perspectiveExtensions

Plug−ins can add their own action sets, views, and various shortcuts to existing perspectives by contributing to the **org.eclipse.ui.perspectiveExtensions** extension point.

The contributions that can be defined for new perspectives (action sets, wizard entries, view layout, view shortcuts, and perspective shortcuts) can also be supplied for an existing perspective. One important difference is that these contributions are specified in the **plugin.xml** markup instead of configuring them into an **IPageLayout.**

The following markup shows how the JDT extends the platform's debug perspective.

```
<extension point="org.eclipse.ui.perspectiveExtensions">
  <perspectiveExtension
targetID="org.eclipse.debug.ui.DebugPerspective">
        <actionSet id="org.eclipse.jdt.debug.ui.JDTDebugActionSet"/>
        <view id="org.eclipse.jdt.debug.ui.DisplayView"
                relative="org.eclipse.debug.ui.ExpressionView"
                relationship="stack"/>
        <view id="org.eclipse.jdt.ui.PackageExplorer"
                relative="org.eclipse.debug.ui.DebugView"
                relationship="stack"
                visible="false"/>
        <view id="org.eclipse.jdt.ui.TypeHierarchy"
                relative="org.eclipse.debug.ui.DebugView"
```

```
                relationship="stack"
                visible="false"/>
        <view id="org.eclipse.search.SearchResultView"
                relative="org.eclipse.debug.ui.ConsoleView"
                relationship="stack"
                visible="false"/>
        <viewShortcut id="org.eclipse.jdt.debug.ui.DisplayView"/>
  </perspectiveExtension>
</extension>
```

The **targetID** is the id of the perspective to which the plug−in is contributing new behavior. The **actionSet** parameter identifies the **id** of a previously declared action set that should be added to the target perspective. This markup is analogous to using **IPageLayout.addActionSet** in the **IPerspectiveFactory**.

Contributing a view to a perspective is a little more involved, since the perspective page layout information must be declared. The **visible** attribute controls whether the contributed view is initially visible when the perspective is opened. In addition to supplying the **id** of the contributed view, the id of a view that already exists in the perspective (a **relative** view) must be specified as a reference point for placing the new view. The **relationship** parameter specifies the layout relationship between the new view and the **relative** view.

- **stack** indicates that the view will be stacked with the relative view in a folder
- **fast** indicates that the view will be shown as a fast view
- **left**, **right**, **top**, or **bottom** indicate that the new view will be placed beside the **relative** view. In this case, a **ratio** between 0.0 and 1.0 must be defined, which indicates the percentage of area in the **relative** view that will be allocated to the new view.

Specifying a **perspectiveShortcut** indicates that another perspective (specified by **id**) should be added to the **Window−>Open Perspective...** menu of the target perspective. This markup is analogous to calling **IPageLayout.addPerspectiveShortcut** in the original perspective definition in the **IPerspectiveFactory**. Plug−ins can also add view shortcuts and new wizard shortcuts in a similar manner.

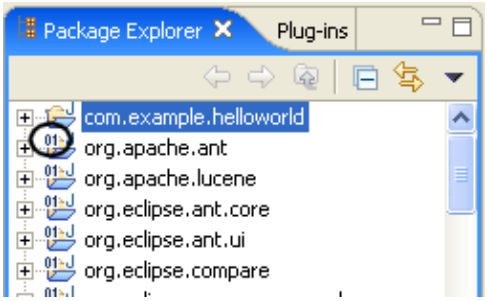
You can also specify one or more views as a valid **showInPart**. The views should be specified by the id used in their **org.eclipse.ui.views** extension contribution. This controls which views are available as targets in the **Navigate > Show In** menu. The ability to specify a "show in" view in the extension markup allows you to add your newly contributed views as targets in another perspective's "show in" menus. See Linking views and editors for more information on "show in."

See **org.eclipse.ui.perspectiveExtensions** for a complete definition of the extension point.

Decorators

Your plug−in can use **decorators** to annotate the images for resources and other objects that appear in the workbench views. Decorators are useful when your plug−in adds functionality for existing resource types. Many of the standard workbench views participate in showing decorations.

For example, PDE contributes decorators that allow you to distinguish between binary and source projects.



The **com.example.helloworld** project is the only source project shown in the navigator. Note how all of the other binary projects show the binary decorator at the top left of the Java project icon. This decorator is contributed by PDE using the **org.eclipse.ui.decorators** extension point.

```
<extension
         point="org.eclipse.ui.decorators">
      <decorator
            lightweight="true"
            quadrant="TOP_LEFT"
            adaptable="true"
            label="%decorator.label"
            icon="icons/full/ovr16/binary_co.png"
            state="false"
            id="org.eclipse.pde.ui.binaryProjectDecorator">
         <description>
            %decorator.desc
         </description>
         <enablement>
            ...
         </enablement>
      </decorator>
 </extension>
```

There are several different ways to supply a decorator implementation. This markup uses the simplest way, known as a **declarative lightweight** decorator. When a declarative lightweight decorator is defined, the markup contains a complete description of the decorator's icon, placement, and enabling conditions. Declarative decorators are useful when only an icon is used to decorate the label. The plug−in need only specify the **quadrant** where the decorator should be overlayed on the regular icon and the **icon** for the overlay. As shown in the picture, the PDE binary icon is overlayed in the top left quadrant of the package icon.

If your plug−in needs to manipulate the label text in addition to the icon, or if the type of icon is determined dynamically, you can use a non−declarative **lightweight** decorator. In this case, an implementation ***class*** that implements **ILightweightLabelDecorator** must be defined. The designated class is responsible for supplying a prefix, suffix, and overlay image at runtime which are applied to the label. The mechanics of concatenating the prefix and suffix with the label text and performing the overlay are handled by the workbench code in a background thread. Thus, any work performed by your plug−in in its **ILightweightLabelDecorator** implementation must be UI−thread safe. (See Executing code from a non−UI thread for more details.)

The following markup shows how the CVS client defines its decorator using this technique:

```
<extension
         point="org.eclipse.ui.decorators">
    <decorator
            objectClass="org.eclipse.core.resources.IResource"
            adaptable="true"
```

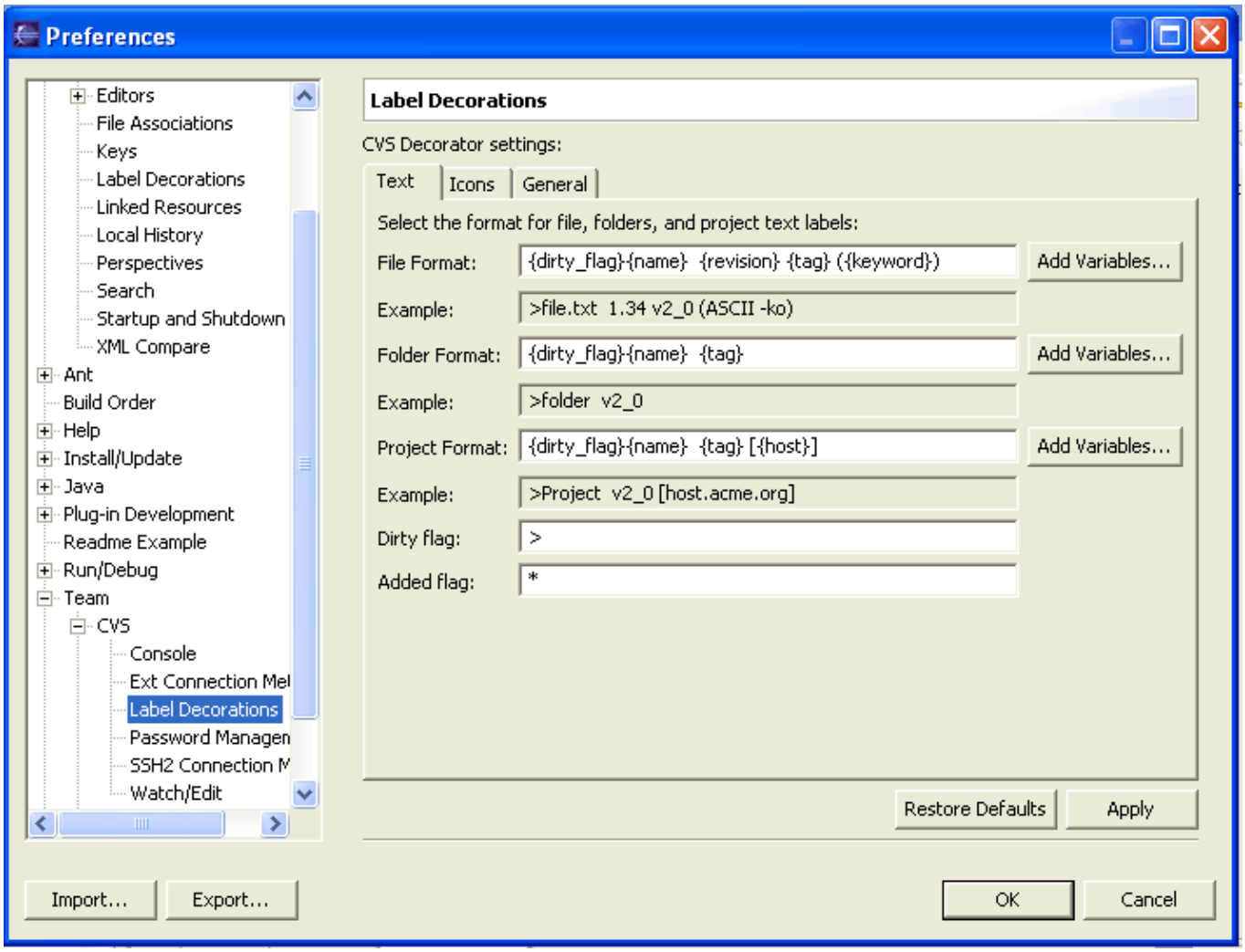
```
            label="%DecoratorStandard.name"
            state="false"
            lightweight= "true"
            quadrant = "BOTTOM_RIGHT"
            class="org.eclipse.team.internal.ccvs.ui.CVSLightweightDecorator"
            id="org.eclipse.team.cvs.ui.decorator">
         <description>
            %DecoratorStandard.desc
         </description>
    </decorator>
</extension>
```

Decorators are ultimately controlled by the user via the workbench **Label Decorations** preferences page. Individual decorators can be turned on and off. Even so, it is a good idea to design your decorators so that they do not overlap or conflict with existing platform SDK decorators. If multiple plug−ins contribute lightweight decorators to the same quadrant, the conflicts are resolved non−deterministically.

Your plug−in may also do all of the image and label management itself. In this case, the **lightweight** attribute should be set to false and the **class** attribute should name a class that implements **ILabelDecorator**. This class allows you to decorate the original label's image and text with your own annotations. It gives you increased flexibility since you aren't limited to prefixes, suffixes, and simple quadrant overlays.

Other attributes of a decorator are independent of the particular implementation style. The **label** and **description** attributes designate the text that is used to name and describe the decorator in the preferences dialog. The **objectClass** names the class of objects to which the decorator should be applied. The **enablement** attribute allows you to describe the conditions under which the object should be decorated. The **adaptable** flag indicates whether objects that adapt to **IResource** should also be decorated. The **state** flag controls whether the decorator is visible by default.

If your decorators include information that is expensive to compute or potentially distracting, you may want to contribute your own preferences that allow the user to further fine−tune the decorator once it is on. This technique is used by the CVS client.



Decorator Update Cycle

Decoration is initiated by refreshing label providers that use the DecoratorManager to provide decoration. As decoration processing is done in the background there will be a period between when the label is requested and the labelProviderChanged event is fired that will be taken up by decoration calculation. During this time decoration on an Object will only be calculated once for effeciency reasons. If the decorator changes during this time it is possible that a stale result will be broadcast as the second and subsequent calls to decorate an element will be ignored.

Decorator contributors should avoid changing thier decorators while decoration is occuring. If this is not possible a second call to decorate an element after the labelProviderChanged in processed will be required.

Workbench key bindings

The workbench defines many keyboard accelerators for invoking common actions with the keyboard. In early versions of the platform, plug–ins could define the accelerator key to be used for their action when the action was defined. However, this strategy can cause several problems:

- Different plug−ins may define the same accelerator key for actions that are not related.
- Plug−ins may define different accelerator keys for actions that are semantically the same.
- Plug−ins may define accelerator keys that later conflict with the workbench (as the workbench is upgraded).

In order to alleviate these problems, the platform defines a configurable key binding strategy that is extendable by plug−ins. It solves the problems listed above and introduces new capabilities:

- The user can control which key bindings should be used.
- Plug−ins can define key bindings that emulate other tools that may be familiar to users of the plug−in.
- Plug−ins can define contexts for key bindings so that they are only active in certain situations.

The basic strategy is that plug−ins use **commands** to define semantic actions. Commands are simply declarations of an action and its associated category. These commands can then be associated with key bindings, actions, and handlers. Commands do not define an implementation for the action. When a plug−in defines an action for an editor, action set, or view, the action can specify that it is an implementation of one of these commands. This allows semantically similar actions to be associated with the same command.

Once a command is defined, a **key binding** may be defined that references the command. The key binding defines the key sequence that should be used to invoke the command. A key binding may reference a **scheme** which is used to group key bindings into different named schemes that the user may activate via the Preferences dialog.

This is all best understood by walking through the workbench and looking at how commands and key bindings are declared. We'll look at all of this from the point of view of defining key bindings for existing workbench actions.

Commands

A **command** is the declaration of a user action by **id**. Commands are used to declare semantic actions so that action implementations defined in action sets and editors can associate themselves with a particular semantic command. The separation of the command from the action implementation allows multiple plug−ins to define actions that implement the same semantic command. The command is what gets associated with a particular key binding.

The workbench defines many common commands in its **plugin.xml** file, and plug−ins are encouraged to associate their own actions with these commands where it makes sense. In this way, semantically similar actions implemented in different plug−ins may share the same key binding.

Defining a command

Commands are defined using the **org.eclipse.ui.commands** extension point. The following comes from the workbench markup:

```
<extension
        point="org.eclipse.ui.commands">
        ...
        <command
                name="%command.save.name"
                description="%command.save.description"
                categoryId="org.eclipse.ui.category.file"
                id="org.eclipse.ui.file.save">
```

```
        </command>
        ...
```

The command definition specifies a **name**, **description**, and **id** for the action. It also specifies the id of a category for the command, which is used to group commands in the preferences dialog. The categories are also defined in the **org.eclipse.ui.commands** extension point:

```
      ...
      <category
            name="%category.file.name"
            description="%category.file.description"
            id="org.eclipse.ui.category.file">
      </category>
      ...
```

Note that there is no implementation specified for a command. A command only becomes concrete when a plug−in associates its action with the command id.

Associating an action with a command

Actions can be associated with a command in code or in the **plugin.xml** for action sets. Your choice depends on where the action is defined.

Actions that are instantiated in code can also be associated with an action definition using **IAction** protocol. This is typically done when the action is created. The **SaveAction** uses this technique when it initializes itself.

```
public SaveAction(IWorkbenchWindow window) {
        ...
        setText...
        setToolTipText...
        setImageDescriptor...
setActionDefinitionId("org.eclipse.ui.file.save");
}
```

(Note: The method name **setActionDefinitionID** could more appropriately be named **setCommandID**. The method name reflects the original implementation of key bindings and uses outdated terminology.)

By invoking **setActionDefinitionID**, the implementation action (**SaveAction**) is associated with the command id that was used in the command definition markup. It is good practice to define constants for your action definitions so that they are easily referenced in code.

If you define an action in an action set, then you typically do not need to instantiate an action yourself. The workbench will do it for you when the user invokes your action from a menu or the keyboard. In this case, you can associate your action with a command ID in your XML markup. The following shows a hypothetical markup for an action set:

```
<extension point = "org.eclipse.ui.actionSets">
           <actionSet id="com.example.actions.actionSet"
                   label="Example Actions"
                   visible="true">
                   <action id="com.example.actions.action1"
                           menubarPath="additions"
                           label="Example Save Action"
                           class="org.example.actions.ExampleActionDelegate"
```

```
   definitionID="org.eclipse.ui.file.save">
                  </action>
                  ...
          </actionSet>
   </extension>
```

The **definitionID** attribute is used to declare a command ID for the action.

Using either technique, associating your action with a command ID causes any key bindings that get defined for the command **org.eclipse.ui.file.save** to invoke your action when appropriate.

Now let's look at how these key bindings get defined.

Key bindings

The association between a command and the key combinations that should invoke the command is called a **key binding**. Plug−ins can define key bindings along with commands in the **org.eclipse.ui.bindings** extension point.

```
...
<key
      sequence="Ctrl+S"
      commandId="org.eclipse.ui.file.save"
      schemeId="org.eclipse.ui.defaultAcceleratorConfiguration">
</key>
...
```

There is our friend **org.eclipse.ui.file.save**. Recall our hypothetical action definition:

```
<extension point = "org.eclipse.ui.actionSets">
          <actionSet id="com.example.actions.actionSet"
                  label="Example Actions"
                  visible="true">
                  <action id="com.example.actions.action1"
                          menubarPath="additions"
                          label="Example Save Action"
                          class="org.example.actions.ExampleActionDelegate"
   definitionID="org.eclipse.ui.file.save">
                  </action>
                  ...
          </actionSet>
</extension>
```

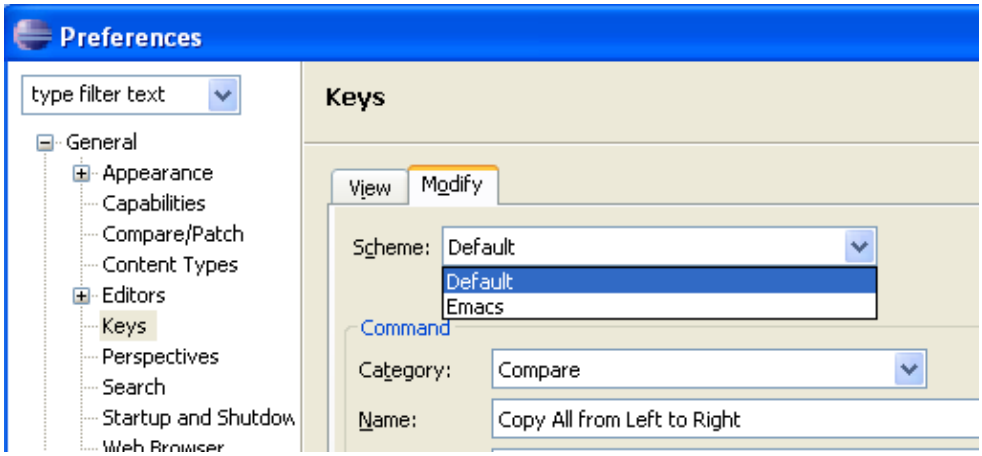
The **sequence** attribute for a key binding defines the key combination that is used to invoke a command. So, it follows that when our example action set is active, our save action will be invoked when the user chooses **Ctrl+S**.

Likewise, when the workbench **SaveAction** is active, the same key combination will invoke it instead, since the workbench uses the same command id for its **SaveAction**.

To complete the example, we need to understand what the **configuration** is all about.

Schemes

Schemes are used to represent a general style or theme of bindings. For example, the Workbench provides a "Default" scheme and an "Emacs" scheme. Only one scheme is active at any given time. End users control which one is active using the general **Preferences** dialog.



From an implementation point of view, schemes are simply named groupings of bindings. A scheme won't accomplish anything on its own unless there are bindings associated with it.

Let's look again at the workbench markup for **org.eclipse.ui.bindings** to find the binding definitions and how a scheme gets associated with a binding.

```
...
<key
       sequence="Ctrl+S"
       commandId="org.eclipse.ui.file.save"
       schemeId="org.eclipse.ui.defaultAcceleratorConfiguration">
</key>
...
<key
       sequence="Ctrl+X Ctrl+S"
       commandId="org.eclipse.ui.file.save"
       schemeId="org.eclipse.ui.emacsAcceleratorConfiguration">
</key>
...
```

There are two different key bindings defined for the "org.eclipse.ui.file.save" command. Note that each one has a different **schemeId** defined. When the default scheme is active, the "Ctrl+S" key binding will invoke the command. When the emacs scheme is active, the sequence "Ctrl+X Ctrl+S" will invoke the command.

Defining new schemes

When your plug–in defines a binding, it will most likely assign it to an existing scheme. However, your plug–in may want to define a completely new style of scheme. If this is the case, you can define a new type of scheme inside the **org.eclipse.ui.bindings** definition. The workbench markup that defines the default and emacs key configurations are shown below:

```
...
<scheme
       name="%keyConfiguration.default.name"
```

```
       description="%keyConfiguration.default.description"
       id="org.eclipse.ui.defaultAcceleratorConfiguration">
</scheme>
<scheme
       name="%keyConfiguration.emacs.name"
       parentId="org.eclipse.ui.defaultAcceleratorConfiguration"
       description="%keyConfiguration.emacs.description"
       id="org.eclipse.ui.emacsAcceleratorConfiguration">
</scheme>
...
```

Note that the **name** defined here is the one used in the preferences page in the list of schemes.

Activating a scheme

The user controls the active scheme via the preferences page. However, you can define the default active scheme as a part of the "plugin_customization.ini" file. It is a preference:

```
org.eclipse.ui/KEY_CONFIGURATION_ID=org.eclipse.ui.defaultAcceleratorConfiguration
```

Contexts and key bindings

A **context** can be specified for a key binding so that the binding is only available when the user is working within a specific context. Contexts are declared in the **org.eclipse.ui.contexts** extension point.

A context can be bound to a key binding by specifying the id of the context when the key binding is defined. For example, if we only wanted the save command to work while the user is editing text, we could specify a context for the key binding:

```
<key
      sequence="Ctrl+S"
      commandId="org.eclipse.ui.file.save"
      contextId="org.eclipse.ui.textEditorScope"
      schemeId="org.eclipse.ui.defaultAcceleratorConfiguration">
</key>
...
```

(See Contexts) for a more detailed discussion of contexts and how they are defined.

Contexts

A **context** can be used to influence what commands are available to the user at any given moment. Contexts are much more dynamic than activities. While an activity represents a broad set of function that is available to the user most of the time, contexts describe a focus of the user at a specific point in time. For example, the commands available to a user while editing text might be different than those available to a user while editing Java text or browsing packages in the package explorer.

Defining a context

Contexts are declared in the **org.eclipse.ui.contexts** extension point. Consider the following context which is defined for editing text:

```
<extension
        point="org.eclipse.ui.contexts">
```

```
        <context
                name="%context.editingText.name"
                description="%context.editingText.description"
                id="org.eclipse.ui.textEditorScope"
                parentId="org.eclipse.ui.contexts.window">
        </context>
```

Contexts are assigned a name and description that are used when showing information about the context to the user. The id of the context is used when binding UI contributions such as commands to a particular context.

Context hierarchies

Contexts are hierarchical in nature. When a context is active, the commands available in the context and in its parent contexts are also available. This is useful for defining levels of contexts that move from very general situations down to more specific contexts. In the context definition above, note that there is an id of a parent assigned to the context:

```
        <context
                name="%context.editingText.name"
                description="%context.editingText.description"
                id="org.eclipse.ui.textEditorScope"
parentId="org.eclipse.ui.contexts.window">
        </context>
```

The parent context defines the more general context of working within a window. Its parent defines an even more general context of working within a window or a dialog.

```
<context
        name="%context.window.name"
        description="%context.window.description"
        id="org.eclipse.ui.contexts.window"
parentId="org.eclipse.ui.contexts.dialogAndWindow">
</context>
<context
        name="%context.dialogAndWindow.name"
        description="%context.dialogAndWindow.description"
        id="org.eclipse.ui.contexts.dialogAndWindow">
</context>
```

Associating a contribution with a context

So far, all we've done is define a hierarchy of contexts. The context becomes useful when it is referenced in the description of another UI contribution. The most common use of contexts is in key bindings. When a context is associated with a key binding, the key binding will only be active when the user is in that context. For example, the following markup specifies the root dialog and window context as the context for a key binding:

```
<keyBinding
        commandId="org.eclipse.ui.edit.cut"
contextId="org.eclipse.ui.contexts.dialogAndWindow"
        keySequence="M1+X"
        keyConfigurationId="org.eclipse.ui.defaultAcceleratorConfiguration">
</keyBinding>
```

Using Context API

The workbench context support includes an API for working with the defined contexts and defining criteria under which a particular context should become enabled. Most plug−ins need not be concerned with this API, but it is useful when defining specialized views or editors that define new contexts.

The starting point for working with contexts in the workbench is **IWorkbenchContextSupport**. Plug−ins can obtain the context support instance from the workbench.

```
IWorkbenchContextSupport workbenchContextSupport = PlatformUI.getWorkbench().getContextSupport();
```

The workbench context support API can be used to add or remove an **EnabledSubmission**, which describes criteria that should cause a particular context to become enabled. Criteria include information such as the active part or active shell. The workbench support also provides access to an **IContextManager**.

```
IContextManager contextManager = workbenchContextSupport.getContextManager();
```

IContextManager defines protocol for getting all defined or enabled context ids, and for getting the associated **IContext** for a particular id. These objects can be used to traverse the definition for a context in API, such as getting the id, name, or id of the parent context. Listeners can be registered on the context manager or on the contexts themselves to detect changes in the definition of a particular context or in the context manager itself. See the package **org.eclipse.ui.contexts** for more information.

Element factories

Element factories are used to recreate workbench model objects from data that was saved during workbench shutdown.

Before we look closely at the element factory extension, we need to review a general technique that is used throughout the platform to add plug−in specific behavior to common platform model objects.

IAdaptables and workbench adapters

When browsing the various workbench classes, you will notice that many of the workbench interfaces extend the **IAdaptable** interface.

Plug−ins use adapters to add specific behavior to pre−existing types in the system. For example, the workbench may want resources to answer a label and an image for display purposes. We know that it's not good design to add UI specific behavior to low−level objects, so how can we add this behavior to the resource types?

Plug−ins can register adapters that add behavior to pre−existing types. Application code can then query an object for a particular adapter. If there is one registered for it, the application can obtain the adapter and use the new behaviors defined in the adapter.

By providing a facility to dynamically query an adapter for an object, we can improve the flexibility of the system as it evolves. New adapters can be registered for platform types by new plug−ins without having to change the definitions of the original types. The pattern to ask an object for a particular adapter is as follows:

```
//given an object o, we want to do "workbench" things with it.
if (!(o instanceof IAdaptable)) {
```

```
   return null;
}
IWorkbenchAdapter adapter = (IWorkbenchAdapter)o.getAdapter(IWorkbenchAdapter.class);
if (adapter == null)
   return null;
// now I can treat o as an IWorkbenchAdapter
...
```

If there is no adapter registered for the object in hand, null will be returned as the adapter. Clients must be prepared to handle this case. There may be times when an expected adapter has not been registered.

The workbench uses adapters to obtain UI information from the base platform types, such as **IResource**. Adapters shield the base types from UI−specific knowledge and allow the workbench to evolve its interfaces without changing the definitions of the base.

Without adapters, any class that might be passed around in the workbench API would have to implement the UI interfaces, which would increase the number of class definitions, introduces tight coupling, and create circular dependencies between the core and UI classes. With adapters, each class implements **IAdaptable** and uses the adapter registry to allow plug−ins to extend the behavior of the base types.

Throughout the workbench code, you'll see cases where a platform core type is queried for an adapter. The query is used to obtain an object that knows how to answer UI oriented information about the type.

Element factories

When the workbench is shut down by the user, it must save the current state of the **IAdaptable** objects that are shown in the workbench. An object's state is stored by saving the primitive data parameters of the object in a special format, an **IMemento**. The id of a factory that can recreate the object from an **IMemento** is also stored and the data is saved in the file system.

When the platform is restarted, the workbench finds the element factory associated with the **IMemento**'s factory id. It finds the factory by checking the plug−in registry for contributions to the **org.eclipse.ui.elementFactories** extension.

The markup is pretty simple. We just have to specify the id of the factory and the corresponding class that implements the factory.

The following code snippet is from the workbench **plugin.xml**.

```
<extension
         point="org.eclipse.ui.elementFactories">
      <factory
            class="org.eclipse.ui.internal.model.ResourceFactory"
            id="org.eclipse.ui.internal.model.ResourceFactory">
      </factory>
      <factory
            class="org.eclipse.ui.internal.model.WorkspaceFactory"
            id="org.eclipse.ui.internal.model.WorkspaceFactory">
      </factory>
      <factory
            class="org.eclipse.ui.part.FileEditorInputFactory"
            id="org.eclipse.ui.part.FileEditorInputFactory">
      </factory>
      <factory
            class="org.eclipse.ui.internal.dialogs.WelcomeEditorInputFactory"
```

```
            id="org.eclipse.ui.internal.dialogs.WelcomeEditorInputFactory">
      </factory>
      <factory
            class="org.eclipse.ui.internal.WorkingSetFactory"
            id="org.eclipse.ui.internal.WorkingSetFactory">
      </factory>
</extension>
```

Accessible user interfaces

The term accessible is used to refer to software that has been designed so that people who have disabilities have a successful interaction with it. Accessible software takes many different kinds of disabilities into account:

- visual – people with color blindness, low vision, or who are completely blind
- audio – people who are hard of hearing or are completely deaf
- mobility – people who have physical impairments that limit their movement and fine motor controls
- cognitive – people who have learning disabilities and may need more consistency or simplicity in their interfaces

Assistive technology

Assistive technology is equipment or software that is used to increase the accessibility of existing operating systems and applications. While it is beyond the scope of this programmer's guide to cover the broad scope of assistive technologies, it is important for you to know that they exist. Why? Because simple things you can do when programming your software or documentation, such as providing alternate text descriptions for images in your HTML, or keyboard equivalents for all of your software actions, can greatly improve the effectiveness of assistive technologies that make use of these techniques.

Accessibility resources

There are some basic coding tips you can use when building plug−in user interfaces that will increase the accessibility of your software. See Tips for Making User Interfaces More Accessible for more information.

IBM's Accessibility Center Website has many useful resources for accessibility, including guidelines and checklists for developing software and web interfaces.

SWT and accessibility

Because SWT uses the operating system's native widgets, user interfaces built with SWT will inherit any assistive technologies that have been installed on the host operating system. SWT implements an interface, **AccessibleListener**, which provides basic accessibility information, such as descriptions of controls, help text, and keyboard shortcuts, to clients. If you are developing assistive technologies that need more information or want to improve upon the basic accessibiliity of the workbench, you can add your own listeners and override the default accessibility behavior in the platform. See the package **org.eclipse.swt.accessibility** for more detail.

Tips for making user interfaces accessible

Below is a series of tips for making something usable to the IAccessibility interface provided by Windows.

Use Groups instead of Labels.

If you use a Label to title a group of related widgets remove the label and replace their parent composite with a Group whose text is the same as the title Label.

Avoid intermediate Composites.

IAccessibility tools will read as far up the parent hierarchy of a widget with focus as there are widgets to read. Be sure there are no widgets without text anywhere in the tree.

Use read−only Texts instead of Labels.

A text can be accessed using the keyboard and should be used if you want the information in a label to be accessible to keyboard navigation. Please note that a label beside a text will be treated as a title and so if you have a title:value pair you wish to show it is only required that you make the value widget a Text.

Read and understand the IBM checklist.

IBM provides a useful checklist for good accessibility at
http://www.ibm.com/able/guidelines/software/accesssoftware.html

Assign mnemonics to all menus and menu items.

Ensure they are unique within a given menu. If a menu is dynamically composed from multiple plugins, it may be better to not assign mnemonics since conflicts cannot be avoided in general (e.g. the **File > New** list, or **Window > Show View** list)

Assign mnemonics to all labels of controls in dialogs / preference pages / property pages (e.g. buttons, checkboxes, radio buttons, etc)

Ensure they are unique within the dialog. Be careful to avoid collisions with the default buttons (e.g. Restore &Defaults, &Apply in preference pages; &Next, &Back, &Finish in wizards). Do not assign mnemonics to OK and Cancel buttons. If you make OK the default button of the shell, and Cancel is equivalent to closing the shell, then Enter and Esc map to these by default. Generally doing something with Esc or Enter is a bad idea.

Ensure that controls that do not have labels are preceded by a label.

If a control does not have its own label (e.g. a text field), use a preceding label ending with ':' and assign a mnemonic to it. Screen readers like JAWS will read this label when the control has focus (see **Window > Preferences > General**)

Avoid extra freestanding labels.

You cannot navigate to freestanding labels with the keyboard and screen readers like JAWS skip these since they do not take focus

Do not assign mnemonics to controls in the main window.

Do not put mnemonics on controls in the main window (other than main menus and main menu items), even if it looks like a dialog (e.g. the form editors in org.eclipse.ui.forms) as these will usually conflict with menu mnemonics

Assign shortcut keys for frequently used functions (and *only* frequently used functions).

There are currently only two ways to hook shortcut keys in SWT:

- ◊ by setting an accelerator on a menu item in the main menu bar (they are ignored in context menus) −− JFace actions have support for this
- ◊ by hooking a key listener on a particular control (e.g. in the implementation of a view or editor)

Consult the table of Eclipse SDK shortcut keys, available within Eclipse from the **General > Keys** preference page, to avoid collision.

Avoid Alt+{key}, Ctrl+Alt+{key} and Ctrl+Space+{key} combinations.

- ◊ Alt+{key} combinations may conflict with menu mnemonics
- ◊ Ctrl+Alt+{key} combinations can conflict with entering special characters on international keyboards (alt Gr = Ctrl+Alt)
- ◊ Ctrl+Space+{key} combinations can conflict with Ctrl−Space, used for mode switching in Asian languages.

Try to save navigation context.

For example, in **Window > Preferences**, we now remember which page you had selected last. This avoids having to navigate through the list each time

Assign a specific person on the team to be responsible for accessibility on your project.

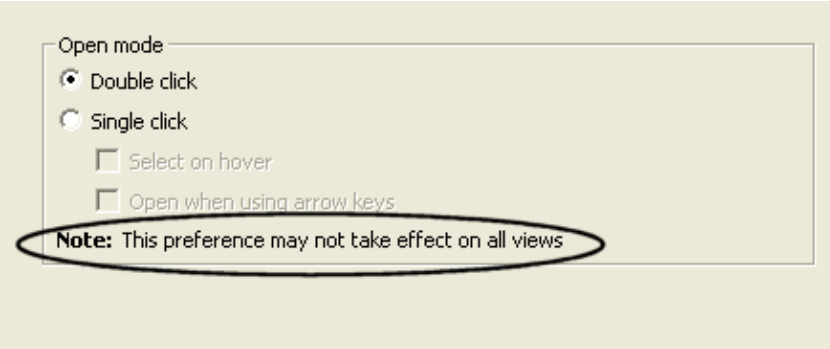
Everything that is important needs an advocate. Make sure that everyone on the team knows that good accessibility is crucial, and is willing to give the person their full cooperation.

Test for accessibility.

Have your team hold an occasional "unplug your mouse day" where they try to use the product using keyboard only. If you are developing on Window, get a copy of JAWS™ and ensure that your UI is usable with it

Honoring single click support

The General **Preferences** allow users to specify whether views should open their objects on single or double click.



Why the disclaimer about this preference not working for all views? Because views contributed by plug−ins must explicitly support this preference in their implementation.

Recall that a view can be implemented by creating SWT controls and writing standard SWT code, or by using JFace viewers to handle the low level details. Honoring the single click preference can be done at either level. Most views that open other objects present them in a structured, list−like view. We'll focus on that

kind of view for now. If your view displays objects in a different manner, you'll likely use the SWT−level concepts to support single click.

Single click in JFace viewers

If you are using a JFace list−oriented viewer to present your objects, supporting single click is straightforward. Instead of using **addDoubleClickListener** to trigger opening the items in your view, use **addOpenListener**. The open listener honors the current workbench preference, firing the open event when the specified mouse event occurs.

You may still wish to use **addDoubleClickListener** for non−open actions, such as expanding the items in a tree on double−click.

Single click in SWT controls

JFace provides a utility class, **OpenStrategy**, to handle the logistics of single and double click at the SWT control level. The **OpenStrategy** is configured by the General **Preferences** dialog so that it honors the current workbench open preference. In fact, the JFace viewers use this class to implement the open listener.

You must create an **OpenStrategy** and associate it with your SWT control. The **OpenStrategy** will hook the appropriate events and interpret them based on the user preferences. Your job is to add an open listener to the strategy that implements the code for open. In this way, you are shielded from knowledge about which widget event triggered the open event.

```
OpenStrategy openHandler = new OpenStrategy(control);
openHandler.addOpenListener(new IOpenEventListener() {
        public void handleOpen(SelectionEvent e) {
        // code to handle the open event.
        ...
        }
}
```

The other workbench preferences for open (select on hover, open using arrow keys) are also handled by **OpenStrategy**. This means that the "right thing" will happen if you use JFace viewers or the **OpenStrategy** class to implement open behavior.

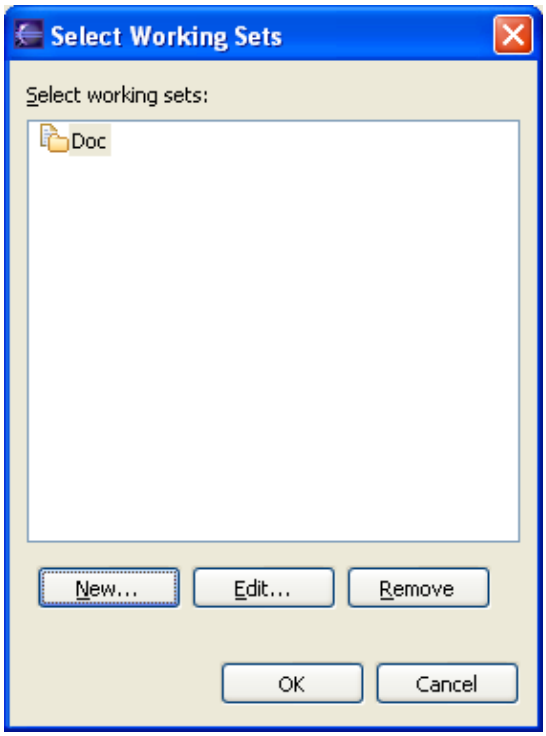
Activating editors on open

When handling an open event, you should use **OpenStrategy.activateOnOpen()** to determine whether an opened editor should be activated by default. Activating an editor switches the focus from the view to the editor, which can be particularly confusing and undesirable in single click mode.

Working sets

Users often find it necessary to filter views such as the navigator view in order to reduce clutter. Plug−ins can assist in filtering using different techniques.

- Resource filters can be used to filter by file name. Plug−ins contribute resource filters that the user can enable using a view's filter selection dialog.
- **Working sets** can be used to filter resources by only including specified resources. Working sets are selected using the view's working set dialog.



If your plug−in implements a view that shows resources (or objects that are adaptable to **IResource**), you should support working sets. **IWorkingSetManager** provides API for manipulating working sets. You can obtain an **IWorkingSetManager** using **IWorkbench** API.

```
IWorkingSetManager manager = workbench.getWorkingSetManager();
```

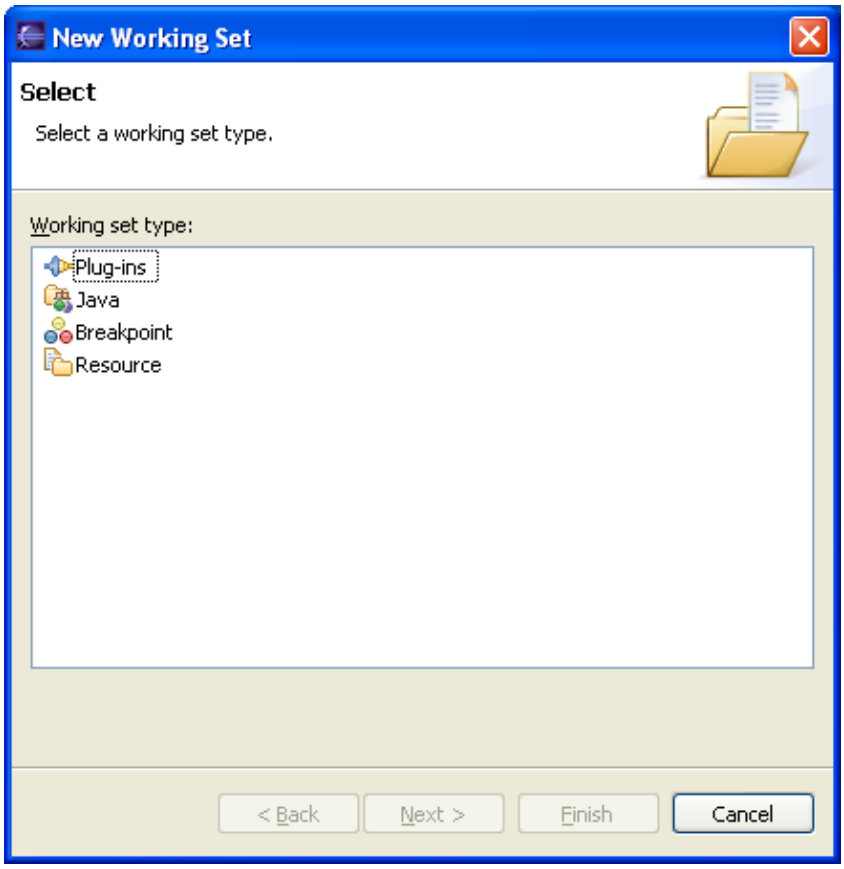
IWorkingSetManager allows you to manipulate and create working sets:

- **createWorkingSetSelectionDialog** – returns a working set dialog that shows the user the current working sets. You can get the selected working sets from the dialog once it is closed.
- **createWorkingSetEditWizard** – returns a working set edit wizard for editing the specified working set
- **getWorkingSets()** – returns a list of all defined working sets
- **getWorkingSet(String name)** – returns a working set specified by name

IWorkingSetManager also provides property change notification as working sets are added, removed, or as they change. If your view or editor needs to respond to changes in the selected working set, it can add a listener for **CHANGE_WORKING_SET_CONTENT_CHANGE.**

Adding new working set types

For many plug−ins, using **IWorkingSetManager** to provide resource filtering is sufficient. If your plug−in needs to define working sets differently, it can register a new type of working set using **org.eclipse.ui.workingSets**. The Java tooling uses this feature to define a Java working set type. Working set types are shown when the user decides to add a working set.



When you define your own type of working set, you can use **IWorkingSet.getId** protocol to ensure that the working set matches the type that you have defined. Any working sets that you create programmatically must have their id set to the id of a working set page that can display the working set elements. This id is used to ensure that the proper working set edit page is launched when the user edits the working set. A null id indicates that the working set should use the default resource working set type.

See the **org.eclipse.ui.workingSets** extension point documentation and **IWorkingSet** protocol for more detail.

Contributing resource filters

The resource filters extension allows plug–ins to define filters that are useful for filtering out file types in the resource navigator view. This extension is useful when special file types are used to represent internal plug–in information but you do not want the files to be shown in the workbench or manipulated by the user.

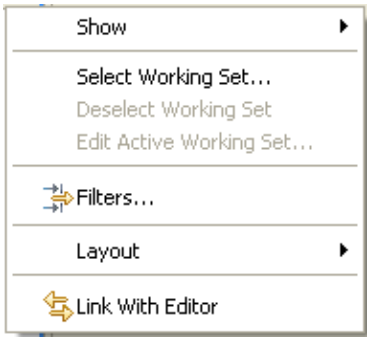
The workbench filters out the pattern "**.***" to exclude internal files such as .metadata from the resource navigator. Likewise, the JDT plug–in filters out "***.class**" files to hide compiled classes.

The markup for the resource filters extension is simple. The following is from the workbench **plugin.xml**.

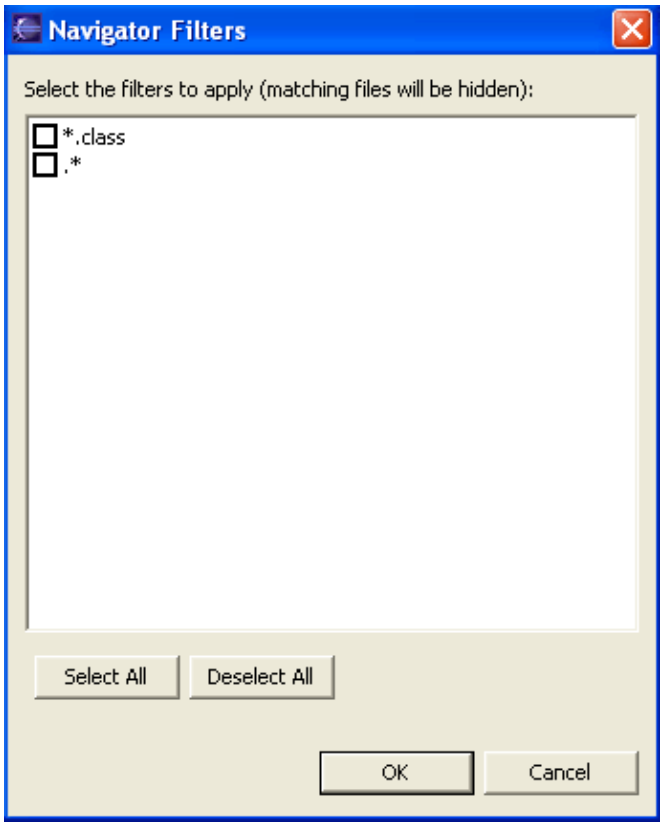
```
<extension
         point="org.eclipse.ui.ide.resourceFilters">
      <filter
            selected="false"
```

```
            pattern=".*">
      </filter>
   </extension>
```

The filters can be enabled by the user using the resource navigator's local pull−down menu.



In addition to declaring the **filter pattern**, the plug−in can use the **selected** attribute to specify whether the filter should be enabled in the resource navigator. This attribute only determines the initial state of the filter pattern. The user can control which filter patterns are active.



Filtering large user interfaces

The rich extensibility mechanisms in the workbench provide many ways for plug−ins to contribute to the platform UI. However, extensibility can introduce its own set of problems. While allowing for a rich set of features contributed by many different developers, it can also create an overwhelming experience for the new user who is trying to navigate through vast menus and preferences pages. As the Eclipse platform matures, the

need for filtering mechanisms that help reduce the UI clutter and guide the user to their desired tasks has become apparent.

The activity and context mechanisms address the problem of too much clutter in the user interface:

- **Activities** allow platform integrators to define large−grained groupings of function that are only shown when a particular user activity is enabled. Users can explicitly (or implicitly through trigger points) enable or disable activities.
- **Contexts** are used to dynamically enable function while the user is performing a specific task. They influence what commands are available to the user at any given moment.

Activities

An **activity** is a logical grouping of function that is centered around a certain kind of task. For example, developing Java software is an activity commonly performed by users of the platform, and the JDT defines many UI contributions (views, editors, perspectives, preferences, etc.) that are only useful when performing this activity. Before we look at the mechanics for defining an activity, let's look at how they are used to help "declutter" the UI.

The concept of an activity is exposed to the user, although perhaps not apparent to a new user. When an activity is enabled in the platform, the UI contributions associated with that activity are shown. When an activity is disabled in the platform, its UI contributions are not shown. Users can enable and disable activities as needed using the **Workbench>Capabilities** preference page. (Activities are referred to as "capabilities" in the user interface, even though we use activity terminology in the API).

Certain user operations serve as **trigger points** for enabling an activity. For example, creating a new Java project could trigger the enabling of the Java development activity. In this way, users are exposed to new function as they need it, and gradually learn about the activities that are available to them and how they affect the UI. When a user first starts the platform, it is desirable for as many activities as possible to be disabled, so that the application is as simple as possible. Choices made in the welcome page can help determine what activities should be enabled.

Activities vs. perspectives

We've seen (in Perspectives) how perspectives are used to organize different view layouts and action sets into tasks. Why do we need activities? While perspectives and activities define similar kinds of tasks, the main difference is how the UI contributions for a plug−in are associated with them. UI contributions are associated with perspectives in the extension definition of the contribution. That is, a plug−in is in charge of determining what perspectives its views and action sets belong to. Plug−ins are also free to define their own perspectives. Even when a perspective is not active, the user can access the views and actions associated with the perspective through commands such as **Show View**.

Activities are a higher level of organization. Individual UI contributions are not aware of activities and do not refer to the activities in their extension definitions. Rather, the activities are expected to be configured at a higher level such as platform integration/configuration or product install. Individual plug−ins typically do not define new activities, unless the plug−in is a systems−level plug−in defined by a systems integrator. In a typical scenario, a systems integrator determines how function is grouped into activities and which ones are enabled by default. Activities are associated with UI contributions using **activity pattern bindings**, patterns that are matched against the id of the UI contributions made by plug−ins. An example will help demonstrate these concepts.

Defining an activity

Activities are defined using the **org.eclipse.ui.activities** extension point. Let's look at a simplified version of how the Eclipse SDK plug−in defines two activities − one for developing Java software and one for developing plug−ins:

```
<extension
        point="org.eclipse.ui.activities">
        <activity
                name="Java Activity"
                description="Developing Java Software"
                id="org.eclipse.javaDevelopment">
        </activity>

        <activity
                name="Plug-in Activity"
                description="Developing Eclipse Plug-ins"
                id="org.eclipse.plugInDevelopment">
        </activity>
        ...
```

Activities are assigned a name and description that can be shown to the user whenever the user is enabling and disabling activities, or otherwise shown information about an activity. The id of the activity is used when defining pattern bindings or other relationships between activities. For example, we can decide that one activity requires another activity.

```
<activityRequirementBinding
        activityId="org.eclipse.plugInDevelopment"
        requiredActivityId="org.eclipse.javaDevelopment">
</activityRequirementBinding>
```

The requirement binding states that the plug−in development activity can only be enabled when the Java development activity is enabled. Related activities can also be bound into **categories**, that are shown to the user when the user is working with activities.

```
<category
        name="Development"
        description="Software Development"
        id="org.eclipse.categories.developmentCategory">
</category>

<categoryActivityBinding
        activityId="org.eclipse.javaDevelopment"
        categoryId="org.eclipse.categories.developmentCategory">
</categoryActivityBinding>

<categoryActivityBinding
        activityId="org.eclipse.plugInDevelopment"
        categoryId="org.eclipse.categories.developmentCategory">
</categoryActivityBinding>
```

The category groups the related development activities together. This category is shown to the user when the user manually configures activities.

Binding activities to UI contributions

Activities are associated with UI contributions using pattern matching. The pattern matching used in activity pattern bindings follows the rules described in the **java.util.regex** package for regular expressions. The patterns used by the workbench are composed of two parts. The first part uses the identifier of the plug−in that is contributing the UI extension. The second part is the id used by plug−in itself when defining the contribution (which may or may not also include the plug−in id as part of the identifier). The following format is used:

```
plug-in-identifier + "/" + local-identifier
```

For example, the following activity pattern binding states that a UI contribution from any JDT plug−in id (`org.eclipse.jdt.*`) is associated with the Java development activity regardless of its local identifier (`.*`).

```
<activityPatternBinding
        activityId="org.eclipse.javaDevelopment"
        pattern="org\.eclipse\.jdt\..*/.*">
</activityPatternBinding>
```

The next binding is more specific. It states that the contribution named `javanature` defined in the JDT core (`org.eclipse.jdt.core`) is associated with the Java development activity.

```
<activityPatternBinding
        activityId="org.eclipse.javaDevelopment"
        pattern="org\.eclipse\.jdt\.core/javanature">
</activityPatternBinding>
```

As you can see, activity pattern bindings can be used to associate large groups of contributions with a particular activity, or to associate very specific contributions to an activity. The following contributions are affected by activities:

- Views and editors
- Perspectives
- Preference and property pages
- Menus and toolbars
- New project wizard

The convention used by the workbench (plug−in id + local id) allows easy binding to plug−ins that do not necessarily follow the naming practice of prefixing their UI contribution identifiers with their plug−in's identifier. Plug−ins that directly interact with the activity API are free to use their own format for identifying contributions and for pattern−matching against those names.

Binding activities to help contributions

Activities are associated with help contributions using the same pattern matching scheme used for UI contributions. The second part of the identifier (the local identifier) indicates the name of the table of contents (TOC) file. For example, the following activity pattern binding associates all TOC files contributed by JDT plug−ins (org.eclipse.jdt.*) with the Java development activity:

```
<activityPatternBinding
        activityId="org.eclipse.javaDevelopment"
        pattern="org\.eclipse\.jdt\..*/.*">
```

```
</activityPatternBinding>
```

When the Java development activity is disabled, help books contributed by JDT plug−ins, or any sub−books (TOCs linked to, or linked by JDT books), even if contributed by a different plug−in, will not show in the help UI. The topics defined in these books will also not show in the search results. In the case where JDT TOCs were not displayed as primary TOCs, but were instead linked from another TOC to appear as sub−trees in a book, disabling the JDT activity has the effect of hiding the sub−trees. The containing book will appear to define less topics in the UI.

Using more specific binding, it is possible to associate activities with selected TOCs from plug−ins that contribute multiple TOCs to the help system. For example, the following activity pattern binding associates the "Examples" TOC with the Java development examples activity.

```
<activityPatternBinding
        activityId="org.eclipse.javaDevelopmentExamples"
        pattern="org\.eclipse\.jdt\.doc\.isv\.topics_Samples.xml">
</activityPatternBinding>
```

With such pattern binding, disabling the Java development examples activity will hide the "Examples" section from the "JDT Plug−in Developer Guide" book.

Using the activities API

The workbench activity support includes an API for working with the defined activities and changing their enabled state. Most plug−ins need not be concerned with this API, but it is useful when implementing function that allows the user to work with activities, or for implementing the trigger points that enable a particular activity. It is assumed that any plug−in that is manipulating activities through API is quite aware of the ways that activities are configured for a particular product. For example, the workbench itself uses the API to trigger the enablement of activities such as Java development. We'll look at how the workbench uses the generic activity API to implement triggers.

The hub of all activity in the workbench is **IWorkbenchActivitySupport**. The activity support works in tandem with an **IActivityManager**. Plug−ins can obtain the activity support instance from the workbench, and the activity manager from there.

```
IWorkbenchActivitySupport workbenchActivitySupport = PlatformUI.getWorkbench().getActivitySupport
IActivityManager activityManager = workbenchActivitySupport.getActivityManager();
```

The following snippet enables the Java development activity (if it is not already enabled). It shows a simplified version of a trigger.

```
...
//the user did something Java related.  Enable the Java activity.
Set enabledActivityIds = new HashSet(activityManager.getEnabledActivityIds());
if (enabledIds.add("org.eclipse.javaDevelopment"))
        workbenchActivitySupport.setEnabledActivityIds(enabledActivityIds);
```

IActivityManager also defines protocol for getting all defined activity and category ids, and for getting the associated **IActivity** or **ICategory** for a particular id. These objects can be used to traverse the definition for an activity or category in API, such as getting the pattern bindings or requirement bindings. Listeners can be registered on the activity manager or on the activities and categories themselves to detect changes in the definition of a particular activity or in the activity manager itself. See the package **org.eclipse.ui.activities** for more information.

Guiding the user through tasks

Even when the platform UI filters out unneeded functionality, there is still a steep learning curve faced by a new user. The platform UI introduces several mechanisms for helping the user choose what task needs to be done and guiding the user through the steps in that task. These mechanisms are used by the workbench itself to help guide Eclipse SDK users through tasks.

- **Cheat sheets** can help guide the user through a series of complex tasks in order to achieve some overall goal. Some steps can be performed by the cheat sheet, and some are described so that the user can manually complete the task.
- **Initial User Experience** allows plug−ins to define pages that help introduce the product to the end user the first time the platform is started. These pages can reference help content or launch the appropriate cheat sheet for a task.

Cheat sheets

Cheat sheets are special views that help guide the user through a series of complex tasks to achieve an overall goal. For example, a cheat sheet could be used to help guide the user through all the steps needed to create, compile, and run a simple Java program. Cheat sheets are launched from the **Help>Cheat Sheets...** menu item. Cheat sheets can also be launched from an intro page.

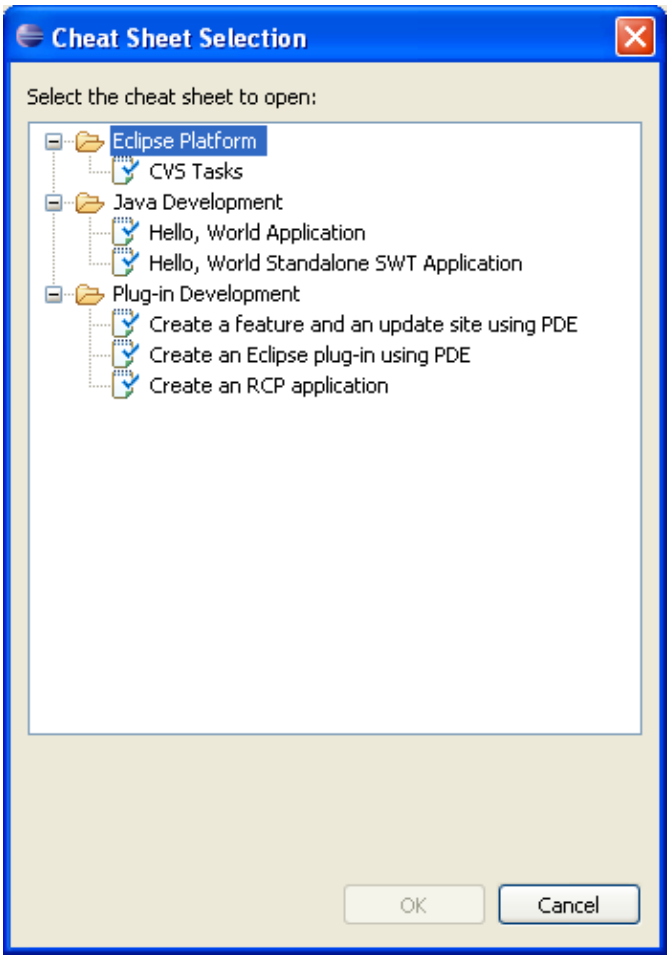
Cheat sheets are defined using the **org.eclipse.ui.cheatsheets.cheatSheetContent** extension point. The cheat sheet content itself is defined in a separate file so that it can be more easily translated into other languages.

Contributing a cheat sheet

Contributing a cheat sheet is pretty straightforward. Let's look at a cheat sheet contributed by the JDT for building a simple Java application.

```
<extension point="org.eclipse.ui.cheatsheets.cheatSheetContent">
        <cheatsheet
                name="%cheatsheet.helloworld.name"
                contentFile="$nl$/cheatsheets/HelloWorld.xml"
                id="org.eclipse.jdt.helloworld">
                <description>%cheatsheet.helloworld.desc</description>
        </cheatsheet>
        ...
```

Much like other workbench contributions, a name, description, and id can be specified for the cheat sheet. The name and description are shown when the user accesses the **Help>Cheat Sheets...** list. A category for the cheat sheet can also be defined if you want to place several cheat sheets into a logical grouping. If no category is specified, the cheat sheet will appear in the **Other** category.



Cheat sheet items

The real work for cheat sheets is done in the content file. The content file is an XML file whose name and location are specified in the **contentFile** attribute. The path for the file is relative to the plug–in's directory. (Note the use of the `$nl$` variable in the directory name, which means the file will be located in a directory specific to the national language of the target environment.)

The file format itself includes overview information about the cheat sheet followed by a description of each step (called an *item*) that the user will perform. At its simplest, an item is just a detailed description of the step that the user should take. However, an item can also specify an action that can be run to perform the step on behalf of the user. Let's look at the first part of the content file (`HelloWorld.xml`) for the Java cheat sheet.

```
<?xml version="1.0" encoding="UTF-8" ?>
<cheatsheet title="Simple Java Application">
        <intro
                href="/org.eclipse.ui.cheatsheets.doc/tasks/tcheatst.htm">
                <description>
Welcome to the Hello, World Java tutorial.
It will help you build the famous "hello world" application and try it out. You will create a jav
Let's get started!
                </description>
</intro>
        <item
                href="/org.eclipse.platform.doc.user/concepts/concepts-4.htm"
                title="Open the Java Perspective">
```

```
<action
                        pluginId="org.eclipse.ui.cheatsheets"
                        class="org.eclipse.ui.internal.cheatsheets.actions.OpenPerspective"
                        param1="org.eclipse.jdt.ui.JavaPerspective"/>
                <description>
Select Window->Open Perspective->Java in the menu bar at the top of the workbench.
This step changes the perspective to set up the Eclipse workbench for Java development.
You can click the "Click to Perform" button to have the "Java" perspective opened automatically.
                </description>
</item>
...
```

The title and intro information are shown at the top of the cheat sheet. Then, the items are described. The first item for this cheat sheet describes how to open the Java perspective. Better still, the **action** attribute specifies a class that can be used to run the action on behalf of the user. The class must implement **IAction**. This is rather convenient, as it allows you to reuse the action classes written for menu or toolbar contributions.

The class for the action can optionally implement **ICheatSheetAction** if the action uses parameters or needs to be aware of the cheat sheet and its state. In this case, the action will be passed an array of parameters and a reference to the **ICheatSheetManager** so that it can request additional information about the cheat sheet. Any necessary parameters can be passed to the action's run method using the **param*N*** attributes.

It is strongly recommended that actions invoked from cheat sheets report a success/fail outcome if running the action might fail. (For example, the user might cancel the action from its dialog.) See **IAction.notifyResult(boolean)** for more detail.

Items do not have to define actions. If your item must be performed manually by the user, you need not specify an action at all. Below is the third step of the Java cheat sheet, which merely tells the user how to code the simple application. When no action is specified, the item description must instruct the user to press the appropriate button after the task has been completed.

```
<item
        href="/org.eclipse.jdt.doc.user/tasks/tasks-54.htm"
        title="Add a System.out.println line in your main method">
        <description>
```

```
Now that you have your HelloWorld class,
In the "public static void main" method, add the following statement:  System.out.println("Hello
        </description>
</item>
```

Additional attributes control whether the item can be skipped completely and what document should be launched if the user requests help during the step. See the **org.eclipse.ui.cheatsheets.cheatSheetContent** extension point documentation for a description of all of the attributes that can be defined inside a cheat sheet.

Subitems

Subitems may be defined to further organize the presentation of an item. Unlike items, subitems do not have to be visited in any particular order. Subitems may also define actions that automatically perform the subtask for the user. Subitem actions are described in the same way as item actions.

Conditional expressions and cheat sheet variables

Conditional expressions can be used to define cheat sheet elements whose content or behavior depends upon a particular condition being true. Conditions are described in the **condition** element of a subitem using arbitrary string values that are matched against the **when** attribute for each choice. Conditions typically reference cheat sheet variables using the form `${var}`, where *var* refers to the name of a cheet sheet variable. A few simple examples will help demonstrate how conditional expressions work.

Conditional subitems can be used to chose one subitem from a list of possible subitems. Only the first subitem whose **when** attribute matches the condition attribute is included in the cheat sheet. For example:

```
<item ...>
        <conditional-subitem condition="${v1}">
                <subitem when="a" label="Step for A." />
                <subitem when="b" label="Step for B." />
        </conditional-subitem>
</item>
```

This item specifies two possible subitems that depend on the value of the variable `v1`. If the variable value is `a`, then the first subitem will be included. If the variable value is `b`, then the second subitem will be included. If the variable is neither value, it is considered an error.

Conditional actions are similar to conditional subitems. The **perform−when** element specifies a condition for performing one action among a list of possible actions. The condition is described the same way, using an arbitrary string that often references a variable. The action whose **when** attribute matches the condition is the one that will be performed. For example:

```
<item ...>
        <perform-when condition="${v1}">
                <action when="a" class="com.example.actionA" pluginId-"com.example" />
                <action when="b" class="com.example.actionB" pluginId-"com.example" />
        </perform-when>
</item>
```

The action to be performed is chosen based on the value of the `v1` variable. If the variable value is neither `a` or `b`, it is considered an error.

Repeated subitems

Repeated subitems describe a subitem that can can expand into 0, 1, or more similar substeps. The substeps are individualized using the special variable `${this}`. This variable will be replaced by the values specified in the **values** attribute. The values attribute is a string of values that are separated by commas. A variable that expands into a list of values may be used in the values attribute. For example:

```
<item ...>
        <repeated-subitem values="${v1}">
                <subitem label="Step ${this}" />
        </repeated-subitem>
</item>
```

If the value of the variable is `1,b,three`, then three subitems will appear in the cheat sheet, each having a unique label ("Step 1," "Step b," "Step three"). The variable can be used in the label or the action paramater value. It can also be accessed from the **ICheatSheetManager** while the action is executing.

Cheat sheet listeners

In some cases, you may want to change other parts of your UI if a cheat sheet is active. For example, you may have an editor that shows special annotations if a cheat sheet is guiding the user through an editing task. In this case, a **listener** can be specified as an attribute of the cheatsheet. The listener attribute must be the fully qualified name of a Java class that subclasses **CheatSheetListener**. Listeners will receive notifications along with an **ICheatSheetEvent** when there is a change in the cheat sheet's life cycle, such as when it opens, closes, or completes.

Contributing attributes to an existing cheat sheet

The **org.eclipse.ui.cheatsheets.cheatSheetItemExtension** extension can be used to contribute arbitrary attributes to a pre−existing cheat sheet. The purpose of this extension point is to allow a plug−in to add additional buttons that will aid the user for a given step. These additional buttons are displayed beside the help icon.

To use this mechanism, you can define any arbitrary attribute inside an item definition in the cheat sheet XML file. The attribute name will be matched against any attributes contributed in extensions to **org.eclipse.ui.cheatsheets.cheatSheetItemExtension**. See the extension point documentation for more detail.

Defining an intro part

The **IIntroPart** interface and the **org.eclipse.ui.intro** extension point make up the generic mechanism that can be used to create your own intro support for a given product. The main purpose of this extension is to define the class that implements **IIntroPart** and to specify the binding between a product id and an intro part. For example, the following contribution defines a hypothetical intro part to be shown by the workbench on startup:

```
<extension
        point="org.eclipse.ui.intro">
        <intro
                class="com.example.SampleIntroPart"
                id="someId">
                icon="someIcon.png"
        </intro>
        <introProductBinding
```

```
                introId="someId"
                productId="com.example.someProductId">
        </introProductBinding>
</extension>
```

This contribution first defines the intro part and assigns it the id "someId". It then binds this intro part to a product whose id is "com.example.someProductId". On platform startup, the class specified in the **class** attribute will be instantiated by the workbench and presented to the user as the introduction to the product. This is the lowest level integration into the **IIntroPart** interface.

The platform supplies its own **IIntroPart** implementation called **CustomizableIntroPart** that allows for the content and presentation of the intro to be customized. Below is the snippet that defines the intro part for the workbench. We won't go over the mechanics of implementing an intro part since we want to focus on defining the intro content. (See the extension point documentation and javadoc referenced above for more detail if you need it.)

```
<extension
        point="org.eclipse.ui.intro">
        <intro
                class="org.eclipse.ui.intro.config.CustomizableIntroPart"
                id="org.eclipse.platform.intro">
        </intro>
        <introProductBinding
                introId="org.eclipse.platform.intro"
                productId="org.eclipse.platform">
        </introProductBinding>
</extension>
```

The above contribution defines the **CustomizableIntroPart** as the intro part to be used for the Eclipse SDK platform. The rest of this discussion shows you how to use and extend this part.

Contributing a HelloWorld Intro Part

We will now contribute a very basic intro part just to illustrate the steps needed to contribute a part implementation to the Workbench and get it to show up as the welcome page. To do this:

1. use the *org.eclipse.ui.intro* extension point to register an intro part implementation and bind this intro part to your product.
2. implement the *org.eclipse.ui.intro.IIntroPart* interface and use this class as the class attribute in the above extension point contribution.
3. run your Eclipse based product with the correct product id.

Here is the *org.eclipse.ui.intro* extension point registration needed:

```
<extension point="org.eclipse.ui.intro">
        <intro class="org.eclipse.ui.intro.HelloWorldIntroPart"
                id="org.eclipse.ui.intro.examples.basic001_introId"
                icon="some_icon.jpg"/>
        <introProductBinding
                introId="org.eclipse.ui.intro.HelloWorld_introId"
                productId="org.eclipse.ui.intro.HelloWorld_product">
        </introProductBinding>
```

```
</extension>
```

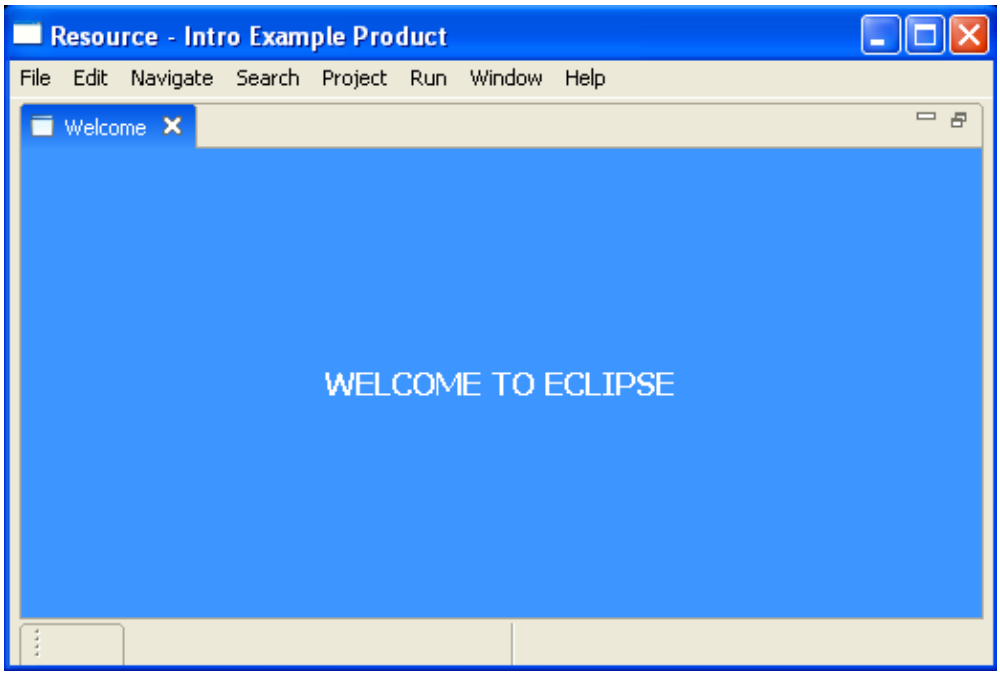
An *intro* part is registered with the workbench. The class that implements this intro part is *org.eclipse.ui.intro..HelloWorldIntroPart* . An icon is also registered with the intro part and it will appear at the top right corner of the intro part window. An *introProductBinding* contribution tells the workbench that the intro part we just created is bound to our product with the id *org.eclipse.ui.intro.HelloWorld_product*. On startup, the workbench looks for the first intro bound to the current product, and instantiates the class registered with this intro contribution.

The second step is to implement the *org.eclipse.ui.intro.IIntroPart* interface. The following is sample code that simply creates a label and centers it in the parent composite. This code can be used to actually create the IntroPart:

```
public void createPartControl (Composite container) {
                Composite outerContainer = new Composite(container, SWT.NONE);
                GridLayout gridLayout = new GridLayout();
                outerContainer.setLayout(gridLayout);
                outerContainer.setBackground
                        (outerContainer.getDisplay().getSystemColor(SWT.COLOR_TITLE_BACKGROUND_GR
                label = new Label(outerContainer, SWT.CENTER);
                label.setText("WELCOME TO ECLIPSE");
                GridData gd = new GridData(GridData.GRAB_HORIZONTAL | GridData.GRAB_VERTICAL);
                gd.horizontalAlignment = GridData.CENTER;
                gd.verticalAlignment = GridData.CENTER;
                label.setLayoutData(gd);
                label.setBackground(outerContainer.getDisplay().
                                getSystemColor(SWT.COLOR_TITLE_BACKGROUND_GRADIENT));

                }
```

The third and last step is to make sure you run the correct product. For example, if you are self hosting, create a new runtime−workbench launch configuration, choose the "Run a product" option, and select *org.eclipse.ui.intro.HelloWorld_product* from the dropdown.
This is what you will see if you run the above HelloWorld sample:



Note that the intro part is in control of the full real–estate of the window. A more elaborate intro part can be created that interacts with the workbench and progressively reveals the functionality of the product.

Using the CustomizableIntroPart

The platform's **CustomizableIntroPart** allows for the content and presentation of the intro to be customized using the **org.eclipse.ui.intro.config** extension point. (This intro config can be extended using the **org.eclipse.ui.intro.configExtension** extension point.) This structure allows product plug–in developers to focus on developing their intro content rather than implementing an intro part scheme from scratch. If a different intro class is specified, then these two extension points are not utilized and the specified class must implement its own scheme for intro content format and configuration.

Defining an intro config

org.eclipse.ui.intro.config describes the id of the intro config that is to show our content, and the name of the XML file that contains the specific definition for the intro content. It is expected that only one intro config should be defined for a given **CustomizableIntroPart**. (Only the first intro config found can be shown in a **CustomizableIntroPart**.)

```
<extension
        id="intro"
        point="org.eclipse.ui.intro.config">
<config
                introId="org.eclipse.platform.intro"
                id="org.eclipse.platform.introConfig"
                content="$nl$/introContent.xml">
        <presentation
                home-page-id="root" standby-page-id="standby">
                        <implementation
```

```
                                ws="win32"
                                style="css/shared.css"
                                kind="html"
                                os="win32">
                        </implementation>
                        <implementation
                                kind="swt">
                        </implementation>
        </presentation>
        </config>
</extension>
```

The path for the file is relative to the plug−in's directory. (Note the use of the `$nl$` variable in the directory name, which means the file will be located in a directory specific to the national language of the target environment.)

The config extension allows you to specify both the content and the presentation of the content. While the **content** element focuses on defining pages, the **presentation** element describes presentation−related attributes that describe how pages will be shown. The page id for the intro home page (in **full mode**) must be specified, and the standby page id (in **standby mode**) is optional. The home page is the page that is shown when the product is first started. A presentation can specify one or more **implementations** for showing the pages. Implementations are specified per platform and windowing system, allowing you to take advantage of platform−specific features for showing page content. For example, the windows platform has a robust HTML browser widget, so an HTML−based implementation is used for intro content. Other platforms without this capability use an SWT−based implementation that maps page descriptions to an SWT−based form. An implementation that does not specify either a windowing system or operating system will be considered the generic implementation; to ensure an intro is shown on all platforms, it is important to define such an implementation. The workbench will first look for an implementation that matches the current operating system and windowing system. If one cannot be found, it will choose the generic implementation. Most of these details are handled at the product configuration level, so we won't discuss them any further here.

Defining intro content

Now we can look at the content itself. Content is described in terms of pages. All pages have an **id** attribute. This is the id that is used when defining the home and standby pages, and other places where there is a reference to a page. Otherwise, the relevant attributes depend on the kind of page that is defined. There are two basic types of pages:

- **Static pages** are plain HTML files. These pages use the normal HTML mechanisms to link to other pages. Static pages need not be defined in the config content file, except for the home page. Since the home page is specified by id (**home−page−id**) in the **presentation** element, there must be a page definition using that id in the content file. This page need only define a **url**. All other subelements will be ignored since the HTML page itself will describe the page content. All other HTML intro pages contributed by the plug−in must be included with the plug−in, but do not need to be specified in the content file. HTML files located in other plug−ins or on the web may be referenced also.
- **Dynamic pages** are described in the XML content file using subelements that describe the content of the page. The subelements are UI items often found in HTML−like pages. Depending on the implementation, these pages will either be dynamically translated to HTML (when the implementation **kind** is html) or else dynamically created as SWT−based UI forms (when the implementation **kind** is swt). The following subelements can be defined in a page:

- A **group** is used to group other subelements and define a consistent style across the group.
- A **link** defines a link that can be displayed using an image and text. The link can navigate to another page and optionally run an intro action. Actions are specified as commands in the URL.
- The **text** and **img** elements show text and image content.
- The **include** element includes a previously defined subelement. The element is referred to by its id.
- The **head** element defines additional HTML to be included in the head section of the page when the html implementation is used.
- The **html** element defines additional HTML to be included in the body of the page when the html implementation is used.

A **title** for a page may also be defined. A page may also specify that its content is defined in a separate **content** file. Breaking up pages into separate files may be useful when performance is a concern, since an intro page's contents won't be initialized until needed.

The best way to get a feel for the content definition format is to browse the implementations in the SDK. The following snippet shows just the first part of the content for the SDK root page, which is the first intro page shown.

```
<introContent>
  <page alt-style="css/root_swt.properties" style="css/root.css" id="root" style-id="page">
    <title style-id="intro-header">Welcome to Eclipse Platform 3.0</title>
    <group id="links-background">
      <group id="page-links">
        <link label="Overview" url="http://org.eclipse.ui.intro/showPage?id=overview" id="overvie
          <text>Find out what Eclipse is all about</text>
        </link>
        <link label="Tutorials" url="http://org.eclipse.ui.intro/showPage?id=tutorials" id="tutor
          <text>Let us guide you through Eclipse end-to-end tutorials</text>
        </link>
        <link label="Samples" url="http://org.eclipse.ui.intro/showPage?id=samples" id="samples"
          <text>Explore Eclipse development through code samples</text>
        </link>
        <link label="Whats New" url="http://org.eclipse.ui.intro/showPage?id=news" id="news" styl
          <text>Find out what is new in this release</text>
        </link>
      </group>
    </group>
```

Elements on a page can also be **filteredFrom** a particular implementation. This allows page designers to design with particular platforms in mind. There are many more powerful attributes that can be used when describing a page and its contents. See the extension point documentation for **org.eclipse.ui.intro.config** and its associated intro content file format specification for a complete reference of valid elements, subelements, and their attributes.

Using XHTML as intro content

Depending on the usage scenario of the intro framework, XHTML files can be contributed as intro content. (A reformulation of the three HTML 4 document types as applications of XML 1.0. For M5, only XHTML 1.0 is supported. XHTML 1.1 (module–based XHTML) can be supported, based on feeThe idea is to use the fact that XHTML is well formed XML and parse each document, manipulating the DOM to allow for contributions and extensions to be merged. Three xml elements from the 3.0 intro markup where used to extend the XHTML 1.0 element list. These where `include`, `anchor`, and `contentProvider`.

- **`include:`** this element can be added to a valid XHTML document to include content from another XHTML document. The content to be included must be a valid XHTML snippet.

  e.g.: `<include path="root/foo" />` will include an element with id `foo` from a welcome page with id `root`.

- **`anchor:`** this element can be added to a valid XHTML document to declare that content can be contributed to this page by other welcome contributions. A page declares locations that are suitable to be extended by defining these anchor points.

  e.g.: `<anchor id="anchor1" />` will allow for contribution into this page from other plugins.

- **`contentProvider:`** this element can be added to a valid XHTML document to establish a hook into the workbench. When the intro framework encounters this element, an interface is called allowing for the manipulation of the DOM of the XHTML page.

  e.g.: `<contentProvider id="contentProviderId" class="org.eclipse.ui.intro.template2.IntroXHTMLContentProvider" pluginId="org.eclipse.ui.intro.template2"> </contentProvider>` will allow for dynamic content to be generated from the `org.eclipse.ui.intro.template2.IntroXHTMLContentProvider` class.

With these three elements, XHTML pages can be used to assemble a pluggable and dynamic welcome pages, just like what used to happen with the custom intro xml markup. PDE has a new template that allow for the creation of a sample RCP application with an Intro. That template is a good sample project for using Intro.

Displaying static HTML content in a CustomizableIntroPart

Just like any intro part implementation, to use a CustomizableIntroPart you need to follow the basic steps to bind it to a product. However, there is an extra step needed to use this intro part and it is binding a "configuration" with it. Just like you bind an intro to a product, you must bind an *introConfig* to a customizableIntroPart. Here is a sample configuration:

```
<extension point="org.eclipse.ui.intro.config">
        <config id = "static001_configId"
                introId="org.eclipse.ui.intro.examples.static_introId"
                content="introContent.xml">
                <presentation  home-page-id="homePageId"
                        standby-page-id="standbyPageId">
                        <implementation os="win32" kind="html" />
                        <implementation kind="swt" />
                </presentation>
        </config>
</extension>
```

In the above contribution a *configuration* is registered with an intro part with id *org.eclipse.ui.intro.examples.static_introId.* (It is assumed that this intro part is a customizable intro part instance that has already been registered with the workbench). This configuration defines the content to be presented in the intro part and dictates how it is presented to the user. The content is defined in an xml markup file, *introContent.xml,* while the presentation is dictated by two *implementation* elements in the

markup.

A config presentation can be either an SWT browser based or a UI forms based presentation. In the above contribution, the presentation will be "html", ie browser based on win32 platforms, while it will be "swt" ie: UI forms based on all other platforms. At runtime, when the workbench is trying to instantiate the CustomizableIntroPart, the operating system is determined and the correct implementation of the presentation is chosen.
Also, a *home−page−id* or root page needs to be specified as it will be the first page displayed by the customizableIntroPart. If a *standby−page−id* is also specified, it will be displayed in the intro part when the intro part is put into standby mode.

The details of what the content file can be found in the extension point documentation. For a simpleexample, and to contribute static content we will use the following as content:

```
<?xml version="1.0" encoding="UTF-8" ?>
<introContent>
        <page id="homePageId" url="http://eclipse.org"/>
        <page id="standbyPageId" url="./static001/standby.html"/>
</introContent>
```

In the above contribution, a simple url is used as the root page, in this case is a url pointing to the eclipse.org web site. This was done for simplicity. The root page could have been any html file, for example, a local html file that loads a flash demo. There is also a standby page defined that will be displayed when the intro is placed into standby mode.

Extending an intro config

An intro configuration can be extended in three ways:

- content of an existing intro config can be extended.
- a custom standby content part, such as Cheat Sheet, can be contributed to provide content for the standby area of the Intro part.
- custom IntroURL actions can be defined.

Extending the content of an intro config

Plug−ins can contribute intro content to a page defined elsewhere. However, the defining page must define an **anchor** attribute that acts as a location placeholder for new content. The SDK overview page defines two anchors for adding JDT and PDE related elements on the overview page.

```
 <group id="page-content">
        <text style-id="page-title" id="page-title">OVERVIEW</text>
        <text style-id="page-description" id="page-description">Eclipse is a kind of universal to
        <group id="overview-links">
                <link label="Workbench basics" url="http://org.eclipse.ui.intro/showHelpTopic?id=
                        <text>Learn about basic Eclipse workbench concepts</text>
        </link>
        <link label="Team support" url="http://org.eclipse.ui.intro/showHelpTopic?id=/org.eclipse
                        <text>Find out how to collaborate with other developers</text>
        </link>
<anchor id="jdtAnchor"/>
```

```
                <anchor id="pdeAnchor"/>
        </group>
</group>
```

These anchors can be referenced by plug−ins that add content to the page. Content is added using the **org.eclipse.ui.intro.configExtension** extension. In addition to extending page content, this extension point also allows one to contribute standby content parts and custom actions.

To extend an existing intro config, you can use the **configExtension** element. In this element, you specify the **configId** of the intro config being extended and the **content** file that describes the new content.

```
<extension
        point="org.eclipse.ui.intro.configExtension">
        <configExtension
                configId="org.eclipse.platform.introConfig"
                content="$nl$/overviewExtensionContent.xml"/>
    ...
</extension>
```

The format of the content file is similar to that of the intro config content, except that it must contain an **extensionContent** element that defines the path to the anchor where the extension content should be inserted.

```
<introContent>
        <extensionContent alt-style="css/swt.properties" style="css/overvi path="overview/page-con
                <link label="Java development" url="http://org.eclipse.ui.intro/showHelpTopic?id=
                        <text>Get familiar with developing Java programs using Eclipse</text>
                </link>
        </extensionContent>
</introContent>
```

After contributing custom content to an intro's predefined anchor points, a given product can bind itself to that intro using the **org.eclipse.ui.intro** discussed above. When the product is run, the intro that was extended will be shown with the additional content. This allows the product to have its own branding and other product−specific information, while reusing a closely related product's intro along with key content of its own.

A given intro could also selectively include pieces of a related product's intro. In this case, the product could define its own intro and intro config, and then reference important elements defined in another intro's config using an **include** in the content file. This mechanism is valuable in situations where related products are built on top of one another and it is necessary to introduce users to key concepts in the higher level products.

Contributing a standby content part

Plug−ins can also implement a part for displaying alternative content when the intro page is in standby mode. For example, the platform defines a standby part that will show a cheat sheet for related intro content. The part is launched using a page link with a specialized URL. Standby parts are launched using a URL containing a special command for showing a standby part, such as `http://org.eclipse.ui.intro/showStandby?partId=somePartId`. The part is defined in the **standbyContentPart** subelement in the **org.eclipse.ui.intro.configExtension** extension. An **id**, **pluginId**, and **class** must be specified for the part. The class must implement **IStandbyContentPart**. The following snippet shows how the platform defines a standby part for showing cheat sheets.

```
<extension point="org.eclipse.ui.intro.configExtension">
```

```
        <standbyContentPart
                id="org.eclipse.platform.cheatsheet"
                class="org.eclipse.platform.internal.CheatSheetStandbyContent"
                pluginId="org.eclipse.platform"/>
</extension>
```

This cheat sheet could be launched from an intro page using a **link** subelement whose URL is `http://org.eclipse.ui.intro/showStandby?partId=org.eclipse.platform.cheatsheet&`
This IntroURL would launch the org.eclipse.platform.cheatsheet standby content part and set its input to "org.eclipse.pde.helloworld". The detailed mechanics for implementing a standby part are beyond the scope of this discussion. See **IStandbyContentPart** and its related classes for more information.

Defining a custom IntroURL action

Using the **org.eclipse.ui.intro.configExtension** extension point, plug−ins can contribute their own custom actions that can be used as a **url** value for a link element in a page. For example, consider the following link:

```
http://org.eclipse.ui.intro/runAction?pluginId=org.eclipse.pde.ui&class=org.ecli
```

This IntroURL will run an action class called **ShowSampleAction**, which is in a package "org.eclipse.pde.ui.internal.samples" in the plug−in "org.eclipse.pde.ui". The id of the sample to run is "org.eclipse.sdk.samples.swt.examples".

To define a custom version of this intro URL, you can use the following markup:

```
<extension point="org.eclipse.ui.intro.configExtension">
        <action
                name="myCommand"
replaces="runAction?pluginId=org.eclipse.pde.ui&class=org.eclipse.pde.ui.internal.samples.ShowSam
        </action>
</extension>
```

With the above extension you can now use the following URL to run the same action:

```
http://org.eclipse.ui.intro/myCommand?id=org.eclipse.sdk.samples.swt.examples
```

The action "myCommand" will be replaced by the value of the **replaces** attribute and any remaining URL parameters will be appended to the end. Once the substitution is made, the resulting URL will be expanded back into:

```
http://org.eclipse.ui.intro/runAction?pluginId=org.eclipse.pde.ui&class=org.ecli
```

Workbench resource support

The Eclipse platform is structured so that you can develop a workbench application even if your application has nothing to do with the platform resource model. However, the workbench does provide support for working with resources in a separate plug−in, **org.eclipse.ui.ide**. This plug−in contains the parts of the SDK workbench that focus on IDE building and manipulating the workspace.

If your plug−in uses the platform resource model, you may want to take advantage of the resource−oriented features in the workbench. These include resource property pages, resource marker UI, resource filtering, and other utilities.

Contributing a property page

You can contribute a property page for an object by using the **org.eclipse.ui.propertyPages** extension point. An object's property page is invoked using the **Properties** menu in any view that shows objects, such as the resource navigator view. This menu is available when a single object is selected.

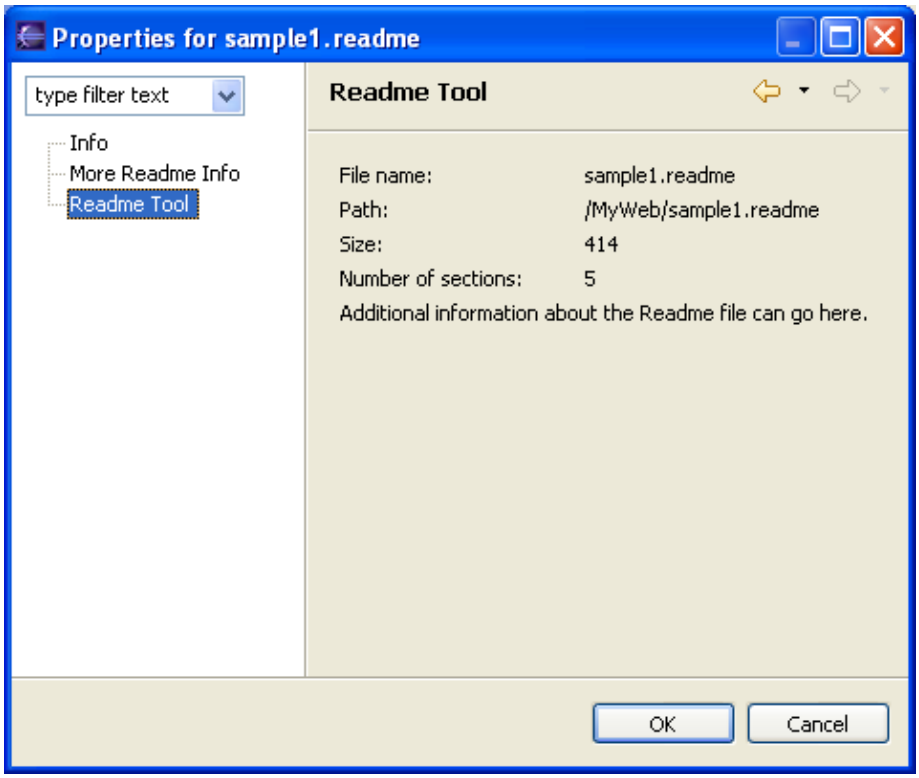
The readme tool contributes two property pages.

```
<extension
     point = "org.eclipse.ui.propertyPages">
        <page
           id="org.eclipse.ui.examples.readmetool.FilePage"
           name="%PropertiesPage.filePage"
           objectClass="org.eclipse.core.resources.IFile"
           class="org.eclipse.ui.examples.readmetool.ReadmeFilePropertyPage"
           nameFilter="*.readme">
        </page>
         <page
            id="org.eclipse.ui.examples.readmetool.FilePage2"
            name="%PropertiesPage.filePage2"
           objectClass="org.eclipse.core.resources.IFile"
           class="org.eclipse.ui.examples.readmetool.ReadmeFilePropertyPage2"
            nameFilter="*.readme">
         </page>
  </extension>
```

When you define a property page, you specify the **objectClass** for which this page is valid. Objects of this class will include your page when the properties are shown. You may optionally supply a **nameFilter** that further refines the class. In the readme tool example, both pages are contributed for objects of type **IFile** with a **.readme** file extension.

Property pages are not limited to workbench resources. All objects showing up in the workbench (even domain−specific objects created by other plug−ins) may have property pages. Any plug−in may register property pages for any object type.

Property pages look a lot like preference pages, except there is no hierarchy or categorization of property pages. In the dialog below, both readme property pages appear in the main list of pages.



Implementing a property page

When the workbench creates and launches a properties page, it sets the selected resource into the page. The page can use the **getElement()** method to obtain its element, an **IAdaptable**.

The pattern for creating property pages is similar to that of preference pages, so we will only focus on what is different. Property pages show information about their element. This information can be obtained by accessing the element in order to query or compute the relevant information. The information can also be stored and retrieved from the resource's properties.

The **ReadmeFilePropertyPage** computes most of its information using its element. The following snippet shows how the number of sections is computed and displayed in a label.

```
...
IResource resource = (IResource) getElement();
...
IAdaptable sections = getSections(resource);
if (sections instanceof AdaptableList) {
   AdaptableList list = (AdaptableList)sections;
   label = createLabel(panel, String.valueOf(list.size()));
...
```

When a property is computed, there is no need for corresponding logic to save the value, since the user cannot update this value.

Properties pages are commonly used for viewing and for setting the application–specific properties of a resource. (See Resource properties for a discussion of session and persistent properties.) Since the property page knows its resource, the resources API can be used in the page to initialize control values or to set new property values based on user selections in the properties page.

The following snippet shows a checkbox value being initialized from a property on a property page's element.

```
private void initializeValues() {
   ...
   IResource resource = (IResource) getElement();
   label.setText(resource.getPersistentProperty("MyProperty"));
   ...
}
```

The corresponding code for saving the checkbox value back into the property looks like this:

```
private void storeValues() {
   ...
   IResource resource = (IResource) getElement();
   resource.setPersistentProperty("MyProperty", label.getText());
   ...
}
```

Marker help and resolution

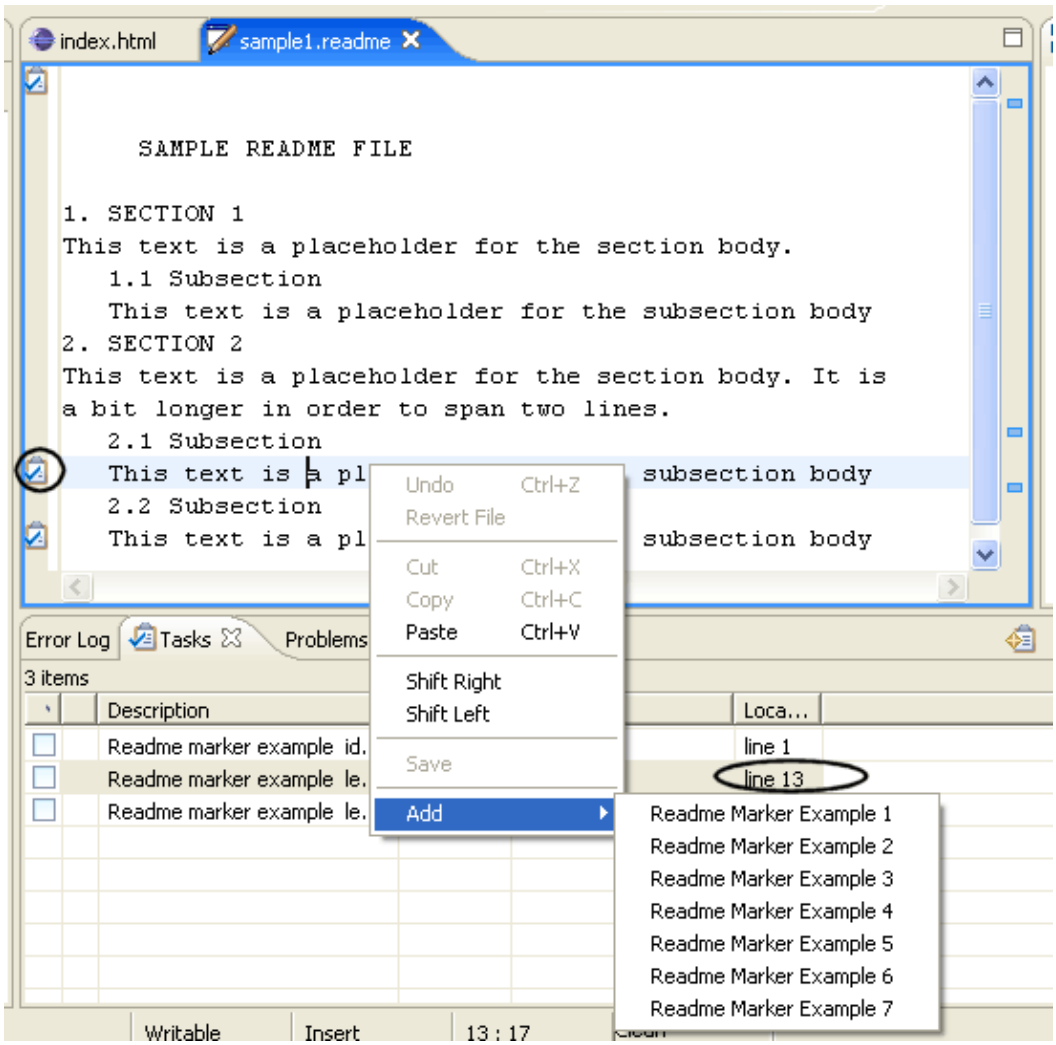
In Resource markers, we saw how plug−ins can define specialized marker types in order to annotate resources with information. The readme tool example defines its own markers in order to demonstrate two marker−related workbench extensions: **marker help** and **marker resolutions**. The marker definition is in the readme plug−in's manifest markup:

```
<extension id="readmemarker" point="org.eclipse.core.resources.markers" name="%ReadmeMarker.name"
        <super type="org.eclipse.core.resources.taskmarker"/>
        <super type="org.eclipse.core.resources.textmarker"/>
        <persistent value="true"/>
        <attribute name="org.eclipse.ui.examples.readmetool.id"/>
        <attribute name="org.eclipse.ui.examples.readmetool.level"/>
        <attribute name="org.eclipse.ui.examples.readmetool.department"/>
        <attribute name="org.eclipse.ui.examples.readmetool.code"/>
        <attribute name="org.eclipse.ui.examples.readmetool.language"/>
</extension>
```

The tool defines a marker that inherits from the platform's text marker and task marker. It also defines named attributes for the marker. Marker attributes can be set and queried.

Since the new readme marker is a kind of text marker, it inherits the text marker attributes. The text marker attributes include the character location of the marker.

Markers can be added to a **.readme** file using the readme editor's popup menu. (The popup menu actions are added dynamically in **ReadmeTextEditor.editorContextMenuAboutToShow(IMenuManager parentMenu)**). Once added, the markers appear on the left side of the editor and in the tasks view.



Contributing marker help

Now we are ready to look at how to add help to the readme tool's markers. Adding marker help is done using the **org.eclipse.ui.ide.markerHelp** extension point. This extension point allows plug–ins to associate a help context id with a particular type of marker. The marker can be qualified by marker type only, or it can be further qualified by the value of one or more of its attributes. The readme tool declares several different help contexts:

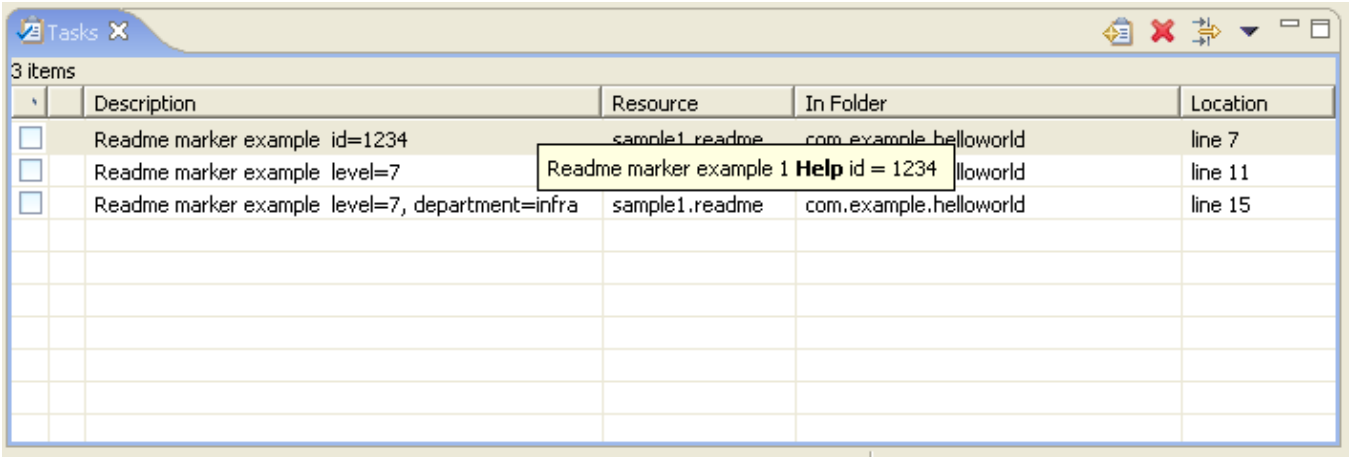
```
<extension point="org.eclipse.ui.ide.markerHelp">
   <markerHelp
      markerType="org.eclipse.ui.examples.readmetool.readmemarker"
      helpContextId="org.eclipse.ui.examples.readmetool.marker_example1_context">
      <attribute name="org.eclipse.ui.examples.readmetool.id" value= "1234"/>
   </markerHelp>
   <markerHelp
      markerType="org.eclipse.ui.examples.readmetool.readmemarker"
      helpContextId="org.eclipse.ui.examples.readmetool.marker_example2_context">
      <attribute name="org.eclipse.ui.examples.readmetool.level" value= "7"/>
   </markerHelp>
   <markerHelp
      markerType="org.eclipse.ui.examples.readmetool.readmemarker"
      helpContextId="org.eclipse.ui.examples.readmetool.marker_example3_context">
      <attribute name="org.eclipse.ui.examples.readmetool.level" value= "7"/>
```

```
    <attribute name="org.eclipse.ui.examples.readmetool.department" value= "infra"/>
  </markerHelp>
  ...
```

Each marker help context is defined for the readme marker type. However, each help context is associated with a different combination of attribute values. The first marker help context will be used for markers whose **id** attribute is set to **"1234"**. The help contexts are defined in the plug−in's **HelpContexts.xml** file:

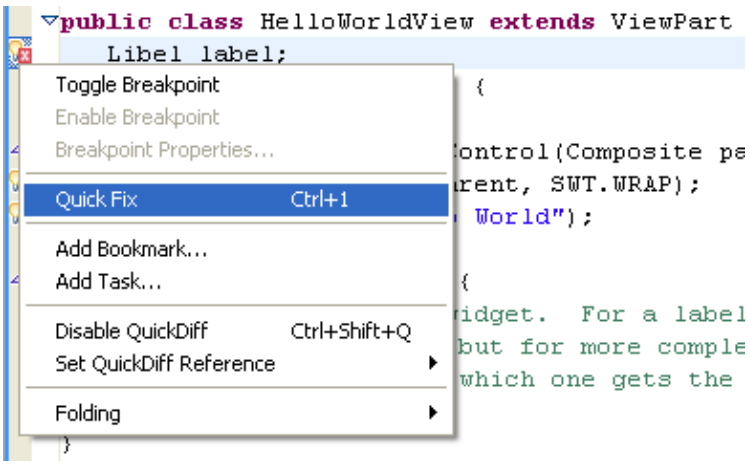
```
<context id="marker_example1_context" >
        <description>Readme marker example 1 <b>Help</b> id = 1234 </description>
</context>
```

Sure enough, when we select a readme marker with id="1234" and select help using F1, we see our help description.



Contributing marker resolution

Plug−ins can also define marker resolutions, so that their problem markers can participate in the workbench **Quick Fix** feature. Users can select a problem marker and choose a **Quick Fix** from a popup containing the list of supplied fixes contributed for the marker.

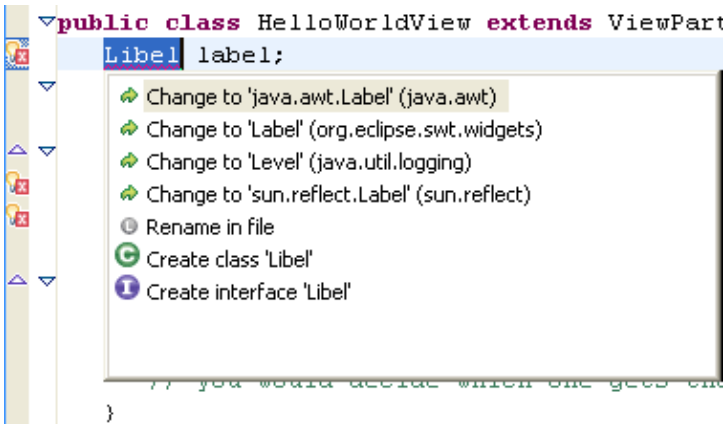


Marker resolutions are contributed using the **org.eclipse.ui.ide.markerResolution** extension point. This extension point allows plug−ins to associate a **class** that implements **IMarkerResolutionGenerator** with a particular type of marker. The marker can be qualified by marker type only, or it can be further qualified by

the value of one or more of its attributes. The JDT contributes a marker resolution for Java problems:

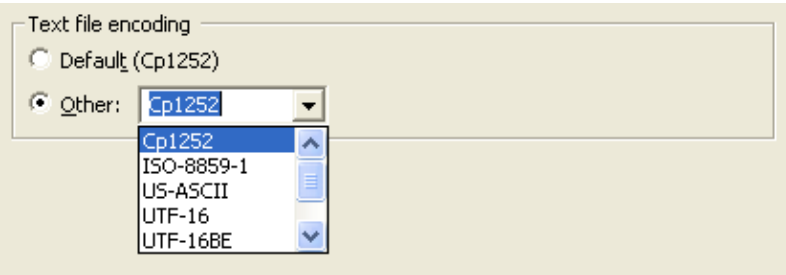
```
<extension
        point="org.eclipse.ui.ide.markerResolution">
        <markerResolutionGenerator
                markerType="org.eclipse.jdt.core.problem"
                class="org.eclipse.jdt.internal.ui.text.correction.CorrectionMarkerResolutionGene
        </markerResolutionGenerator>
</extension>
```

The marker resolution generator is responsible for returning an array of marker resolutions (**IMarkerResolution**) that will be shown in the **Quick Fix** popup. The resolution will be **run()** if the user selects one of the fixes.



Text file encoding

If your plug–in reads text files, it should honor the **text file encoding** preference in the workbench.



Text files are encoded differently depending on the platform and the locale. Most of the time, using the default text file encoding for the locale of the host operating system is good enough. However, a user may want to work with text files that originate from another source. Given the ability to use the platform in a networked team environment, it's certainly possible that users will want to work with text files that use a different encoding scheme than their native encoding scheme so that they can easily interchange files with another team.

For this reason, the workbench defines its own encoding profile that is specified by the user in the **Preferences** dialog. Users may choose from the available encoding choices in the **General > Editors** preference page or type in their own encoding. Plug–ins that interpret text files, such as editors and builders, should consult the workbench encoding preference rather than assume that the installed operating system encoding is in use.

You can obtain the encoding preference using **ResourcesPlugin.getEncoding().** This encoding should be passed to **java.io** readers instead of using the default system encoding. If you need to track changes to this preference, you can hook a listener on the **ResourcesPlugin** preferences and react to changes in **ResourcesPlugin.PREF_ENCODING.** The following example comes from the default text editor:

```
public void initialize(StatusTextEditor textEditor) {

        fTextEditor= textEditor;

        fPropertyChangeListener= new Preferences.IPropertyChangeListener() {
                public void propertyChange(Preferences.PropertyChangeEvent e) {
                        if (ResourcesPlugin.PREF_ENCODING.equals(e.getProperty()))
                                setEncoding(null, false);
                }
        };

        Preferences p= ResourcesPlugin.getPlugin().getPluginPreferences();
        p.addPropertyChangeListener(fPropertyChangeListener);

        fEncodingActionGroup= new EncodingActionGroup(fTextEditor);
        fEncodingActionGroup.update();
}
```

Users may also change the encoding for a particular file in the **Edit > Encoding** menu of an editor. If you are manipulating text inside an open editor, you should use **IEncodingSupport.getEncoding()** instead in order to get the encoding for the particular editor. The following example shows how to obtain this information from an editor:

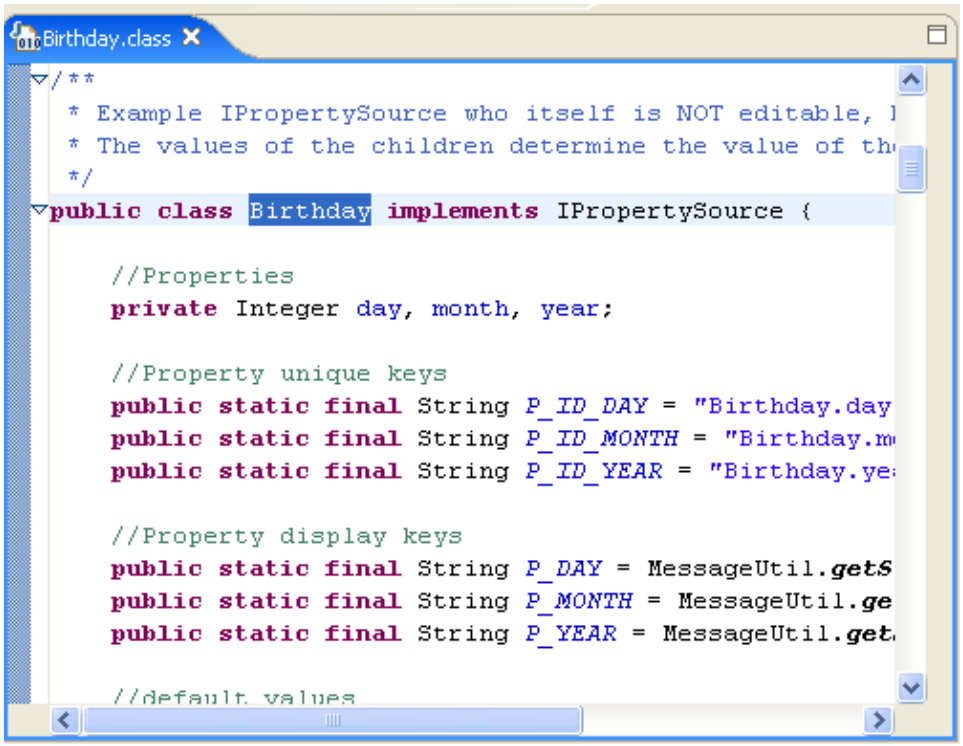
```
IEncodingSupport encodingSupport = (IEncodingSupport) editor.getAdapter(IEncodingSupport.class);
String encoding = encodingSupport.getEncoding();
```

Editors

We have seen how plug−ins can contribute an editor to the workbench, but we haven't yet looked at the implementation of an editor.

There is no "typical" implementation pattern for an editor, because editors usually provide application−specific semantics. A tool that edits and manages a particular content type will provide customized behavior for manipulating the data represented by the resource.

Editors can come in all shapes and sizes. If a plug−in's editor is text−based, then the editor can either use the existing default text editor, or create a customized text editor by using the facilities provided in the platform. The latter approach is used by the Java example editor.



If a plug−in's editor is not text based, then a custom editor must be implemented by the plug−in. There are several approaches for building custom editors, all of which depend on the look and behavior of the editor.

- Form−based editors can layout controls in a fashion similar to a dialog or wizard. The Plug−in Development Environment (PDE) uses this approach in building its manifest editors.
- Graphics intensive editors can be written using SWT level code. For example, an editor could create its own SWT window for displaying the information, or it could use a custom SWT control that is optimized for the application.
- List−oriented editors can use JFace list, tree, and table viewers to manipulate their data.

Once the implementation model for the editor has been determined, implementing the editor is much like programming a stand−alone JFace or SWT application. Platform extensions are used to add actions, preferences, and wizards needed to support the editor. But the internals of the editor are largely dependent on your application design principles and internal model.

Workbench editors

Although the implementation of a workbench editor will be specific to your plug−in and the content that you want to edit, the workbench provides a general structure for building an editor. The following concepts apply to all workbench editors.

Editor parts and their inputs

An editor must implement **IEditorPart** and is often built by extending the **EditorPart** class. An editor implements its user interface in the **createPartControl** method. This method is used to assemble the SWT widgets or JFace viewers that present the editor contents.

An **editor input** is a description of something to be edited. You can think of an editor input as a file name, though it is more general. **IEditorInput** defines the protocol for an editor input, including the name of the input and the image that should be used to represent it in the labels at the top of the editor.

Two generic editor inputs are provided in the platform. **IFileEditorInput** represents an input that is a file in the file system. **IStorageEditorInput** represents an input that is a stream of bytes. These bytes may come from sources other than the file system.

Resetting the editor input

If your editor can support the replacement of the editor's input object on the fly, you should implement **IReusableEditor**. Implementing this interface allows the workbench to "recycle" your editor. Workbench user preferences allow the user to dictate that editors should be reused after a certain number of them are open.

Navigating the editor input

If you want to implement a navigation history in your editor, you should implement **INavigationLocationProvider**. This provides a mechanism for the workbench to request a current navigation location (**INavigationLocation**) as needed to keep a navigation history. The workbench handles the mechanics of the navigation user interface. Your **INavigationLocation** will be notified when it needs to restore the editor to the location that it represents.

The rest of your editor's implementation depends on the content that you are trying to present. We'll look next at the most common type of editor – the text editor.

Documents and partitions

The platform text framework defines a document model for text and provides a viewer that displays text using this model. We will start by looking at the Java editor example and how it uses this model. We will not focus on the basic mechanics of registering an editor extension, since we've already seen this in the section discussing **org.eclipse.ui.editors**. Instead, we'll look at the specifics of how the editor class is implemented in the example.

Document providers and documents

In the workbench, an editor is typically opened when the user selects a domain element (such as a file or an element stored inside an archive file) and opens it. When the editor is created, it is associated with an editor input (**IEditorInput**), which describes the object being edited.

The Java editor example opens when the user opens a file with the "*.jav" extension. In this case, the input to the editor is an **IFileEditorInput**. The platform text framework assumes little about the editor input itself. It works with a presentation model, called an **IDocument**, for the input, so that it can effectively display and manipulate text.

This means that there must be a way to map from an expected domain model (the editor input) to the presentation model. This mapping is defined in an **IDocumentProvider**. Given an editor input, the document provider returns an appropriate **IDocument**.

The Java editor example inherits the **TextFileDocumentProvider** defined by the plug−in **org.eclipse.ui.editors**. The extension **org.eclipse.ui.editors.documentProviders** is used to define mappings between editor input types (or file extensions) and document providers. The editors plug−in defines its document provider as follows:

```
<extension
      point="org.eclipse.ui.editors.documentProviders">
   <provider
         class="org.eclipse.ui.editors.text.TextFileDocumentProvider"
         inputTypes="org.eclipse.ui.IStorageEditorInput"
         id="org.eclipse.ui.editors.text.StorageDocumentProvider">
   </provider>
</extension>
```

This extension point allows plug−ins to register document providers and associate them with either a file extension or an editor input class. Since the Java editor example does not define its own document provider extension, it inherits the generic document provider specified for all input types that are **IStorageEditorInput**. When the user opens a file for editing, the platform manages the details of creating the proper document provider instance. If a specific document provider is registered for the file extension, that one will be used. If there is no specific document provider for the file extension, then the editor input type will be used to find the appropriate provider.

By using the generic platform document provider, the Java editor example can take advantage of all of the features of the document provider, such as file buffering and other optimizations.

Document setup

Since the Java editor uses the platform text document provider, how can it supply any specialized behavior for handling Java files?

The extension **org.eclipse.core.filebuffers.documentSetup** is used to define mappings between file extensions and an **IDocumentSetupParticipant**. The setup participant will set up the document with any special features once it has been provided to the editor.

```
<extension
        id="ExampleJavaDocumentSetupParticipant"
        name="%documentSetupParticipantName"
        point="org.eclipse.core.filebuffers.documentSetup">
```

```
        <participant
                extensions="jav"
class="org.eclipse.ui.examples.javaeditor.JavaDocumentSetupParticipant">
        </participant>
</extension>
```

This extension definition is what gives the example a chance to setup the document for Java specific tasks. So what does **JavaDocumentSetupParticipant** do? We'll look at a simplified version of the **setup** method.

```
        public void setup(IDocument document) {
                ...
                IDocumentPartitioner partitioner= new FastPartitioner(JavaEditorExamplePlugin.get
                partitioner.connect(document);
                ...
        }
```

The setup code configures an object called a **partitioner**.

Partitions

The partitioner (**IDocumentPartitioner**) is responsible for dividing the document into non−overlapping regions called partitions. Partitions (represented by **ITypedRegion**) are useful for treating different sections of the document differently with respect to features like syntax highlighting or formatting.

In the case of the Java editor example, the document is divided into partitions that represent the javadoc comments, multi line comments, and everything else. Each region is assigned a content type and its position in the document. Positions are updated as the user edits text.

Rule based document partitioning

It is up to each editor to determine the appropriate implementation for a document partitioner. Support is provided in **org.eclipse.jface.text.rules** for rule−based document scanning. Using a rule−based scanner allows an editor to use the **FastPartitioner** provided by the framework.

```
IDocumentPartitioner partitioner= new FastPartitioner(JavaEditorExamplePlugin.getDefault().getJav
```

RuleBasedPartitionScanner is the superclass for rule based scanners. Subclasses are responsible for enumerating and implementing the rules that should be used to distinguish tokens such as line delimiters, white space, and generic patterns when scanning a document. The example's **JavaPartitionScanner** defines rules for distinguishing single line comments, character constants, javadoc, multi line comments, and words. This is done in the scanner's constructor:

```
public JavaPartitionScanner() {
        super();
        IToken javaDoc= new Token(JAVA_DOC);
        IToken comment= new Token(JAVA_MULTILINE_COMMENT);

        List rules= new ArrayList();
        // Add rule for single line comments.
        rules.add(new EndOfLineRule("//", Token.UNDEFINED));

        // Add rule for strings and character constants.
        rules.add(new SingleLineRule("\"", "\"", Token.UNDEFINED, '\\'));
        rules.add(new SingleLineRule("'", "'", Token.UNDEFINED, '\\'));

        // Add special case word rule.
```

```
        rules.add(new WordPredicateRule(comment));

        // Add rules for multi-line comments and javadoc.
        rules.add(new MultiLineRule("/**", "*/", javaDoc, (char) 0, true));
        rules.add(new MultiLineRule("/*", "*/", comment, (char) 0, true));

        IPredicateRule[] result= new IPredicateRule[rules.size()];
        rules.toArray(result);
        setPredicateRules(result);
}
```

See the classes in **org.eclipse.jface.text.rules** for more details about defining rules and the types of rules availables. We'll look at the scanners again when we look at syntax coloring.

Syntax coloring

Syntax coloring is provided in the platform text framework using a model of damage, repair, and reconciling. For each change applied to a document, a presentation reconciler determines which region of the visual presentation should be invalidated and how to repair it. Different strategies can be used for different content types in the document.

Implementing syntax coloring (and doing so with a presentation reconciler) is optional. By default, **SourceViewerConfiguration** does not install a presentation reconciler since it does not know the document model used for a particular editor, and has no generic behavior for syntax highlighting.

In order to use the reconciling classes to implement syntax highlighting, your editor's source viewer configuration must be configured to define a presentation reconciler. Once again, we start with **JavaSourceViewerConfiguration** to see how a presentation reconciler is defined for our editor.

```
public IPresentationReconciler getPresentationReconciler(ISourceViewer sourceViewer) {

        PresentationReconciler reconciler= new PresentationReconciler();
        ...
        return reconciler;
}
```

To understand what a presentation reconciler does, we must first look at the concepts of damage, repair, and reconciling.

Damage, repair, and reconciling

As the user modifies text in an editor, parts of the editor must be redisplayed to show the changes. Computing the text that must be redisplayed is known as computing **damage**. When syntax coloring is involved, the amount of damage caused by an editing operation becomes more extensive, since the presence or absence of a single character could change the coloring of the text around it.

Damagers (**IPresentationDamager**) determine the region of a document's presentation which must be rebuilt because of a document change. A presentation damager is assumed to be specific to a particular document content type (or region). It must be able to return a damage region that is valid input for a presentation repairer (**IPresentationRepairer**). A repairer must be able to derive all of the information it needs from a damage region in order to successfully describe the **repairs** that are needed for a particular content type.

Reconciling describes the overall process of maintaining the presentation of a document as changes are made in the editor. A presentation reconciler (**IPresentationReconciler**) monitors changes to the text through its associated viewer. It uses the document's regions to determine the content types affected by the change and notifies a damager that is appropriate for the affected content type. Once the damage is computed, it is passed to the appropriate repairer which will construct repair descriptions that are applied to the viewer to put it back in sync with the underlying content.

The classes in **org.eclipse.jface.text.reconciler** define additional support classes for synchronizing a document model with external manipulation of the document.

Presentation reconcilers should be provided with a repairer and damager pair for each content type to be found in the document. It is up to each editor to determine the appropriate implementation for a presentation reconciler. However, the platform provides support in **org.eclipse.jface.text.rules** for using rule−based document scanners to compute and repair damage. Default damagers and repairers are defined in this package. They can be used along with the standard reconcilers in **org.eclipse.jface.text.presentation** to implement syntax coloring by defining scanning rules for the document.

Rule based reconciling

Now we have enough background to look in detail at the creation of the example presentation reconciler. Recall that the Java editor example implements a **JavaPartitionScanner** which partitions the document into content types representing javadoc, multi line comments, and everything else.

For each of these content types, a damager/repairer pair must be specified. This is done below using the **PresentationReconciler** and the **DefaultDamagerRepairer**.

```
JavaColorProvider provider= JavaEditorEnvironment.getJavaColorProvider();
PresentationReconciler reconciler= new PresentationReconciler();

DefaultDamagerRepairer dr= new DefaultDamagerRepairer(JavaEditorEnvironment.getJavaCodeSc
reconciler.setDamager(dr, IDocument.DEFAULT_CONTENT_TYPE);
reconciler.setRepairer(dr, IDocument.DEFAULT_CONTENT_TYPE);

dr= new DefaultDamagerRepairer(new SingleTokenScanner(new TextAttribute(provider.getColor
reconciler.setDamager(dr, JavaPartitionScanner.JAVA_DOC);
reconciler.setRepairer(dr, JavaPartitionScanner.JAVA_DOC);

dr= new DefaultDamagerRepairer(new SingleTokenScanner(new TextAttribute(provider.getColor
reconciler.setDamager(dr, JavaPartitionScanner.JAVA_MULTILINE_COMMENT);
reconciler.setRepairer(dr, JavaPartitionScanner.JAVA_MULTILINE_COMMENT);

return reconciler;
```

Note that the example provide scanners for each content type.

The default content type is set up with a **JavaCodeScanner** so that keywords can be detected and colored. The **JavaCodeScanner** builds rules for detecting different kinds of tokens, such as single line comments, white space, and words. It describes the colors that should be used for words of different token types.

The other content types are set up with a **SingleTokenScanner** and given a color to be used for tokens in these content types.

All of the details for damaging and repairing the proper parts of the documents according to the scanning rules are handled by **DefaultDamagerRepairer**. These details typically don't need to be understood by plug−in

code. Your plug−in should focus on building a set of rules that are appropriate for partitioning and scanning its editor content.

Dynamically installing a reconciler

The Java editor example provides a subclass of **SourceViewerConfiguration** for installing the presentation reconciler as seen earlier. A presentation reconciler can also be installed dynamically on a text viewer using **IPresentationReconciler** protocol. There is no particular runtime benefit to doing it either way, but putting all of the pluggable behavior overrides in a subclass of **SourceViewerConfiguration** provides the advantage of consolidating all of the behavioral overrides in one place. The dynamic protocol may be useful when different presentation reconcilers are attached to a viewer throughout the life of an editor.

Configuring a source viewer

So far we've looked at **SourceViewer** in the context of managing source code annotations.

The **SourceViewer** is also the central hub for configuring your editor with pluggable behavior such as text hovering and syntax highlighting. For these features, the editor supplies a **SourceViewerConfiguration** that is used to configure the **SourceViewer** when it is created. The Java example editor need only to supply a **SourceViewerConfiguration** appropriate for its needs. The following snippet shows how the **JavaTextEditor** creates its configuration:

```
protected void initializeEditor() {
        super.initializeEditor();
setSourceViewerConfiguration(new JavaSourceViewerConfiguration());
        ...
```

What does the **JavaSourceViewerConfiguration** do? Much of its behavior is inherited from **SourceViewerConfiguration**, which defines default strategies for pluggable editor behaviors such as auto indenting, undo behavior, double−click behavior, text hover, syntax highlighting, and formatting. Public methods in **SourceViewerConfiguration** provide the helper objects that implement these behaviors.

If the default behavior defined in **SourceViewerConfiguration** does not suit your editor, you should override **initializeEditor()** as shown above and set your own source viewer configuration into the editor. Your configuration can override methods in **SourceViewerConfiguration** to supply customized helper objects that implement behavior for your editor. The following snippet shows two of the ways the **JavaSourceViewerConfiguration** supplies customized helper objects for the Java editor example:

```
public IAnnotationHover getAnnotationHover(ISourceViewer sourceViewer) {
        return new JavaAnnotationHover();
}

public IAutoIndentStrategy getAutoIndentStrategy(ISourceViewer sourceViewer, String contentType)
        return (IDocument.DEFAULT_CONTENT_TYPE.equals(contentType) ? new JavaAutoIndentStrategy()
}
```

In the first method, a customized helper class is provided for implementing annotation hovering. In the second method, the default content type of the document is queried to determine whether a customized auto−indent strategy or the default strategy should be used.

See the API reference for **SourceViewerConfiguration** for all the ways you can configure a source viewer by overriding methods.

Source viewers and annotations

The editor and its corresponding text viewer are largely responsible for the implementation of the document's presentation and the configuration of any needed helper classes. (See Viewers if you are not familiar with the concept of a viewer.)

A **TextViewer** handles all of the low level details of mapping the document model and its partitions into the colored and formatted text that a user sees. For source code style editors, a **SourceViewer** is provided. A source viewer introduces the notion of source code annotations. These annotations can be shown in a vertical ruler on the left side of the text, an overview ruler on the right side of the text, or as colored squigglies underneath text.

SourceViewer and its helper classes are used throughout the **AbstractTextEditor** hierarchy. The package **org.eclipse.jface.text.source** defines this viewer and the other classes supporting annotation presentation.

Annotations and rulers

Annotations, like partitions, are largely dependent on the kind of document being edited. The **IAnnotationModel** for a document is what holds the annotations, enumerates them on request, and listens for text changes in order to keep the annotations up to date with the text. Annotation models are registered in the **org.eclipse.core.filebuffers.annotationModelCreation** extension. This extension point allows plug−ins to register a class that will create an annotation model appropriate for a given file extension. The Java Editor example does not use this extension point, so it inherits the annotation model defined by the platform.

```
<extension
        point="org.eclipse.core.filebuffers.annotationModelCreation">
        <factory
                extensions="*"
class="org.eclipse.ui.texteditor.ResourceMarkerAnnotationModelFactory">
        </factory>
</extension>
```

The supplied factory class will create a **ResourceMarkerAnnotationModel** for files with any extension. This class displays annotations that represent a marker on a resource in the workspace. (See Resource markers for more information on markers.) It assigns an image and description to each marker and monitors its resource for changes in the markers.

To see how an annotation model is displayed in a text editor, we'll examine the platform text editor and its use of rulers and annotations. The specifics of how different annotations are shown in the rulers and text can be controlled by the user in the **General > Editors > Text Editors > Annotations** preferences.

Vertical ruler

A vertical ruler to the left of the editing area is used by platform text editors to show text ranges and line−based annotations adjacent to their text line.

```
        return year;
    }
    /* (non-Javadoc)
     * Method declared on IPropertySource
     */
    public boolean isPropertySet(Object property) {
        if (P_ID_DAY.equals(property))
            return getDay() != DAY_DEFAULT;
        if (P_ID_MONTH.equals(property))
            return getMonth() != MONTH_DEFAULT;
        if (P_ID_YEAR.equals(property))
            return getYear() != YEAR_DEFAULT;
        return false;
    }
    /* (non-Javadoc)
     * Method declared on IPropertySource
     */
```

These annotations are described in the supplied **ResourceMarkerAnnotationModel**. This model is set into the **SourceViewer** when the source viewer is initialized by the editor. The following snippet from **AbstractTextEditor** shows how the document and the annotation model are associated with the viewer.

```
private void initializeSourceViewer(IEditorInput input) {

        IAnnotationModel model= getDocumentProvider().getAnnotationModel(input);
        IDocument document= getDocumentProvider().getDocument(input);

        if (document != null) {
fSourceViewer.setDocument(document, model);
                ...
```

Once the source viewer is configured with the proper document and annotation model, it has enough information to present the document and ensure the correct annotations are shown in the vertical ruler to the left. The model is associated with the ruler when the document is set. The following snippet shows what happens when a document is set into the source viewer. It has been simplified from the actual code in **SourceViewer** for clarity:

```
public void setDocument(IDocument document, IAnnotationModel annotationModel) {
        ...
        // create visual annotation model from the supplied model and store
        // in fVisualAnnotationModel
        ...
        if (fVerticalRuler != null)
fVerticalRuler.setModel(fVisualAnnotationModel);
```

In this way, the ruler is associated with the appropriate annotation model.

Let's look at the ruler itself. It is created by the text editor and then connected with the editor's viewer. Since the Java editor example does not define any special behavior for rulers, it inherits the ruler as defined in **TextEditor**.

```
protected IVerticalRuler createVerticalRuler() {
        CompositeRuler ruler= new CompositeRuler();
        ruler.addDecorator(0, new AnnotationRulerColumn(VERTICAL_RULER_WIDTH));
        if (isLineNumberRulerVisible())
                ruler.addDecorator(1, createLineNumberRulerColumn());
```

```
        return ruler;
}
```

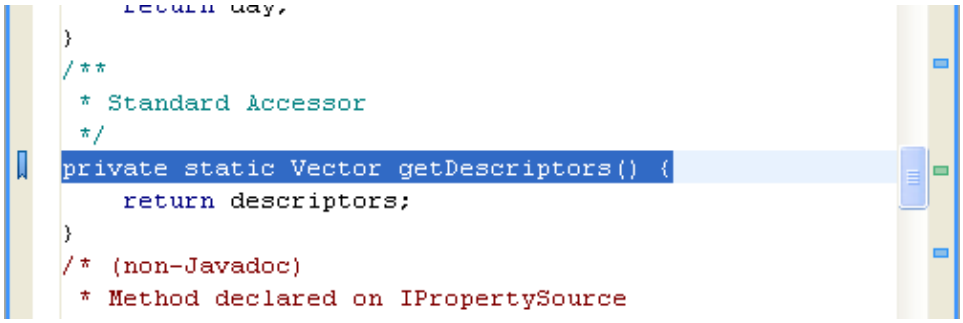
The text editor uses a CompositeRuler. This ruler does not have a visual presentation of its own. The presentation is provided by a list of decorators that show columns (**IVerticalRulerColumn**) in the ruler. In this example, a ruler column that shows annotations (**AnnotationRulerColumn**) is always added, and a line number ruler column is added based on user preferences. The annotation ruler column handles the particulars of displaying the annotation images in the proper locations.

Despite all the classes involved in showing a ruler, note that the example editor needed only to subclass framework classes to get ruler behavior. **JavaDocumentProvider** inherits an appropriate marker annotation model from **FileDocumentProvider**. The JavaTextEditor inherits the ruler presentation from **TextEditor**.

Overview ruler

An overview ruler on the right hand side of the editing area is used to show annotations concerning the entire document. These annotations are shown relative to their position in the document and do not move as the user scrolls the document. There usually is a corresponding annotation on the vertical ruler when that portion of the document is visible.

The vertical ruler below shows that there are two tasks in the document and one bookmark. Since the bookmarked text is visible, its annotation is also shown on the left.



The user can navigate to the location of the annotation in the code by clicking on the annotation itself.

The types of annotations shown in the overview ruler are determined by adding annotation types to the ruler. In the following snippet from **SourceViewerDecorationSupport**, annotation types are dynamically added to the ruler. (See next section for more information about **SourceViewerDecorationSupport**.)

```
private void showAnnotationOverview(Object annotationType) {
        if (fOverviewRuler != null) {
                Color c= getAnnotationTypeColor(annotationType);
                fOverviewRuler.setAnnotationTypeColor(annotationType, c);
                int l= getAnnotationTypeLayer(annotationType);
                fOverviewRuler.setAnnotationTypeLayer(annotationType, l);
                fOverviewRuler.addAnnotationType(annotationType);
                fOverviewRuler.update();
        }
}
```

The overview ruler is also supplied with an **IAnnotationAccess** that is used to provide information about a particular annotation, such as its type and how it is to be displayed. The **TextEditor** uses a **DefaultMarkerAnnotationAccess** which interprets annotations according to their marker types and consults

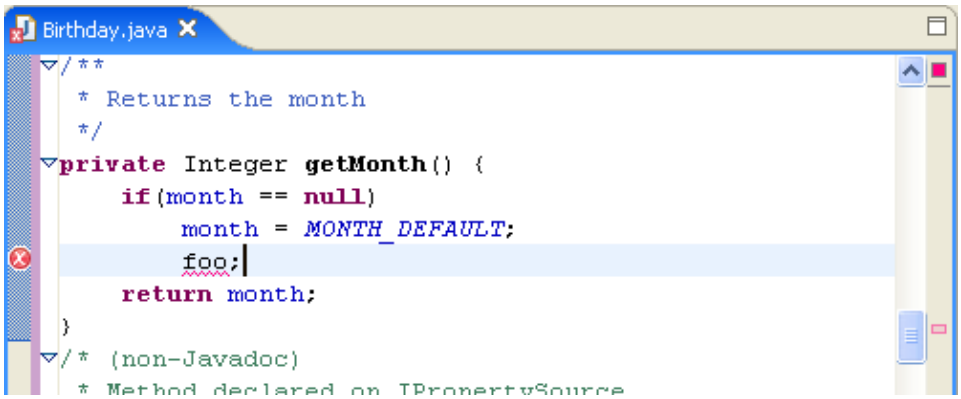
the user preferences to see which marker types should be shown in the overview ruler.

```
protected IAnnotationAccess createAnnotationAccess() {
        return new DefaultMarkerAnnotationAccess(fAnnotationPreferences);
}
```

Consult the implementation of **DefaultMarkerAnnotationAccess** and **MarkerAnnotation** for more detail about presenting markers in the overview ruler.

Text annotations

In addition to showing annotations in the rulers, a source viewer can show annotations as colored squiggly marks in the text.



We'll look again at the creation of the source viewer in **TextEditor**.

```
protected ISourceViewer createSourceViewer(Composite parent, IVerticalRuler ruler, int styles) {

        ...
        ISourceViewer sourceViewer= new SourceViewer(parent, ruler, fOverviewRuler, isOverviewRul
fSourceViewerDecorationSupport= new SourceViewerDecorationSupport(sourceViewer, fOverviewRuler, f
        configureSourceViewerDecorationSupport();

        return sourceViewer;
}
```

The class **SourceViewerDecorationSupport** handles many of the decorations shown in a source viewer, including text annotations, colored margins, colored cursor lines, and the like. It is configured with the user preferences so that it can respond to dynamic updates of user preference changes. Most editors need not be concerned with the details of how these decorations are painted. (See **SourceViewerDecorationSupport** and related classes such as **AnnotationPainter** if you must!). The important thing to know is what decorations are available so that the **SourceViewer** and its supporting **SourceViewerDecorationSupport** are configured correctly.

Configuring a SourceViewerDecorationSupport

Let's look at the configuration used by **TextEditor** for the decoration support.

```
protected void configureSourceViewerDecorationSupport() {

Iterator e= fAnnotationPreferences.getAnnotationPreferences().iterator();
        while (e.hasNext())
```

```
                fSourceViewerDecorationSupport.setAnnotationPreference((AnnotationPreference) e.n
        fSourceViewerDecorationSupport.setAnnotationPainterPreferenceKeys(DefaultMarkerAnnotation

        fSourceViewerDecorationSupport.setCursorLinePainterPreferenceKeys(CURRENT_LINE, CURRENT_L
        fSourceViewerDecorationSupport.setMarginPainterPreferenceKeys(PRINT_MARGIN, PRINT_MARGIN_
        fSourceViewerDecorationSupport.setSymbolicFontName(getFontPropertyPreferenceKey());
}
```

Note that the annotation preferences are used to define annotation types for all of the annotations shown in the user preferences. This includes annotations contributed by any plug−in and is not limited to the workbench−supplied annotations. If you do not wish to show all available annotations in your editor, you should override this method and set up the **SourceViewerDecorationSupport** with only those types you want to show.

Text and ruler hover

Hover support is provided in the platform text framework, allowing you to implement informational hovers (or infopops) over the text and the rulers shown in your editor.

Hover support is optional. By default, **SourceViewerConfiguration** does not install hover behavior since there is no useful general information to show. In order to provide text or ruler hover, your editor's source viewer configuration must be configured to define a pluggable hover object.

Let's look again at **JavaSourceViewerConfiguration** to see which methods define the hover behavior:

```
public ITextHover getTextHover(ISourceViewer sourceViewer, String contentType) {
        return new JavaTextHover();
}
public IAnnotationHover getAnnotationHover(ISourceViewer sourceViewer) {
        return new JavaAnnotationHover();
}
```

Hover helper classes can also be installed dynamically using **SourceViewer** protocol (**setTextHover** and **setAnnotationHover**). There is no particular runtime benefit to doing it either way, but putting all of the pluggable behavior overrides in a subclass of **SourceViewerConfiguration** provides the advantage of consolidating all of the definitions in one place.

Let's look at the specifics of providing both kinds of hover.

Text hover

Text hover allows you to provide informational text about text shown in the editor. This is done using the **ITextHover** interface. A text hover is responsible for computing the region that should be used as the source of hover information, given an offset into the document. It is also responsible for providing the informational text about a specific region. **JavaTextHover** is pretty simple. It checks to see if the supplied offset for hover is contained inside the text selection. If so, it supplies the selection range as hover region.

```
public class JavaTextHover implements ITextHover {

        ...

        public IRegion getHoverRegion(ITextViewer textViewer, int offset) {
                Point selection= textViewer.getSelectedRange();
                if (selection.x <= offset && offset < selection.x + selection.y)
```

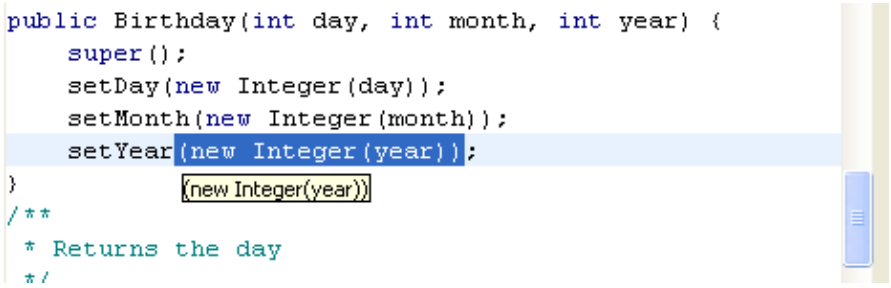
```
                        return new Region(selection.x, selection.y);
                return new Region(offset, 0);
        }
}
```

Given its own computed hover region, it obtains the selected text from its document and returns that as the hover info.

```
public class JavaTextHover implements ITextHover {

        public String getHoverInfo(ITextViewer textViewer, IRegion hoverRegion) {
                if (hoverRegion != null) {
                        try {
                                if (hoverRegion.getLength() > -1)
                                        return textViewer.getDocument().get(hoverRegion.getOffset
                        } catch (BadLocationException x) {
                        }
                }
                return JavaEditorMessages.getString("JavaTextHover.emptySelection");
        }
        ...
}
```

Sure enough, we can see that if we hover over a selection in the editor, the hover text shows the selection.



More complicated contextual information can be used to compute useful hover information. Examples of this can be found in the **JavaTextHover** implemented with the JDT editor.

Ruler hover

Hover on the vertical ruler is useful for showing show line−oriented information. The hover class is configured as described above. **IAnnotationHover** is the interface for ruler hover objects. Although the name implies that the hover is designed for annotations in the ruler, it is really up to an individual editor to detemine what is appropriate. A ruler hover is responsible for returning the info string associated with a particular line number, regardless of the presence of markers on that line.

The Java example editor's **JavaAnnotationHover** implements hover for all lines. It uses the line number to obtain all of the text on the hover line and returning it as the info string.

```
public String getHoverInfo(ISourceViewer sourceViewer, int lineNumber) {
        IDocument document= sourceViewer.getDocument();

        try {
                IRegion info= document.getLineInformation(lineNumber);
                return document.get(info.getOffset(), info.getLength());
        } catch (BadLocationException x) {
```

```
        }
        return null;
}
```

```
        private Integer getDay() {
    if(day == null) (day == null)
                day = DAY_DEFAULT;
            return day;
}
```

Since the hover has access to the document and the source viewer, it has all the context needed to make more complicated contextual decisions about the info that should be shown. For example, the annotation model could be retrieved from the source viewer in order to provide hover info for any annotations shown in the vertical ruler. The **JavaAnnotationHover** provided by the JDT editor provides this capability.

Content assist

Content assist allows you to provide context sensitive content completion upon user request. This functionality is implemented by the platform text framework in **org.eclipse.jface.text.contentassist**. Popup windows (infopops) are used to propose possible text choices to complete a phrase. The user can select these choices for insertion in the text. Content assist also supports contextual infopops for providing the user with information that is related to the current position in the document.

Implementing content assist is optional. By default, **SourceViewerConfiguration** does not install a content assistant since it does not know the document model used for a particular editor, and has no generic behavior for content assist.

In order to implement content assist, your editor's source viewer configuration must be configured to define a content assistant. This is done in the Java editor example inside the **JavaSourceViewerConfiguration**.

```
public IContentAssistant getContentAssistant(ISourceViewer sourceViewer) {

        ContentAssistant assistant= new ContentAssistant();
        assistant.setContentAssistProcessor(new JavaCompletionProcessor(), IDocument.DEFAULT_CONT
        assistant.setContentAssistProcessor(new JavaDocCompletionProcessor(), JavaPartitionScanne

        ...
        return assistant;
}
```

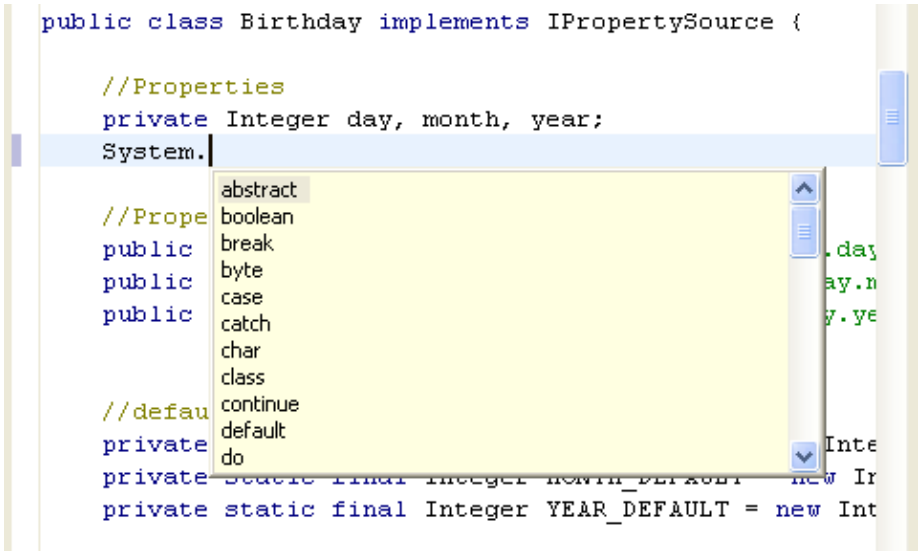
Content assist behavior is defined in the interface **IContentAssistant**. Setting up a content assistant is somewhat similar to setting up syntax highlighting. The assistant should be configured with different phrase completion strategies for different document content types. The completion strategies are implemented using **IContentAssistProcessor**. A processor proposes completions and computes context information for an offset within particular content type.

Content assist processors

Not all content types need to have content assistance. In the Java example editor, content assist processors are provided for the default content type and javadoc, but not for multi−line comments. Let's look at each of these processors.

The **JavaCompletionProcessor** is quite simple. It can only propose keywords as completion candidates. The keywords are defined in a field, `fgProposals`, and these keywords are always proposed as the candidates:

```
public ICompletionProposal[] computeCompletionProposals(ITextViewer viewer, int documentOffset) {
        ICompletionProposal[] result= new ICompletionProposal[fgProposals.length];
        for (int i= 0; i < fgProposals.length; i++) {
                IContextInformation info= new ContextInformation(fgProposals[i], MessageFormat.fo
                result[i]= new CompletionProposal(fgProposals[i], documentOffset, 0, fgProposals[
        }
        return result;
}
```



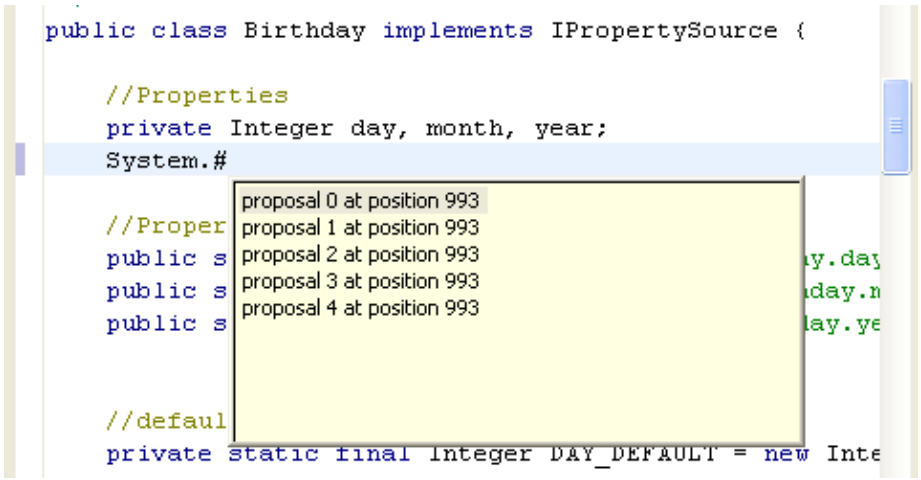
Completion can be triggered by user request or can be automatically triggered when the "(" or "." character is typed:

```
public char[] getCompletionProposalAutoActivationCharacters() {
        return new char[] { '.', '(' };
}
```

In addition to proposing completions, the **JavaCompletionProcessor** defines context information that can be requested by the user. Context information includes a description of the pieces of information available in a given context and the detailed information message.

In the Java editor example, the information is not really contextual. An array containing five similar context information objects is computed for the current offset when the user requests context info. All of these context information objects define a context that contains the five characters in front of the offset and the five after the offset. If any one of these five proposals is selected, the detailed information will appear near the cursor and will stay as long as the cursor is within the context of the five characters around the offset.

```
public IContextInformation[] computeContextInformation(ITextViewer viewer, int documentOffset) {
        IContextInformation[] result= new IContextInformation[5];
        for (int i= 0; i < result.length; i++)
                result[i]= new ContextInformation(
                        MessageFormat.format(JavaEditorMessages.getString("CompletionProcessor.Co
                        MessageFormat.format(JavaEditorMessages.getString("CompletionProcessor.Co
        return result;
}
```



This context information is shown automatically when the "#" character is typed:

```
public char[] getContextInformationAutoActivationCharacters() {
        return new char[] { '#' };
}
```

Content assist configuration

The appearance and behavior of content assist can be configured using **IContentAssistant**. For example, you can configure the auto activation time out, and the orientation and color of information popups.

```
public IContentAssistant getContentAssistant(ISourceViewer sourceViewer) {

        ContentAssistant assistant= new ContentAssistant();
        ...
        assistant.enableAutoActivation(true);
        assistant.setAutoActivationDelay(500);
        assistant.setProposalPopupOrientation(IContentAssistant.PROPOSAL_OVERLAY);
        assistant.setContextInformationPopupOrientation(IContentAssistant.CONTEXT_INFO_ABOVE);
        assistant.setContextInformationPopupBackground(JavaEditorEnvironment.getJavaColorProvider

        return assistant;
}
```

Registering editor actions

The text editor framework provides many utility classes that aid in presenting and updating text and source code. Now we will turn our attention to the workbench in which the editor is but one part. How does the editor interact with other workbench features such as context menus, menu bars, and tool bars?

Editor menu bar actions

To understand how editors register themselves with the workbench and provide actions for the workbench menu bar, see the section discussing org.eclipse.ui.editors. We won't rehash that information here. We'll just

take a quick look at the markup where the Java example editor registers its editor.

```
<extension
      point="org.eclipse.ui.editors">
   <editor
         name="%javaEditorName"
         icon="icons/obj16/java.png"
         extensions="jav"
         contributorClass="org.eclipse.ui.examples.javaeditor.JavaActionContributor"
         class="org.eclipse.ui.examples.javaeditor.JavaEditor"
         id="org.eclipse.ui.JavaEditor">
   </editor>
</extension>
```

Workbench menu bar actions are contributed by the **JavaActionContributor**. It implements actions that are placed in the workbench **Edit** menu and the workbench tool bar.

```
public JavaActionContributor() {
        super();
        fContentAssistPr new RetargetTextEditorAction(JavaEditorMessages.getResourceBundle(), "Con
        ...
        fContentAssistTip= new RetargetTextEditorAction(JavaEditorMessages.getResourceBundle(), "
        ...
        fTogglePresentation= new PresentationAction();
}
```

The first two actions are defined as retargetable text editor actions. The principle is similar to the retargetable actions provided by the workbench. Retargetable text editor actions represent menu entries which the action contributor dynamically binds to corresponding actions provided by the active editor. When the active editor changes, the action to which a retargetable text editor action is bound changes as well. The following snippet shows that the editor action contributor finds the corresponding action by asking the editor for an action of a given id:

```
protected final IAction getAction(ITextEditor editor, String actionId) {
        return (editor == null ? null : editor.getAction(actionId));
}

public void setActiveEditor(IEditorPart part) {
        super.setActiveEditor(part);
        ITextEditor editor= null;
        if (part instanceof ITextEditor)
                editor= (ITextEditor) part;
        fContentAssistProposal.setAction(getAction(editor, "ContentAssistProposal"));
        fContentAssistTip.setAction(getAction(editor, "ContentAssistTip"));
        fTogglePresentation.setEditor(editor);
        fTogglePresentation.update();
}
```

The id must be the same under which the action is registered with the editor as given here for the **JavaTextEditor**. (See also next section.):

```
protected void createActions() {
        super.createActions();

        IAction a= new TextOperationAction(JavaEditorMessages.getResourceBundle(), "ContentAssist
a.setActionDefinitionId(ITextEditorActionDefinitionIds.CONTENT_ASSIST_PROPOSALS);
        setAction("ContentAssistProposal", a);
```

```
        a= new TextOperationAction(JavaEditorMessages.getResourceBundle(), "ContentAssistTip.", t
        a.setActionDefinitionId(ITextEditorActionDefinitionIds.CONTENT_ASSIST_CONTEXT_INFORMATION
        setAction("ContentAssistTip", a);
}
```

The third action in the contributor is a concrete action added to the workbench tool bar. It toggles the state of the editor between showing the highlighted range (as dictated by the Java example's content outliner) and showing the entire file. This action only appears in the tool bar.

Editor context menus

The editor context menus are created and managed in the **AbstractTextEditor** and **TextEditor** framework.

The method **createActions** is used to register actions with the editor. This includes actions appropriate for the editor context menus or any actions contributed in extension definitions. In the Java example editor, only the actions that get bound to the retargetable actions are created. However, the Java example editor also inherits the actions created by **TextEditor** and its superclasses. These actions can be used in the editor context menus.

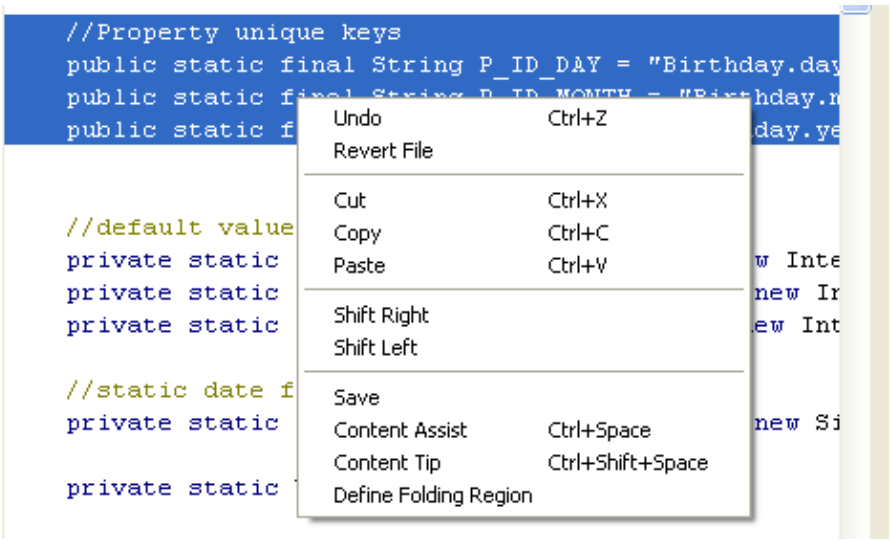
The **TextEditor** method **editorContextMenuAboutToShow** is used in the framework to allow editors to add actions to the context menu for the editing area. You can use a menu path to decide exactly where your action should appear. Valid menu paths inside the editor context menu are defined in the implementation of this method in **AbstractTextEditor**.

There are several ways to add an action to this menu. The first way is by adding an action using only the id under which it is registered with the editor. For example, the **JavaTextEditor** adds its actions for content assistance to the menu when this method is called. Actions will not appear in the menu when no action is registered under the used id.

```
public void editorContextMenuAboutToShow(MenuManager menu) {
        super.editorContextMenuAboutToShow(menu);
        addAction(menu, "ContentAssistProposal");
        addAction(menu, "ContentAssistTip");
}
```

The superclass **TextEditor** adds actions a second way – by specifying a menu group in the context menu for placing the action. In this case the actions (**Shift Left**, **Shift Right**) do appear in the context menu in the group defined by **AbstractTextEditor**.

```
protected void editorContextMenuAboutToShow(IMenuManager menu) {
        super.editorContextMenuAboutToShow(menu);
        addAction(menu, ITextEditorActionConstants.GROUP_EDIT, ITextEditorActionConstants.SHIFT_R
        addAction(menu, ITextEditorActionConstants.GROUP_EDIT, ITextEditorActionConstants.SHIFT_L
}
```



The method **rulerContextMenuAboutToShow** is used in the same way before the ruler's context menu is shown. The implementation of this method in **AbstractTextEditor** defines the groups in which items can be added to the menu.

Menu ids

The editor context and ruler context menus can be assigned ids so that other plug–ins can contribute to these menus in their extensions. The scheme for establishing menu ids is more flexible since the original version of the platform. However, the framework can run in a compatibility mode in order to remain compatible with plug–ins developed for the original version. You can use **AbstractTextEditor.setCompatibilityMode()** to control this behavior. The default setting is true.

1.0 compatible menu ids

When the compatibility mode is true, the ids of the editor and ruler context menus can be set using **AbstractTextEditor** protocol. The methods **setEditorContextMenuId** and **setRulerContextMenuId** can be used for this purpose. Resetting the ids can be useful if you want to prevent inheriting menus that were contributed to superclass menus. For example, the **JavaTextEditor** in the example resets its context menu ids to be Java specific in order to prevent inheriting any generic text contributions from other plug–ins.

```
protected void initializeEditor() {
        super.initializeEditor();
        JavaEditorEnvironment.connect(this);
        setSourceViewerConfiguration(new JavaSourceViewerConfiguration());
setEditorContextMenuId("#JavaEditorContext");
        setRulerContextMenuId("#JavaRulerContext");
}
```

If no id is set anywhere in the concrete hierarchy, the default ids defined by **AbstractTextEditor** will be used.

1.0 non–compatible menu ids

The editor context menu id is always `<editor id>.EditorContext`, where `<editor id>` is the id of the editor . The id of an editor is defined in the xml declaration of the editor. The ruler context menu id is always `<editor id>.RulerContext`.

Other text editor responsibilities

The Java example editor inherits a lot of useful default behavior from **AbstractTextEditor**. The text editing framework handles several other responsibilities that you can customize by overriding methods in **AbstractTextEditor**. Browse the implementation of this class and its subclasses to see how behavior is customized in the framework.

The following are some of the useful framework features that can be configured.

Preference handling

Text editors typically contribute user preferences that control the presentation and behavior of the editor. In the text framework, each text editor instance has an associated preference store that is used for accessing user preferences. This preference store can be set up by your editor, or you can inherit from preference stores already used in the framework.

In the case of the Java example editor, it inherits the preference store initialized by **TextEditor**. This is the preference store defined by the workbench editors plug−in.

```
protected void initializeEditor() {
        ...
        setPreferenceStore(EditorsPlugin.getDefault().getPreferenceStore());
}
```

The editors plug−in preferences can be manipulated in the **General > Editors** and **General > Editors > Text Editors** preference pages.

If you do not want to use the standard workbench text preferences for your editor, you can set a different preference store. This is typically done by overriding **initializeEditor** and setting your own preference store. If you do use your own preference store, you will also need to override the method **handlePreferenceStoreChanged()** which is triggered whenever a preference is updated.

Key bindings

Key binding contexts are useful for establishing a lookup order for key bindings. Having contextual key bindings reduces the chances of different plug−ins contributing conflicting key sequences. By default, the workbench operates in a generic context for working with windows or dialogs. When a text editor becomes active, it is responsible for resetting the context to the text editing context, so that editor specific key bindings will be active.

In the platform text framework, each text editor instance has an associated array of key binding scopes. It is responsible for setting the correct scopes when it becomes active. **AbstractDecoratedTextEditor** defines this scope and takes care of making it active. The scope is assigned in a method that is called from the constructor:

```
protected void initializeKeyBindingScopes() {
        setKeyBindingScopes(new String[] { "org.eclipse.ui.textEditorScope" });
}
```

The argument to the method is an array of ids that have been defined for contexts. If you want your editor to define its own key binding context, then you can override this method in your editor class, or set the scope dynamically using **setKeybindingScopes**.

The context itself must be defined with the corresponding id in the **org.eclipse.ui.contexts** extension point. The following is the definition for the text editing context.

```
<extension
        point="org.eclipse.ui.contexts">
        <context
                name="%context.editingText.name"
                description="%context.editingText.description"
                id="org.eclipse.ui.textEditorScope"
                parentId="org.eclipse.ui.contexts.window">
        </context>
        ...
```

(Note: We use the terms **scope** and **context** interchangeably in this discussion. The method names in the text classes still refer to key binding contexts as scopes. These method names reflect the original implementation of contexts as scopes and use outdated terminology.)

Building a help plug–in

In this example, we assume that a documentation author has already supplied you with the raw documentation in the form of HTML files. The granularity and structure of these files is completely up to the documentation team. Once the documentation is delivered, setting up the plug–in and topics can be done independently.

We start by assuming that the documentation has already been provided in the following tree.

```
html/
   concepts/
      concept1.html
      concept1_1.html
      concept1_2.html
   tasks/
      task1.html
      task2.html
      task3_1.html
      task3_2.html
   ref/
      ref1.html
      ref2.html
```

We will assume that the plug–in name is **com.example.helpexample**.

The first step is to create a plug–in directory, **com.example.helpexample** underneath the platform **plugins** directory. The **doc** sub tree shown above should be copied into the directory.

Documentation plug–ins need a manifest just like code plug–ins. The following markup defines the documentation plug–in.

```
<?xml version="1.0" ?>
<plugin name="Online Help Sample"
   id="com.example.helpexample"
   version="1.0"
   provider-name="MyExample" />
```

Table of contents (toc) files

Now that we have our sample content files we can create a table of contents (**toc**) file. A toc file defines the key entry points into the HTML content files by mapping a topic label to a reference in one of the HTML files.

Applications that are being migrated to the platform can reuse existing documentation by using the toc file to define entry points into that documentation.

A plug−in can have one or more toc files. Our example documentation is organized into three main categories: concepts, tasks and reference. How do we make toc files that represent this structure?

We could make one large toc file, or we could create a separate toc file for each main category of content. This decision should be made according to the way your documentation teams work together. If a different author owns each category, it might be preferable to keep separate toc files for each category. It is not dictated by the platform architecture.

In this example, we will create a toc file for each major content category. For such a small number of files, having separate toc files for each category may not be necessary. We will build this example as if we had many more files or had separate authors who own each content category.

Our files look like this:

toc_Concepts.xml

```
<toc label="Concepts">
   <topic label="Concept1" href="html/concepts/concept1.html">
      <topic label="Concept1_1" href="html/concepts/concept1_1.html"/>
      <topic label="Concept1_2" href="html/concepts/concept1_2.html"/>
   </topic>
</toc>
```

toc_Tasks.xml

```
<toc label="Tasks">
   <topic id="plainTasks" label="Plain Stuff">
      <topic label="Task1" href="html/tasks/task1.html"/>
      <topic label="Task2" href="html/tasks/task2.html"/>
   </topic>
   <topic id="funTasks" label="Fun Stuff" >
      <topic label="Task3_1" href="html/tasks/task3_1.html"/>
      <topic label="Task3_2" href="html/tasks/task3_2.html"/>
   </topic>
</toc>
```

toc_Ref.xml

```
<toc label="Reference">
   <topic label="Ref1" href="html/ref/ref1.html"/>
   <topic label="Ref2" href="html/ref/ref2.html"/>
</toc>
```

A topic can be a simple link to content. For example, "Task1" provides a **label** and an **href** linking to the content. A topic can also be a hierarchical grouping of sub topics with no content of its own. For example,

"Fun Stuff" has only a **label** and sub topics, but no **href** . Topics can do both, too. "Concept1" has an **href** and sub topics.

Help server and file locations

The platform utilizes its own documentation server to provide the actual web pages for your plug−in's documentation. A custom server allows the platform to handle HTML content in a browser independent manner and to provide plug−in aware support. The main difference to you as a plug−in developer is that you have a little more flexibility in the way you structure your files and specify your links.

A documentation plug−in can run from a jar file or unpacked into plug−in directory during installation. A plug−in archive jar is not expanded into a plug−in directory when value of `unpack` attribute of the `plugin` element is specified as true in the feature manifest. In such plug−in, the documentation is compressed in the plug−in's jar, together with other plug−in files.

In plug−ins that run unpacked, the documentation can be delivered in a zip file, avoiding problems that may result when a large number of files are present in a plug−in directory. In our example plug−in, we created a sub−directory called **html**. Alternatively, we could have placed our html files into a zip file called **doc.zip**. This zip file must mimic the file structure underneath the plug−in directory. In our case, it must contain the sub−directory **html** and all the contents underneath **html**.

Note that for plug−ins running from a jar, there is no need for documentation to be additionally contained in doc.zip, and such set−up of doc.zip in an unexploded plug−in jar is not supported by help system

When resolving file names in a plug−in that runs unpacked, the help server looks in the **doc.zip** file for documents before it looks in the plug−in directory itself. When used as a link, the argument in an **href** is assumed to be relative to the current plug−in. Consider the following link:

```
<topic label="Ref1" href="html/ref/ref1.html"/>
```

The help plug−in will look for this file as follows:

- look in **doc.zip** for the file **/html/ref/ref1.html**
- look for the file **ref1.html** in the **/html/ref** sub−directory structure underneath the plug−in directory.

A fully qualified link can be used to refer to any content on the web.

```
<topic label="Ref1" href="http://www.example.com/myReference.html"/>
```

National language and translated documentation

The platform help system uses the same national language directory lookup scheme used by the rest of the platform for finding translated files. (See Locale specific files for an explanation of this directory structure.) If you are using a **doc.zip** file, you should produce a **doc.zip** file for **each** locale and place it inside the correct locale directory. (You should not replicate the **nl** locale directory structure inside the doc.zip file.)

In addition to locale specific directories, help system checks windowing system and operating system directories when locating help resources. Look−up is performed in the following order: ***ws***, ***os***, ***nl*** subdirectories, then the root of the plug−in, until the resource is located. Documents, and other resource, like images which differ between system, should be placed under ws or os directories for specific platform.

Cross plug−in referencing

The **href** argument can also refer to content from another plug−in. This is done by using a special cross plug−in referencing notation that is resolved by the help server:

```
<topic label="Ref1" href="../"another_plugin_id"/ref/ref1.html"/>
```

For example, you could link to this chapter of the programmer's guide using the following topic:

```
<topic label="Help Chapter in Platform Doc" href="../org.eclipse.platform.doc.isv/guide/help.h
```

Note: When referencing content from another plug−in, be sure to use the plug−in's **id**, as declared in its **plugin.xml** file, not its directory name. While these are often the same in practice, it's important to check that you are using the id and not the directory name.

Referencing the Product Plug−in.

Branding information is often placed in a plug−in defining a product as explained in Defining a Product. Help resources in the product plug−in can be referenced from the table of contents or topics using special identifier "PRODUCT_PLUGIN" for the plug−in ID. For example,

```
href="../PRODUCT_PLUGIN/book.css"
```

refers to a style sheet residing in the plug−in for the currently running product.

Completing the plug–in manifest

We started this example by creating our plug–in and document files. Next we created toc files to describe the organization of our content. The remaining piece of work is to pull everything together into a master toc and update our **plugin.xml** to actually contribute the master toc.

We start by creating a **toc.xml** to contribute the three tocs we created initially. Instead of providing an **href** for each topic, we use the **link** attribute to refer to our existing toc files.

```
<toc label="Online Help Sample" topic="html/book.html">
        <topic label="Concepts">
                <link toc="toc_Concepts.xml" />
        </topic>
        <topic label="Tasks">
                <link toc="toc_Tasks.xml" />
        </topic>
        <topic label="Reference">
                <link toc="toc_Ref.xml" />
        </topic>
</toc>
```

Then we update the **plugin.xml** to contribute our master toc:

```
   <extension point="org.eclipse.help.toc">
      <toc file="toc.xml" primary="true" />
   </extension>
```

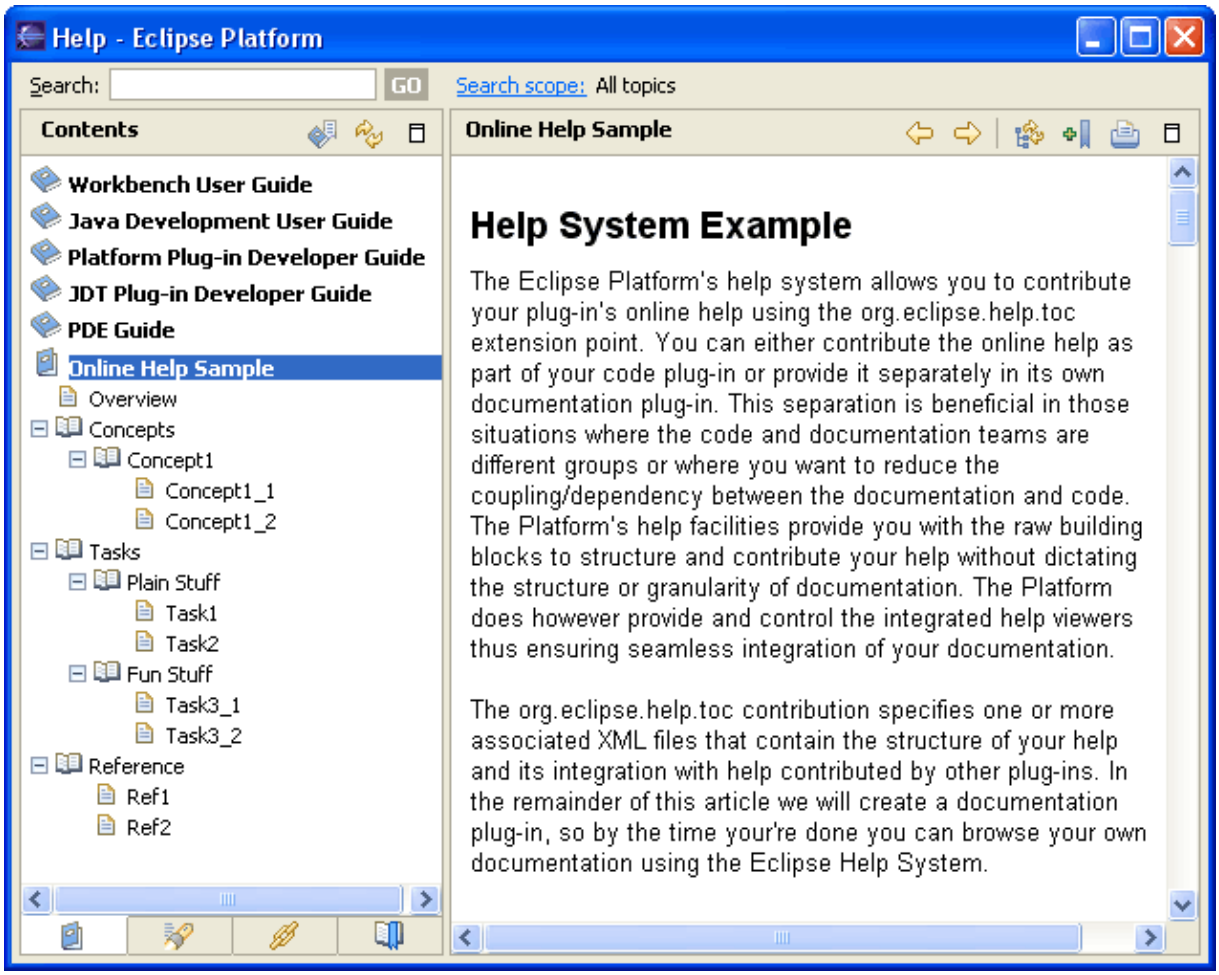
Note the use of the **primary** attribute. Setting this attribute to true indicates that the toc should always appear in the navigation, even if it is not referenced by any other toc. This way, our "master" toc is always guaranteed to show up in the topics list. It appears at the top level list of books since no other toc references it.
Note: If more files were associated with this toc but not present in the navigation, but just linked from other topics, then to have those topics available to the search engine we would have to use the **extradir** attribute in the toc.

Finally, we contribute our individual toc files.

```
   <extension point="org.eclipse.help.toc">
       <toc file="toc_Concepts.xml" />
       <toc file="toc_Tasks.xml" />
       <toc file="toc_Reference.xml" />
   </extension>
```

These toc files will not appear in the top level list of books because we did not set the **primary** attribute. Toc files that are not designated as primary will only appear in the documentation web if they are referred to from some toc that is a primary toc or is linked in by a primary toc.

That's it. If you copy your plug–in directory to the platform's **plugins** directory, start the platform, and choose **Help–>Help Contents**, you should see your example appear in the list of books. If you click on the "Online Help Sample", you'll see your toc structure:


Help - Eclipse Platform
Search:
GO
Search scope: All topics
Contents
Online Help Sample
Workbench User Guide
Java Development User Guide
Platform Plug-in Developer Guide
JDT Plug-in Developer Guide
PDE Guide
Online Help Sample
Overview
Concepts
Concept1
Concept1_1
Concept1_2
Tasks
Plain Stuff
Task1
Task2
Fun Stuff
Task3_1
Task3_2
Reference
Ref1
Ref2
Help System Example
The Eclipse Platform's help system allows you to contribute your plug-in's online help using the org.eclipse.help.toc extension point. You can either contribute the online help as part of your code plug-in or provide it separately in its own documentation plug-in. This separation is beneficial in those situations where the code and documentation teams are different groups or where you want to reduce the coupling/dependency between the documentation and code. The Platform's help facilities provide you with the raw building blocks to structure and contribute your help without dictating the structure or granularity of documentation. The Platform does however provide and control the integrated help viewers thus ensuring seamless integration of your documentation.
The org.eclipse.help.toc contribution specifies one or more associated XML files that contain the structure of your help and its integration with help contributed by other plug-ins. In the remainder of this article we will create a documentation plug-in, so by the time your're done you can browse your own documentation using the Eclipse Help System.

Building nested documentation structures

As plug−ins contribute function to the platform, it's common to add documentation that describes the new function. How can this documentation be structured so that the user sees a cohesive and complete set of documentation instead of many individual contributions? The table of contents definition provides mechanisms for building documentation in both a top−down and bottom−up fashion.

Top−down nesting

Top−down nesting refers to the technique of defining a master table of contents which refers to all other included tocs. Top−down nesting is a convenient method for breaking up known content into smaller pieces. With top−down nesting, the **link** attribute is used in the table of contents definition to refer to linked tocs rather than providing an **href**.

```
<toc label="Online Help Sample" topic="html/book.html">
        <topic label="Concepts">
                <link toc="toc_Concepts.xml" />
        </topic>
        <topic label="Tasks">
                <link toc="toc_Tasks.xml" />
        </topic>
        <topic label="Reference">
                <link toc="toc_Ref.xml" />
        </topic>
</toc>
```

The basic structure stays the same (Concepts, Tasks, Reference), but the individual tocs are free to evolve. They in turn might link to other sub−tocs.

Bottom−up composition

Bottom−up composition is more flexible in that it lets new plug−ins decide where the documentation should exist in the toc structure. Bottom−up composition is accomplished using **anchor** attributes. A toc defines named anchor points where other plug−ins can contribute documentation. In our example, we could add anchors so that plug−ins can contribute additional material between the concepts, tasks, and reference sections.

```
<toc label="Online Help Sample" topic="html/book.html">
        <topic label="Concepts">
                <link toc="toc_Concepts.xml" />
 anchor id="postConcepts" />
        </topic>
        <topic label="Tasks">
                <link toc="toc_Tasks.xml" />
 anchor id="postTasks" />
        </topic>
        <topic label="Reference">
                <link toc="toc_Ref.xml" />
 anchor id="postReference" />
        </topic>
</toc>
```

Other plug−ins can then contribute to the anchor from their plug−in. This is done using the **link_to** attribute when defining a toc.

```
<toc link_to="../com.example.helpexample/toc.xml#postConcepts" label="Late breaking info about co
        <topic>
                ...
        </topic>
</toc>
```

Dynamic topics

In addition to static HTML files present in doc.zip or file system under plug−in directory, help can display documents that are dynamically generated by the documentation plug−in. Such plug−in needs to contain Java code capable of producing contents that would otherwise be read from static files by help system. If a class implementing org.eclipse.help.IHelpContentProducer is contributed by a plug−in to help system using org.eclipse.help.contentProducer extension point, the help system will call getInputStream for every document accessed from this plug−in. If that results in not null InputStream, it will be sent to the browser for displaying. If IHelpContentProdcuer returns null, help system will default to searching doc.zip and plug−in directory for a document.

Content producer example

For example, an implementation of IHelpConentProducer as follows:

```
package com.my.company.doc;
public class DynamicTopics implements IHelpContentProducer {
        public InputStream getInputStream(
                String pluginID,
                String name,
                Locale locale) {
                if (name.indexOf("dynamic") >= 0)
return new ByteArrayInputStream(
                                ("<html><body>Content generated "
                                        + new Date().toString()
                                        + ".</body></html>")
                                        .getBytes());
                else
       null;            return
        }
}
```

identified by extension in plugin.xml file as:

```
<extension point="org.eclipse.help.contentProducer"
    name="dynamicTopics"
    id="my.company.doc.dynamicTopics">
    <contentProducer producer="com.my.company.doc.DynamicTopics" />
</extension>
```

will produce an HTML document content for all document references that have a word "dynamic" as part of the path or file name.

Constraints

A plug−in has a complete freedom of the method of method or underlying framework that is used to produce the content. It should however ensure that content is generated in correct language and encoded accordingly that a browser can display it. A locale used by the user is provided to the getInputStream method. If it is incapable of providing translatable content, it should default to content for the default locale of the platform.

Context−sensitive help

A focused set of help topics that is related to the current context can be shown to users on demand using **context−sensitive help**. This form of user assistance is delivered to users when a platform−specific trigger is activated (e.g. F1 key on Windows, Ctrl+F1 on GTK, Help key on Carbon). Until Eclipse 3.1, context−sensitive help was presented in infopop windows. Since 3.1, a new Help view is the preferred way to deliver context−sensitive information to the user.

Context−sensitive help can be associated with widgets statically using context IDs, or dynamically using context providers. The new help view also adds dynamic help capability by searching help for topics relevant for the current context.

Declaring a context id

The **setHelp** method in **org.eclipse.ui.help.IWorkbenchHelpSystem** is used to associate a context id with a `Control`, `IAction`, `Menu`, or `MenuItem`. The context id should be fully qualified with the plug−in id. For example, the following snippet associates the id "com.example.helpexample.panic_button" with a button in the application.

```
PlatformUI.getWorkbench().getHelpSystem().setHelp(myButton, com.example.helpexample.panic_button)
```

The following UI controls cannot have context ids (and therefore cannot have context−sensitive help):

- Toolbar buttons (ToolItem)
- CTabItem
- TabItem
- TableColumn
- TableItem
- TableTreeItem
- TreeItem

Widgets that do not get focus should not be assigned context ids, since they will never trigger a context−sensitive help.

Describing and packaging context−sensitive help content

Content−sensitive help is described by associating the context id declared in the UI code with a description and list of links to related topics in the online help. These associations are made inside an XML file. You can create any number of XML files containing context help associations for each plug−in. The description and links for each context id is made inside **<context>** elements in the XML file. Each context element can have an optional **<description>** element which is used to describe the UI object and any number of **<topic>** elements which link to the on−line documentation.

Since 3.1, context elements can optionally override the default title used to present the context help information in the Help view.

```
<contexts>
        <context id="panic_button" title="Panic Button Title">
                <description>This is the panic button.</description>
                <topic href="tasks/pushing_the_panic_button.htm" label="Pushing the panic button"
                <topic href="reference/panic_button.htm" label="Panic Button Reference"/>
        </context>
        ...
</contexts>
```

Once the contexts have been described in the XML file (or files), you are ready to refer to the context files in your plug−in manifest. Note that the context id is not fully qualified above. This is allowed, as long as the context file is contributed in the manifest of the plug−in that defined the context id. In other words, the context id is resolved to the id of the plug−in that contributed the XML file.

A plug−in contributes context files using the **org.eclipse.help.contexts** extension point.

```
<extension point="org.eclipse.help.contexts">
   <contexts name="myContextHelp.xml"  />
</extension>
```

You can reference context files from other plug−ins by including the **plugin** attribute. This allows you to group all of your documentation, including content−sensitive help, in one plug−in, and refer to it from the UI code plug−in or some other related plug−in.

```
<extension point="org.eclipse.help.contexts">
   <contexts name="myContextHelp.xml" plugin="com.example.helpExample"  />
</extension>
```

As you can see, you have a lot of flexibility in organizing your contexts into one or more files contained in one or more plug−ins. The main consideration is that the context ids in the files resolve correctly. If you do not fully qualify a context id, then you must contribute the context XML files in the plug−in that declared the context ids. If you use fully qualified context ids in your context XML file, then you have complete flexibility in the location of your XML files and which plug−in contributes the contexts.

Context−sensitive help from multiple plug−ins

Another level of flexibility is the ability to contribute context−sensitive help for the same context id from different plug−ins. This is useful, for example, if there are different sets of documentation plug−ins that may or may not be installed in a user's configuration. This allows each documentation plug−in to declare its

contexts independently. The end user will see the merged context−sensitive help content for all plug−ins that contributed contexts for the widget's id.

Note that the fully qualified context id for the widget must be used, since none of the documentation plug−ins declared the context id. When multiple plug−ins contribute context−sensitive help for the same context ID, the content defined in the plug−in that declared the context (the UI plug−in) is shown first. Additional descriptions and links are appended in no guaranteed order.

Dynamic creation of context help

In addition to statically associating widgets and context Ids, it is possible to provide this information dynamically for a more dynamic context−sensitive help capability. Help system uses context Ids to locate the matching org.eclipse.help.IContext object. The new Help view tracks activation of the workbench parts (views and editors) and checks if they adapt to org.eclipse.help.IContextProvider interface. If they do, the view will use the context provider to locate the `IContext` object and get the required information from it. This object can be cached or created on the fly.

Workbench parts that want to create the context object dynamically should adapt to the `IContextProvider.class` object as a key:

```
public Object getAdapter(Class key) {
        if (key.equals(IContextProvider.class)) {
                return new MyContextProvider();
        }
        return super.getAdapter(key);
}
```

The context provider interface requires implementation of three methods:

```
public class MyContextProvider implements IContextProvider {
        int getContextChangeMask() {
                return NONE;
        }
        IContext getContext(Object target) {
                return myContext;
        }
        String getSearchExpression(Object target) {
                return null;
        }
}
```

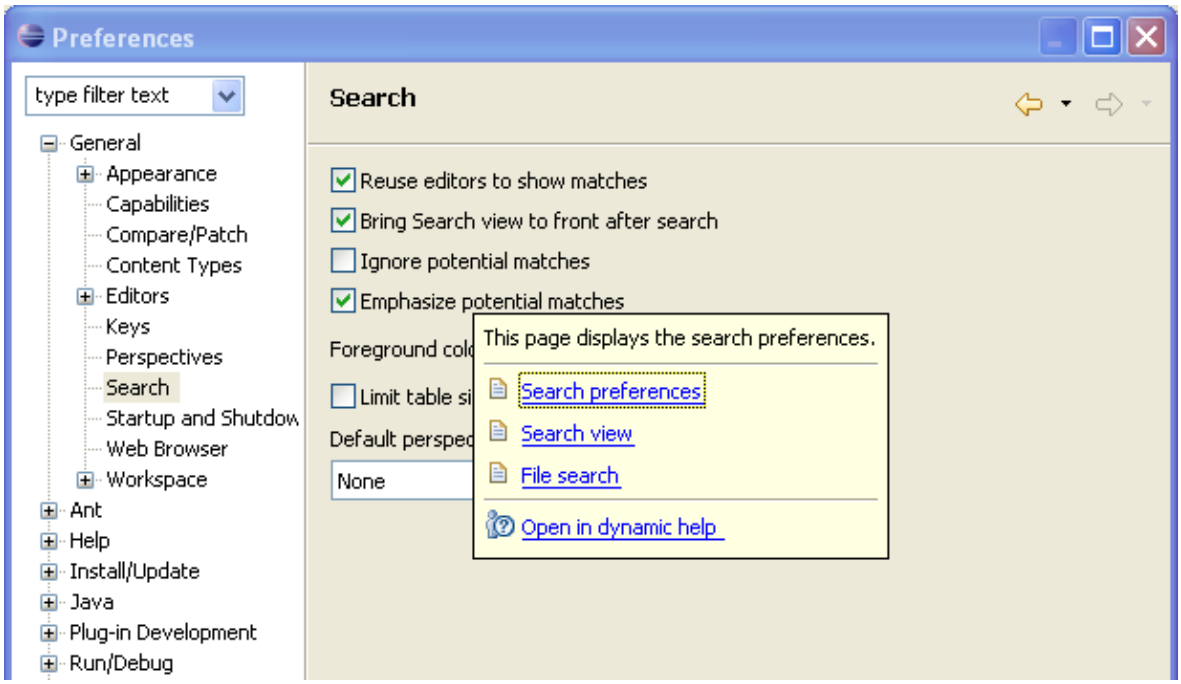
If context change mask returns `NONE`, context object will need to be provided when the workbench part is activated. If `SELECTION` is returned, you will need to provide context object that is sensitive to the current selection in the part. Each time part selection provider fires a selection change event, the context provider will be asked to provide context object.

Optionally, search expression for the dynamic help can be provided. Otherwise, a combination of the part name and perspective name will be used with good results in most cases.

Infopops

An infopop is a small window associated with a particular SWT widget in a plug–in's user interface. It displays context–sensitive help and links to related help topics for the widget. It is activated when the user puts focus on the widget and presses the `F1` key (`Ctrl+F1` on GTK, and `Help` key on Carbon).

Since Eclipse 3.1, the preferred way to show the context help is in the help window. However, it is still possible to configure context help presentation to use infopops in the Help preferences. For example, here is an infopop that has been defined on the General/Search page:



Since Eclipse 3.1, infopops have a link to show the currently displayed context help in the new Help view.

Information Search

Since Eclipse 3.1, search in the workbench has been partitioned into two major categories: development artifact search and information search. The former is handled by the search dialog opened from the Search menu. The later is available from Help>Search menu item which opens the new Help view into the Search page.

The new information search facility uses multiple search engines run in parallel using the same search expression. Eclipse provides a number of preconfigured search engines. New engines can be added programmatically or by the user. All the results are collated in the Search page directly.

Federated search engine types

The new federated information search in Help system uses the notion of **search engine types** and **search engines**. An engine type is a meta−engine from which a number of concrete search engines can be created by parameterization.

New engine types are contributed through the org.eclipse.help.ui.searchEngine:

```
<extension point="org.eclipse.help.ui.searchEngine">
        <engineType
                scopeFactory="com.example.xyz.XYZScopeFactory"
                label="XYZ Search"
                class="com.example.xyz.search.XYZSearch"
                icon="icons/etool16/xyzsearch.gif"
                pageClass="com.example.xyz.search.XYZSearchPage"
                id="com.example.xyz.XYZSearch">
                <description>
                        Instances of XYZ Search search the XYZ site.
                </description>
        </engineType>
```

This extension point is used to plug in search participants in the information search. Each search engine can be configured individually. When search is initiated, each search engine is executed as a background job, and the results are collated in the help view immediately under the query.

Search engines defined here will not automatically show up as federated search participants until engine product binding is established, unless `productId` attribute is left undefined. For engines that define it, only those bound to a particular product will show up when that product is running.

Search engines can simply compose a URL and provide only one hit containing that URL as `href`. Popular search engines for which API support requires license can be plugged in like this. On the other end of the spectrum, search engines can communicate with the server and receive individual hits with information like label, href, short description, score etc. Local help engine can produce hits this way.

Regardless of the search mechanism, engines can provide various search scope settings using JFace preference pages. These pages are shown when 'Advanced Settings' link is followed from the Help view. In addition to root preference pages defined with the engine, additional preference sub−pages can be plugged in for more advanced settings.

Scope settings are loaded and stored using `IPreferenceStore` objects. Scope settings for all engines are grouped together under a named **scope set**. When first opened, default scope set ('Default') is created, but users can define more scope sets and flip between them.

Since federated search support is part of `org.eclipse.help.base` plug−in, a factory is needed to create search scope objects from the data in the preference store. Clients that plug in scope preference pages are required to plug in scope factories as well.

Engines defined in this extension point do not show up in the UI by default. What is shown there is a concrete **instance** of a search engine that can be individually modified. Products can pre−configure the help system with a number of instances of the registered engine types, possibly parameterized to perform in a desired way. In addition, users can add their own instances of the registered engines and configure them to their liking:

```
<engine
        enabled="true"
        engineTypeId="com.example.xyz.search.XYZSearch"
        id="com.example.xyz.XYZSearch"
        label="XYZ Search">
</engine>
<engine
        enabled="true"
        engineTypeId="org.eclipse.help.ui.web"
        id="org.eclipse.sdk.Eclipse"
        label="%search.Eclipse.label">
        <description>
                %search.Eclipse.desc
        </description>
        <param
                name="url"
                value="http://eclipse.org/search/search.cgi?q={expression}&amp;ul=&amp;ps=
        </param>
</engine>
```

Active help

Active help is the ability to invoke Eclipse code from on−line documentation. It is implemented by including some JavaScript in your documentation that describes a class that should be run inside the Eclipse platform.

For example, instead of writing, "Go to the Window Menu and open the message dialog," your on−line help can include a link that will open your application's message dialog for the user. Active help links look like hyperlinks in the on−line help.

Below is an active help link that opens a message dialog in the workbench. We will take a look at how to make this work.

Click here for a Message.

Writing the help action

The interface **ILiveHelpAction** is used to build an active help action.

It is straightforward to implement an **ILiveHelpAction.** You must implement two methods.

- **run() –** This method is called to run the live help action. This method will be called by the help system from a non−UI thread, so UI access must be wrapped in a **Display.syncExec()** method.
- **setInitializationString(String) –** This method is called to initialize your action with a String data parameter you can specify in the HTML which runs the JavaScript **liveAction**. If you don't need initialization data, you can just implement this method to do nothing. This method is called before **run()**.

Here is a simple implementation of a live help action that opens a message dialog. We don't need any information from the JavaScript, so the initialization data is ignored.

```
package org.eclipse.platform.doc.isv.activeHelp;

import org.eclipse.help.ILiveHelpAction;
import org.eclipse.jface.dialogs.MessageDialog;
import org.eclipse.swt.widgets.*;
import org.eclipse.ui.*;
/**
 * Sample Active Help action.
 */
public class ActiveHelpOpenDialogAction implements ILiveHelpAction {

     public void setInitializationString(String data) {
          // ignore the data.  We do not use any javascript parameters.
     }

     public void run() {
          // Active help does not run on the UI thread, so we must use syncExec
          Display.getDefault().syncExec(new Runnable() {
               public void run() {
                    IWorkbenchWindow window =
                         PlatformUI.getWorkbench().getActiveWorkbenchWindow();
                    if (window != null) {
                         // Bring the Workbench window to the top of other windows;
                         // On some Windows systems, it will only flash the Workbench
                         // icon on the task bar
                         Shell shell = window.getShell();
                         shell.setMinimized(false);
                         shell.forceActive();
                         // Open a message dialog
                         MessageDialog.openInformation(
                              window.getShell(),
                              "Hello World.",
                              "Hello World.");
                    }
               }
          });
     }
}
```

Invoking the action from HTML

To include active help links in your documentation, you must first declare the use of the active help JavaScript in the **HEAD** tag of your HTML:

```
<script language="JavaScript" src="../../org.eclipse.help/livehelp.js> </script>
```

The live help JavaScript is located in the **org.eclipse.help** plug−in. You can refer to it using the help system's cross plug−in referencing technique.

In this example, we have to navigate up two levels in the directory structure. The document with the active help link is located in the sub−directory **guide** underneath our plug−in directory. So we must navigate up one directory to reach our plug−in's root, and another level to reach the "virtual" location of all plug−ins. Then we can refer to the **org.eclipse.help** plug−in.

In the body of your documentation, you invoke the liveAction script.

```
<a href='javascript:liveAction(
        "org.eclipse.platform.doc.isv",
        "org.eclipse.platform.doc.isv.activeHelp.ActiveHelpOpenDialogAction",
        "")'>Click here for a Message.</a>
```

The parameters for **liveAction** are

- the ID of the plug−in that contains the action
- the name of the class that implements the action
- the String that will be passed to the live help action using **setInitializationString**. We don't need to pass any information from the HTML page, so we just pass an empty string.

Tips for debugging active help

The code and markup that triggered our active help link looks pretty straightforward. But what do you do if your active help link doesn't seem to work?

Test your action ahead of time

If your action implementation is fairly involved, you should invoke the action yourself with some test code inside Eclipse. This way, you'll know that the action is error−free before invoking it from the JavaScript.

Ensure the JavaScript is running

You can modify "plugins/org.eclipse.help_2.1.0/**liveHelp.js**" to include a call to the **alert** function as the first statement in the **liveAction** function:

```
function liveAction(pluginId, className, argument)
{
        alert("liveAction called");
        ...
```

The **alert** function opens a warning dialog in the browser and can be used to verify that the **liveAction** was properly invoked in your HTML. If you don't see a warning dialog when you click on your help link, then you have a problem in the HTML markup.

Debug the active help action

Once you know that the JavaScript is running, you can debug your action from inside Eclipse. To do this, you can set a breakpoint in your help action class and start up a self−hosted Eclipse instance. You must test your active help with the Help browser from the newly launched Eclipse instance, not from your host instance, since the JavaScript from your help HTML calls a servlet on the Eclipse help server that launched the browser.

If nothing happens after you've set up the breakpoint and clicked on the active help link, it's likely that your plug−in and active help class were not correctly specified in the JavaScript.

Once you've managed to stop at the breakpoint in your action, you can debug the action as you would any other Java code.

Make sure your UI code is wrapped in Display.syncExec

A common runtime problem is improperly accessing UI code from the thread that invokes the active help. If your live help action came from code that ran originally in a UI thread, it will need to be modified to handle the fact that it is running from a non−UI thread.

```
public void run() {
                // Active help does not run on the UI thread, so we must use syncExec
                Display.getDefault().syncExec(new Runnable() {
                                                        public void run() {
                                                                //do the UI work in here;
                                                        }
                });
        }
```

Search support

If your plug−in defines and manipulates its own resource types, you may have special requirements for searching a resource. For example, the Java IDE plug−in implements a search engine specialized for Java files.

The **search** plug−in allows you to add a specialized page describing your search to the workbench search dialog. This allows you to obtain any specialized information needed from the user and perform a search using your plug−in's internal model.

You should also provide a specialized class for displaying the search results. Abstract implementations of a search result page are provided to give you a head start.

These services are contributed using search plug−in extension points.

Contributing a search page

When the user selects a resource and chooses the search command, the search plug−in launches a dialog containing pages for different kinds of content searches. These pages are contributed using the **org.eclipse.search.searchPages** extension point.

The markup for contributing a search page is straightforward. The following example is the JDT plug−in's contribution of the Java search page:

```
<extension point="org.eclipse.search.searchPages">
   <page id="org.eclipse.jdt.ui.JavaSearchPage"
                icon="icons/full/obj16/jsearch_obj.png"
                label="%JavaSearchPage.label"
                sizeHint="460,160"
                extensions="java:90, jav:90"
                showScopeSection="true"
                canSearchEnclosingProjects="true"
class="org.eclipse.jdt.internal.ui.search.JavaSearchPage">
        </page>
</extension>
```

The **class** that implements the search page must be specified. This class must implement the **ISearchPage** nterface and typically extends **DialogPage**. The **label** and **icon** that can be used to describe the search in the search dialog are also specified. Additional parameters control the size of the page and the location of the page within the search dialog.

The **extensions** attribute specifies the resources on which the search page can operate. It is formatted as a comma separated list of file extensions. Each file extension should be followed by a numeric weight value, where 0 is the lowest weight, separated by a colon. The weight value is a relative value used to allow the search infrastructure to find the search page most appropriate for a given resource.

If a search page can search all possible resources then "*" should be used.

Implementing the search page

The protocol for **ISearchPage** is simple. Your search page must implement `performAction()` which is called when the **Search** button is pressed. Of course, your particular search implementation depends on your plug−in's function, but it is typical to open a results viewer in this method using the **NewSearchUI** method `activateSearchResultView()`.

Your plug−in is responsible for showing its results in the search result view.

Contributing a search result page

You can contribute a customized search results page using the **org.eclipse.search.searchResultViewPages** extension point.

When contributing a search result view page, you specify the class of search result that the page should be used for, and the name of the class that implements the page. The following example is the JDT plug−in's contribution of the Java search results page:

```
<extension
        id="JavaSearchResultPage"
        point="org.eclipse.search.searchResultViewPages">
        <viewPage
                id="org.eclipse.jdt.ui.JavaSearchResultPage"
                searchResultClass="org.eclipse.jdt.internal.ui.search.JavaSearchResult"
                class="org.eclipse.jdt.internal.ui.search.JavaSearchResultPage">
        </viewPage>
</extension>
```

The **class** must implement the **ISearchResultPage** interface and often extends **AbstractTextSearchViewPage**.

Compare support

If your plug−in defines and manipulates its own resource types, you may have special requirements for comparing resources. Resources are often compared when working with local history or with files from a repository. The **compare** plug−in supports merging of multiple content streams and the implementation of advanced compare views. Services provided by this plug−in include:

- Interfaces for merging multiple text streams into a single output stream
- Two and three−way compare and merge components for hierarchical structures inferred from text
- Differencing engines for hierarchical structures and character ranges in text

These services are used in the platform to assist with user tasks such as integrating patch files and comparing/merging the workspace with local history.

Compare viewers

All compare viewers are standard JFace viewers that expect an input object implementing the **ICompareInput** interface.

Compare viewers are said to be content−oriented if they compare flat inputs such as text or images and structure−oriented if they compare hierarchically structured input elements.

Implementing a content viewer

The compare plug−in allows you to supply specialized viewers for viewing and merging content differences between unstructured elements.

Simple content viewers

A **content viewer** is used in places where only a single input is available and therefore no compare is necessary. A typical example for this is the "**Restore from Local History**" function. The **org.eclipse.compare.contentViewers** extension point allows you to define a specialized content viewer that does not compare its inputs.

```
<extension
        point="org.eclipse.compare.contentViewers">
        <viewer
                extensions="java,java2"
                class="org.eclipse.jdt.internal.ui.compare.JavaTextViewerCreator"
                id="org.eclipse.jdt.internal.ui.compare.JavaTextViewerCreator">
        </viewer>
        <contentTypeBinding
                contentTypeId="org.eclipse.jdt.core.javaSource"
                contentViewerId="org.eclipse.jdt.internal.ui.compare.JavaTextViewerCreator">
        </contentTypeBinding>
</extension>
```

Specialized viewers contributed by your plug−in are designated in the **viewer** element. You must specify the **id** of the viewer and the **class** that creates it. You may also specify any file **extensions** for which the content viewer should be used.

You may also use the **contentTypeBinding** element to associate a content type with a content viewer.

Content merge viewers

A **content merge viewer** performs a two−way or three−way compare of its inputs and presents the result side−by−side or in any other suitable way. The viewer lets the user merge between the inputs. Content merge viewers are common for text or images.

If the standard merge viewers are not appropriate for your plug−in's function, you may choose to implement your own content merge viewer. Your content merge viewer should be registered with the platform using the **org.eclipse.compare.contentMergeViewers** extension point. The following markup shows the definition of specialized content merge viewers for viewing Java files and properties files in the Java IDE:

```
<extension
        point="org.eclipse.compare.contentMergeViewers">
        <viewer
                extensions="java,java2"
                class="org.eclipse.jdt.internal.ui.compare.JavaContentViewerCreator"
                id="org.eclipse.jdt.internal.ui.compare.JavaContentViewerCreator">
        </viewer>
        <contentTypeBinding
                contentTypeId="org.eclipse.jdt.core.javaProperties"
                contentMergeViewerId="org.eclipse.compare.TextMergeViewerCreator">
        </contentTypeBinding>
        <contentTypeBinding
                contentTypeId="org.eclipse.jdt.core.javaSource"
                contentMergeViewerId="org.eclipse.jdt.internal.ui.compare.JavaContentViewerCreato
        </contentTypeBinding>
</extension>
```

Similar to content viewers, specialized merge viewers contributed by your plug−in are designated in the **viewer** element. You must specify the **id** of the viewer and the **class** that creates it. You may also specify any file **extensions** for which the content merge viewer should be used.

Also similar to content viewers, you can use **contentTypeBinding** to associate a content type with a merge viewer. The JDT plug−in binds content merge viewers to two different content types: Java source and Java properties files.

ContentMergeViewer is an abstract compare and merge viewer with two side−by−side content areas and an optional content area for a common ancestor (for three−way compare). Because the implementation makes no assumptions about the content type, the subclass is responsible for dealing with the specific content type.

ImageMergeViewer in **org.eclipse.compare.internal** shows how to implement a simple merge viewer for images using a **ContentMergeViewer**. A **ContentMergeViewer** accesses its model by means of a content provider which must implement the **IMergeViewerContentProvider** interface.

Text merging

If your viewer uses text, additional classes that compare and merge text content can be used.

TextMergeViewer is the concrete subclass of **ContentMergeViewer** used for comparing and merging text content. A text merge viewer uses the **RangeDifferencer** to perform a textual, line−by−line comparison of two (or three) input documents.

For text lines that differ, the **TextMergeViewer** uses an **ITokenComparator** to find the longest sequences of matching and non−matching tokens. The **TextMergeViewer**'s default token compare works on characters separated by white space. If a different strategy is needed (for example, Java tokens in a Java−aware merge viewer), clients can create their own token comparators by implementing the **ITokenComparator** interface.

TextMergeViewer works on whole documents and on sub ranges of documents. For partial documents, the viewer's input must be an **IDocumentRange** instead of an **IDocument**.

Range differencing

RangeDifferencer finds the longest sequences of matching and non−matching comparable entities in text content. Its implementation is based on an objectified version of the algorithm described in: *A File Comparison Program,* by Webb Miller and Eugene W. Myers, Software Practice and Experience, Vol. 15, Nov. 1985. Clients must supply an input to the differencer that implements the **IRangeComparator** interface. **IRangeComparator** breaks the input data into a sequence of entities and provides a method for comparing one entity with the entity in another **IRangeComparator**.

For example, to compare two text documents and find the longest common sequences of matching and non−matching lines, the implementation of **IRangeComparator** must break the document into lines and provide a method for testing whether two lines are considered equal. See **org.eclipse.compare.internal.DocLineComparator** for an example of how this can be done.

The differencer returns the differences among these sequences as an array of **RangeDifference** objects. Every single **RangeDifference** describes the kind of difference (no change, change, addition, deletion) and the corresponding ranges of the underlying comparable entities in the two or three inputs.

Implementing a structure viewer

A structure merge viewer performs a two−way or three−way compare of its inputs, presents the result in a hierarchical view, and lets the user merge between the inputs. Structure merge viewers are common for workspace resources or the members of an archive file.

Tree−like structure viewers

Because the implementation of many structure compare viewers is based on a tree, the compare plug−in provides a generic tree−based **StructureDiffViewer**. Your plug−in is responsible for supplying a **structure creator** that breaks a single input object into a hierarchical structure. The **StructureDiffViewer** performs the compare on the resulting structure and displays the result as a tree.

You designate a structure creator for your plug−in using the **org.eclipse.compare.structureCreators** extension. Much like content viewers, a structure creator can be specified for a set of file **extensions**, or a **contentTypeBinding** can be used to associate a content type with a particular structure creator. We won't review the markup here since it's so similar to content viewers. The JDT plug−in defines several contributions for **org.eclipse.compare.structureCreators**.

Other hierarchical structure viewers

In some cases, the tree−based **StructureDiffViewer** may not be appropriate for your plug−in. The **org.eclipse.compare.structureMergeViewers** extension point allows you to define your own implementation for a structure merge viewer. A structure merge viewer can be specified for file **extensions**, or a

contentTypeBinding can be used to associate a content type with a particular structure merge viewer. See the JDT plug−in for examples of **org.eclipse.compare.structureMergeViewers** contributions.

The search plug−in provides several utility classes to help you implement a search viewer.

Differencer

Differencer is a differencing engine for hierarchically structured data. It takes two or three inputs and performs a two−way or three−way compare on them.

If the input elements to the differencing engine implement the **IStructureComparator** interface, the engine recursively applies itself to the children of the input element. Leaf elements must implement the **IStreamContentAccessor** interface so that the differencer can perform a byte wise comparison on their contents.

There are several good examples of differencers included in the platform implementation:

- **ResourceNode** implements both interfaces (and more) for platform workspace resources (**org.eclipse.core.resources.IResource**).
- **DocumentRangeNode** is used to compare hierarchical structures that are superimposed on a document. Nodes and leaves correspond to ranges in a document (**IDocumentRange**). Typically, **DocumentRangeNode**s are created while parsing a document and they represent the semantic entities of the document (e.g. a Java class, method or field). The two subclasses **JavaNode** (in **org.eclipse.jdt.internal.ui.compare**) and **PropertyNode** (in **org.eclipse.jdt.internal.ui.compare**) are good examples for this.

By default the differencing engine returns the result of the compare operation as a tree of **DiffNode** objects. A **DiffNode** describes the changes among two or three inputs. The type of result nodes can be changed by overriding a single method of the engine.

Difference Viewers

A tree of **DiffNode**s can be displayed in a **DiffTreeViewer**. The **DiffTreeViewer** requires that inner nodes of the tree implement the **IDiffContainer** interface and leaves implement the **IDiffElement** interface.

The typical steps to compare hierarchically structured data and to display the differences are as follows:

1. Map the input data into a tree of objects implementing both the **IStructureComparator** and **IStreamContentAccessor** interfaces
2. Perform the compare operation by means of the **Differencer**
3. Feed the differencing result into the **DiffTreeViewer**

The **StructureDiffViewer** is a specialized **DiffTreeViewer** that automates the three steps from above. It takes a single input object of type **ICompareInput** from which it retrieves the two or three input elements to compare. It uses an **IStructureCreator** to extract a tree containing **IStructureComparator** and **IStreamContentAccessor** objects from them. These trees are then compared with the differencing engine and the result is displayed in the tree viewer.

The **ZipFileStructureCreator** is an implementation of the **IStructureCreator** interface and makes the contents of a zip archive available as a hierarchical structure of **IStructureComparator**s which can be easily compared by the differencing engine (**Differencer**). It is a good example for how to make structured files

available to the hierarchical compare functionality of the compare plug−in.

Merging multiple streams

The search plug−in allows you to customize views that assist the user in merging different content streams. In some cases, however, the ability to merge streams without the assist of a user is desirable. The extension point **org.eclipse.compare.streamMergers** allows you to contribute a class that merges three different input streams into a single output stream. Stream mergers can be associated with file extensions or bound to a particular content type. The search plug−in defines a stream merger for merging three streams of plain text:

```
<extension
        point="org.eclipse.compare.streamMergers">
        <streamMerger
                extensions="txt"
                class="org.eclipse.compare.internal.merge.TextStreamMerger"
                id="org.eclipse.compare.internal.merge.TextStreamMerger">
        </streamMerger>
        <contentTypeBinding
                contentTypeId="org.eclipse.core.runtime.text"
                streamMergerId="org.eclipse.compare.internal.merge.TextStreamMerger">
        </contentTypeBinding>
</extension>
```

The stream merger itself is described in the **streamMerger** element. You must specify the **id** of the merger and the **class** that implements it. You may also specify any file **extensions** for which the the stream merger should be used.

You may also use the **contentTypeBinding** element to associate a content type with a stream merger.

Stream mergers must implement **IStreamMerger**. This simple interface merges the contents from three different input streams into a single output stream. The not−so−simple implementation depends upon your plug−in and its content types.

New **IStreamMerger**s can be created for registered types with the createStreamMerger methods of **CompareUI**.

Advanced compare techniques

This section provides additional information about advanced API in the compare plug−in.

Writing compare operations

A compare operation must be implemented as a subclass of **CompareEditorInput**. A **CompareEditorInput** runs a (potentially lengthy) compare operation under progress monitor control, creates a UI for drilling−down into the compare results, tracks the dirty state of the result in case of merge, and saves any changes that occurred during a merge.

CompareUI defines the entry point to initiate a configurable compare operation on arbitrary resources. The result of the compare is opened into a compare editor where the details can be browsed and edited in dynamically selected structure and content viewers.

NavigationAction is used to navigate (step) through the individual differences of a **CompareEditorInput**.

CompareConfiguration configures various UI aspects of compare/merge viewers like title labels and images, or whether a side of a merge viewer is editable. It is passed to the **CompareEditorInput** on creation.

When implementing a hierarchical compare operation as a subclass of **CompareEditorInput**, clients must provide a tree of objects where each node implements the interface **IStructureComparator**. This interface is used by the hierarchical differencing engine (**Differencer**) to walk the tree.
In addition every leaf of the tree must implement the **IStreamContentAccessor** interface in order to give the differencing engine access to its stream content.

BufferedContent provides a default implementation for the **IStreamContentAccessor** and **IContentChangeNotifier** interfaces. Its subclass **ResourceNode** adds an implementation for the **IStructureComparator** and **ITypedElement** interfaces based on platform workbench resources (**IResource**). It can be used without modification as the input to the differencing engine.

Compare functionality outside of compare editors

If you want to use compare functionality outside of the standard compare editor (for example, in a dialog or wizard) the compare plug−in provides additional helper classes.

CompareViewerPane is a convenience class which provides a label and local toolbar for a compare viewer (or any other subclass of a JFace viewer). Its abstract subclass **CompareViewerSwitchingPane** supports **dynamic viewer switching**, that is the viewer installed in the pane is dynamically determined by the pane's input object.

EditionSelectionDialog is a simple selection dialog where one input element can be compared against a list of historic variants (**editions**) of the same input element. The dialog is used to implement functions like ″**Replace with Local History"** on workbench resources.

In addition it is possible to specify a subsection of the input element (e.g. a method in a Java source file) by means of a **path**. In this case the dialog compares only the subsection (as specified by the path) with the corresponding subsection in the list of editions. This functionality can be used to implement ″**Replace with Element from Local History"** for Java Elements.

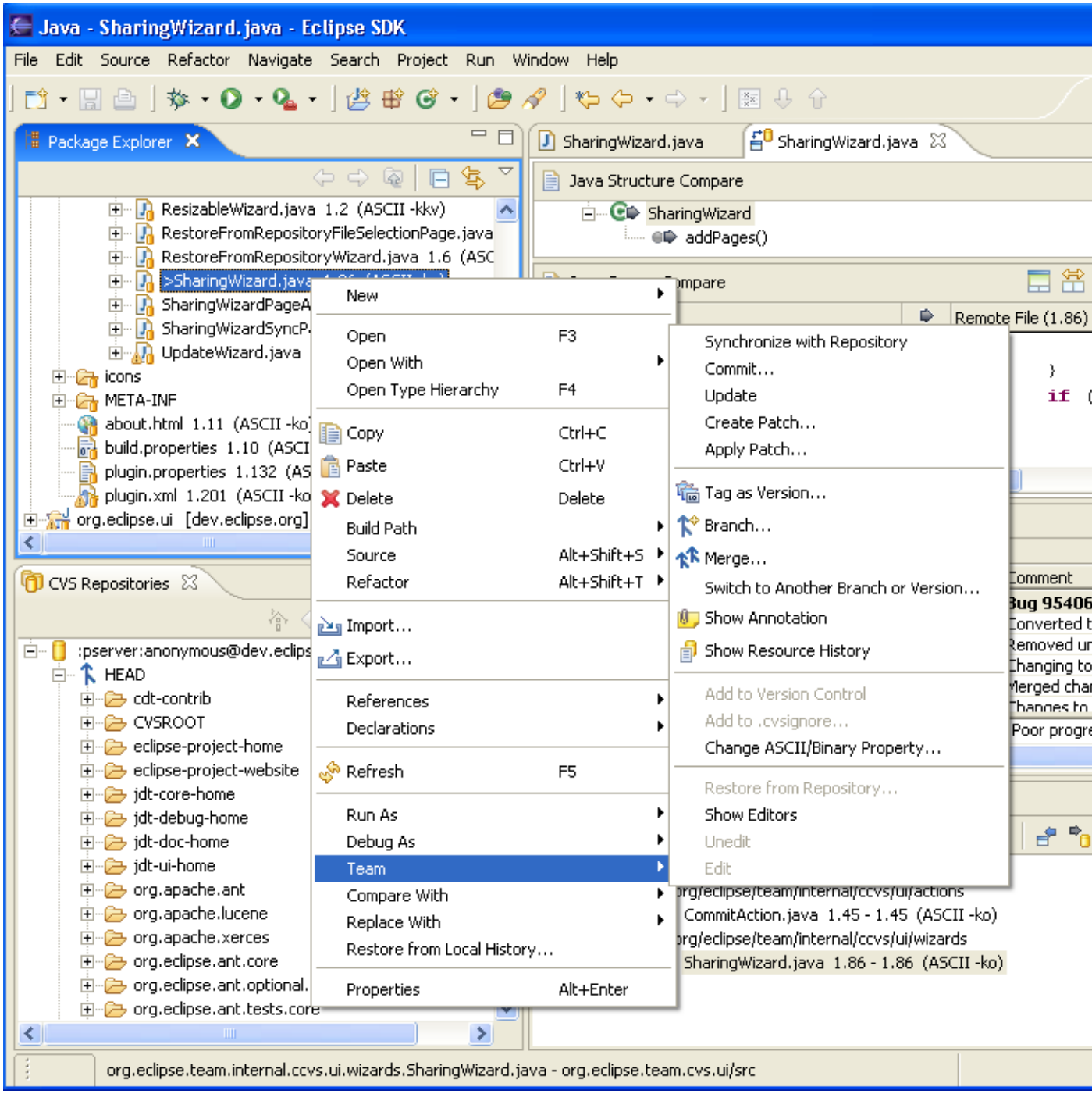
The **EditionSelectionDialog** requires that the editions implement the **IStreamContentAccessor** and **IModificationDate** interfaces. The **HistoryItem** is a convenience class that implements these interfaces for **IFileState** objects.

Rich Team Integration

Integrating your repository's support with the platform starts with good solid design. The goal is to integrate the workflow that your repository users know with the concepts defined in the workbench. Because there are many ways to extend workbench UI and functionality, you have a lot of flexibility in how you achieve integration. So where to start?

Building a team provider is not just a matter of learning Team API. (Subsequent sections will focus on the specific support introduced by the team plug−in.) It's a matter of understanding workbench integration. So let's start with the big picture. We'll be using the CVS client as a case study for integrating a team provider with the platform. Let's look at some of the function the CVS provider supplies and what workbench and team

facilities you can use to achieve similar levels of integration.



The CVS client integrates seamlessly with the existing workbench resource perspective. It allows users to configure a project for CVS, adds functionality to a resource's menu, decorates resources with team–specific information, provides customized views that show team–specific information, adds team–oriented tasks to the task list... The list goes on and on. How can your provider achieve similar integration? Here are some basic steps to start with and links for information (both team–specific and workbench–oriented) on these topics.

Getting started

Define a **RepositoryProvider** that represents your implementation.	• Define your provider using **org.eclipse.team.core.repository**. • Subclass **RepositoryProvider** and implement the necessary methods.
Provide a configuration wizard so that users can associate your provider with their projects.	• Contribute a wizard using **org.eclipse.team.ui.configurationWizards**.
Add your actions to the Team menu.	• Add your actions to the Team menu. • Use the **popupMenus** extension to define the menu items.

Enhancing resource views

Add provider−specific properties to the properties page for a resource.	• Implement and contribute property pages to show team−specific properties for your resource.
Implement specialized decorators to show team−related attributes	• Contribute decorators to resource views.
Reduce clutter by filtering out any resources that are used in implementing team support.	• Use team−private resources to hide implementation files and folders.

Handling user editing and changes to resources

Intervene in the saving of resources so you can check permissions before a user changes a file.	• Implement the fileModificationValidator hook. • Use **validateSave** to prevent or intervene in saving of files.
Intervene before a user edits a file to see if it's allowed.	• Implement the fileModificationValidator hook. • Use **validateEdit** to prevent or intervene in saving of files.
Track changes to resources in the workspace so you can allow associated changes in the repository.	• Use the move/delete hooks to prevent or enhance moving and deleting of resources. • See **IMoveDeleteHook** for more detail about what you can do.
Ensure that the proper resource locks are obtained for resource	• Use the resource rule factory to ensure that the proper rules are obtained for resource

operations that invoke the move/delete hook or fileModificatonValidator.	operations. • See **ResourceRuleFactory** for more details.
Enable the use of linked resources	• See **Team and linked resources**

Streamlining repository−related tasks

Provide an easy way to export a description of your projects.	• Use project sets to export your projects without exporting the content so that users can rebuild projects from the repository.
Reduce clutter in the repository by ignoring files that can be regenerated.	• Honor the **ignore** extension when handling files and use ignore for your plug−in's derived files.

Enhancing platform integration

Add provider−specific preferences to the preferences page.	• Add your preferences to the Team category. • Build your preference page using workbench support.
Implement custom views to show detailed information about repositories or their resources.	• Use the **views** extension to contribute a view. • See the CVS provider's repository view for an example.
Add your views or actions to existing workbench perspectives if appropriate.	• Use the **perspectiveExtensions** extension to add your plug−in's shortcuts or views to existing perspectives.
Implement a repository−specific perspective to streamline repository administration or browsing.	• Use the **perspectives** extension to define your own perspective, views, short cuts, and page layout.

Adding team actions

The team UI plug−in defines a popup menu extension in order to consolidate all team−related actions in one place. The team menu includes many subgroup slots so that team provider plug−ins can contribute actions and have some amount of control over the order of items in the menu. The following markup is from the team UI's plug−in manifest:

```
<extension
     point="org.eclipse.ui.popupMenus">
        <objectContribution
           id="org.eclipse.team.ui.ResourceContributions"
```

```
            objectClass="org.eclipse.core.resources.IResource" adaptable="true">
                <menu
    team.main"          id="
                        path="additions"
                        label="%TeamGroupMenu.label">
<separator name="group1"/>
                        <separator name="group2"/>
                        <separator name="group3"/>
                        <separator name="group4"/>
                        <separator name="group5"/>
                        <separator name="group6"/>
                        <separator name="group7"/>
                        <separator name="group8"/>
                        <separator name="group9"/>
                        <separator name="group10"/>
                        <separator name="targetGroup"/>
                        <separator name="projectGroup"/>
                </menu>
        ...
</extension>
```

A team menu is added to the popup menu of all views that show resources (or objects that adapt to resources.) Your plug−in can use the id of this menu and the separator groups in order to add your own menu items. There is nothing to keep you from defining your own popup menus, action sets, or view and editor actions. However, adding your actions to the predefined team menu makes it easier for the end user to find your actions.

Let's look at a CVS action that demonstrates some interesting points:

```
<extension
    point="org.eclipse.ui.popupMenus">
     <objectContribution
           objectClass="org.eclipse.core.resources.IFile"
           adaptable="true"
           id="org.eclipse.team.ccvs.ui.IFileContributions">
        <filter
              name="projectPersistentProperty"
              value="org.eclipse.team.core.repository=org.eclipse.team.cvs.core.cvsnature">
        </filter>
        <action
              label="%IgnoreAction.label"
              tooltip="%IgnoreAction.tooltip"
              class="org.eclipse.team.internal.ccvs.ui.actions.IgnoreAction"
              menubarPath="team.main/group3"
              helpContextId="org.eclipse.team.cvs.ui.team_ignore_action_context"
              id="org.eclipse.team.ccvs.ui.ignore">
        </action>
        ...
```

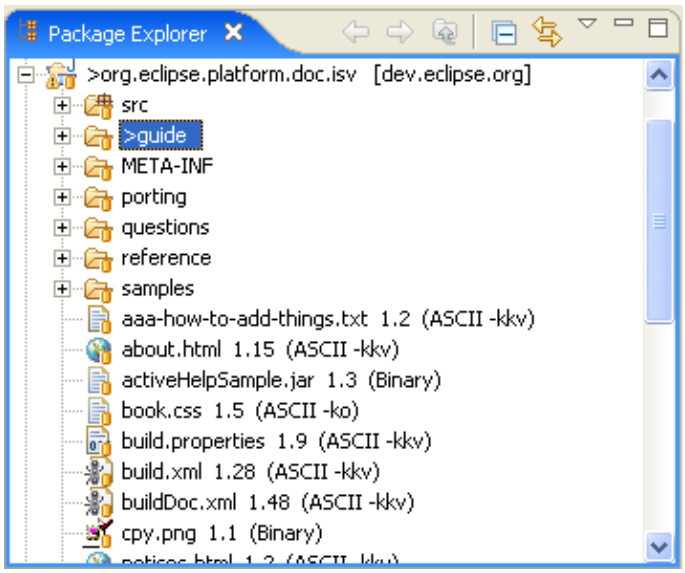
Note that the action is contributed using the **org.eclipse.ui.popupMenus** workbench extension point. Here are some team−specific things happening in the markup:

- the action is filtered by a project persistent property which identifies team providers. The value of the property must be of the format **"org.eclipse.team.core.repository=<your repository id>"** where **<your repository id>** is the **id** provided in the **org.eclipse.team.core.repository** markup. This filter ensures that the CVS popup menu items only appear for files that appear in projects that have been mapped to the CVS repository id.
- the action is added to a group in the menu that was specified above in the team UI plug−in

The implementation of an action is largely dependent on your specific provider.

Team decorators

Since any view that shows resources can contain projects that are configured with different team providers, it is helpful for team providers to contribute decorators that distinguish resources configured for their repository. The CVS client uses decorators to show information such as a dirty flag (the > symbol), tags, keywords (e.g. "(ASCII –kkv)"), and revisions (e.g. "1.15"). Icons can also be decorated (e.g. the ▯ symbol indicates that the resource is managed by CVS).



Some decorators may be expensive to compute, so it's a good idea to allow the users some control over the use, or even content, of decorators. The CVS client provides a preference page that allows users to control the presentation and content of decorators.

See **org.eclipse.ui.decorators** for a complete description of the decorator extension point. The CVS decorator markup is as follows:

```
<extension
     point="org.eclipse.ui.decorators">
  <decorator
        objectClass="org.eclipse.core.resources.IResource"
        adaptable="true"
        label="%DecoratorStandard.name"
        state="false"
        class="org.eclipse.team.internal.ccvs.ui.CVSDecorator"
        id="org.eclipse.team.cvs.ui.decorator">
     <description>
        %DecoratorStandard.description
     </description>
  </decorator>
</extension>
```

Team and linked resources

A project may contain resources that are not located within the project's directory in the local file system. These resources are referred to as linked resources.

Consequences for Repository Providers

Linked resources can pose particular challenges for repository providers which operate directly against the file system. This is a consequence of the fact that linked resources by design do not exist in the immediate project directory tree in the file system.

Providers which exhibit the following characteristics may be affected by linked resources:

1. Those which call out to an external program that then operates directly against the file system.
2. Those which are implemented in terms of IResource but assume that all the files/folders in a project exist as direct descendents of that single rooted directory tree.

In the first case, lets assume the user picks a linked resource and tries to perform a provider operation on it. Since the provider calls a command line client, we can assume that the provider does something equivalent to first calling **IResource**.getLocation().toOSString(), feeding the resulting file system location as an argument to the command line program. If the resource in question is a linked resource, this will yield a file/folder outside of the project directory tree. Not all command line clients may expect and be able to handle this case. In short, if your provider ever gets the file system location of a resource, it will likely require extra work to handle linked resources.

The second case is quite similar in that there is an implicit assumption that the structure of the project resources is 1:1 with that of the file system files/folders. In general, a provider could be in trouble if they mix IResource and java.io.File operations. For example, for links, the parent of **IFile** is not the same as the java.io.File's parent and code which assumes these to be the same will fail.

Backwards Compatibility

It was important that the introduction of linked resources did not inadvertently break existing providers. Specifically, the concern was for providers that reasonably assumed that the local file system structure mirrored the project structure. Consequently, by default linked resources can not be added to projects that are mapped to such a provider. Additionally, projects that contain linked resources can not by default be shared with that provider.

Strategies for Handling Linked Resources

In order to be "link friendly", a provider should allow projects with linked resources to be version controlled, but can disallow the version controlling of linked resources themselves.

A considerably more complex solution would be to allow the versioning of the actual linked resources, but this should be discouraged since it brings with it complex scenarios (e.g. the file may already be version controlled under a different project tree by another provider). Our recommendation therefore is to support version controlled projects which contain non−version controlled linked resources.

Technical Details for Being "Link Friendly"

Repository provider implementations can be upgraded to support linked resources by overriding the **RepositoryProvider.canHandleLinkedResources()** method to return ***true***. Once this is done, linked resources will be allowed to exist in projects shared with that repository provider. However, the repository provider must take steps to ensure that linked resources are handled properly. As mentioned above, it is strongly suggested that repository providers ignore all linked resources. This means that linked resources (and their children) should be excluded from the actions supported by the repository provider. Furthermore, the repository provider should use the default move and delete behavior for linked resources if the repository provider implementation overrides the default **IMoveDeleteHook**.

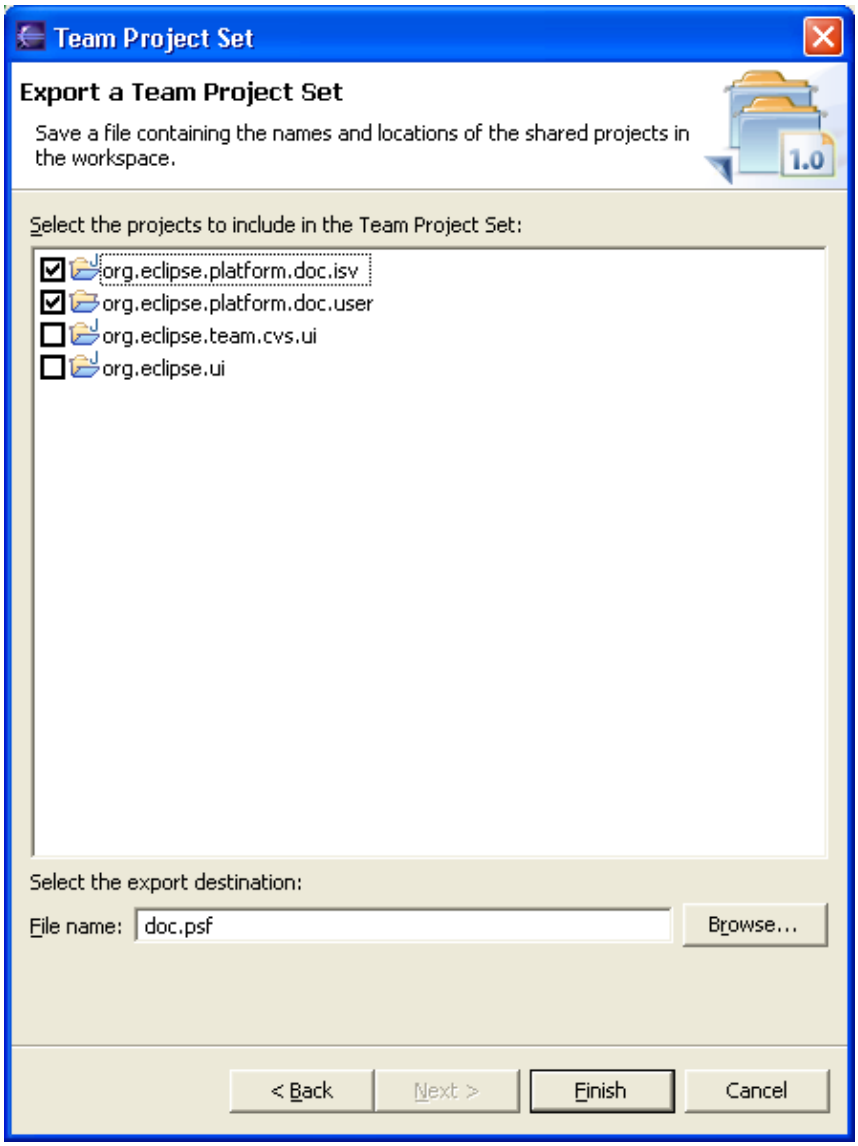
Team providers can use **IResource.isLinked()** to determine if a resource is a link. However, this method only returns true for the root of a link. The following code segment can be used to determine if a resource is the child of a link.

```
String linkedParentName = resource.getProjectRelativePath().segment(0);
IFolder linkedParent = resource.getProject().getFolder(linkedParentName);
boolean isLinked = linkedParent.isLinked();
```

Repository providers should ignore any resource for which the above code evaluates to ***true***.

Project sets

Since the resources inside a project under version control are kept in the repository, it is possible to share projects with team members by sharing a reference to the repository specific information needed to reconstruct a project in the workspace. This is done using a special type of file export for **team project sets**.



In 3.0, API was added to ProjectSetCapability to allow repository providers to declare a class that implements project saving for projects under their control. When the user chooses to export project sets, only the projects configured with repositories that define project sets are shown as candidates for export. This API replaces the old project set serialization API (see below).

The project set capability class for a repository provider is obtained from the RepositoryProviderType class which is registered in the same extension as the repository provider. For example:

```
<extension point="org.eclipse.team.core.repository">
    <repository
          typeClass="org.eclipse.team.internal.ccvs.core.CVSTeamProviderType"
          class="org.eclipse.team.internal.ccvs.core.CVSTeamProvider"
          id="org.eclipse.team.cvs.core.cvsnature">
    </repository>
</extension>
```

Prior to 3.0, The **org.eclipse.team.core.projectSets** extension point allowed repository providers to declare a class that implements project saving for projects under their control. When the user chooses to export project sets, only the projects configured with repositories that define project sets are shown as candidates for export.

For example, the CVS client declares the following:

```
<extension point="org.eclipse.team.core.projectSets">
        <projectSets id="org.eclipse.team.cvs.core.cvsnature" class="org.eclipse.team.internal.cc
</extension>
```

The specified class must implement **IProjectSetSerializer**. Use of this interface is still supported in 3.0 but has been deprecated.

File types

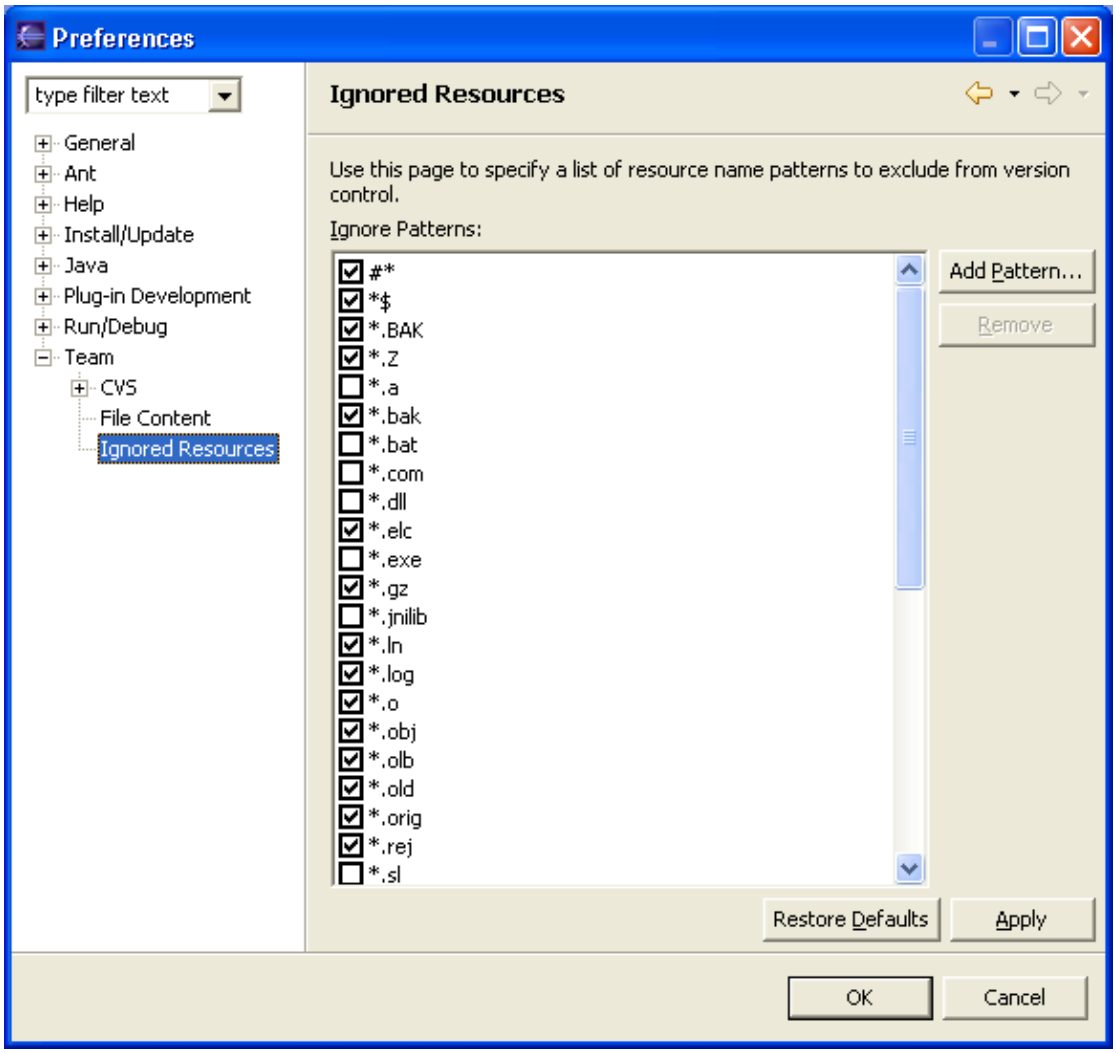
The presence of a repository management system may dictate special handling needs for files. For example, some files should be omitted from version control. Some providers have special handling for text vs. binary files. The team plug−in defines extension points that allow other plug−ins to provide information about their file types. In all cases, special handling is ultimately left up to the user via the team Preferences page. These extensions allow plug−ins to seed the preferences with values appropriate for the plug−in.

Ignored files

In several cases, it may be unnecessary to keep certain files under repository control. For example, resources that are derived from existing resources can often be omitted from the repository. For example, compiled source files, (such as Java ".class" files), can be omitted since their corresponding source (".java") file is in the repository. It also may be inappropriate to version control metadata files that are generated by repository providers. The **org.eclipse.team.core.ignore** extension point allows providers to declare file types that should be ignored for repository provider operations. For example, the CVS client declares the following:

```
<extension point="org.eclipse.team.core.ignore">
        <ignore pattern = ".#*" selected = "true"/>
</extension>
```

The markup simply declares a file name **pattern** that should be ignored and a **selected** attribute which declares the default selection value of the file type in the preferences dialog. It is ultimately up to the user to decide which files should be ignored. The user may select, deselect, add or delete file types from the default list of ignored files.

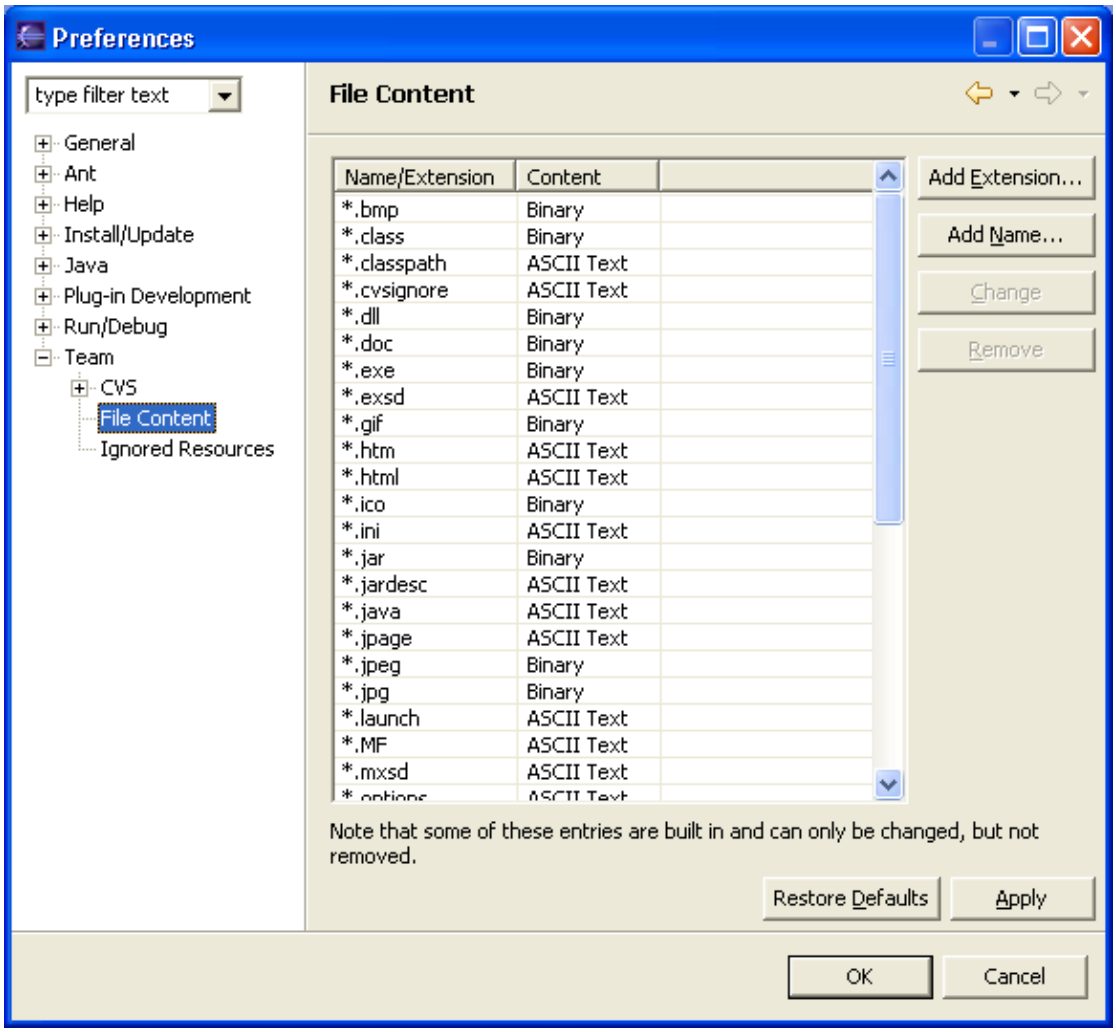


Text vs. binary files

Some repositories implement different handling for text vs. binary files. The **org.eclipse.team.core.fileTypes** extension allows plug–ins to declare file types as text or binary files. For example, the Java tooling declares the following:

```
<extension point="org.eclipse.team.core.fileTypes">
  <fileTypes extension="java" type="text"/>
  <fileTypes extension="classpath" type="text"/>
  <fileTypes extension="properties" type="text"/>
  <fileTypes extension="class" type="binary"/>
  <fileTypes extension="jar" type="binary"/>
  <fileTypes extension="zip" type="binary"/>
</extension>
```

The markup lets plug–ins define a file type by **extension** and assign a **type** of text or binary. As with ignored files, it is ultimately up to the user to manage the list of text and binary file types.

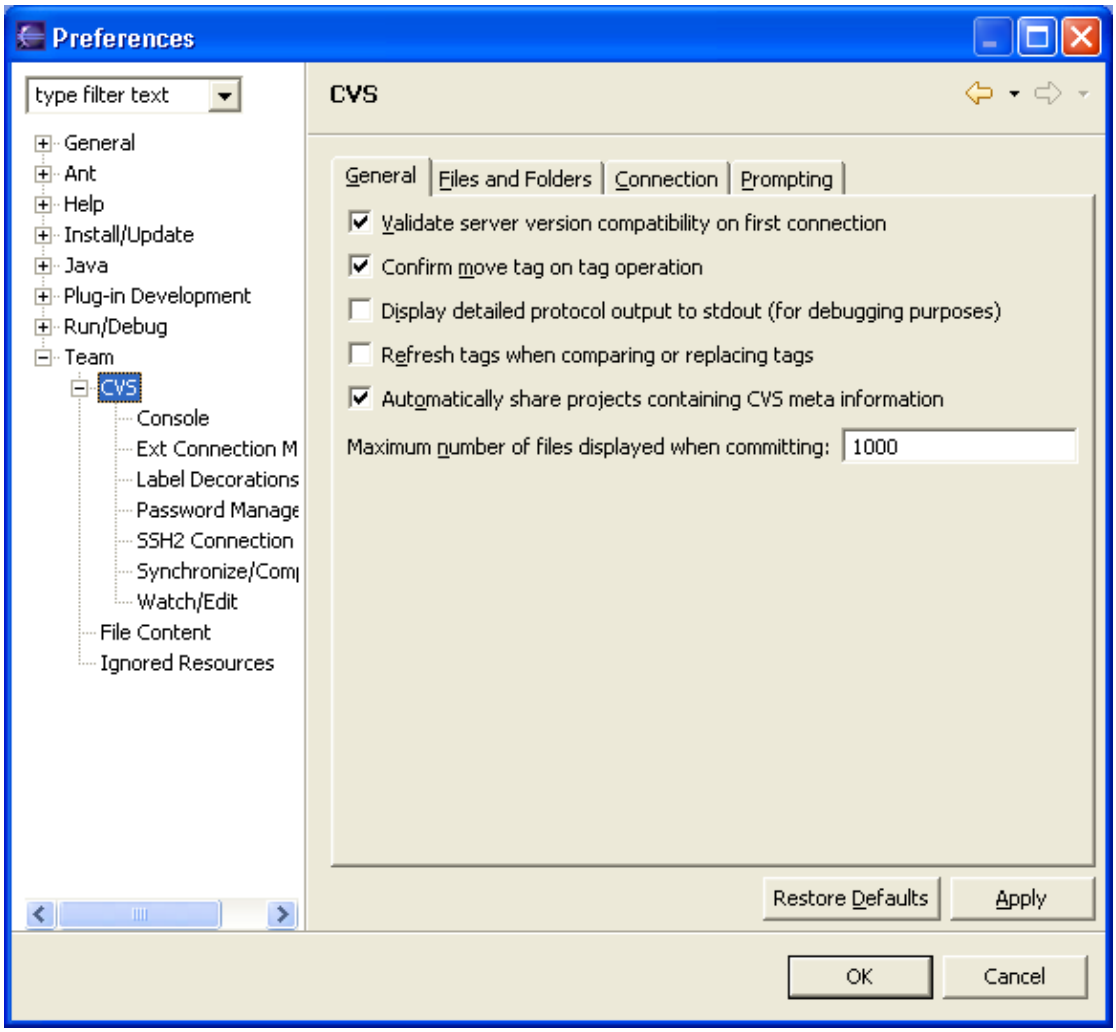


Adding preferences and properties

Preferences and properties can be contributed by team UI plug–ins using the standard techniques. The only difference for a team plug–in is that preferences should be contributed using the team category, so that all team related preferences are grouped together. The CVS markup for the main preferences page looks like this:

```
<extension
      point="org.eclipse.ui.preferencePages">
   <page
         name="%PreferencePage.name"
         category="org.eclipse.team.ui.TeamPreferences"
         class="org.eclipse.team.internal.ccvs.ui.CVSPreferencesPage"
         id="org.eclipse.team.cvs.ui.CVSPreferences">
   </page>
</extension>
```

The preferences dialog shows the CVS preferences underneath the team category.



Properties are added as described by **org.eclipse.ui.propertyPages**. There is no special team category for properties, since a resource can only be configured for one repository provider at a time. However, you must set up your property page to filter on the team project persistent property (similar to the way we filtered resources for popup menu actions.)

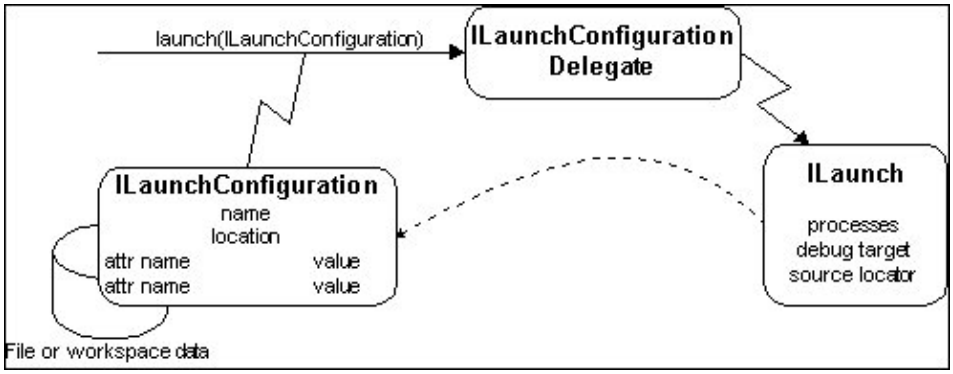
```
<extension
  point="org.eclipse.ui.propertyPages">
   <page
         objectClass="org.eclipse.core.resources.IFile"
         adaptable="true"
         name="%CVS"
         class="org.eclipse.team.internal.ccvs.ui.CVSFilePropertiesPage"
         id="org.eclipse.team.ccvs.ui.propertyPages.CVSFilePropertiesPage">
      <filter
            name="projectPersistentProperty"
            value="org.eclipse.team.core.repository=org.eclipse.team.cvs.core.cvsnature">
      </filter>
   </page>
   ...
```

Launching a program

The platform debug plug−ins allow your plug−in to extend the platform so that your partiticular type of program can be launched from the workbench, obtaining input from the user if necessary. A unique type of program that can be launched in the platform is called a **launch configuration type**. The class **ILaunchConfiguration** is used to describe a type of configuration. A launch configuration keeps a set of named attributes that can be used to store data specific for a particular kind of launcher.

For each launch configuration type, there are different **modes** in which the configuration can be launched. The platform defines modes for running, debugging, or profiling a program defined by a particular configuration. Plug−ins are free to implement any or all of these launch modes for their particular launch configuration, or define new launch modes for any launch configuration.

Plug−ins that contribute additional types of launchers do so by providing an **ILaunchConfigurationDelegate** (or **ILaunchConfigurationDelegate2**) that knows how to launch a program given the expected type and mode for launch configuration. Once the program is launched, an **ILaunch** object is used to represent the launched session. This object can be queried for information such as running processes, debug session information, and source code location. A launch knows the configuration that was used to create it.



Users interact with a launch configuration dialog to set up the parameters for different types of launches. These configurations can be stored in a file to be shared with other users or stored locally in the workspace.

Adding launchers to the platform

Your plug−in can add launch configuration types to the platform using the **org.eclipse.debug.core.launchConfigurationTypes** extension point. This extension point allows you to declare a configuration type using a unique id. You must provide a corresponding implementation of **ILaunchConfigurationDelegate**. The delegate is responsible for launching its launch configuration in a specified mode. Optionally, you can implement **ILaunchConfigurationDelegate2**, which enhances the delegate interface to allow your delegate to abort a launch, build relevant projects in the workspace before a launch, and control the creation of the launch object that is used in a launch.

In addition to defining the delegate, you can specify which launch modes are supported by your delegate, and a name that should be used when showing launchers of this type to the user.

The following markup shows how the Java tools declare a Java launch configuration for launching local Java programs:

```
<extension point = "org.eclipse.debug.core.launchConfigurationTypes">
```

```
   <launchConfigurationType
           id="org.eclipse.jdt.launching.localJavaApplication"
   name="%localJavaApplication"
           delegate="org.eclipse.jdt.internal.launching.JavaLocalApplicationLaunchConfigurationDe
           modes= "run, debug"
           sourceLocatorId="org.eclipse.jdt.launching.sourceLocator.JavaSourceLookupDirector"
       sourcePathComputerId="org.eclipse.jdt.launching.sourceLookup.javaSourcePathComputer">
   </launchConfigurationType>
</extension>
```

This extension defines a launch configuration delegate that can be used to run or debug programs that are launched using the local Java launch configuration.

Defining new launch modes

We mentioned previously that the platform defines launch modes for running, debugging, or profiling a program. These modes are defined using the **org.eclipse.debug.core.launchModes** extension point. This extension point allows you to declare a launch mode by defining its string mode name and the label that should be shown to the user to describe the mode. The following markup shows the definition of the platform's three standard launch modes:

```
<extension point="org.eclipse.debug.core.launchModes">
        <launchMode
                label="%run"
                mode="run">
        </launchMode>
        <launchMode
                label="%debug"
                mode="debug">
        </launchMode>
        <launchMode
                label="%profile"
                mode="profile">
        </launchMode>
</extension>
```

Note that the mode is not associated with any particular launch configuration type. As shown earlier, that association occurs when a launch delegate is specified for a configuration type.

Defining launch delegates

Since launch modes can be specified independently of launch configuration types, it's possible that new modes are defined that are not implemented by the original delegate for a launch configuration. In this case, a plug−in may define a launch delegate that implements a particular mode for a particular launch configuration type. This can be done using the **org.eclipse.debug.core.launchDelegates** extension point. This extension point allows you to define a launch delegate that implements the specified modes for a given configuration type. The following markup shows how you could define a delegate for profiling a local Java application:

```
<extension point="org.eclipse.debug.core.launchDelegates">
  <launchDelegate
   id="com.example.MyJavaProfileDelegate"
   delegate="com.example.MyJavaProfileDelegate"
   type="org.eclipse.jdt.launching.localJavaApplication"
   modes="profile">
  </launchDelegate>
 </extension>
```

The specification of the delegate is similar to the way it is done when defining a launch configuration type, except that in this case the type of launch configuration is specified along with the supported modes. As seen previously, the delegate must implement **ILaunchConfigurationDelegate**, and can optionally implement **ILaunchConfigurationDelegate2** for more control over the launch sequence.

Other references

We Have Lift−off: The Launching Framework in Eclipse provides a start to finish example for defining your own launch type.

Obtaining a program's source code

For certain kinds of launch modes, it may be important to obtain the source code that corresponds with the current execution point in the code. This is typically important when debugging or profiling a program. Several different extension points are provided by the debug plug−in that allow plug−ins to register classes that can assist with locating source code.

Source locators

ISourceLocator and **IPersistableSourceLocator** define interfaces for mapping from an executing program back to the source code.

Source locators are typically implemented to work with a corresponding launch configuration and launch configuration delegate. A source locator id may be specified when a launch configuration type is defined, or it may be associated programmatically with a launch configuration using the **ILaunchConfiguration.ATTR_SOURCE_LOCATOR_ID** attribute. In either case, at some point the id of a source locator for a configuration must be resolved to the class that actually implements **IPersistableSourceLocator**. The association between a source locator id and its class is established using the **org.eclipse.debug.core.sourceLocators** extension point.

The following markup is from the Java tooling:

```
<extension point = "org.eclipse.debug.core.sourceLocators">
        <sourceLocator
                id = "org.eclipse.jdt.debug.ui.javaSourceLocator"
                class="org.eclipse.jdt.debug.ui.JavaUISourceLocator"
                name="%javaSourceLocator"/>
</extension>
```

Since launch configurations can be persisted, source locator ids may will be stored with the launch configuration. When it's time to instantiate a source locator, the debug plug−in looks up the source locator id attribute and instantiates the class associated with that id.

The implementation for source lookup necessarily depends on the type of program being launched. However, the platform defines an abstract implementation for a source locator that looks up source files on a given path that includes directories, zip files, jar files, and the like. To take advantage of this implementation, your plug−in can extend **AbstractSourceLookupDirector**. All that is needed from the specific implementation is the ability to provide an appropriate **ISourceLookupParticipant**, which can map a stack frame to a file name. See the extenders of **AbstractSourceLookupDirector**. for examples.

Source path computers

The **AbstractSourceLookupDirector** searches for source files according to a particular source code lookup path. This path is expressed as an array of **ISourceContainer**. The source containers that should be searched for source are typically computed according to the particulars of the source configuration that is being launched. **ISourcePathComputer** defines the interface for an object that computes the appropriate source path for a launch configuration. A source path computer, much like a source locator, is specified by id, and can be specified in the extension definition for a launch configuration type, or associated programmatically by setting the **ISourceLocator.ATTR_SOURCE_PATH_COMPUTER_ID** attribute for the launch configuration. The id for a source path computer is associated with its implementing class in the **org.eclipse.debug.core.sourcePathComputers** extension point. The following markup shows the definition used by JDT for its Java source path computer:

```
<extension point="org.eclipse.debug.core.sourcePathComputers">
        <sourcePathComputer
                id="org.eclipse.jdt.launching.sourceLookup.javaSourcePathComputer"
                class="org.eclipse.jdt.launching.sourcelookup.containers.JavaSourcePathComputer">
        </sourcePathComputer>
        ...
```

The source path computer is responsible for computing an array of **ISourceContainer** that represents the source lookup path. For example, the Java source path computer considers the classpath when building the path.

Source container types

The containers specified as part of a source lookup path must implement **ISourceContainer**, which can search the container represented for a named source element. Different kinds of source containers may be needed to represent the different kinds of places source code is stored. For example, the JDT defines source containers that represent source in a Java project, source on the classpath, and source in a package fragment. The source containers used for a launch configuration may be stored by id in the launch configuration. Since launch configurations can be persisted, there must be a way to associate the the id of a source container with its implementation class. This is done using the **org.eclipse.debug.core.sourceContainerTypes** extension point. The following example comes from the JDT:

```
<extension point="org.eclipse.debug.core.sourceContainerTypes">
        <sourceContainerType
                id="org.eclipse.jdt.launching.sourceContainer.javaProject"
                name="%javaProjectSourceContainerType.name"
                description="%javaProjectSourceContainerType.description"
                class="org.eclipse.jdt.internal.launching.JavaProjectSourceContainerTypeDelegate"
        </sourceContainerType>
        ...
```

Comparing launch configurations

We've seen how a plug−in can use named attributes and values to store important data with a launch configuration. Since the interpretation of a plug−in's attributes are not known by the platform, an extension point is provided that allows you to supply a comparator for a specific attribute. This comparator is used to determine whether attributes of the specified name are equal. In many cases, the simple string compare provided by java.lang.Object.equals(Object) is suitable for comparing attributes. This technique will be used if no comparator has been provided. However, some attribute values may require special handling, such as stripping white space values from text before comparing for equality.

Comparators are contributed using the **org.eclipse.debug.core.launchConfigurationComparators** extension point.

The Java tools supply launch configuration comparators for comparing program source paths and class paths.

```
<extension point = "org.eclipse.debug.core.launchConfigurationComparators">
        <launchConfigurationComparator
                id = "org.eclipse.jdt.launching.classpathComparator"
                class = "org.eclipse.jdt.internal.launching.RuntimeClasspathEntryListComparator"
                attribute = "org.eclipse.jdt.launching.CLASSPATH"/>
        <launchConfigurationComparator
                id = "org.eclipse.jdt.launching.sourcepathComparator"
                class = "org.eclipse.jdt.internal.launching.RuntimeClasspathEntryListComparator"
                attribute = "org.eclipse.jdt.launching.SOURCE_PATH"/>
</extension>
```

Comparators must implement the interface **java.util.Comparator**.

Process factories

When a launch configuration launches its program, it is responsible for invoking the executable program in the requested mode. The implementation for a launch will depend on the specifics of each launch configuration, but most plug−ins will build a command line and call a runtime exec to start the program. The **DebugPlugin** class implements a convenience method for invoking a runtime exec and handling the possible exceptions. Clients can specify the command line and working directory for the exec.

```
Process p = DebugPlugin.exec(cmdLine, workingDirectory);
```

Once the **java.lang.Process** for the executing program has been created, it needs to be managed by the debug plug−in. For starters, the process needs to be associated with the **ILaunch** that represents the launched program. The debug plug−in defines a wrapper for a system process, **IProcess**, that allows clients to access the associated **ILaunch** and assign their own named attributes to the process. In addition, **IProcess**, defines a label for the process and associates an **IStreamsProxy** with the process that gives clients access to the input, output, and error streams of the system process. This process wrapper can also be created using a utility method in **DebugPlugin**.

```
IProcess process= DebugPlugin.newProcess(launch, p, "My Process);
```

A map of named attributes can also be supplied.

Many plug−ins can simply rely on the utility methods in **DebugPlugin** for launching the system process and wrapping it in an **IProcess**. For those plug−ins that need more control in the creation of the wrapper, a **process factory** can be associated with a launch configuration. The process factory is used to create an **IProcess** that meets the special needs of the plug−in. The process factory is referenced by id, and should be stored in the **DebugPlugin.ATTR_PROCESS_FACTORY_ID** attribute of the launch configuration.

The association between the process factory id and the class that implements it is made using the **org.eclipse.debug.core.processFactories** extension point.

The following example shows how the Ant plug−in sets up a process factory for its launches:

```
<extension point="org.eclipse.debug.core.processFactories">
        <processFactory
                class="org.eclipse.ant.internal.ui.launchConfigurations.RemoteAntProcessFactory"
```

```
                id="org.eclipse.ant.ui.remoteAntProcessFactory">
        </processFactory>
</extension>
```

It is the responsibility of the registering plug−in to store the process factory id in the proper launch configuration attribute.

Launching Java applications

In Process factories, we saw how a system process is used to invoke a program and how wrapping it with an **IProcess** allows clients to access launch−related information about the process. These techniques are general and can be used to launch any external program from the platform.

Additional classes provided in the **org.eclipse.jdt.debug** plug−in provide support for launching a specific JRE with a specified classpath, arguments, VM arguments, and a main type. See the following references for more specific information about launching a Java application:

- Launching Java Applications Programmatically
- We Have Lift−off: The Launching Framework in Eclipse

Handling errors from a launched program

If you have defined your own type of launch configuration, it's likely that you will want to handle errors or other status information that arises during the running of the program. For example, you may want to prompt or alert the user when certain types of errors occur during a launch, or provide information messages for certain status changes in the program. Since it's good practice to separate UI handling from core function, you do not want to have direct references from your launch delegate to status handling classes.

This problem is addressed by the **org.eclipse.debug.core.statusHandlers** extension point. It allows you to associate a status handler with a specific status code. Using this extension point, you can define all of the possible status and error codes in your launch delegate and core classes, while registering unique handlers for the different status codes from another plug−in.

The extension point does not designate any association between a status handler and a launch configuration. It is up to the implementation of the launch delegate to detect errors, find the appropriate status handler, and invoke it. The extension merely provides a registry so that the status handlers can be found for particular status codes. **DebugPlugin** provides a utility method for obtaining a specific status handler.

```
IStatusHandler handler = DebugPlugin.getDefault().getStatusHandler(status);
```

Status handlers should implement **IStatusHandler**. The status handling class is specified in the extension definition, along with its associated status code and the plug−in that is expected to generate the status codes.

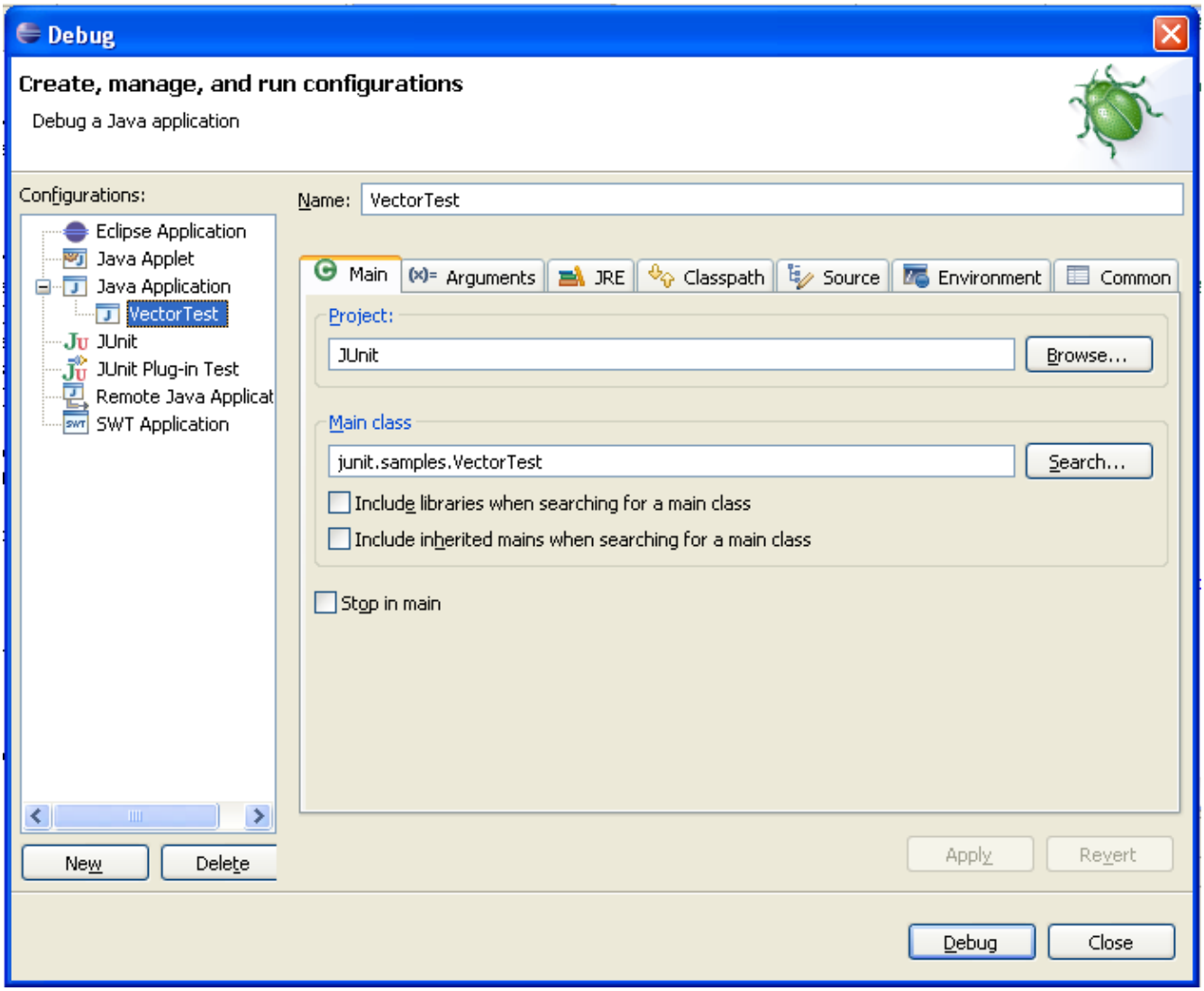
The following markup shows how the Java tools declare status handlers:

```
<extension point = "org.eclipse.debug.core.statusHandlers">
        <statusHandler
                id="org.eclipse.jdt.debug.ui.statusHandler.vmConnectTimeout"
                class="org.eclipse.jdt.internal.debug.ui.launcher.VMConnectTimeoutStatusHandler"
                plugin ="org.eclipse.jdt.launching"
                code="117">
        </statusHandler>
        ...
```

```
</extension>
```

Launch configuration dialog

Launch configurations can most easily be visualized by looking at their corresponding UI. Users interact with a launch configuration dialog to create instances of the different types of launch configurations that have been contributed by plug–ins. Each type of launch configuration defines a group of tabs that collect and display information about the configuration. The tab group for running a local Java application is shown below.



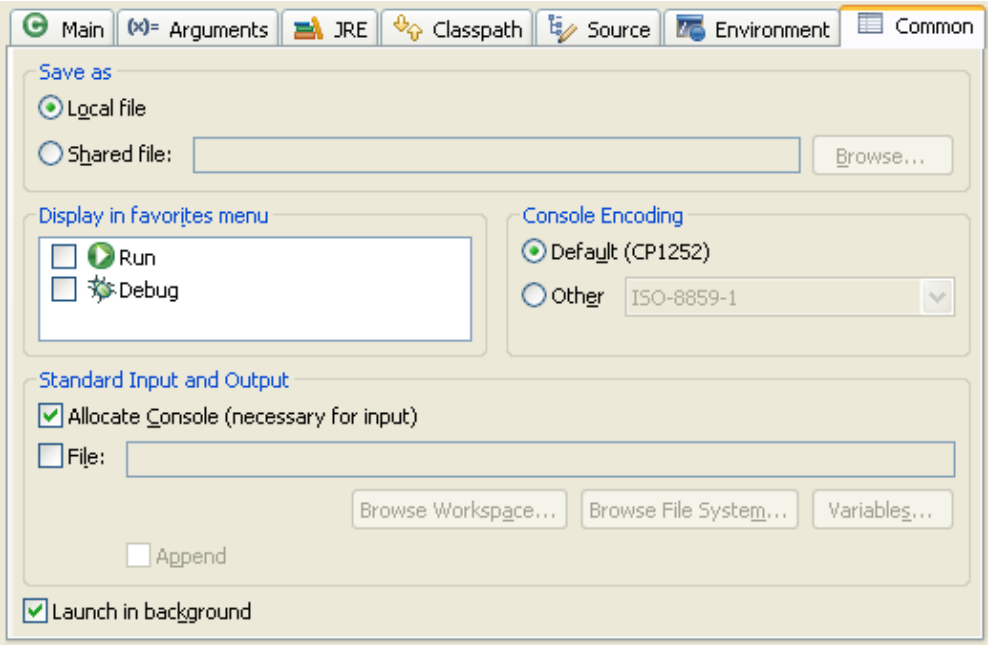
The tabs are contributed using the **org.eclipse.debug.ui.launchConfigurationTabGroups** extension point. In this extension, the id of a configuration type (defined using **org.eclipse.debug.core.launchConfigurationTypes**) is associated with the class that implements **ILaunchConfigurationTabGroup**.

It's possible that a some tab groups are only appropriate when launching the configuration in a particular mode. If this is the case, then one or more **mode** elements can be specified along with the class. For each mode, the tab group can be given a unique description. If no mode is specified, then the tab group will be shown on all modes that do not have a mode–specific tab group contribution. The Java application tab group is defined for run and debug modes:

```
<extension
      point="org.eclipse.debug.ui.launchConfigurationTabGroups">
   <launchConfigurationTabGroup
         type="org.eclipse.jdt.launching.localJavaApplication"
         helpContextId="org.eclipse.jdt.debug.ui.launchConfigHelpContext.local_java_applicatio
         class="org.eclipse.jdt.internal.debug.ui.launcher.LocalJavaApplicationTabGroup"
         id="org.eclipse.jdt.debug.ui.launchConfigurationTabGroup.localJavaApplication"
         bannerImage="icons/full/wizban/java_app_wiz.png">
         <launchMode
               mode="debug"
               perspective="org.eclipse.debug.ui.DebugPerspective"
               description="%localJavaApplicationTabGroupDescription.debug">
         </launchMode>
         <launchMode
               mode="run"
               description="%localJavaApplicationTabGroupDescription.run">
         </launchMode>
   </launchConfigurationTabGroup>
   ...
```

Note that a perspective may also be specified with a mode. This will cause the platform to switch to the specified perspective when the program is launched in that mode.

Your tab group class is responsible for creating the necessary tabs and displaying and saving the relevant data from the launch configuration attributes. A tab that is common to all configurations, **CommonTab**, is already implemented and can be created by any configuration. This tab manages the saving of the launch configuration as well as collecting common preferences.



Launch configuration type images

The image that is shown for a launch configuration type in the launch dialog is contributed using the **org.eclipse.debug.ui.launchConfigurationTypeImages** extension point. This extension associates an image file with the id of a configuration type.

The markup for the Java application image is as follows:

```
<extension
        point="org.eclipse.debug.ui.launchConfigurationTypeImages">
                <launchConfigurationTypeImage
                        icon="icons/full/etool16/java_app.png"
                        configTypeID="org.eclipse.jdt.launching.localJavaApplication"
                        id="org.eclipse.jdt.debug.ui.launchConfigurationTypeImage.localJavaApplic
                </launchConfigurationTypeImage>
                ...
```

Launch shortcuts

Once a launch configuration has been defined using the dialog, it can be shown directly in the appropriate menu, rather than having to open the launch configuration dialog again. When a launch configuration is shown directly in a menu, we refer to it as a **launch shortcut**. The **org.eclipse.debug.ui.launchShortcuts** extension point is used to register these shortcuts. In the extension definition, you can specify in which modes the shortcuts are shown. For each shortcut, you must specify an implementation of **ILaunchShortcut**. This class is used to launch a program given a particular selection in a view or editor.

You may also specify when the shortcut should be shown. The **contexttualLaunch** element allows you to describe applicable modes and enabling conditions for the shortcut. This is best demonstrated by example. The following markup registers shortcuts for launching a Java application:

```
  <extension
         point="org.eclipse.debug.ui.launchShortcuts">
      <shortcut
            label="%JavaApplicationShortcut.label"
            icon="icons/full/etool16/java_app.png"
            helpContextId="org.eclipse.jdt.debug.ui.shortcut_local_java_application"
            modes="run, debug"
            class="org.eclipse.jdt.internal.debug.ui.launcher.JavaApplicationLaunchShortcut"
            id="org.eclipse.jdt.debug.ui.localJavaShortcut">
         <contextualLaunch>
           <enablement>
             <with variable="selection">
               <count value="1"/>
               <iterate>
                 <or>
                   <test property="org.eclipse.debug.ui.matchesPattern" value="*.java"/>
                   <test property="org.eclipse.debug.ui.matchesPattern" value="*.class"/>
                   <instanceof value="org.eclipse.jdt.core.IJavaElement"/>
                 </or>
                 <test property="org.eclipse.jdt.debug.ui.hasMainType"/>
               </iterate>
             </with>
           </enablement>
                 </contextualLaunch>
         ...
      </shortcut>
      ...
```

See Boolean expressions and action filters for an explanation of the XML syntax for enabling conditions. The complete syntax is described in the documentation for **org.eclipse.debug.ui.launchShortcuts**.

Debugging a program

When you define a launch configuration for running a program, you can specify which modes (run, debug, profile, etc.) are supported by your program. If you support debug mode, then you need to implement a debug model and UI that allow users to interact with your programs while they are under debug. The core platform debug plug−in provides support for:

- a generic debug model
- debug events and listeners
- breakpoint management
- expression management

The debug UI plug−in provides a framework for showing your debug model in the UI. It also includes utility classes for implementing common UI tasks.

Since it's difficult to discuss generic debugging in any meaningful detail, we'll review the platform debug model and UI classes from the perspective of the Java debugger.

How to Write an Eclipse Debugger provides a start to finish example for building a debugger using a simple push down automata (PDA) assembly language as an example.

Platform debug model

The platform debug model defines generic debug interfaces that are intended to be implemented and extended in concrete, language−specific implementations.

Artifacts

The model includes classes that represent different artifacts in a program under debug. All of the artifacts implement **IDebugElement** in addition to their own interfaces. The model includes definitions for the following artifacts:

- Debug targets (**IDebugTarget**) − a debuggable execution context, such as a process or virtual machine
- Expressions (**IExpression**) − a snippet of code that can be evaluated to produce a value
- Memory blocks (**IMemoryBlock**) − a contiguous segment of memory in an execution context
- Registers (**IRegister**) − a named variable in a register group
- Register groups (**IRegisterGroup**) − a group of registers assigned to a stack frame
- Stack frames (**IStackFrame**) − an execution context in a suspended thread containing local variables and arguments
- Threads (**IThread**) − a sequential flow of execution in a debug target containing stack frames
- Values (**IValue**) − the value of a variable
- Variables (**IVariable**) − a visible data structure in a stack frame or value
- Watch expressions (**IWatchExpression**) − an expression that updates its value when provided with a particular context

Plug−ins that implement language−specific debuggers typically extend these interfaces to include language−specific behavior. All debug elements can return the id of the plug−in that originated them. This is important for registering other classes that are associated with a debug model, such as UI classes.

Actions

The model also includes interfaces that define a set of debug actions that are common behaviors among debug artifacts. These interfaces are implemented by debug elements where appropriate. They include the following actions:

- Disconnect (**IDisconnect**) − the ability to end a debug session with a target program and allow the target to continue running
- Step (**IStep**) − the ability to step into, over, and return from the current execution point
- Step filters (**IStepFilters**) − the ability to enable or disable filtering of step operations so that a debug target can apply filters to steps when appropriate
- Suspend and resume (**ISuspendResume**) − the ability to suspend and resume execution
- Terminate (**ITerminate**) − the ability to terminate an execution context
- Modify a value (**IValueModification**) − the ability to modify the value of a variable

If you look at the definitions for the platform debug elements, you will see that different elements implement different debug actions. Standard interfaces for the elements and their behaviors allow the platform to provide abstract implementations of utility classes and UI support that can be extended for concrete implementations of debuggers.

Events

Debug events (**DebugEvent**) are used to describe events that occur as a program is being debugged. Debug events include the debug element that is associated with the event. Each kind of debug element has a specific set of events that it supports as documented in **DebugEvent**. Debugger implementations can add application specific debug events using a designation of **DebugEvent.MODEL_SPECIFIC** as the kind of event. A client data field can be used in this case to add model−specific information about the event.

Debugger UI classes typically listen to specific events for elements in order display information about changes that occur in the elements. Debug events arrive in groups called **debug event sets**. Events that occur at the same point of execution in a program arrive in the same set. Clients should implement an **IDebugEventSetListener** and register the listener with the **org.eclipse.debug.core** plug−in in order to receive debug events.

Breakpoints

Breakpoints allow users to suspend the execution of a program at a particular location. Breakpoints are typically shown in the UI along with the source code. When a breakpoint is encountered during execution of a program, the program suspends and triggers a **SUSPEND** debug event with **BREAKPOINT** as the reason.

If your plug−in needs to show breakpoints in its UI, you can add an **IBreakpointListener** to the **IBreakpointManager**. The **IBreakpointManager** is the central authority over all breakpoints. Breakpoints are added and removed using the breakpoint manager, which in turn informs any listeners about breakpoint activity. The operation of breakpoints can be enabled or disabled using the breakpoint manager. The breakpoint manager can be obtained from the **DebugPlugin**:

```
IBreakpointManager mgr = DebugPlugin.getDefault().getBreakpointManager();
```

Plug−ins that define their own debug models and launch configurations often need to define their own breakpoint types. You can implement breakpoints for your particular debug model by defining a class that implements **IBreakpoint**.

Breakpoints are implemented using resource markers. Recall that resource markers allow you to associate meta information about a resource in the form of named attributes. By implementing a breakpoint using markers, the debug model can make use of all the existing marker function such as persistence, searching, adding, deleting, and displaying in editors.

Why is it important to know about markers when using breakpoints? When you create a breakpoint type, you must also specify an associated marker type. Every extension of **org.eclipse.debug.core.breakpoints** should be accompanied by an extension of **org.eclipse.core.resources.markers**. This is best demonstrated by looking at the extensions defined by the Java tooling for Java breakpoints.

```
<extension id="javaBreakpointMarker" point="org.eclipse.core.resources.markers">
        <super type="org.eclipse.debug.core.breakpointMarker"/>
</extension>

<extension id="javaExceptionBreakpointMarker" point="org.eclipse.core.resources.markers">
        <super type="org.eclipse.jdt.debug.javaBreakpointMarker"/>
        <persistent value="true"/>
        <attribute name="org.eclipse.jdt.debug.core.caught"/>
        <attribute name="org.eclipse.jdt.debug.core.uncaught"/>
        <attribute name="org.eclipse.jdt.debug.core.checked"/>
</extension>

<extension point="org.eclipse.debug.core.breakpoints">
        <breakpoint
                id="javaExceptionBreakpoint"
                markerType="org.eclipse.jdt.debug.javaExceptionBreakpointMarker"
                class="org.eclipse.jdt.internal.debug.core.breakpoints.JavaExceptionBreakpoint">
        </breakpoint>
</extension>
```

The debug plug−in defines a special type of marker, **org.eclipse.debug.core.breakpointMarker**. When you define a breakpoint marker for your debugger, you should declare it using this marker as a super type. This allows the debug model to find all possible breakpoints within a source file by searching for subtypes of its marker. In the example above, the **javaExceptionBreakpointMarker** has a super type, **javaBreakpointMarker**, whose super type is the **breakpointMarker**. The **javaExceptionBreakpoint** (defined in the breakpoint extension) designates the **javaExceptionBreakpointMarker** as its marker.

What does all of this mean? When the debug code obtains a source code file, it can search for all markers whose super type is **org.eclipse.debug.core.breakpointMarker**. Having found all of the markers, it can then use the extension registry to map the markers to their associated breakpoint classes. In this way, the platform debug code can generically find all breakpoint types that have been set on a particular source file.

Expressions

An **expression** is a snippet of code that can be evaluated to produce a value. The context for an expression depends on the particular debug model. Some expressions may need to be evaluated at a specific location in the program so that its variables can be referenced. **IExpression** defines a general interface for debug expressions.

An expression manager (**IExpressionManager**) keeps track of all of the expressions in the workspace. It will also fire events to interested listeners as expressions are added, removed, or changed.

Expressions can be used to implement "inspectors," or "scrapbooks" that let users evaluate code snippets. The Java tooling uses expressions to implement the expression generated when the user inspects the source code.

A **watch expression** is an expression that is repeatedly evaluated as the program executes. **IWatchExpression** defines a specialized kind of **IExpression** that updates the value of the expression when supplied with a new debug context. Watch expressions are used to implement "watch lists," which show changes in the value of an expression as the program executes.

Debug model presentation

Since there is a generic, uniform model for debug elements in the platform, it's possible to provide a starting point for implementing a debugger UI. The heart of the debugger UI support is the **debug model presentation** (**IDebugModelPresentation**). The debug model presentation is responsible for providing labels, images, and editors associated with specific debug elements.

Plug−ins that define their own debug model typically provide a debug model presentation for displaying debug elements in the model. This is accomplished using the **org.eclipse.debug.ui.debugModelPresentations** extension point. This extension point allows an implementation of **IDebugModelPresentation** to be associated with the identifier of a particular debug model.

Recall that debug model elements know the id of their originating debug model. This means that given any debug element, the debug platform can obtain the id of the debug model and then query the extension registry for any corresponding debug model presentations.

The markup for adding a debug model presentation looks like this:

```
<extension point = "org.eclipse.debug.ui.debugModelPresentations">
         <debugModelPresentation
             class = "org.eclipse.jdt.internal.debug.ui.JDIModelPresentation"
             id = "org.eclipse.jdt.debug"
             detailsViewerConfiguration = "org.eclipse.jdt.internal.debug.ui.display.DetailsViewe
          </debugModelPresentation>
</extension>
```

An optional **detailsViewerConfiguration** can be specified in addition to the debug model presentation. The details viewer must extend the JFace **SourceViewerConfiguration** class. The meaning of "details" is interpreted by the debug model. The details are computed by the debug model presentation and passed to the details viewer. For example, the Java debugger uses the details viewer to show code assist in the variables view when expressions are evaluated.

Implementors of **IDebugModelPresentation** may also implement **IDebugEditorPresentation** when more control is needed over the editor that is displaying source code for a stack frame. Implementors may control the editor's position or the annotations in the source.

Debug UI utility classes

In addition to defining a general framework for showing a debug UI, the debug UI plug−in includes classes

that implement useful utility methods for implementing a debugger UI.

DebugUITools groups many of these utilities and includes methods for the following:

- Saving and/or building the workspace before launching a particular launch configuration
- Storage and retrieval of images on behalf of debug UI clients
- Lookup of the debug model presentation associated with a given debug model
- Retrieval of the debug plug−in preference store
- Information about the current program under debug, such as the current context, process, or console
- Opening the launch configuration dialog with various defaults
- Enabling or disabling the use of step filters

IDebugView provides common function for debug views. It provides access to an underlying viewer and its debug model presentation. Typically, clients should extend **AbstractDebugView** rather than implement the interface from scratch. **AbstractDebugView** provides many useful functions:

- Storage of actions in an action registry
- Generic handling of the underlying viewer's context menu
- General implementations of delete key and double click function
- Mechanism for displaying an error message in the view

Platform Ant support

Ant is a Java−based build tool that uses XML−based configuration files to describe build tasks. The Eclipse platform allows you to run Ant buildfiles from your plug−in and contribute new Ant tasks, types and properties using extension points. The rest of this discussion assumes that you have a basic understanding of Ant.

Running Ant buildfiles programmatically

The Ant support built into Eclipse allows plug−ins to programmatically run Ant buildfiles. This is done via the **AntRunner** class included in the **org.eclipse.ant.core** plug−in.

The following code snippet shows an example of how to use the **AntRunner** from within code of another plug−in:

```
import org.eclipse.ant.core.AntRunner;
import org.eclipse.core.runtime.IProgressMonitor;

...

public void runBuild() {
        IProgressMonitor monitor = ...
        AntRunner runner = new AntRunner();
        runner.setBuildFileLocation("c:/buildfiles/build.xml");
        runner.setArguments("-Dmessage=Building -verbose");
        runner.run(monitor);
}
```

If a progress monitor is used, it is made available for the running tasks. See Progress Monitors for more details.

Note that only one Ant build can occur at any given time if the builds do not occur in separate VMs. See AntRunner.isBuildRunning();

Special care for native libraries if build occurs within the same JRE as the workspace

Every time an Ant buildfile runs in Eclipse a new classloader is created. Since a library can only be loaded by one classloader in Java, tasks making use of native libraries could run into problems during multiple buildfile runs. If the previous classloader has not been garbage collected at the time the new classloader tries to load the native library, an exception is thrown indicating the problem and the build fails. One way of avoiding this problem is having the library load be handled by a class inside a plug−in library. The task can make use of that class for accessing native methods. This way, the library is loaded by the plug−in classloader and it does not run into the load library conflict.

Ant tasks provided by the platform

The platform provides some useful Ant tasks and properties that interact with the workspace. They can be

used with buildfiles that are set to build within the same JRE as the workspace.

eclipse.refreshLocal

This task is a wrapper to the **IResource.refreshLocal()** method. Example:

```
<eclipse.refreshLocal resource="MyProject/MyFolder" depth="infinite"/>
```

- **resource** is a resource path relative to the workspace
- **depth** can be one of the following: **zero**, **one** or **infinite**

eclipse.incrementalBuild

When the **project** attribute is supplied, this task is a wrapper to **IProject.build()**. Otherwise, this task is a wrapper to the method: **IWorkspace.build()**. In both cases, the kind of build is always **IncrementalProjectBuilder#INCREMENTAL_BUILD**

Examples:

```
<eclipse.incrementalBuild/>
```

```
<eclipse.incrementalBuild project="MyProject"/>
```

- **project** the name of the project to be built

eclipse.convertPath

Converts a file system path to a resource path or vice−versa. The resulting value is assigned to the given property. The **property** attribute must be specified, as well as either the **fileSystemPath** or **resourcePath** attribute. When a file system path is supplied, this task is a wrapper to **IWorkspaceRoot.getContainerForLocation(IPath)**. When a resource path is supplied, this task is a wrapper to **IResource.getLocation()**.

Examples:

```
<eclipse.convertPath fileSystemPath="${basedir}" property="myPath"/>
```

```
<eclipse.convertPath resourcePath="MyProject/MyFile" property="myPath"/>
```

Contributing tasks and types

When your plug−in contributes Ant tasks and types, the tasks and types have access to all of the classes inside the contributing plug−in. For example, the **eclipse.refreshLocal** task contributed by **org.eclipse.core.resources** plug−in is a wrapper for the **IResource.refreshLocal()** method.

Tasks and types contributed by plug−ins must not be placed in any of the plug−in libraries. They have to be in a separate JAR. This means that the plug−in classes do not have access to the tasks and types provided by the

plug−in. (See Why a separate JAR for tasks and types? for more information.)

The **org.eclipse.ant.core.antTasks** extension point provides an example of how to specify a new task in the **plugin.xml** file.

Progress Monitors

The Eclipse Ant support provides access to an **IProgressMonitor** if one is passed when invoking the AntRunner. One of the advantages of having access to a progress monitor is that a long−running task can check to see if the user has requested its cancellation. The progress monitor object is obtained from the Ant project's references. Note that a monitor is only made available if the method **AntRunner.run(IProgressMonitor)** was called with a valid progress monitor. The following code snippet shows how to obtain a progress monitor from the task's project:

```
import org.apache.tools.ant.BuildException;
import org.apache.tools.ant.Task;
import org.eclipse.ant.core.AntCorePlugin;
import org.eclipse.core.runtime.IProgressMonitor;

public class CoolTask extends Task {

public void execute() throws BuildException {
        IProgressMonitor monitor =
                (IProgressMonitor) getProject().getReferences().get(AntCorePlugin.ECLIPSE_PROGRES
        if (monitor == null) {
                ...
        } else {
                ...
        }
}
}
```

Important rules when contributing tasks and types

The following should work as a checklist for plug−in developers:

- The JAR containing the tasks must not be a plug−in library (declared in <library></library>).
- The task or type can reference any class available for the plug−in but plug−in classes must not access the tasks or types.
- Native libraries should be loaded by the plug−in library classes and not tasks or types.

Why a separate JAR for tasks and types?

There are basically two requirements for running Ant in Eclipse that do not fit the plug−in model very well:

- Change the Ant classpath at runtime
- Change the Ant version at runtime

During runtime plug−in classloaders cannot have their classpaths expanded and plug−ins cannot change their dependencies. At the same time having separate JARs for the tasks and types is a good isolation from the plug−in classloading mechanism. Having these extra JARs declared by a plug−in permits adding the

contributing plug−in to the Ant classpath as well.

Developing Ant tasks and types within Eclipse

The following guidelines should be followed when developing and debugging Ant tasks and types within Eclipse. These requirements stem from the fact that Ant tasks and types must be loaded by the Ant classloader, rather than a plug−in classloader, when Ant is run in the same VM as Eclipse. To avoid having the Ant tasks and types loaded by a plug−in classloader, the tasks and types need to be stored in a location that is not visible to any plug−in classloader. Also see Contributing tasks and types.

- Contributed Ant tasks or types should be defined in their own source folder within a plug−in (i.e. separate from the source folders containing regular plug−in classes)
- Each source folder containing the Ant tasks and types should have its own output location that does not overlap with the output location of the regular plug−in classes.
- When testing/debugging the new Ant tasks or types, the project contributing the Ant tasks or types must be configured to **exclude** the output folders containing the Ant tasks and types. Use the **Properties** dialog for the project to correctly configure the **Self−Hosting** configuration by removing the Ant output directories from the plug−ins classpath.

Expanding the Ant classpath

Plug−ins can contribute extra JARs to the Ant classpath. The plug−in contributing the JARs is also added to the Ant classpath. As a consequence, classes inside the extra JARs have access to all classes available for the plug−in. These extra JARs cannot be plug−in libraries; they must be separate JARs. The consequences are that plug−in classes do not have access to the classes provided by these extra JARs. The **org.eclipse.ant.core.extraClasspathEntries** extension point provides an example of how to specify the extra JARs in the plugin.xml file.

Packaging and delivering Eclipse based products

The Eclipse platform is designed so you can add plug−ins that provide function for the software development tools community. Commercial software vendors can build, brand, and package products using the platform as a base technology. These products can be sold and supported commercially.

The Eclipse SDK can be downloaded and used as a Java IDE and Eclipse plug−in development tool, but it is not marketed as a commercial product. The platform provides the raw ingredients for a product without a box, label, or price tag. It defines a file and directory structure that lets you easily customize the platform's about dialog and splash screen to brand your product.

The license governing the Eclipse platform gives you a lot of freedom over how to build and configure a product. However, Eclipse based products will coexist more easily on a user's system if the products use similar standards for packaging, configuring, and installing their products.

Customizing a primary feature

Product customization works differently when using the primary feature mechanism. The branding information for the feature is located in a plug−in identified by the primary feature (or the plug−in of the same name as the primary feature if none is specified). The files that designate and define branding information for our hypothetical acmeweb application are highlighted in the sample directory structure below:

```
acmeweb/
acmeweb.exe   (product executable − invokes eclipse.exe and specifies the primary feature)
        eclipse/
                .eclipseproduct
                eclipse.exe
                startup.jar
                install.ini
                .config/
                platform.cfg
                jre/
                features/
com.example.acme.acmefeature_1.0.0/     (primary feature)
                                feature.xml
                plugins/
com.example.acme.acmefeature_1.0.0/     (plug−in for primary feature.  Contains branding info.)
                                plugin.xml
about.ini
                                about.html
                                about.mappings
                                about.properties
                                acme.png
                                plugin_customization.ini
                                splash.jpg
                                welcome.xml
                        com.example.acme.acmewebsupport_1.0.0/
                        ...
                links/
                        ...
```

The plug−in associated with a primary feature is where the branding information for a product is specified. There are many customizable aspects of a product. Product−level customizations are defined using the **about.ini** file and other files described therein. Products can also control the default preference values of other plug−ins. This is done using the **plugin_customization.ini** file.

Referring once again to our sample product's primary feature plug−in, let's look closer at how the product is customized.

```
com.example.acme.acmefeature_1.0.0/
                plugin.xml
about.ini
                about.html
                about.mappings
                about.properties
                acme.png
                plugin_customization.ini
                plugin_customization.properties
                splash.bmp
                welcome.xml
```

We'll review the same customizations that we reviewed for the products extension point, focusing on how the specification differs using the primary feature mechanism.

About dialogs

As discussed with the product−level customization, all features and plug−ins should contribute an **about.html** file that provides information about that particular plug−in.

The primary feature also supplies the information and graphics for the overall product. Additional files are used to specify this information.

- **about.ini** specifies the about text and images for features, windows, and the about dialog itself. It also specifies the welcome page. See Customization using about.ini for a complete description of the format of this file.
- **about.properties** should be used to hold translated strings from the **about.ini** file (using "%var" as the value in about.ini paired with an entry for the key "var" in about.properties. This file is a **java.io.Properties** format file.
- **about.mappings** can be used to fill in values for fill−in fields in the about text. This is useful when the about text contains information specific to a particular install, such as license key, install date, or licensed user. For example, the about text could be defined as "AcmeWeb is licensed to {0}". The about.mappings file could be generated at install time based on input from the user. The final form should contain a mapping for the field, such as "0=Joe Q. Webuser". The about.mappings file is a **java.io.Properties** format file.

Window images

A 16x16 pixel color image can be used to brand windows created by the product. It will appear in the upper left hand corner of product windows. It is specified in the **windowImage** attribute in the **about.ini** file. The path should be specified as a plug−in relative path. A proper entry for the directory structure shown above would be as follows:

```
windowImage=acme.png
```

Welcome page

Plug−ins using the Eclipse 2.1 Welcome mechanisms should specify the welcome page file in the welcomePage attribute in the about.ini file. The path should be specified as a plug−in relative path. A proper

entry for the directory structure shown above would be as follows:

```
welcomePage=welcome.xml
```

You can also specify an national language lookup for the file. (See Locale specific files for more detail.)

```
welcomePage=$nl$/welcome.xml
```

Splash screens

The product splash screen is supplied in a **splash.bmp** file located in the primary feature plug−in directory. The image should be supplied in 24−bit color BMP format (RGB format) and should be approximately 500x330 pixels in size. If splash screens need to be customized for different locales, they can be placed in a fragment of the primary feature's plug−in.

Preferences defaults

The **plugin_customization.ini** file is used to set the default preference values for preferences defined by other plug−ins. This file is a **java.io.Properties** format file. Typically this file is used to set the values for preferences that are published as part of a plug−in's public API. That is, you are taking a risk if you refer to preferences that are used by plug−ins but not defined formally in the API.

One common customization is to set the default perspective for the workbench. This preference is defined in the **org.eclipse.ui** plug−in. The following example assumes that the product should be launched with the resource perspective as the default perspective.

```
org.eclipse.ui/defaultPerspectiveId=org.eclipse.ui.resourcePerspective
```

If you discover you need to change the default value for one of another plug−in's preferences, consult the API documentation for that plug−in to see if the preference is considered public.

The **plugin_customization.properties** file contains translated strings for the **plugin_customization.ini** file.

About.ini File Format

Last revised March 25, 2003 for Eclipse 2.1

A feature's "about" information is given by properties in the "`about.ini`" file located within the feature's *plug−in*. This is a java.io.Properties format file (ISO 8859−1 character encoding with \u*xxxx* Unicode escapes), with the following keys:

- "`aboutText`" − A multi−line text description of the feature containing copyright, license terms, license registration info, etc. URLs found in text are presented as links. Line breaks are indicated by "`\n`"; lines should be no longer than 75 characters each, and there is a limit of 15 lines. This information is shown in the main "about" dialog (used in conjunction with a half−sized image), and on the secondary "about" dialog describing individuals feature. All features are required to have this property.
- "`windowImage`" − A 16x16 pixel color image used to brand windows created by a product. The value of this property is a plug−in−relative path to an image file. This property is only required for primary features; other features may omit it.
- "`featureImage`" − A 32x32 pixel color image used as an icon for this feature. The value of this property is a plug−in−relative path to an image file. This image is shown on the "about" dialog describing individual features, and is also used on the main "about" dialog to indicate that the feature is present. This property is optional. Related features supplied by a single provider should contain copies of the identical image file; this allows duplicate images to be eliminated.
- "`aboutImage`" − A large color image that appears in the main "about" dialog for a product. The value of this property is a plug−in−relative path to an image file. This property is only required for primary features; other features should omit it. Both graphic−and−text, and graphic only layouts are supported, depending on the size of image provided. A half−size image of up to 250 pixels wide by 330 pixels high is shown on the main "about" dialog with text ("`aboutText`" property) beside it. A full−size image of up to 500 pixels wide by 330 pixels high is shown without text.
- "`welcomePage`" − A welcome page. The value of this property is a plug−in−relative path to an XML−format file containing the feature's welcome page. This property is recommended for all major features; other features may omit it.
- "`welcomePerspective`" − The preferred perspective in which to show the welcome page. The value of this property is a workbench perspective id. This property is recommended for features with welcome pages that show better in the context of the feature's views and editors. Features that do not have welcome pages, or have no strong opinion where the welcome pages is shown, should omit it.
- "`appName`" − A short, non−translatable string used to associate a name with the application on platforms where this makes sense. In operating environments with the Motif window system, the value is used for Motif resource lookup (see xrdb and related). In addition, on the CDE window manager (used on Solaris and AIX), this is the name which shows up under the icon for windows which are minimized to the desktop. This property is required for primary features; other features may omit it.
- "`tipsAndTricksHref`" − Tips and tricks section of help documentation. The value of this property is a reference to a help book containing the feature's tips and trick section; for example, `tipsAndTricksHref=/org.eclipse.jdt.doc.user/tips/jdt_tips.html`. This property is recommended for all major features; other features may omit it.

Locale specific files

Fragments are a convenient way to package national language translations. Let's look more closely at the

directory structure used for installing locale−specific translation files. This directory structure is used regardless of whether the translated files are packaged in a fragment or delivered in the original plug−in.

There are three mechanisms for locating locale specific files in a plug−in.

- **Platform core mechanism** (the platform's runtime locale−specific sub−directory search)
- **Java resource bundles** (java.util.ResourceBundle)
- **The plugin.properties mechanism** (Translating values from the plugin.xml files)

It is important to understand which mechanism is used to access any given file that must be translated so that you'll know what to name the file and where to put it in the file system relative to the plug−in.

Platform core mechanism

The platform core defines a directory structure that uses locale−specific subdirectories for files that differ by locale. Translated files are placed in a directory called **nl** under the plug−in. For example, the following install tree shows a trivial (no code) plug−in with locale−specific translations of its **about.properties** file. The various translations are shown as coming from a plug−in fragment rather than the plug−in itself. This is typical for shipping translations separately from the base, but you could also place the **nl** sub−directory under the plug−in itself.

```
acmeweb/
  eclipse/
    plugins/
      com.example.acme.acmewebsupport_1.0.0/
        plugin.xml
        about.properties    (default locale)
      com.example.acme.fragmentofacmewebsupport_1.0.0/
        fragment.xml   (a fragment of com.example.acme.acmewebsupport 1.0.0)
        nl/
          fr/
            about.properties  (French locale)
            CA/
              about.properties  (French Canadian locale)
            FR/
              EURO/
                about.properties (French France Euros)
          en/
            about.properties  (English locale)
            CA/
              about.properties  (English Canadian locale)
            US/
              about.properties (English US locale)
         de/
            about.properties (German locale)
```

The files to be translated are not contained in JAR files. Each file should have exactly the same file name, but be located in subdirectories underneath the **nl** sub−directory in the fragment's (or plug−in's) root.

Only the most specific file is accessed at runtime. The file paths are searched as part of the **Platform.find**, **IPluginDescriptor.find** and **Plugin.find** mechanism. For example, suppose the default locale is **en_CA**, and a plug−in searches for the **about.properties** as follows:

```
somePlugin.find("$nl$/about.properties");
```

The method will return a URL corresponding to the first place **about.properties** is found according to the following order:

```
com.example.acme.acmewebsupport_1.0.0/nl/en/CA/about.properties
com.example.acme.fragmentofacmewebsupport_1.0.0/nl/en/CA/about.properties
 ...            <any other fragments>
com.example.acme.acmewebsupport_1.0.0/nl/en/about.properties
com.example.acme.fragmentofacmewebsupport_1.0.0/nl/en/about.properties
 ...
com.example.acme.acmewebsupport_1.0.0/about.properties
com.example.acme.fragmentofacmewebsupport_1.0.0/about.properties
```

This mechanism is used by plug−ins to search for well known file names inside other plug−ins. This includes the following well known file names:

- **preferences.properties** (externalized strings for plug−in −specific preference default overrides)
- **about.properties** (externalized strings for feature "about" information)
- **plugin_customization.properties** (externalized strings for product−specific preference default overrides)
- **splash.bmp** (product−specific splash screens)

*(Note: The **plugin.properties** and **fragment.properties** are conspicuously absent from this list. They are treated in a sightly different way described below.)*

Java resource bundles

The standard Java handling of property resource bundles is used for other files. Translated files are contained in a JAR file, with each properties file having a locale−specific name, such as "**message_en_CA.properties**". The files are in package−specific subdirectories and may appear in the plug−in itself or one of its fragments. Each translated properties file may be partial since lookup of keys accesses a well−defined chain of properties files.

The plugin.properties mechanism

The mechanism used to translate plugin.properties files uses the Java resource bundles naming convention. However the files must be located in the root of the plug−in or in the root of a fragment of this plug−in. The same rules apply to the translation of MANIFEST.MF.

Defining NL fragments

The shape of NL fragments has evolved slightly since 2.1. Previously all translation files (including the plugin.properties) were provided in a jar. This was inconsistent since the plugin.properties file was provided at the root of the plug−in.
To adapt your NL fragment to the new model, remove the plugin.properties translation files from the jar and put them at the root of the fragment as siblings of fragment.xml. For example, the new shape of the NL fragment for org.eclipse.ui.workbench is the following:

```
org.eclipse.ui.workbench.nl/
   fragment.xml
   plugin_fr.properties
   plugin_pt_BR.properties
   ...
   nl1.jar
```

Plug−ins and fragments

Features are described in terms of the plug−ins that comprise them. This means that plug−ins are the fundamental unit for packaging function.

While features are organized for the purposes of distributing and updating products, plug−ins are organized to facilitate the development of the product function among the product team. The development team determines when to carve up program function into a separate plug−in.

Plug−ins are packaged in a plug−in archive file and described using a plug−in manifest file, **plugin.xml**.

Plug−in **fragments** are separately packaged files whose contents are treated as if they were in the original plug−in archive file. They are useful for adding plug−in functionality, such as additional national language translations, to an existing plug−in after it has been installed. Fragments are ideal for shipping function that may trail the initial product release, since they can be used to add function without repackaging or reinstalling the original plug−in. When a fragment is detected by the platform, its contents are merged with the function in the original plug−in. In other words, if you query the platform plug−in registry, you would see the extensions and other function provided in the fragment as if it was in the original plug−in.

Fragments are described using a fragment manifest file, **fragment.xml.** It is similar to the plug−in manifest file. Since a fragment shares many properties with its plug−in, some attributes in the plug−in manifest are not valid, such as the plug−in class and plug−in imports.

Plug−in archive files can contain plug−ins or fragments.

Plug−in Archives

Plug−ins and plug−in fragments are individually packaged as separate Java .jars. Standard Java jar facilities are used for constructing plug−in archives. There is no distinction made between a plug−in archive containing a plug−in and one containing a plug−in fragment.

The recommended convention for naming the plug−in archives is
`<id>_<version>.jar`

Where `<id>` is the plug−in or fragment identifier and `<version>` is the full version identifier contained in the respective plugin.xml or fragment.xml. Note that this is a recommended convention that minimizes chance of collisions, but is not required by the Eclipse architecture. For example, the following are valid plug−in archive names:

```
org.eclipse.platform_1.0.3.jar
org.eclipse.ui.nl_2.0.jar
my_plugin.jar
```

Internally, each plug−in archive packages all the relevant plug−in or fragment files relative to its plug−in or fragment directory (but not including the directory path element). The archive has the following structure

```
plugin.xml *OR* fragment.xml
other plug-in or fragment files and subdirectories
META-INF/
    Java jar manifest and security files
```

Product installation guidelines

The platform provides standard tools for updating and extending products. In order to participate in the platform mechanisms for updating and extending products, your packaged product should follow the following guidelines. This will allow your product to peacefully coexist with, or even enhance, other Eclipse based products.

Consider again the sample directory structure for the acmeweb product:

```
acmeweb/
        acmeweb.exe
        eclipse/
                .eclipseproduct
                eclipse.exe
                startup.jar
                install.ini
                .config/
                        platform.cfg
                jre/
                features/
                        com.example.acme.acmefeature_1.0.0/
                                feature.xml
                        ...
                plugins/
                        com.example.acme.acmefeature_1.0.0/
                                plugin.xml
                                about.ini
```

```
                about.html
                about.mappings
                about.properties
                acme.png
                plugin_customization.ini
                splash.jpg
                welcome.xml
        com.example.acme.acmewebsupport_1.0.0/
        ...
links/
        ...
```

Where did these files come from? Let's look at the product content from the perspective of the development team. The installed files can be grouped into five main categories:

- top−level product files (such as the acmeweb.exe)
- product features and plug−ins
- the Eclipse platform itself
- the Java runtime environment (JRE)
- files generated by the installation process itself

A proper installation script will produce the appropriate directory structure by doing the following:

- allow the user to specify the top level directory of the install (such as **c:\acmeweb.** We will refer to it as **acmeweb** for the remaining steps.)
- ensure that a product is not already installed in the location
- copy the files as follows:
  - Top−level product files are copied to **acmeweb**
  - Eclipse files are copied to **acmeweb/eclipse** using the expected feature and plugin directory structures
  - JRE files are copied to **acmeweb/eclipse/jre**. If a JRE is already located elsewhere, then the application shortcut should be setup to invoke eclipse with the **−vm** command line argument so that the location of the JRE is known by the platform
  - Product features and plug−ins are copied to **acmeweb/eclipse/features** and **acmeweb/eclipse/plugins**
- create a marker file, **.eclipseproduct**, in **acmeweb/eclipse**. The marker file is a **java.io.Properties** format file that indicates the name, id, and version of the product.
- store any necessary install info (user, license, date) that is to be shown in the about dialog in **acmeweb/eclipse/plugins/com.example.acmefeature_1.0.0/about.mappings**
- if the primary feature mechanism (pre R3.0) is used to define the product, replace the **acmeweb/eclipse/install.ini** with one that sets the property **feature.default.id** to the product's primary feature
- invoke the product executable using the **−initialize** option. This causes the platform to quietly perform all time−consuming first−time processing and cache the results, so that when the user starts the product it comes up promptly in an open−for−business state.

Multi−user issues

When a product is installed with the intention of sharing the installation among multiple users, care must be taken to separate individual user's data (such as the **workspace** directory) from the shared product install directory.

Uninstall issues

When a product is uninstalled, the following concepts are important.

- **all** files in the **eclipse/features** and **eclipse/plugins** directories should be removed, even if they weren't originally installed by the installation program. This ensures that files added by the platform update manager are removed when the product is removed.
- except for the point above, **only** those files installed by the installation program in other directories should be removed. It is important to preserve any important data, such as the **workspace** directory, so that reinstalling the product after an uninstall will produce expected results.

Reinstalling the product

When the product is already installed, the installer should allow a service update or upgrade if one is available. The existence of the product can be confirmed by looking for **acmeweb/eclipse/.eclipseproduct**. The information in this marker file can be used to confirm with the user that the correct product is being updated. The availability of updates can be confirmed with pattern matches against feature names. For example, the presence of **acmeweb/eclipse/plugins/com.example.acmefeature_1.0.1** would indicate that the 1.0.1 version update had already been applied to the product.

Once the validity of the reinstall is established, the install program should copy or replace files as needed. If the version of the underlying Eclipse platform has not changed, there is a good chance that complete directories can be ignored. The version numbers appended to the platform features and plugins can be used to determine whether any changes underneath a plug−in or feature's directory are necessary.

Additional information on installing products can be found in How to write an Eclipse installer.

How to write an Eclipse installer

Last modified 15:20 Friday June 18, 2004

Eclipse−based products need to be correctly installed on the end user's computer. Special−purpose packaging tools, such as InstallShield and RPM, are often used to build executable installers that automate installing, updating, and uninstalling. This note describes how to write an installer for an Eclipse−based product, and for separately−installable extensions to Eclipse−based products.

We assume that a product development team is responsible for providing the raw ingredients that will need to find their way to end users' computers packaged as an executable installer. The creation of executable installers is scripted, as are the install time actions needed to interact with the end user and deposit files on their computer. This note described in detail what these installers need to do and how they should work.

This note should be treated as a recipe for the person responsible for writing an installer for an Eclipse−based products. Two good reasons why we recommend all installers writers follow our recipe:

- **Product and extension interoperability.** By behaving in standard ways, an installer for one Eclipse−based product or extension automatically works with products and extensions laid down by other installers. Otherwise the idiosyncrasies of one product's installer would require matching quirks in all extension installers that expected to work with that product.
- **Uniformity of install time user interaction.** All installers for Eclipse−based products and extension should interact with the user in the same manner. There is nothing to having gratuitous variety in this matter.

Product installer creation script

A product installer should be self−contained − the kind of thing that could be distributed on a CD and installed on any machine with a suitable operating system.

Eclipse requires a Java2 Java Runtime Environment (JRE) to run Java code. JREs are licensed software, obtained from Java vendors. With a license to redistribute a JRE from a JRE vendor, a company can include a JRE with its product, and install it on the end user's computer at the same time as the product. The alternative is to require that a JRE be pre−installed on the end user's computer, and associated with at product install time. One way or the other, an Eclipse−based product requires a suitable JRE, and the product installer must play a role in either installing a JRE or locating and linking to a pre−existing JRE.

Assume that a JRE is to be installed with the product. A directory containing the JRE is one input to the installer creation script. Denote this directory <*JRE*>. This directory must have a standard JRE directory structure, with the Java executable is located at `jre/bin/java.exe` and the class library at `jre/lib/rt.jar` below the <*JRE*> directory. For reference, the skeletal structure of this directory looks like:

```
<JRE>/
  jre/
    bin/
      java.exe
    lib/
      rt.jar
```

There are additional files (and subdirectories) in these directories; we've only shown a sample to give the general structure. Italicized names in italics are product−specific.

The second input to the installer creation script is a directory, <*product head*>, containing the product−specific executable launcher and any files unrelated to Eclipse. For reference, the skeletal structure of this directory would look like (italics indicate file names that will vary from product to product):

```
<product head>/
  acmeproduct.exe
```

The third input to the installer creation script is a directory, <*product body*>, containing the features and plug−ins developed for the product. For reference, the skeletal structure of this directory would look like:

```
<product body>/
  eclipse/
    features/
      com.example.acme.acmefeature_1.0.0/
        feature.xml
      com.example.acme.otherfeature_1.0.0/
        feature.xml
    plugins/
      com.example.acme.acmefeature_1.0.0/
        plugin.xml
        about.ini
        about.properties
        about.mappings
        plugin_customization.ini
        splash.bmp
      com.example.acme.otherfeature_1.0.0/
        plugin.xml
        about.ini
        about.properties
        about.mappings
      com.example.acme.myplugin_1.0.0/
        plugin.xml
        myplugin.jar
      com.example.acme.otherplugin_1.0.0/
        plugin.xml
        otherplugin.jar
```

The fourth input to the installer creation script is a directory, <*platform*>, containing the features and plug−ins for the Eclipse platform itself and any third−party tools being included. This directory also includes the standard Eclipse executable launcher, `eclipse.exe`, (named `eclipse` on Unix operating environment), its companion `startup.jar`, and any other Eclipse platform files required to be at the root of the install. For reference, the skeletal structure of this directory would look like:

```
<platform>
  eclipse/
    eclipse.exe
    startup.jar
    features/
```

```
    org.eclipse.platform_2.0.0/
    org.eclipse.platform.win32_2.0.0/
    org.eclipse.jdt_2.0.0/
    org.eclipse.pde_2.0.0/
  plugins/
    org.eclipse.platform_2.0.0/
    org.eclipse.core.runtime_2.0.0/
    org.eclipse.core.boot_2.0.0/
    org.eclipse.core.resources_2.0.0/
    org.eclipse.ui_2.0.0/
    org.eclipse.jdt_2.0.0/
    org.eclipse.jdt.core_2.0.0/
    org.eclipse.jdt.ui_2.0.0/
    org.eclipse.pde_2.0.0/
    org.eclipse.pde.core_2.0.0/
    org.eclipse.pde.ui_2.0.0/
    (more org.eclipse.* plug-in directories)
```

The exact contents of the <*JRE*>, <*product head*>, <*product body*>, and <*platform*> input directories determine what files will eventually be installed on the end user's computer.

The final inputs to the installer creation script are the id and version strings for the product's primary feature; e.g., "*com.example.acme.acmefeature*", and "*1.0.0*"; and the name of the product executable; e.g., "*acmeproduct*.exe". For products that do not require their own product executable, this would be the path of the standard Eclipse executable launcher "eclipse/eclipse.exe". These strings have special significance to the installer, appearing in file and directory names, and in the contents of marker files created at install time.

At install time, the installer should behave in the standard manner (further details follow the list of steps):

1. warn user to exit all programs
2. introduce the product to be installed
3. if appropriate, ask the user for the name of the registered owner and for the license key
4. display the product's licensing agreement and ask the user to accept
5. recommend a location on the disk to install the product (but allow user to override this default)
6. check that a product or extension is not already stored at the specified location
7. ask user to confirm all details of the install
8. create marker file to mark root of product install
9. copy files to disk (see below)
10. if appropriate, insert name of registered owner and license key into the "about " description
11. create a desktop shortcut to run the product executable
12. create an appropriate entry to allow the user to uninstall the product
13. launch the product executable with –initialize option to perform all first–time processing
14. offer to show the product release notes ("readme" file)

If the location specified in step 5 is <*install*>, the installer copies all the files in the <*JRE*>, <*platform*>, <*product*>, and <*product plug-ins*> directories into <*install*>.

Input file	Installed file
<*JRE*>/*	<*install*>/eclipse/*

<product head>/*	*<install>*/*
<product body>/*	*<install>*/*
<platform>/*	*<install>*/*

The marker file created in step 8 is *<install>*/eclipse/.eclipseproduct is used to mark a directory into which an Eclipse−based product has been installed, primarily for extension installers to locate. This marker file is a java.io.Properties format file (ISO 8859−1 character encoding with "\" escaping) and contains the following information that identifies the product to the user and distinguishes one Eclipse−based product from one another:

```
name=Acme Visual Tools Pro
id=com.example.acme.acmefeature
version=1.0.0
```

The values of the "id" and "version" property are inputs to the installer creation script; the name of the product is presumably known and hard−wired. (Products would not ordinarily access this marker file; only product and extension installers write or read it.)

Step 6 requires checking for an existing *<install>*/eclipse/.eclipseproduct or *<install>*/eclipse/.eclipseextension file. A product cannot be installed in exactly the same place as another product or extension.

After installing all files, the top−level structure of the install directory would contain the following files and subdirectories (and perhaps others):

```
<install>/
  acmeproduct.exe
  eclipse/
    .eclipseproduct
    eclipse.exe
    startup.jar
    features/
    plugins/
    jre/
```

If a product installer solicits license information from the user, such as the name of the registered owner and the license key, this information should make it into the product "about" dialog (step 10).

This is done by recording the user responses in the "about.mapping" file in the primary feature's plug−in. For example, at *<install>*/plugins/*com.example.acme.acmefeature_1.0.0*/about.mapping. The "about.mapping" file may be pre−existing in the <*product head*> input, or may need to be created by the installer at install time. The keys are numbers; the value of the "*n*" key is substituted for the substring "{*n*}" in the "aboutText" property. For example, if a license key was field number 0, an "about.mapping" file containing a line like "0=T42−24T−ME4U−U4ME" should be created.

N.B. The "about.mapping" file is a java.io.Properties format file (ISO 8859−1 character encoding with "\" escaping). When the native character encoding at install time is different from ISO 8859−1, the installer is responsible for converting the native character encoding to Unicode and for adding "\" escapes where required. Escaping is required when the strings contain special characters (such as "\") or non−Latin characters. For example, field number 1 containing the first 3 letters of the Greek alphabet would be written

"1=\u03B1\u03B2\u03B3".

At step 12, the product installer launches the product executable, <*install*>/*acmeproduct*.exe, with the special –initialize option [exact details TBD]. This causes the Eclipse platform to quietly perform all time–consuming first–time processing and cache the results, so that when the user starts the product it comes up promptly in an open–for–business state.

Uninstaller behavior

At uninstall time, the uninstaller should behave in the standard manner:

1. warn user to exit all programs, especially the product being uninstalled
2. ask user to confirm that the product is to be uninstalled
3. remove all installed files from the <*install*> directory, and **all** files in <*install*>/eclipse/features and <*install*>/eclipse/plugins including ones put there by parties other than this installer (e.g., by the Eclipse update manager)
4. remove desktop shortcut for the product executable
5. remove entry for product uninstaller
6. inform user of any files that were not removed

When the product is uninstalled, files deposited at install time should be deleted, along with updated features and plug–ins created by the Eclipse update manager. **Important:** At uninstall time, there may be other directories and files in the <*install*> directory, notably <*install*>/eclipse/workspace/, <*install*>/eclipse/links/, and <*install*>/eclipse/configuration/, that contain important data which must be retained when the product is uninstalled. The user must be able to uninstall and reinstall a product at the same location without losing important data.

Installer behavior when product already installed

When the product is already installed on the user's computer, the installer should allow a service update or version upgrade to be applied to the installed product.

At install time, the installer should behave in the standard manner:

1. warn user to exit all programs, especially the product being updated
2. locate the installed product to be updated, if necessary by searching the disk for an existing product install or by allowing the user to locate it
3. determine where this installer is a compatible update
4. if appropriate, ask the user for the name of the registered owner and for the license key
5. display the product's updated licensing agreement and ask the user to accept
6. ask user to confirm all details of the update
7. update files to disk (see below)
8. if required, alter the desktop shortcut to run the product executable
9. should add modified or newly added files to the list of ones to be removed at uninstall time (where feasible)
10. offer to show the product release notes ("readme" file)

In step 2, an installed product can be recognized by the presence of an "eclipse" directory immediately containing a file named ".eclipseproduct". The parent of the "eclipse" directory is a product's install directory; i.e., <*install*>/eclipse/.eclipseproduct. The information contained within this

marker file should be shown to the user for confirmation that the correct product is being updated (there may be several Eclipse−based product on the user's computer).

The installer should perform compatibility checks in step 3 by simple pattern matching against subdirectories in the `<install>/eclipse/features` directory. For example, the presence of a folder matching "*com.example.acme.otherfeature_1.0.1*" would ensure that a certain service update had been applied to the installed product.

For step 7, the installer may delete or replace any of the files that it originally installed, and add more files. **Important:** Several files and directories, including `<install>/eclipse/workspace/`, `<install>/eclipse/configuration`, may be co−located with the install and contain important data files which need to be retained when the product is upgraded.

In upgrade situations, there is a good chance that most of the files below `<install>/eclipse/plugins/` are the same (likewise for `<install>/eclipse/features/`). There is significant opportunity for optimization in `<install>/eclipse/plugins/` since the sub−directory name, which embeds the plug−in (or fragment) version number, changes if and only iff any of the files below it change. In other words, there is no need to touch any files in `<install>/eclipse/plugins/org.eclipse.ui_2.0.0/` if this sub−directory should also exist after the upgrade; if any of the plug−in's files were to change, the plug−in's version number is revised, causing the files for the upgraded plug−in to be installed in a parallel directory `<install>/eclipse/plugins/org.eclipse.ui_2.0.1/`.

Associating a JRE installed elsewhere

The JRE is expected to be located at `<install>/eclipse/jre/bin/javaw.exe`. If it is located elsewhere, the absolute path should be specified using the `-vm` option on the command line; e.g., `-vm C:\j2jre1.3.0\jre\bin\javaw.exe`. In which case, the installer should add this option to the command line of the desktop shortcut it creates.

Extension installer creation script

By extension we mean a separately installable set of features and their plug−ins that can be associated with, and used from, one ore more Eclipse−based products installed on the same computer. In contrast to a product, an extension is not self−contained; an extension does not include a product executable, the Eclipse platform, a JRE.

Without loss of generality, assume that an extension consists of a single feature. The first input to the installer creation script is a directory, **<***extension***>**, containing its feature and plug−ins. We are assuming that an extension has no files that are related to Eclipse; if it did, they would go in `<extension>/`, and not in `<extension>/eclipse/`. For reference, the skeletal structure of this directory would look like:

```
<extension>/
  eclipse/
    features/
      com.example.wiley.anvilfeature_1.0.0/
        feature.xml
    plugins/
      com.example.wiley.anvilfeature_1.0.0/
        plugin.xml
```

```
    about.ini
    about.properties
    about.mappings
com.example.wiley.mainplugin_1.0.0/
com.example.wiley.otherplugin_1.0.0/
```

The exact contents of the <*extension*> input directory determines what files will eventually be installed on the end user's computer.

The final inputs to the installer creation script are the id and version strings for the extension's feature; e.g., "*com.example.wiley.anvil*" and "*1.0.0*". These strings have special significance to the installer, appearing in file and directory names, and in the contents of marker files created at install time.

An extension installer is similar to a product installer in most respects. The areas where it differs are highlighted below:

At install time, the installer behaves in the standard manner:

1. warn user to exit all programs
2. introduce the extension to be installed
3. if appropriate, ask the user for the name of the registered owner and for the license key
4. display the extension's licensing agreement and ask the user to accept
5. recommend a location on the disk to install the extension (but allow user to override this default)
6. check that a product or a different extension is not already stored at the specified location
7. ask user which product(s) to are to use this extension (search disk; browse; or skip)
8. optionally, determine if extension is compatible with selected products
9. ask user to confirm all details of the install
10. create marker file to mark root of extension install
11. copy files to disk (see below)
12. insert name of registered owner and license key into the "about " description
13. create an appropriate entry to allow the user to uninstall the extension
14. write link file in each of the selected products to associate extension with product
15. offer to show the extension release notes ("readme" file)

If the location specified in step 5 is <*install*>, the installer copies all the files in the <*extension*> directory into <*install*> in step 11.

Input file	Installed file
<*extension*>/*	<*install*>/*

For step 7, any Eclipse product might be a candidate. Eclipse–based product can be recognized by the presence of a <*product install*>/eclipse/.eclipseproduct file; the user should be able to request a limited disk search for installed products (a "search for installed products" button), or would navigate to a directory containing a product (i.e., a "browse" button).

The installer should perform compatibility checks in step 8 by simple pattern matching against subdirectories in the <*product install*>/eclipse/features directory. For example, the presence of a folder matching "org.eclipse.jdt_2.*" means that JDT is included in the installed product.

The marker file created in step 10 is <*install*>/eclipse/.eclipseextension is used to mark a directory into which an Eclipse–based extension has been installed, primarily for extension installers to locate

(analogous to a product's `.eclipseproduct` marker file). This marker file is a java.io.Properties format file (ISO 8859−1 character encoding with "\" escaping) and contains the following information that identifies the extension to the user and distinguishes one Eclipse−based extension from one another:

```
name=Wiley Anvil Enterprise Edition
id=com.example.wiley.anvilfeature
version=1.0.0
```

The values of the "id" and "version" property are inputs to the installer creation script; the name of the extension is presumably known and hard−wired. (Products would not ordinarily access this marker file; only product and extension installers write or read it.)

After installing all files, the top−level structure of the install directory would contain the following files and subdirectories:

```
<install>/
  eclipse/
    .eclipseextension
    features/
    plugins/
```

The only significant difference from a product installer is that an extension installer also creates link files in other Eclipse−based products already installed on the user's computer. (This saves the user from having to manually associate the new extension from within each product using the Eclipse update manager.)

The link file created in step 14 is `<product install>/eclipse/links/com.example.wiley.anvilfeature.link`; that is, the file has the same name of as the extension's feature directory less the version number suffix. A link file is a java.io.Properties format file (ISO 8859−1 character encoding with "\" escaping). The key is "path" and the value is the absolute path of the installed extension, `<install>`; e.g., an entry might looks like `"path=C:\\Program Files\\Wiley\\Anvil".` The installer is responsible for converting from native character encoding to Unicode and adding "\" escapes where required. Escaping is usually required since `<install>` typically contains special characters (such as "\") and may mention directories with non−Latin characters in their names. The product reads link files when it starts up. The installer keeps a record of any link files it creates so that they can be located when the extension is updated or uninstalled.

Uninstaller behavior

At un install time, the un installer should behave in the standard manner:

1. warn user to exit all programs, especially products using the extension being uninstalled
2. ask user to confirm that the extension is to be un installed
3. remove all installed files from the <*install*> directory, and **all** files in <*install*>`/eclipse/features` and <*install*>`/eclipse/plugins` including ones put there by parties other than this installer (e.g., by the Eclipse update manager)
4. if feasible, remove the link file from any products to which it had been added
5. remove entry for extension uninstaller
6. inform user of any files that were not removed

When an extension is uninstalled, all plug−in and feature files should be deleted; there are no important data

files to be kept in these subdirectories. This allows the user to uninstall an extension completely, including any updates applied by the Eclipse update manager.

Installer behavior when extension already installed

When the extension is already installed on the user's computer, the installer should allow a service update or version upgrade to be applied to the installed extension.

At install time, the installer should behave in the standard manner:

1. warn user to exit all programs, especially products using the extension being updated
2. locate the installed extension to be updated, if necessary by searching the disk for an existing extension install or by allowing the user to locate it
3. determine where this installer is a compatible update
4. if appropriate, ask the user for the name of the registered owner and for the license key
5. display the product's updated licensing agreement and ask the user to accept
6. ask user to confirm all details of the update
7. update files on disk (see below)
8. should add modified or newly added files to the list of ones to be removed at uninstall time (where feasible)
9. offer to show the extension release notes ("readme" file)

In step 2, an installed extension can be recognized by the presence of an "`eclipse`" directory immediately containing a file named "`.eclipseextension`". The parent of the "`eclipse`" directory is an extension's install directory; i.e., `<install>/eclipse/.eclipseextension`. The information contained within this marker file should be shown to the user for confirmation that the correct extension is being updated (there may be several Eclipse−based extension on the user's computer).

For step 7, the installer should not delete or overwrite any of the files that it originally installed; rather, it should only add the files for new versions of features and plug−in, and possibly rewrite the marker file `<install>/eclipse/.eclipseextension`. Leaving the old versions around gives the user the option to back out of the update. As with upgrading a product install, there is no need to touch any files in `<install>/eclipse/plugins/com.example.wiley.otherplugin_1.0.0/` if this sub−directory should also exist after the upgrade; if any of the plug−in's files were to change, the plug−in's version number is revised, causing the files for the upgraded plug−in to be installed in a parallel directory `<install>/eclipse/plugins/com.example.wiley.otherplugin_1.0.1/`.

Product extensions

An **extension** is a set of Eclipse features and plug−ins that are designed to extend the functionality of already−installed Eclipse based products. Extensions are installed separately, but used only in conjunction with other Eclipse based products. This means that an extension does not need to install a JRE, the Eclipse platform, or a primary feature. The recommended directory structure for extensions allows a single installation to be used with multiple Eclipse based products.

The following directory structure shows how an extension for a hypothetical product, **betterwebs**, could be used to extend the function of the **acmeweb** product.

```
betterwebs/
        eclipse/ (directory for installed Eclipse files)
                .eclipseextension       (marker file)
```

```
        (installed features)
                com.example.betterwebs.betterfeature_1.0.0/
                        feature.xml
          plugins/
                com.example.betterwebs.betterfeature_1.0.0/
                        plugin.xml
                        about.html
                com.example.betterwebs.betterwebsupport_1.0.0/
```

The relationship between an extension and the product that it is designed to enhance is set up in the **links** directory of the original product. Recall the following directory in the acmeweb product:

```
acmeweb/
        ...
        eclipse/ (directory for installed Eclipse files)
                ...
                jre/
        (installed features)
                ...
          plugins/
                ...
          links/
                com.example.betterwebs.betterfeature.link
```

When an extension is installed, it creates a link file in the **links** directory of any product that it is intending to extend. This link file makes the original product aware of the existence of the extension.

Installing and uninstalling extensions

The install process for extensions is similar to that for products except for the following differences:

- Determine which already−installed product is to be extended (by asking the user or searching the computer for the appropriate marker file)
- Create an **.eclipseextension** marker file (instead of an **.eclipseproduct** marker file). The format and content are similar to the product markers.
- Create a link file for the extension and write it into the **links** directory of the associated product. The link file has the same name as the extension's feature directory without the version suffix. The link file is a **java.io.Properties** format file which defines the path to the installed extension.

The uninstall process for extensions is similar to that for products except that the uninstall must remove the link file from any products where it added one.

Additional information on installing extensions can be found in How to write an Eclipse installer.

A product extension can be "softly" linked to an eclipse installation by using the update manager: open **Help > Software Updates > Manage Configuration** and click on the "Add an extension location" link on the right pane.

Updating a product or extension

By following the prescribed procedures for packaging and installing products, we can take advantage of the **platform update manager,** which treats products and extensions in a uniform way and allows users to discover and install updated versions of products and extensions.

Before looking at the implementation of such a server, it's important to revisit some important concepts:

- The platform provides a framework for defining **features** and update **sites**. The platform itself defines a concrete implementation of features and sites. This concrete implementation is what allows the update server to upgrade and install additional features.
- The platform **update server** can be used to **update** products by installing new versions of features. It can also be used to **install** or **update** extensions by adding or upgrading features. This is only possible for products and extensions that conform to the platform's concrete implementation of features and sites and conform to the appropriate install guidelines.
- Developers are free to use native installers and uninstallers to upgrade their own products and extensions without regard to sites and the update manager.

That said, what do we do if we want to fully participate in the platform implementation of product updating and use its update server?

Feature and plug−in packaging

The previous example product and extension directory structures show how features and plug−ins are laid out once they are installed. In order to install features using the update server, the features must be packaged in a feature archive file. This is described in Feature Archive Files.

Plug−ins and fragments must be packaged according to the format described in Plug−in Archive Files.

Update server layout

The update server must be a URL−accessible server with a fixed layout. The list of available features and plug−ins provided by the server is described in a site map file, **site.xml**. The update server URL can be specified as a full URL to the site map file, or a URL of a directory path containing the site map. The site map file contains a list of all the available features and the location of the feature archives on the server. It also describes the locations of the plug−in archives that are referenced in the feature manifest.

A simple site layout for our example web product and extension could look something like this:

```
<site root>/
        site.xml
        features/ (contains feature archive files)
                com.example.acme.acmefeature_1.0.1.jar
                com.example.betterwebs.betterfeature_1.0.1.jar
                ...
        plugins/ (contains plug-in archive files)
                com.example.acme.acmefeature_1.0.1.jar
                com.example.acme.acmewebsupport_1.0.3.jar
                com.example.betterwebs.betterfeature_1.0.1.jar
                com.example.betterwebs.betterwebsupport_1.0.1.jar
                ...
```

The complete definition for the site map is described in Update Server Site Map.

Update servers and policies

An Eclipse update server is provided for updating the platform itself. In addition, the platform update UI allows users can maintain a list of update servers that can be searched for new features. Any site that conforms

to the specified update server layout may be added to the list. Users can choose to manually or automatically search for additional features or upgrades to their installed features.

Some organizations may wish to have more control over how their user installations are updated. This can be accomplished with an update **policy** file that specifies which features can be updated and which servers can be used to update a particular feature. See Update Policy Control for a complete definition of the policy file and how it is used.

Update server site map

The default Eclipse update server is any URL−accessible server. The default implementation assumes a fixed−layout server. The content of the server (in terms of available features and plug−ins) is described in a site map file, *site.xml*. This file can be manually maintained, or can be dynamically computed by the server.

Site Map

The update server URL can be specified as a full URL to the site map file, or a URL of a directory path containing the site map file (similar to index.html processing). The site map site.xml format is defined by the following dtd:

```
<?xml encoding="ISO-8859-1"?>

<!ELEMENT site (description?, feature*, archive*, category-def*)>
<!ATTLIST site
    type          CDATA #IMPLIED
    url           CDATA #IMPLIED
    mirrorsURL    CDATA #IMPLIED
>

<!ELEMENT description (#PCDATA)>
<!ATTLIST description
    url           CDATA #IMPLIED
>

<!ELEMENT feature (category*)>
<!ATTLIST feature
    type          CDATA #IMPLIED
    id            CDATA #IMPLIED
    version       CDATA #IMPLIED
    url           CDATA #REQUIRED
    patch         (false | true) false

    os            CDATA #IMPLIED
    nl            CDATA #IMPLIED
    arch          CDATA #IMPLIED
    ws            CDATA #REQUIRED
>

<!ELEMENT archive EMPTY>
<!ATTLIST archive
    path          CDATA #REQUIRED
    url           CDATA #REQUIRED
>

<!ELEMENT category EMPTY>
<!ATTLIST category
    name          CDATA #REQUIRED
>
```

```
<!ELEMENT category-def (description?)>
<!ATTLIST category-def
    name          CDATA #REQUIRED
    label         CDATA #REQUIRED
>
```

The element and attribute definitions are as follows:

- <site> – defines the site map
  - type – optional site type specification. The value refers to a type string registered via the install framework extension point. If not specified, the type is assumed to be the default Eclipse site type (as specified in this document).
  - url – optional URL defining the update site baseline URL (used to determine individual <feature> and <archive> location). Can be relative or absolute. If relative, is relative to site.xml. If not specified, the default is the URL location of the site.xml file.
  - mirrorsURL – optional URL pointing to a file that contains update site mirror definitions. This URL can be absolute or relative to this site. The mirrors file is described later in this document.
- <description> – brief description as simple text. Intended to be translated.
  - url – optional URL for the full description as HTML. The URL can be specified as absolute of relative. If relative, the URL is relative to site.xml.
    Note, that for NL handling the URL value should be separated to allow alternate URLs to be specified for each national language.
- <feature> – identifies referenced feature archive
  - type – optional feature type specification. The value refers to a type string registered via the install framework extension point. If not specified, the type is assumed to be the default feature type for the site. If the site type is the default Eclipse site type, the default feature type is the packaged feature type (as specified in this document).
  - id – optional feature identifier. The information is used as a performance optimization to speed up searches for features. Must match the identifier specified in the feature.xml of the referenced archive (the url attribute). If specified, the version attribute must also be specified.
  - version – optional feature version. The information is used as a performance optimization to speed up searches for features. Must match the version specified in the feature.xml of the referenced archive (the url attribute). If specified, the id attribute must also be specified.
  - url – required URL reference to the feature archive. Can be relative or absolute. If relative, it is relative to the location of the site.xml file. **Note**: the default site implementation allows features to be accessed without being explicitly declared using a <feature> entry. By default, an undeclared features reference is interpreted as "features/<id>_<version>.jar". **Note**: for better lookup performance, always define the id and version attributes.
  - patch – optional attribute to denote that this is a patch (special type of feature). **Note**: for better lookup performance, always define this attribute.
  - os – optional operating system specification. A comma–separated list of operating system designators defined by Eclipse (see Javadoc for `org.eclipse.core.runtime.Platform`). Indicates this feature should only be installed on one of the specified OS's. If this attribute is not specified, the feature can be installed on all systems (portable implementation). This information is used as a hint by the installation and update support (user can force installation of feature regardless of this setting).
  - arch – optional machine architecture specification. A comma–separated list of architecture designators defined by Eclipse (see Javadoc for `org.eclipse.core.runtime.Platform`). Indicates this feature should only be

installed on one of the specified systems. If this attribute is not specified, the feature can be installed on all systems (portable implementation). This information is used as a hint by the installation and update support (user can force installation of feature regardless of this setting).
    - ws – optional windowing system specification. A comma–separated list of window system designators defined by Eclipse (see Javadoc for `org.eclipse.core.runtime.Platform`). Indicates this feature should only be installed on one of the specified WS's. If this attribute is not specified, the feature can be installed on all systems (portable implementation). This information is used as a hint by the installation and update support (user can force installation of feature regardless of this setting).
    - nl – optional locale specification. A comma–separated list of locale designators defined by Java. Indicates this feature should only be installed on a system running with a compatible locale (using Java locale–matching rules). If this attribute is not specified, the feature can be installed on all systems (language–neutral implementation). This information is used as a hint by the installation and update support (user can force installation of feature regardless of this setting).
- <archive> – identifies referenced "storage" archive (the actual files referenced via the `<plugin>` or `<data>` elements in the feature manifest). The site simply manages archives as a path–to–URL map. The default Eclipse site implementation does not require the <archive> section to be included in the site map (site.xml). Any archive reference not explicitly defined as part of an <archive> section is assumed to be mapped to a url in the form "<archivePath>" relative to the location of the site.xml file.
    - path – required archive path identifier. This is a string that is determined by the feature referencing this archive and is not otherwise interpreted by the site (other than as a lookup token).
    - url – required URL reference to the archive. Can be relative or absolute. If relative, it is relative to the location of the site.xml file.
- <category–def> – an optional definition of a category that can be used by installation and update support to hierarchicaly organize features
    - name – category name. Is specified as a path of name tokens separated by /
    - label – displayable label. Intended to be translated.
- <category> – actual category specification for a feature entry
    - name – category name

Note, that in general the feature.xml manifest documents should specify UTF–8 encoding. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
```

Translatable text contained in the site.xml can be separated into site<_locale>.properties files using Java property bundle conventions. Note that the translated strings are used at installation time (ie. do not employ the plug–in fragment runtime mechanism). The property bundles are located relative to the site.xml location.

Default Site Layout

```
<site root>/
    site.xml
    features/
        feature archives   (eg. org.eclipse.javatools_1.0.1.jar)
        <featureId>_<featureVersion>/    (optional)
            non-plug-in files for feature
```

```
    plugins/
        plug-in archives    (eg. org.eclipse.ui_1.0.3.jar)
```

Mirrors File

The update mirrors file (the one pointed at by the mirrorsURL attribute of <site>) contains definition for update site mirrors. It format is defined by the following dtd:

```
<?xml encoding="ISO-8859-1"?>

<!ELEMENT mirrors (mirror*))>


<!ELEMENT mirror EMPTY>
<!ATTLIST mirror
    url           CDATA #REQUIRED
    label         CDATA #REQUIRED
>
```

- <mirrors> – defines the available update site mirrors
- <mirror> – defines a mirror site
    - ♦ url – the URL of the mirror site
    - ♦ label – displayable label. Intended to be translated.

Controlling Access

The default Eclipse site implementation provides support for http access with basic user authentication (userid and password).

Custom access control mechanisms can be added to base Eclipse in one of 2 ways:

- by supplying server–side logic on the update server (eg. implementing servlets that compute the site.xml map, and control access to individual archives based on some user criteria)
- by supplying a custom concrete implementation of the site object (installed on the client machine, update server specified `<site type="">`). The custom concrete site implementation, together with any server–side logic support the required control mechanisms.

Eclipse provides an example demonstrating an implementation of an access mechanism based on feature key files.

Eclipse Update Policy Control

Eclipse Update allows users to search for updates to the currently installed features. For each installed feature, Update uses the embedded URL to connect to the remote server and search for new versions. If there are updates, Eclipse allows users to initiate the install procedure. After downloading, installing and restarting the platform, new feature version is ready for use.

In companies with many users of the same Eclipse−based product (typically a commercial one), several problems can arise from this model:

1. Updates for very large products (e.g. 500+ plug−ins) are also large. I/T support teams may not like the idea of hundreds of developers individually downloading 500MEG updates to their individual machines. In addition to the bandwidth hit, such a large download request may fail, leading to repeated attempts and increased developers' downtime.
2. Some companies explicitly don't want the developers downloading updates directly from the Internet. For example, they can set up a local support team that may not be ready to handle requests related to the version of the product already available from the provider's update site. They may want to restrict updates and fixes to the internally approved list. Ideally, they would do that by setting up 'proxy' update sites on the LAN (behind the firewall).
3. Once updates are set in the proxy sites as above, administrators need a way of letting users know that updates are available.

2. Update policy to the rescue

2.1 Support for creating local (proxy) update sites

First step for a product administrator would be to set up a local Eclipse update site on a server connected to the company's LAN (behind the firewall). The update site would be a subset of the product's update site on the Internet because it would contain only features and plug−ins related to the updates that the company wants applied at the moment. Technically, this site would be a regular Eclipse update site with site.xml, feature and plug−in archives.

Administrators would construct this site in two ways:

1. Product support teams would make a zip file of the update site readily available for this particular purpose. Administrators would simply need to download the zip file from the product support web page using the tool of their choice and unzip it in the local server. This approach is useful for very large zip files that require modern restartable downloading managers (those that can pick up where they left off in case of the connection problems).
2. Eclipse Update provides a tool to mirror remote update sites entirely or allow administrators to select updates and fixes to download. This mirroring capability would be fully automated and would greatly simplify administrator's task but it relies on Update network connection support.

2.2 Common update policy control

Since features have the update site URL embedded in the manifest, they are unaware of the local update sites set up by the administrators. It is therefore important to provide **redirection capability**. This and other update policy settings can be set for an Eclipse product by creating an update policy file and configuring Update to use that file when searching.

The file in question uses XML format and can have any name. The file can be set in **Preferences>Install/Update** in the **Update Policy** field. The text field is empty by default: users may set the URL of the update policy file. The file is managed by the local administrator and is shared for all the product installations. Sharing can be achieved in two ways:

- If users install the product: users are told to open the preference page and enter the provided URL
- If administrators install the product: administrators edit the file 'plugin_customization.ini' in the primary product feature and set the default value of the 'updatePolicyFile' property as follows:

```
org.eclipse.update.core/updatePolicyFile = <URL value>
```

  This will cause all the installations to have this file set by default.

The policy file must conform to the following DTD:

```
<?xml encoding="ISO-8859-1"?>

<!ELEMENT update-policy (url-map)*>
<!ATTLIST update-policy
>

<!ELEMENT url-map EMPTY>
        pattern CDATA #REQUIRED
        url     CDATA #REQUIRED
>
```

url–map

- **pattern –** a string that represents prefix of a feature ID (up to and including a complete ID). A value of "*" matches all the features.
- **url –** a URL of the alternative update site that should be used if the feature ID begins with the pattern. If the string is empty, features matching pattern will not be updateable.

This element is used to override Update URLs embedded in feature manifests. When looking for new updates, Eclipse search will check the update policy (if present) and check if **url–map** for the matching feature prefix is specified. If a match is found, the mapped URL will be used **instead** of the embedded one. This way, administrators can configure Eclipse products to search for updates in the local server behind the firewall. Meanwhile, third–party features installed by Eclipse Update will continue to be updated using the default mechanism because they will not find matches in the policy.

Several **url–map** elements may exist in the file. Feature prefixes can be chosen to be less or more specific. For example, to redirect all Eclipse updates, the pattern attribute would be `"org.eclipse"`. Similarly, it is possible to use a complete feature ID as a pattern if redirection is required on a per–feature basis.

Patterns in the file may be chosen to progressively narrow the potential matches. This may result in multiple matches for a given feature. In this case, the **match with a longest pattern** will be used. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<update-policy>
        <url-map pattern="org.eclipse" url="URL1"/>
        <url-map pattern="org.eclipse.jdt" url="URL2"/>
</update-policy>
```

In the case above, all Eclipse features will be updated from URL1, except `org.eclipse.jdt` that will use URL2.

Update policy files do not contain translatable strings and therefore do not require special NL handling. In general, the files should use UTF−8 encoding.

2.3 Automatic discovery of updates

Automatic updates will allow Eclipse to run update search on a specified schedule (on each startup (the default), once a day, once a week etc.).

3. Summary

Here is the complete sequence of steps that comprise the solution:

1. Administrator allocates a server on the company LAN for hosting local product updates. Initially it contains no update sites. The machine must have an HTTP server running.
2. Administrator sets up an update policy file on that server and instructs all users to set the update policy preference the provided URL.
3. As the product provider ships updates and fixes on their update sites, administrator downloads supported updates onto the local server.
4. Automatic update executed at the scheduled frequency when the client's product is up picks up the local updates and notifies the user
5. User chooses to install the discovered updates

Deploying eclipse based application with Java Web Start

Applications built on eclipse 3.1 can now be deployed using Java Web Start.

Java Web Start "is an application−deployment technology that gives you the power to launch full−featured applications with a single click from your web browser".

The prerequisites to start eclipse from Java Web Start are:

- The deployed application must be eclipse 3.1 based;
- All deployed plug−ins must be jar'ed;
- All plug−ins must be signed since the application need full permission from the client.

The following steps describe how to setup a Java Web Start site serving up a feature based RCP application.

Step 1, creating a wrappering feature

- Create a feature including all the features that are listed in your product definition;
- Copy in a folder of your feature the startup.jar;
- Add the following line to the build.properties of the feature.

```
root=<folderContainingStartup.jar>/
```

Step 2, exporting the wrappering feature and the startup.jar

Note. Before proceeding with this step make sure to have a keystore available. Eclipse does not provide any facility to create keystores. You need to use keytool. In addition, ensure that the eclipse you are developing with is running on a Java SDK instead of a JRE. If this constraint is not satisfied, the jar signing will fail.

- Select the wrappering feature and do File > Export > Feature. In the wizard, select the wrappering feature, choose the "directory" option to export your jnlp application to, and check the option "package features and plug–ins as individual JAR archives". On the next page of the wizard, fill in the information relative to your keystore in the "Signing JAR Archives" section. Then in the "JNLP section", enter the name of the server that will serve up your application and the level of JRE required to start your application. That last value will be used to in the generated JNLP files to fill in the value of `<j2se version="1.4+" />`. Click finish.
- Once the export is done you should have the following structure on disk

```
site/   (The root of your jnlp site)
  startup.jar
  features/
    WrapperingFeature_1.0.0.jar
    WrapperingFeature_1.0.0.jnlp
    com.xyz.abc_1.0.0.jar
    com.xyz.abc_1.0.0.jnlp
    ...
  plugins/
    org.eclipse.core.runtime_3.1.0.jar
    com.foo.baz_1.0.0.jnlp
    ...
```

- Using the same keystore than the one used to export the feature, sign startup.jar using jarsigner. Tips: if you don't change keystore, replace the startup.jar from the feature with this signed version so you don't have to do this manual step on every export.

Step 3, creating the main jnlp file

A Java Web Start application is described by JNLP files. They replace the eclipse.exe and the config.ini files by some equivalent mechanism. For example, JNLP has its own mechanism to control splash screen, ways to pass parameters and define what constitutes the application.

When you did the export, all the simple JNLP files have been created, so you are left with writing the main file that will control the application. Because the majority of the main file is common to all applications, it is recommended to start from the following self documented template.

On the site serving up your application, the file must be located in the same folder than startup.jar. Once you will be done editing this file, your application will be ready.

```
<?xml version="1.0" encoding="UTF-8"?>
<jnlp
    spec="1.0+"
    codebase="http://myCompany.org/jnlpServer"
    href="mail.jnlp"> <!-- URL to the site containing the jnlp application. It should match the v
  <information>
    <!-- user readable name of the application -->
    <title> Mail Application </title>
    <!-- vendor name -->
    <vendor>My company</vendor>
```

```
    <!-- vendor homepage -->
    <homepage href="My company website" />
    <!-- product description -->
    <description>This is a mail client</description>
    <icon kind="splash" href="splash.gif"/>
  </information>

  <!--request all permissions from the application. This does not change-->
  <security>
    <all-permissions/>
  </security>

  <!-- The name of the main class to execute. This does not change-->
  <application-desc main-class="org.eclipse.core.launcher.WebStartMain">
    <argument>-nosplash</argument>
  </application-desc>

  <resources>
    <!-- Reference to the startup.jar. This does not change -->
    <jar href="startup.jar"/>

    <!-- Reference to all the plugins and features consituting the application -->
    <!-- Here we are refering to the wrappering feature since it transitively refers to all the o
    <extension
        name="Wrappering feature"
        href="features/Wrappering_1.0.0.jnlp"/>

    <!-- Information usually specified in the config.ini -->
    <property
        name="osgi.instance.area"
        value="@user.home/Application Data/mail"/>
    <property
        name="osgi.configuration.area"
        value="@user.home/Application Data/mail"/>

    <!-- The id of the product to run, like found in the overview page of the product editor -->
    <property
        name="eclipse.product"
        value="mail.product"/>
  </resources>

  <!-- Indicate on a platform basis which JRE to use -->
  <resources os="Mac">
    <j2se version="1.5+" java-vm-args="-XstartOnFirstThread"/>
  </resources>
  <resources os="Windows">
    <j2se version="1.4+"/>
  </resources>
  <resources os="Linux">
    <j2se version="1.4+"/>
  </resources>
</jnlp>
```

Tips: once you have created this file, you can store it in the wrappering feature in the same folder than the startup.jar, such that on every export you will get the complete structure.

Plug-ins based application

Even though your RCP application does not use features, Java Web Start-ing it is possible.

To do so, it is recommended to create a wrappering feature in order to facilitate the creation of the main jnlp file and ease the deployement. This wrappering feature will list all the plug−ins of your application. Once the feature has been updated copy the generated JNLP file and modify it to become your main JNLP file.

Known limitations

- Eclipse Update and Java Web Start
  Those two deployment technologies can work together but under the following restrictions: plug−ins installed by Java Web Start can not be updated by Update and vice−versa. Features and plug−ins installed by Java Web Start can't be refered in the prerequisites of features that needs to be installed by Update;
- Help can not be deployed through Java Web Start. However it could be installed using eclipse Update, or the server serving your application could run a help server;
- Request to exit the application with a restart code are ignored;
- On the mac, applications can only be webstarted by clients using java 1.5.

Building a Rich Client Platform application

While the Eclipse platform is designed to serve as an open tools platform, it is architected so that its components could be used to build just about any client application. The minimal set of plug−ins needed to build a rich client application is collectively known as the **Rich Client Platform**.

Applications that don't require a common resource model can be built using a subset of the platform. These rich applications are still based on a dynamic plug−in model, and the UI is built using the same toolkits and extension points. The layout and function of the workbench is under fine−grained control of the plug−in developer in this case. Up until now, we've been adding all of our visible function to the platform SDK workbench. In a rich client application, we are responsible for defining the application and its workbench.

When we say that the Rich Client Platform is the minimal set of plug−ins needed to build a platform application with a UI, we mean that your application need only require two plug−ins, **org.eclipse.ui** and **org.eclipse.core runtime**. However, rich client applications are free to use any API deemed necessary for their feature set, and can require any plug−ins above the bare minimum. The Map of platform plug−ins is a useful reference when determing what plug−ins should be required when using various platform API.

The main thing that differentiates a rich client application from the platform workbench is that the application is responsible for defining which class should be run as the main application. We'll look at how this is done in the context of an example.

Eclipse Platform
Map of Platform Plug−ins

The Eclipse Platform provides basic support for plug−ins, resources, and the workbench.

The Eclipse Platform itself is divided up into a number of separate plug−ins. The following table shows which API packages are found in which plug−ins as of Eclipse 3.1. This table is useful for determining which plug−ins a given plug−in should include as prerequisites.

API Package	Required plug−in id	N
org.apache.tools.ant[.*] org.apache.tools.bzip2 org.apache.tools.mail org.apache.tools.tar org.apache.tools.zip	org.apache.ant	
org.apache.lucene[.*]	org.apache.lucene	
org.eclipse.ant.core	org.eclipse.ant.core	
org.eclipse.compare org.eclipse.compare.contentmergeviewer org.eclipse.compare.rangedifferencer org.eclipse.compare.structuremergeviewer	org.eclipse.compare	
org.eclipse.core.commands org.eclipse.core.commands.common org.eclipse.core.commands.contexts org.eclipse.core.commands.operations	org.eclipse.core.commands	
org.eclipse.core.expressions	org.eclipse.core.expressions	
org.eclipse.core.filebuffers org.eclipse.core.filebuffers.manipulation	org.eclipse.core.filebuffers	
org.eclipse.core.resources org.eclipse.core.resources.refresh org.eclipse.core.resources.team	org.eclipse.core.resources	
org.eclipse.core.runtime org.eclipse.core.runtime.content org.eclipse.core.runtime.dynamichelpers org.eclipse.core.runtime.jobs org.eclipse.core.runtime.preferences org.eclipse.osgi[.*] org.osgi[.*]	org.eclipse.core.runtime	[
org.eclipse.core.boot (obsolete) *org.eclipse.core.runtime.model (obsolete)*	org.eclipse.core.runtime.compatibility	[
org.eclipse.core.variables	org.eclipse.core.variables	
org.eclipse.debug.core org.eclipse.debug.core.model org.eclipse.debug.core.sourcelookup org.eclipse.debug.core.sourcelookup.containers	org.eclipse.debug.core	
	org.eclipse.debug.ui	

org.eclipse.debug.ui org.eclipse.debug.ui.actions org.eclipse.debug.ui.console org.eclipse.debug.ui.memory org.eclipse.debug.ui.sourcelookup		
org.eclipse.help	org.eclipse.help	
org.eclipse.help.browser org.eclipse.help.search org.eclipse.help.standalone	org.eclipse.help.base	
org.eclipse.help.ui org.eclipse.help.ui.browser	org.eclipse.help.ui	
org.eclipse.jface org.eclipse.jface.action org.eclipse.jface.action.images org.eclipse.jface.bindings org.eclipse.jface.bindings.keys org.eclipse.jface.bindings.keys.formatting org.eclipse.jface.commands org.eclipse.jface.contexts org.eclipse.jface.dialogs org.eclipse.jface.dialogs.images org.eclipse.jface.images org.eclipse.jface.operation org.eclipse.jface.preference org.eclipse.jface.preference.images org.eclipse.jface.resource org.eclipse.jface.util org.eclipse.jface.viewers org.eclipse.jface.viewers.deferred org.eclipse.jface.window org.eclipse.jface.wizard org.eclipse.jface.wizard.images	org.eclipse.ui	[
org.eclipse.jface.contentassist org.eclipse.jface.contentassist.images org.eclipse.jface.text (split) org.eclipse.jface.text.contentassist org.eclipse.jface.text.formatter org.eclipse.jface.text.hyperlink org.eclipse.jface.text.information org.eclipse.jface.text.link (split) org.eclipse.jface.text.presentation org.eclipse.jface.text.reconciler org.eclipse.jface.text.rules org.eclipse.jface.text.source (split) org.eclipse.jface.text.source.projection org.eclipse.jface.text.source.projection.images org.eclipse.jface.text.templates (split) org.eclipse.jface.text.templates.persistence	org.eclipse.jface.text	[
	org.eclipse.text	[

org.eclipse.jface.text (split) org.eclipse.jface.text.link (split) org.eclipse.jface.text.projection org.eclipse.jface.text.source (split) org.eclipse.jface.text.templates (split) org.eclipse.text.edits		
org.eclipse.ltk.core.refactoring org.eclipse.ltk.core.refactoring.participants	org.eclipse.ltk.core.refactoring	
org.eclipse.ltk.ui.refactoring	org.eclipse.ltk.ui.refactoring	
org.eclipse.search.ui org.eclipse.search.ui.text	org.eclipse.search	
org.eclipse.swt org.eclipse.swt.accessibility org.eclipse.swt.awt org.eclipse.swt.browser org.eclipse.swt.custom org.eclipse.swt.dnd org.eclipse.swt.events org.eclipse.swt.graphics org.eclipse.swt.layout org.eclipse.swt.printing org.eclipse.swt.program org.eclipse.swt.widgets	org.eclipse.ui	[
org.eclipse.swt.ole.win32	org.eclipse.swt.win32.win32.x86	[
org.eclipse.team.core org.eclipse.team.core.subscribers org.eclipse.team.core.synchronize org.eclipse.team.core.variants	org.eclipse.team.core	
org.eclipse.team.ui org.eclipse.team.ui.synchronize	org.eclipse.team.ui	
org.eclipse.ui.browser	org.eclipse.ui.browser	
org.eclipse.ui.cheatsheets	org.eclipse.ui.cheatsheets	
org.eclipse.ui.console org.eclipse.ui.console.actions	org.eclipse.ui.console	
org.eclipse.ui.editors.text org.eclipse.ui.editors.text.templates org.eclipse.ui.texteditor (split)	org.eclipse.ui.editors	[
org.eclipse.ui.forms org.eclipse.ui.forms.editor org.eclipse.ui.forms.events org.eclipse.ui.forms.widgets	org.eclipse.ui.forms	
org.eclipse.ui (split) org.eclipse.ui.about org.eclipse.ui.actions (split) org.eclipse.ui.activities org.eclipse.ui.application org.eclipse.ui.branding org.eclipse.ui.browser	org.eclipse.ui	[

org.eclipse.ui.commands org.eclipse.ui.contexts org.eclipse.ui.dialogs (split) org.eclipse.ui.handlers org.eclipse.ui.help org.eclipse.ui.intro org.eclipse.ui.keys org.eclipse.ui.model (split) org.eclipse.ui.operations org.eclipse.ui.part (split) org.eclipse.ui.plugin org.eclipse.ui.preferences org.eclipse.ui.presentations org.eclipse.ui.progress org.eclipse.ui.testing org.eclipse.ui.themes org.eclipse.ui.views org.eclipse.ui.wizards		
org.eclipse.ui (split) org.eclipse.ui.actions (split) org.eclipse.ui.dialogs (split) org.eclipse.ui.ide org.eclipse.ui.ide.dialogs org.eclipse.ui.model (split) org.eclipse.ui.part (split) org.eclipse.ui.views.bookmarkexplorer org.eclipse.ui.views.framelist org.eclipse.ui.views.markers org.eclipse.ui.views.navigator org.eclipse.ui.views.properties org.eclipse.ui.views.tasklist org.eclipse.ui.wizards.datatransfer org.eclipse.ui.wizards.newresource	org.eclipse.ui.ide	[
org.eclipse.ui.intro.config	org.eclipse.ui.intro	
org.eclipse.ui.views.contentoutline org.eclipse.ui.views.properties	org.eclipse.ui.views	
org.eclipse.ui.contentassist org.eclipse.ui.texteditor (split) org.eclipse.ui.texteditor.link org.eclipse.ui.texteditor.quickdiff org.eclipse.ui.texteditor.spelling org.eclipse.ui.texteditor.templates	org.eclipse.ui.workbench.texteditor	[
org.eclipse.update.configurator	org.eclipse.update.configurator	
org.eclipse.update.configuration org.eclipse.update.core org.eclipse.update.core.model org.eclipse.update.operations org.eclipse.update.search org.eclipse.update.standalone	org.eclipse.update.core	

org.eclipse.update.ui	org.eclipse.update.ui

Note 1: Plug−ins needing access to the Eclipse runtime API must list `org.eclipse.core.runtime` (or `org.eclipse.core.runtime.compatibility`) as a prerequisite plug−in. `org.eclipse.core.runtime` re−exports API from the OSGi−specific plug−ins (e.g., `org.eclipse.osgi`). The OSGi plug−ins should never be explicitly listed as prerequisites.

Note 2: These pre−3.0 API packages are obsolete and have been moved to the `org.eclipse.core.runtime.compatibility` plug−in.

Note 3: Plug−ins needing access to the JFace API must list `org.eclipse.ui` as a prerequisite plug−in. `org.eclipse.ui` re−exports API from the JFace plug−in. The `org.eclipse.jface` plug−in should never be explicitly listed as a prerequisite.

Note 4: Some of the JFace text packages are split between the `org.eclipse.jface.text` and the `org.eclipse.text` plug−ins.

Note 5: Plug−ins needing access to the SWT API must list `org.eclipse.ui` as a prerequisite plug−in. `org.eclipse.ui` re−exports API from the SWT plug−in. The `org.eclipse.swt` plug−in should never be explicitly listed as a prerequisite.

Note 6: The `org.eclipse.ui.texteditor` package is split between the `org.eclipse.ui.editors` and the `org.eclipse.ui.workbench.texteditor` plug−ins.

Note 7: Plug−ins needing access to the Workbench UI API must list `org.eclipse.ui` as a prerequisite plug−in. `org.eclipse.ui` re−exports API from the `org.eclipse.ui.workbench` plug−in. The `org.eclipse.ui.workbench` plug−in should never be explicitly listed as a prerequisite.

Note 8: Some of the UI packages are split between the `org.eclipse.ui` and the `org.eclipse.ui.ide` plug−ins.

Note 9: The plug−in `org.eclipse.swt.win32.win32.x86` is available on Win32/x86 platforms only.

The browser example

We will look at how to build a Rich Client Platform application by going through a simple web browser example. This example is not included in the R3.0 SDK, but can be downloaded from the project org.eclipse.ui.examples.rcp.browser. If you are working in Eclipse, you may simply check out the project from the Eclipse CVS repository (see the Eclipse CVS How−To if you are not familiar with the procedure for checking out projects from CVS).

To run the RCP Browser example from within the Eclipse SDK:

1. Load the project `org.eclipse.ui.examples.rcp.browser` from the Eclipse CVS repository.
2. Choose **Run>Run...** from the workbench menu bar and create a new "Run−time workbench" configuration named "Browser Example".
3. On the **Arguments** tab, select **Run a product** and select "org.eclipse.ui.examples.rcp.browser.product" from the drop−down.

4. On the **Plug–ins** tab, select **Choose plug–ins and fragments to launch from the list** so that you can select which plug–ins are needed.
5. Press **Deselect All** to start with a clean slate.
6. Check "org.eclipse.ui.examples.rcp.browser"
7. Press **Add Required Plug–ins**.
8. Check "org.eclipse.update.configurator"
9. Run or debug the new run configuration.

As you can see, it's hard to tell that this application has anything at all to do with Eclipse (apart from the default web site that it browses!). There is no resource navigator, no mention of the Eclipse Platform, and none of the familiar menu bar items from the platform workbench. (The few Eclipse–related features, such as the window icon, could also be reconfigured if desired.)

Hopefully this example helps clarify what's exciting about the Rich Client Platform. Let's take a look under the covers so we can learn what's involved in building one. We'll assume that you are familiar with the basic workbench extensions discussed in Plugging into the workbench.

Defining a rich client application

The definition of a rich client application plug−in starts out similarly to the other plug−ins we've been studying. The only difference in the first part of the markup is that the list of required plug−ins is much smaller than we've been used to!

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin
   id="org.eclipse.ui.examples.rcp.browser"
   name="%pluginName"
   version="3.0.0"
   provider-name="%providerName">

   <runtime>
      <library name="browser.jar">
      </library>
   </runtime>
   <requires>
      <import plugin="org.eclipse.core.runtime"/>
      <import plugin="org.eclipse.ui"/>
   </requires>
```

Up to now, we've contributed function to the platform workbench by declaring extensions that add elements to the workbench. In all of the `plugin.xml` content that we've reviewed so far, we've only looked at individual contributions to a workbench that is assumed to be there. On the rich client platform, there is no application already defined. Your rich client plug−in is the one responsible for specifying the class that should be executed when the platform is started. This is done in the **org.eclipse.core.runtime.applications** extension.

```
   <extension
         point="org.eclipse.core.runtime.applications"
         id="app"
         name="%appName">
      <application>
         <run
            class="org.eclipse.ui.examples.rcp.browser.BrowserApp">
         </run>
      </application>
   </extension>
```

In this extension, we specify the class that should be run when the platform is first started. This class must implement **IPlatformRunnable**, which simply means that it must implement a **run** method. The run method is responsible for creating the SWT display and starting up a workbench. The class **PlatformUI** implements convenience methods for performing these tasks.

```
        public Object run(Object args) throws Exception {
                Display display = PlatformUI.createDisplay();
                try {
                        int code = PlatformUI.createAndRunWorkbench(display,
new BrowserAdvisor());
                        // exit the application with an appropriate return code
                        return code == PlatformUI.RETURN_RESTART
```

```
                                        ? EXIT_RESTART
                                        : EXIT_OK;
                } finally {
                        if (display != null)
                                display.dispose();
                }
        }
```

The call to **createAndRunWorkbench** will not return until the workbench is closed. The SWT event loop and other low−level logistics are handled inside this method. At this stage, it's not so important that you understand the underlying mechanics in running an SWT application. This code can be copied to your rich client application with minimal changes. In fact, the hook for you to add your own functionality is the **WorkbenchAdvisor** that is passed as an argument when the workbench is created. Let's take a closer look.

Customizing the workbench

The "entry point" for supplying custom workbench behavior is the designation of a **WorkbenchAdvisor** for configuring the workbench. Your rich client plug−in should extend this abstract class to provide the application−specific configuration for the workbench. The browser example does this using the **BrowserAdvisor** class.

```
        ...
        int code = PlatformUI.createAndRunWorkbench(display,
new BrowserAdvisor());
        ...
```

A workbench advisor is responsible for overriding methods to configure the workbench with its desired layout and features, such as the action bar items or intro page.

The workbench life−cycle

Life−cycle methods provided by the workbench advisor allow your application to hook into the creation of the workbench at any point in time and influence the behavior. The following list of advisor life−cycle methods that can be overridden comes from the javadoc for **WorkbenchAdvisor**.

- `initialize` – called first; before any windows; use to register things
- `preStartup` – called second; after initialize but before first window is opened; use to temporarily disable things during startup or restore
- `postStartup` – called third; after first window is opened; use to reenable things temporarily disabled in previous step
- `postRestore` – called after the workbench and its windows have been recreated from a previously saved state; use to adjust the restored workbench
- `preWindowOpen` – called as each window is being opened; use to configure aspects of the window other than action bars
- `fillActionBars` – called after `preWindowOpen` to configure a window's action bars
- `postWindowRestore` – called after a window has been recreated from a previously saved state; use to adjust the restored window
- `postWindowCreate` – called after a window has been created, either from an initial state or from a restored state; used to adjust the window
- `openIntro` – called immediately before a window is opened in order to create the introduction component, if any.
- `postWindowOpen` – called after a window has been opened; use to hook window listeners, etc.

- `preWindowShellClose` – called when a window's shell is closed by the user; use to pre–screen window closings
- `eventLoopException` – called to handle the case where the event loop has crashed; use to inform the user that things are not well
- `eventLoopIdle` – called when there are currently no more events to be processed; use to perform other work or to yield until new events enter the queue
- `preShutdown` – called just after event loop has terminated but before any windows have been closed; allows the application to veto the shutdown
- `postShutdown` – called last; after event loop has terminated and all windows have been closed; use to deregister things registered during initialize

As you can see, a rich client application has a lot of control over how the workbench is configured and implemented. In the browser example, the primary function of the **BrowserAdvisor** is to configure the action bars with menu items appropriate for a browser. This is done in the **fillActionBars** method:

```
public void fillActionBars(IWorkbenchWindow window, IActionBarConfigurer configurer, int
        ...
        BrowserActionBuilder builder = new BrowserActionBuilder(window);
        getWorkbenchConfigurer().getWindowConfigurer(window).setData(BUILDER_KEY, builder
        builder.fillActionBars(configurer, flags);
}
```

In this method, the workbench is configured with a specialized action builder. This action builder is used to fill the action bars of the workbench. We'll look at the details for how the actions are specified in Defining the actions. For now, we are focusing on how we configure the workbench.

Note the use of the **getWorkbenchConfigurer()** method above. The **IWorkbenchConfigurer** and **IWorkbenchWindowConfigurer** are used in conjunction with the **WorkbenchAdvisor** to customize the window. These classes allow you to override many aspects of workbench creation at different levels. For example, the **IWorkbenchWindowConfigurer** defines protocol that assumes a particular configuration of controls in the workbench window, such an action bar, status line, perspective bar, cool bar, etc. Its protocol allows you customize and populate these items. The **IWorkbenchConfigurer** operates at a higher level, allowing you to store application–specific data with the workbench. The **WorkbenchAdvisor** provides access to these configurers in the life–cycle methods noted above. Lower level methods inside **WorkbenchAdvisor** may be overridden to completely replace default behavior. For example, your workbench advisor could override the method that creates the SWT controls in the window in order to provide a completely different implementation for the main window.

In other words, there are many ways to customize the workbench and several different levels at which these techniques can be used. The javadoc for **WorkbenchAdvisor**, **IWorkbenchConfigurer**, and **IWorkbenchWindowConfigurer** includes a complete description of the available protocol. See also the complete implementation of **BrowserAdvisor** for comments on alternate implementations.

Defining the actions

The primary customization provided by the **BrowserAdvisor** in the browser example is the designation of the action bar content for the workbench window:

```
public void fillActionBars(IWorkbenchWindow window, IActionBarConfigurer configurer, int
        ...
        BrowserActionBuilder builder = new BrowserActionBuilder(window);
        getWorkbenchConfigurer().getWindowConfigurer(window).setData(BUILDER_KEY, builder
        builder.fillActionBars(configurer, flags);
```

```
        }
```

Let's take a closer look at how these actions are defined in the **BrowserActionBuilder**. In particular, let's look at the actions that are handled by the browser view.

```
private void makeActions() {
                ...
                backAction = new RetargetAction("back", "&Back");
                backAction.setToolTipText("Back");
                backAction.setImageDescriptor(images.getImageDescriptor(ISharedImages.IMG_TOOL_BA
                window.getPartService().addPartListener(backAction);

                forwardAction = new RetargetAction("forward", "&Forward");
                forwardAction.setToolTipText("Forward");
                forwardAction.setImageDescriptor(images.getImageDescriptor(ISharedImages.IMG_TOOL
                window.getPartService().addPartListener(forwardAction);

                stopAction = new RetargetAction("stop", "Sto&p");
                stopAction.setToolTipText("Stop");
                window.getPartService().addPartListener(stopAction);

                refreshAction = new RetargetAction("refresh", "&Refresh");
                refreshAction.setToolTipText("Refresh");
                window.getPartService().addPartListener(refreshAction);
                ...
        }
```

The actions are defined as retargetable actions so that individual views can implement the handler actions. The **BrowserView** associates its handler actions with the window's retargetable actions when it creates the controls for the view:

```
        private Browser createBrowser(Composite parent, final IActionBars actionBars) {

                ...
                actionBars.setGlobalActionHandler("back", backAction);
                actionBars.setGlobalActionHandler("forward", forwardAction);
                actionBars.setGlobalActionHandler("stop", stopAction);
                actionBars.setGlobalActionHandler("refresh", refreshAction);
                ...
        }
```

These actions are created when the view is first created.

```
        private Action backAction = new Action("Back") {
                public void run() {
                        browser.back();
                }
        };
```

See Retargetable actions for a complete discussion of retargetable actions and how they are defined and implemented.

Making UI contributions

So far, we've seen that the main difference between a rich client platform plug−in and an Eclipse SDK plug−in is that the rich client plug−in is responsible for specifying the class that should be run when the platform is started. This class creates and runs a workbench window that is configured appropriately. What

else is different about a rich client application? Not much, actually.

Once the infrastructure for the application workbench is in place, the techniques for adding function to the workbench are the same as those we used when we were extending the platform SDK workbench. The workbench UI extension points are used to add views, editors, menus, and all of the other contributions we know and love. In the case of the browser example, we'll be adding extensions for a perspective and a couple of views.

We were introduced to these extensions in Plugging into the workbench. For completeness, we'll take a look at how the browser example uses these extensions, but we'll assume that we already have a working knowledge of these concepts.

Adding the perspective

When a rich client application uses the **WorkbenchAdvisor** as the primary means for customizing the workbench, it must supply a perspective that is shown in the workbench window. This perspective must be identified in the application's workbench advisor class. The following is specified in the **BrowserAdvisor** class:

```
public String getInitialWindowPerspectiveId() {
        return BrowserApp.BROWSER_PERSPECTIVE_ID;
}
```

While **BrowserApp** defines:

```
public static final String PLUGIN_ID = "org.eclipse.ui.examples.rcp.browser";
public static final String BROWSER_PERSPECTIVE_ID = PLUGIN_ID + ".browserPerspective";
```

The corresponding perspective is defined in the browser plug–in's manifest:

```
<extension
      point="org.eclipse.ui.perspectives">
   <perspective
         id="org.eclipse.ui.examples.rcp.browser.browserPerspective"
         name="%perspectives.browser.name"
         class="org.eclipse.ui.examples.rcp.browser.BrowserPerspectiveFactory"
         fixed="true"/>
</extension>
```

The **BrowserPerspectiveFactory** is responsible for laying out the views appropriately.

```
public void createInitialLayout(IPageLayout layout) {
        layout.addView(BrowserApp.BROWSER_VIEW_ID, IPageLayout.RIGHT, .25f, IPageLayout.I
        layout.addPlaceholder(BrowserApp.HISTORY_VIEW_ID, IPageLayout.LEFT, .3f, IPageLay
        IViewLayout historyLayout = layout.getViewLayout(BrowserApp.HISTORY_VIEW_ID);
        historyLayout.setCloseable(true);
        layout.setEditorAreaVisible(false);
}
```

The browser perspective defines two views (one visible, with a placeholder for the other), and makes the editor area invisible. For a complete discussion of perspectives and their corresponding layout, see Perspectives.

Adding views

The browser example defines two views in its workbench. One view shows the browser content and the other displays the history of visited links. We first saw these views when they were added to the browser's perspective:

```
        public void createInitialLayout(IPageLayout layout) {
layout.addView(BrowserApp.BROWSER_VIEW_ID, IPageLayout.RIGHT, .25f, IPageLayout.ID_EDITOR_AREA);
                layout.addPlaceholder(BrowserApp.HISTORY_VIEW_ID, IPageLayout.LEFT, .3f, IPageLay
                IViewLayout historyLayout = layout.getViewLayout(BrowserApp.HISTORY_VIEW_ID);
                historyLayout.setCloseable(true);
                layout.setEditorAreaVisible(false);
        }
```

The corresponding views are also defined in the browser plug–in's manifest:

```
   <extension
         point="org.eclipse.ui.views">
      <category
            id="org.eclipse.ui.examples.rcp.browser"
            name="%views.category.name"/>
      <view
            id="org.eclipse.ui.examples.rcp.browser.browserView"
            name="%views.browser.name"
            icon="icons/eclipse.png"
            class="org.eclipse.ui.examples.rcp.browser.BrowserView"
            category="org.eclipse.ui.examples.rcp.browser"
            allowMultiple="true"/>
      <view
            id="org.eclipse.ui.examples.rcp.browser.historyView"
            name="%views.history.name"
            icon="icons/eclipse.png"
            class="org.eclipse.ui.examples.rcp.browser.HistoryView"
            category="org.eclipse.ui.examples.rcp.browser"/>
   </extension>
```

The **BrowserView** and **HistoryView** create the necessary SWT controls for showing the browser content and history. The implementation of these views is no different for rich client plug–ins, so we won't review them here. See the example classes and org.eclipse.ui.views for more information.

Other Reference Information

The following specifications, white papers, and design notes describe various aspects of the Eclipse Platform.

Basic platform information

- Runtime options
- Starting Eclipse from Java
- API Rules of Engagement
- Naming Conventions
- Glossary of terms
- Map of Platform Plug−ins
- Plug−in Manifest
- Project Description File
- OSGi Bundle Manifest
- Message Bundles in Eclipse 3.1
- Multi−user Installs

User interface information

- Tips for making user interfaces accessible

Help information

- Pre−built documentation index
- Installing the stand−alone help system
- Installing the help system as an infocenter
- Help system preferences

Product install and configuration information

- How to write an Eclipse Installer
- About.ini File Format
- Plug−in archives
- Feature manifest
- Feature archives
- The platform.xml File
- Update server site map
- Update policy control
- Running update manager from command line

Eclipse.org articles index

The eclipse corner site contains technical articles about many topics of interest. Check the index regularly for the latest technical information regarding Eclipse.

The Eclipse runtime options

Version 3.1 – Last revised June 15, 2005

The Eclipse platform is highly configurable. Configuration input takes the form of command line arguments and System property settings. In many cases the command line arguments are simply short cuts for setting the related System properties. In fact, there are many more System property settings than command line arguments.

Command line arguments

Listed below are the command line arguments processed by various parts of the Eclipse runtime. Many of these values can also be specified using System properties either on the command line using –D VM arguments, by specifying their values in a config.ini file or by using a <launcher>.ini file. Using the two latter techniques it is possible to customize your Eclipse without using command line arguments at all.

For each argument in the list, its corresponding System property key is given (in {}). Also given is the Eclipse runtime layer in which the command line argument is processed (in ()). This is useful for people replacing parts of the runtime to suit special needs.

–application <id> (Runtime)
: equivalent to setting eclipse.application to <id>

–arch <architecture> (OSGi)
: equivalent to setting osgi.arch to <architecture>

–clean (OSGi)
: equivalent to setting osgi.clean to "true"

–configuration <location> (Main)
: equivalent to setting osgi.configuration.area to <location>

–console [port] (OSGi)
: equivalent to setting osgi.console to [port] or the empty string if the default port is to be used (i.e., when the port is not specified)

–consoleLog (Runtime)
: equivalent to setting eclipse.consoleLog to "true"

–data <location> (OSGi)
: equivalent to setting osgi.instance.area to <location>

–debug [options file] (OSGi)
: equivalent to setting osgi.debug to [options file] or the empty string to simply enable debug (i.e., if the options file location is not specified)

–dev [entries] (OSGi)
: equivalent to setting osgi.dev to [entries] or the empty string to simply enable dev mode (i.e., if entries are not specified)

–endSplash <command> (Main)
: specifies the command to use to take down the splash screen. Typically supplied by the Eclipse executable.

–feature <feature id> (Runtime)
: equivalent to setting eclipse.product to <feature id>

–framework <location> (Main)
: equivalent to setting osgi.framework to <location>

–initialize (Main)

initializes the configuration being run. All runtime related data structures and caches are refreshed. Any user/plug−in defined configuration data is not purged. No application is run, any product specifications are ignored and no UI is presented (e.g., the splash screen is not drawn)

−install <location> (Main)
equivalent to setting osgi.install.area to <location>

−keyring <location> (Runtime)
the location of the authorization database on disk. This argument has to be used together with the −password argument.

−name <string> *NEW*
The name to be displayed in task bar item when the application starts up. When not set, the name is the name of the executable.

−nl <locale> (OSGi)
equivalent to setting osgi.nl to <locale>

−noExit (OSGi)
equivalent to setting osgi.noShutdown to "true"

−noLazyRegistryCacheLoading (Runtime)
equivalent to setting eclipse.noLazyRegistryCacheLoading to "true"

−noRegistryCache (Runtime)
equivalent to setting eclipse.noRegistryCache to "true"

−noSplash (Executable, Main)
controls whether or not the splash screen is shown

−os <operating system> (OSGi)
equivalent to setting osgi.os to <operating system>

−password <password> (Runtime)
the password for the authorization database

−pluginCustomization <location> (Runtime)
equivalent to setting eclipse.pluginCustomization to <location>

−product <id> (OSGi)
equivalent to setting eclipse.product to <id>

−showSplash <command> (Main)
specifies the command to use to show the splash screen. Typically supplied by the Eclipse executable.

−startup <location> (Executable) *NEW*
The location of jar used to startup eclipse. The jar referred to must have the Main−Class attribute set. If this parameter is not set, the startup.jar located in the same folder than the executable is used.

−user <location> (OSGi)
equivalent to setting osgi.user.area to <location>

−vm <path to java executable> (Executable, Main)
when passed to the Eclipse executable, this option is used to locate the Java VM to use to run Eclipse. It must be the full file system path to an appropriate Java executable. If not specified, the Eclipse executable uses a search algorithm to locate a suitable VM. In any event, the executable then passes the path to the actual VM used to Java Main using the −vm argument. Java Main then stores this value in eclipse.vm.

−vmargs [vmargs*] (Executable, Main)
when passed to the Eclipse, this option is used to customize the operation of the Java VM to use to run Eclipse. If specified, this option must come at the end of the command line. Even if not specified on the executable command line, the executable will automatically add the relevant arguments (including the class being launched) to the command line passed into Java using the −vmargs argument. Java Main then stores this value in eclipse.vmargs.

−ws <window system> (OSGi)
equivalent to setting osgi.ws to <window system>

Obsolete command line arguments

The following command line arguments are no longer relevant or have been superceded and are consumed by the runtime and not passed on to the application being run to maintain backward compatibility. .

−boot
    see −configuration
−classLoaderProperties
    no longer relevant
−firstUse
    no longer relevant
−newUpdates
    no longer relevant
−noPackagePrefixes
    no longer relevant
−noUpdate
    no longer relevant
−plugins
    no longer relevant
−update
    no longer relevant

Others

The following command line arguments are defined by various Eclipse plug−ins and are only supported if the defining plug−in is installed, resolved and activated.

−noVersionCheck (workbench)
    <description>
−perspective (workbench)
    <description>
−refresh (org.eclipse.core.resources)
    <description>
−showLocation (org.eclipse.ui.ide.workbench)
    <description>
−allowDeadlock
    <description>

System properties

The following System properties are used by the Eclipse runtime. Note that those starting with "osgi" are specific to the OSGi framework implementation while those starting with "eclipse" are particular to the Eclipse runtime layered on top of the OSGi framework.

Many of these properties have command line equivalents (see the command line arguments section and the value in braces { }). Users are free to use either command line or property settings to specify a value. Properties can be set in the following ways:

- use −DpropName=propValue as a VM argument to the Java VM
- set the desired property in the config.ini file in the appropriate configuration area

eclipse.application {−application}
: the identifier of the application to run. The value given here overrides any application defined by the product (see eclipse.product) being run

eclipse.commands
: a new−line separated list of all command−line arguments passed in when launching Eclipse

eclipse.consoleLog
: if "true", any log output is also sent to Java's System.out (typically back to the command shell if any). Handy when combined with −debug

eclipse.debug.startupTime
: the time in milliseconds when the Java VM for this session was started

eclipse.exitcode
: <description>

eclipse.exitdata
: <description>

eclipse.log.backup.max
: the max number of backup log files to allow. The oldest backup log file will be deleted after the max number of backup log files is reached as a result of rotating the log file. The default value is "10". A negative or zero value will cause the default value to be used.

eclipse.log.size.max
: the max size in Kb that the log file is allowed to grow. The log file is rotated when the file size exceeds the max size. The default value is "1000". A negative value will cause the default value to be used. A zero value indicates no max log size.

eclipse.noExtensionMunging
: if "true", legacy registry extension are left as−is. By default such extensions are updated to use the new extension point ids found in Eclipse 3.0.

eclipse.noLazyRegistryCacheLoading {−noLazyRegistryCacheLoading}
: if "true", the platform's plug−in registry cache loading optimization is deactivated. By default, configuration elements are loaded from the registry cache (when available) only on demand, reducing memory footprint. This option forces the registry cache to be fully loaded at startup.

eclipse.noRegistryCache {−noRegistryCache}
: if "true", the internal extension registry cache is neither read or written

eclipse.pluginCustomization {−pluginCustomization}
: the file system location of a properties file containing default settings for plug−in preferences. These default settings override default settings specified in the primary feature. Relative paths are interpreted relative to the current working directory for eclipse itself.

eclipse.product {−product}
: the identifier of the product being run. This controls various branding information and what application is used.

eclipse.startTime
: This property is set at the time eclipse is started. The value of this property a string representation of the value returned by System.currentTimeMillis(). This value is not intended to be set by users.

eclipse.vm {−vm}
: the path to the Java executable used to run Eclipse. This information is used to construct relaunch command lines.

eclipse.vmargs {−vmargs}
: lists the VM arguments used to run Eclipse. This information is used to construct relaunch command lines.

osgi.adaptor
: the class name of the OSGi framework adaptor to use.

osgi.arch {−arch}
: see −arch

osgi.baseConfiguration.area
: specifies a base configuration that is used when osgi.configuration.area is not specified.

osgi.bundles
: The comma−separated list of bundles which are automatically installed and optionally started once the system is up and running. Each entry is of the form:

```
<URL | simple bundle location>[@ [<start-level>] [":start"]]
```

: If the start−level (>0 integer) is omitted then the framework will use the default start level for the bundle. If the "start" tag is added then the bundle will be marked as started after being installed. Simple bundle locations are interepreted as relative to the framework's parent directory. The start−level indicates the OSGi start level at which the bundle should run. If this value is not set, the system computes an appropriate default.

osgi.clean
: if set to "true", any cached data used by the OSGi framework and eclipse runtime will be wiped clean. This will clean the caches used to store bundle dependency resolution and eclipse extension registry data. Using this option will force eclipse to reinitialize these caches.

osgi.configuration.cascaded
: if set to "true", this configuration is cascaded to a parent configuration. The parent configuration is specified by the osgi.sharedConfiguration.area. See the section on locations for more details.

osgi.configuration.area {−configuration}
: the configuration location for this run of the platform. The configuration determines what plug−ins will run as well as various other system settings. See the section on locations for more details.

osgi.configuration.area.default
: the default configuration location for this run of the platform. The configuration determines what plug−ins will run as well as various other system settings. This value (i.e., the default setting) is only used if no value for the osgi.configuration.area is set. See the section on locations for more details.

osgi.console {−console}
: if set to a non−null value, the OSGi console (if installed) is enabled. If the value is a suitable integer, it is interpreted as the port on which the console listens and directs its output to the given port. Handy for investigating the state of the system.

osgi.console.class
: the class name of the console to run if requested

osgi.debug {−debug}
: if set to a non−null value, the platform is put in debug mode. If the value is a string it is interpreted as the location of the .options file. This file indicates what debug points are available for a plug−in and whether or not they are enabled. If a location is not specified, the platform searches for the .options file under the install directory.

osgi.dev {−dev}
: if set to the empty string, dev mode is simply turned on. This property may also be set to a comma−separated class path entries which are added to the class path of each plug−in or a URL to a Java properties file containing custom classpath additions for a set of plug−ins. For each plug−in requiring a customized dev time classpath the file will contain an entry of the form

```
<plug-in id>=<comma separated list of classpath entries to add>
```

: where plug−in id "*" matches any plug−in not otherwise mentioned.

osgi.framework
: the URL location of the OSGi framework. Useful if the Eclipse install is disjoint. See the section on locations for more details.

osgi.frameworkClassPath

a comma separated list of classpath entries for the OSGi framework implementation. Relative locations are interpreted as relateve to the framework location (see osgi.framework)

osgi.framework.extensions

a comma−separated list of framework extensions. Each entry is of the form:

```
<URL | simple bundle location>
```

If a simple bundle location is specified (not a URL) then a search is done in parent directory of the org.eclipse.osgi bundle. Framework extensions may be used to run Eclipse with a different framework adaptor. The framework extension may contain an eclipse.properties file to set system properties. For example, a framework extension that provides a framework adaptor implementation can specify what the adaptor class is by setting the osgi.adaptor property.

osgi.framework.shape

set to the shape of the Eclipse OSGi Framework implementation. This property is set when the Eclipse platform is started and is not intended by be set by the user. The value "jar" indicates that the Eclipse OSGi Framework is contained in a single jar. The value "folder" indicates that the Eclipse OSGi Framework is contained in a directory.

osgi.java.profile

a URL to the JRE profile file to use. A JRE profile contains values for the properties org.osgi.framework.system.packages and org.osgi.framework.bootdelegation. If the osgi.java.profile is not set then a profile is selected based on the java.specification.version value of the running JRE.

osgi.java.profile.bootdelegation

a java profile osgi.java.profile may contain a "org.osgi.framework.bootdelegation" property. This value may be used to set the system property "org.osgi.framework.bootdelegation". The osgi.java.profile.bootdelegation indicates the policy for bootdelegation to be used. The following values are valid (default is ignore):

- **ignore −** indicates that the "org.osgi.framework.bootdelegation" value in the java profile should be ingored. The system property "org.osgi.framework.bootdelegation" will be used to determine which packages should be delegated to boot.
- **override −** indicates that the "org.osgi.framework.bootdelegation" in the java profile should override the system property "org.osgi.framework.bootdelegation".
- **none −** indicates that the "org.osgi.framework.bootdelegation" in the java profile AND the system properties should be ignored. This is the most strict option. Running with this option causes the framework to use the OSGi R4 strict boot delegation model.

osgi.install.area {−install}

the install location of the platform. This setting indicates the location of the basic Eclipse plug−ins and is useful if the Eclipse install is disjoint. See the section on locations for more details.

osgi.instance.area {−data}

the instance data location for this session. Plug−ins use this location to store their data. For example, the Resources plug−in uses this as the default location for projects (aka the workspace). See the section on locations for more details.

osgi.instance.area.default

the default instance data location for this session. Plug−ins use this location to store their data. For example, the Resources plug−in uses this as the default location for projects (aka the workspace). This value (i.e., the default setting) is only used if no value for the osgi.instance.area is set. See the section on locations for more details.

osgi.locking

the locking type to use for this run of the platform. The valid locking types are "java.io", "java.nio", and "none". The default value is "java.nio" unless the JRE does not support "java.nio" then "java.io" is the default.

osgi.manifest.cache

the location where generated manifests are discovered and generated. The default is in the configuration area but the manifest cache can be stored separately.

osgi.nl {−nl}
: the name of the locale on which Eclipse platform will run. NL values should follow the standard Java locale naming conventions.

osgi.noShutdown {−noExit}
: if "true", the VM will not exit after the eclipse application has ended. This is useful for examining the OSGi framework after the eclipse application has ended.

osgi.os {−os}
: the operating system value. The value should be one of the Eclipse processor architecture names known to Eclipse (e.g., x86, sparc, ...).

osgi.parentClassLoader
: the classloader type to use as the parent classloader for all bundles installed in the Framework. The valid types are the following:
  - **app** − the application classloader.
  - **boot** − the boot classloader.
  - **ext** − the extension classloader.
  - **fwk** − the framework classloader.

osgi.requiredJavaVersion
: The minimum java version that is required to launch Eclipse. The default value is "1.4.1".

osgi.resolverMode
: the mode used to resolve bundles installed in the Framework. A value of "strict" puts the resolver in strict mode. The default resolver mode is not strict. When the resolver is in strict mode the Framework will enforce access restriction rules when loading classes and resources from exported packages which specify the x−internal or x−friends directives.

osgi.sharedConfiguration.area
: the shared configuration location for this run of the platform. If the osgi.configuration.cascaded property is set to "true" then shared configuration area is used as the parent configuration.

osgi.splashLocation
: the absolute URL location of the splash screen (.bmp file) to to show while starting Eclipse. This property overrides any value set in osgi.splashPath.

osgi.splashPath
: a comma separated list of URLs to search for a file called splash.bmp. This property is overriden by any value set in osgi.splashLocation.

osgi.syspath
: set to the path where the eclipse OSGi Framework (org.eclipse.osgi) implementation is located. For example, "<eclipse install path>/eclipse/plugins". This property is set when the Eclipse platform is started and is not intended by be set by the user.

osgi.user.area {−user}
: the location of the user area. The user area contains data (e.g., preferences) specific to the OS user and independent of any Eclipse install, configuration or instance. See the section on locations for more details.

osgi.user.area.default
: the default location of the user area. The user area contains data (e.g., preferences) specific to the OS user and independent of any Eclipse install, configuration or instance. This value (i.e., the default setting) is only used if no value for the osgi.user.area is set. See the section on locations for more details.

osgi.ws {−ws}
: the window system value. The value should be one of the Eclipse window system names known to Eclipse (e.g., win32, motif, ...).

Locations

The Eclipse runtime defines a number of *locations* which give plug−in developers context for reading/storing data and Eclipse users a control over the scope of data sharing and visibility. Eclipse defines the following notions of location:

User (−user) {osgi.user.area} [@none, @noDefault, @user.home, @user.dir, filepath, url]
: User locations are specific to, go figure, users. Typically the user location is based on the value of the Java `user.home` system property but this can be overridden. Information such as user scoped preferences and login information may be found in the user location.

Install (−install) {osgi.install.area} [@user.home, @user.dir, filepath, url]
: An install location is where Eclipse itself is installed. In practice this location is the directory (typically "eclipse") which is the parent of the startup.jar or eclipse.exe being run. This location should be considered read−only to normal users as an install may be shared by many users. It is possible to set the install location and decouple startup.jar from the rest of Eclipse.

Configuration (−configuration) {osgi.configuration.area} [@none, @noDefault, @user.home, @user.dir, filepath, url]
: Configuration locations contain files which identify and manage the (sub)set of an install to run. As such, there may be many configurations per install. Installs may come with a default configuration area but typical startup scenarios involve the runtime attempting to find a more writable configuration location.

Instance (−data) {osgi.instance.area} [@none, @noDefault, @user.home, @user.dir, filepath, url]
: Instance locations contain user−defined data artifacts. For example, the Resources plug−in uses the instance area as the workspace location and thus the default home for projects. Other plugins are free to write whatever files they like in this location.

While users can set any of these locations, Eclipse will compute reasonable defaults if values are not given. The most common usecase for setting location is the instance area or, in the IDE context, the workspace. To run the default Eclipse configuration on a specific data set you can specify:

```
eclipse -data c:\mydata
```

More detail

Locations are URLs. For simplicity, file paths are also accepted and automatically converted to file: URLs. For better control and convenience, there are also a number of predefined symbolic locations which can be used. Note that not all combinations of location type and symbolic value are valid. A table below details which combinations are possible. Since the default case is for all locations to be set, valid and writable, some plug−ins may fail in other setups even if they are listed as possible. For example, it is unreasonable to expect a plugin focused on user data (e.g., the Eclipse Resources plug−in) to do much if the instance area is not defined. It is up to plug−in developers to choose the setups they support and design their function accordingly.

@none
: Indicates that the corresponding location should never be set either explicitly or to its default value. For example, an RCP style application which has no user data may use osgi.instance.area=@none to prevent extraneous files being written to disk. @none must not be followed by any additional path segments.

@noDefault
: Forces a location to be undefined or explicitly defined (i.e., Eclipse will not automatically compute a default value). This is useful where you want to allow for data in the corresponding location but the

Eclipse default value is not appropriate. @noDefault must not be followed by any additional path segments.

@user.home
: Directs Eclipse to compute a location value relative to the user's home directory. @user.home can be followed by additional path segments. In all cases, the string "@user.home" is simply replaced with the value of the Java "user.home" System property. For example, setting
  osgi.instance.area=@user.home/myWorkspace
  results in a value of
  file:/users/bob/myWorkspace

@user.dir
: Directs Eclipse to compute a location value relative to the current working directory. @user.dir can be followed by additional path segments. In all cases, the string "@user.dir" is simply replaced with the value of the Java "user.dir" System property. For example, setting
  osgi.instance.area=@user.dir/myWorkspace
  results in a value of
  file:/usr/share/eclipse/myWorkspace

location/value	supports default	file/URL	@none	@noDefault	@user.home	@user.dir
instance	yes	yes	yes	yes	yes	yes (default)
configuration	yes	yes	yes*	yes*	yes	yes
install	no	yes	no	no	yes	yes
user	yes	yes	yes	yes	yes	yes

* indicates that this setup is technically possible but pragmatically quite difficult to manage. In particular, without a configuration location the Eclipse runtime may only get as far as starting the OSGi framework.

Read–only Locations

A location may be specified as a read–only location by appending ".readOnly" to the location property and setting it to the value "true". The following properties can be used to specify their corresponding locations as read–only:

- osgi.configuration.area.readOnly osgi.configuration.area
- osgi.sharedConfiguration.area.readOnly osgi.sharedConfiguration.area
- osgi.instance.area.readOnly osgi.instance.area
- osgi.user.area.readOnly osgi.user.area

Launcher ini file

The eclipse.exe and more generally executables for RCP applications now read their parameters from an associated ini file. This file offers a platform independent way to pass in arguments that previously had to be specified directly on the command line such as vm or vm arguments. Although all parameters can be specified in this file, it recommend for maintainability and consistency across various installations to only specifiy the vm location and the vm arguments in this ini file and use the config.ini file for others.

File format

This file must be named after the executable name (for example, eclipse.exe will read eclipse.ini, whereas launcher.exe will read launcher.ini) and every parameter must be specified on a new line in the file. Here is an

example of such a file specifying the vm location and some parameters:

```
-vm
c:/myVm/java.exe
-vmargs
-Dms40M
```

Starting Eclipse from Java

Last modified 09:00 Wednesday June 23, 2004

The Eclipse Platform makes heavy use of Java class loaders for loading plug−ins. Even the Eclipse Runtime itself and the OSGi framework need to be loaded by special class loaders. Client programs, such as a Java main program or a servlet, cannot directly reference any part of Eclipse directly. Instead, a client must use the **EclipseStarter** class in **org.eclipse.core.runtime.adaptor** to start the platform, invoking functionality defined in plug−ins, and shutting down the platform when done.

Clients that do not need to access any particular function, but just need to start the platform, can use `org.eclipse.core.launcher.Main.run()` in `startup.jar`. However, clients that need to invoke specific function should use **EclipseStarter**. See the javadoc inside this class for details.

Eclipse platform
API rules of engagement

Version 0.15 – Last revised 12:00 May 30, 2001

Here are the rules of engagement for clients of the Eclipse platform API (and other components).

What it means to be API

The Eclipse platform defines API elements for use by its clients, namely ISVs writing plug–ins. These plug–ins may in turn define API elements for their clients, and so on. API elements are the public face: they carry a specification about what they are supposed to do, and about how they are intended to be used. API elements are supported: the Eclipse platform team will fix implementation bugs where there is a deviation from the specified behavior. Since there is often a high cost associated with breaking API changes, the Eclipse platform team will also try to evolve API elements gracefully through successive major releases.

How to tell API from non–API

By their very nature, API elements are documented and have a specification, in contrast to non–API elements which are internal implementation details usually without published documentation or specifications. So if you cannot find the documentation for something, that's usually a good indicator that it's not API.

To try to draw the line more starkly, the code base for the platform is separated into API and non–API packages, with all API elements being declared in designated API packages.

- **API package –** a Java package that contains at least one API class or API interface. The names of API packages are advertised in the documentation for that component; where feasible, all other packages containing only implementation details have "internal" in the package name. The names of API packages may legitimately appear in client code. For the Eclipse platform proper, these are:

- `org.eclipse.foo.*` – for example, `org.eclipse.swt.widgets`, `org.eclipse.ui`, or `org.eclipse.core.runtime`
- `org.eclipse.foo.internal.*` – not API; internal implementation packages
- `org.eclipse.foo.examples.*` – not API; these are examples
- `org.eclipse.foo.tests.*` – not API; these are test suites

- **API class or interface –** a `public` class or interface in an API package, or a `public` or `protected` class or interface member declared in, or inherited by, some other API class or interface. The names of API classes and interfaces may legitimately appear in client code.
- **API method or constructor –** a `public` or `protected` method or constructor either declared in, or inherited by, an API class or interface. The names of API methods may legitimately appear in client code.
- **API field –** a `public` or `protected` field either declared in, or inherited by, an API class or interface. The names of API fields may legitimately appear in client code.

Everything else is considered internal implementation detail and off limits to all clients. Legitimate client code must never reference the names of non–API elements (not even using Java reflection). In some cases, the Java language's name accessibility rules are used to disallow illegal references. However, there are many

cases where this is simply not possible. Observing this one simple rule avoids the problem completely:

- **Stick to officially documented APIs.** Only reference packages that are documented in the ***published API Javadoc*** for the component. Never reference a package belonging to another component that has "internal" in its name−−−these are never API. Never reference a package for which there is no published API Javadoc−−−these are not API either.

General rules

The specification of API elements is generated from Javadoc comments in the element's Java source code. For some types of elements, the specification is in the form of a contract. For example, in the case of methods, the contract is between two parties, the caller of the method and the implementor of the method. The fundamental ground rule is:

- **Honor all contracts.** The contracts are described in the published Javadoc for the API elements you are using.

The term "must", when used in an API contract, means that it is incumbent on the party to ensure that the condition would always be met; any failure to do so would be considered a programming error with unspecified (and perhaps unpredictable) consequences.

- **You must honor "must".** Pay especially close heed to conditions where "must" is used.

Other common sense rules:

- **Do not rely on incidental behavior.** Incidental behavior is behavior observed by experiment or in practice, but which is not guaranteed by any API specification.
- **Do not treat null as an object.** Null is more the lack of an object. Assume everything is non−null unless the API specification says otherwise.
- **Do not try to cheat with Java reflection.** Using Java reflection to circumvent Java compiler checking buys you nothing more. There are no additional API contracts for uses of reflection; reflection simply increases the likelihood of relying on unspecified behavior and internal implementation detail.
- **Use your own packages.** Do not declare code in a package belonging to another component. Always declare your own code in your own packages.

Calling public API methods

For most clients, the bulk of the Eclipse API takes the form of public methods on API interfaces or classes, provided for the client to call when appropriate.

- **Ensure preconditions.** Do ensure that an API method's preconditions are met before calling the method. Conversely, the caller may safely assume that the method's postconditions will have been achieved immediately upon return from the call.
- **Null parameters.** Do not pass null as a parameter to an API method unless the parameter is explicitly documented as allowing null. This is perhaps the most frequently made programming error.
- **Restricted callers.** Do not call an API method that is documented as available only to certain callers unless you're one of them. In some situations, methods need to be part of the public API for the benefit of a certain class of callers (often internal); calling one of these methods at the wrong time has unspecified (and perhaps unpredictable) consequences.

- **Debugging methods.** Do not call an API method labelled "for debugging purposes only". For example, most `toString()` methods are in this category.
- **Parameter capture.** Do not pass an array, collection, or other mutable object as a parameter to an API method and then modify the object passed in. This is just asking for trouble.

Instantiating platform API classes

Not all concrete API classes are intended to be instantiated by just anyone. API classes have an instantiation contract indicating the terms under which instances may be created. The contract may also cover things like residual initialization responsibilities (for example, configuring a certain property before the instance is fully active) and associated lifecycle responsibilities (for example, calling `dispose()` to free up OS resources hung on to by the instance). Classes that are designed to be instantiated by clients are explicitly flagged in the Javadoc class comment (with words like "Clients may instantiate.").

- **Restricted instantiators.** Do not instantiate an API class that is documented as available only to certain parties unless you're one of them. In some situations, classes need to be part of the public API for the benefit of a certain party (often internal); instantiating one of these classes incorrectly has unspecified (and perhaps unpredictable) consequences.

Subclassing platform API classes

Only a subset of the API classes were designed to be subclassed. API classes have a subclass contract indicating the terms under which subclasses may be declared. This contract also covers initialization responsibilities and lifecycle responsibilities. Classes that are designed to be subclassed by clients are explicitly flagged in the Javadoc class comment (with words like "Clients may subclass.").

- **Restricted subclassers.** Do not subclass an API class that is not intended to be subclassed. Treat these classes as if they had been declared final. (These are sometimes referred to as "soft final" classes).

Calling protected API methods

Calling inherited protected and public methods from within a subclass is generally allowed; however, this often requires more care to correctly call than to call public methods from outside the hierarchy.

Overriding API methods

Only a subset of the public and protected API methods were designed to be overridden. Each API method has a subclass contract indicating the terms under which a subclass may override it. By default, overriding is not permitted. It is important to check the subclass contract on the actual method implementation being overridden; the terms of subclass contracts are not automatically passed along when that method is overridden.

- **Do not override a public or protected API method unless it is explicitly allowed.** Unless otherwise indicated, treat all methods as if they had been declared final. (These are sometimes known as "soft final" methods). If the kind of overriding allowed is:
    - "**implement**" – the abstract method declared on the subclass must be implemented by a concrete subclass

- "**extend**" − the method declared on the subclass must invoke the method on the superclass (exactly once)
- "**re−implement**" − the method declared on the subclass must not invoke the method on the superclass
- "**override**" − the method declared on the subclass is free to invoke the method on the superclass as it sees fit

- **Ensure postconditions.** Do ensure that any postconditions specified for the API method are met by the implementation upon return.
- **Proactively check preconditions.** Do not presume that preconditions specified for the API method have necessarily been met upon entry. Although the method implementation would be within its rights to not check specified preconditions, it is usually a good idea to check preconditions (when feasible and reasonably inexpensive) in order to blow the whistle on misbehaving callers.
- **Null result.** Do not return null as a result from an API method unless the result is explicitly documented (on the specifying interface or superclass) as allowing null.
- **Return copies.** Do not return an irreplaceable array, collection, or other mutable object as the result from an API method. Always return a copy to avoid trouble from callers that might modify the object.

Implementing platform API interfaces

Only a subset of the API interfaces were designed to be implemented by clients. API interfaces have a contract indicating the terms under which it may be implemented. Interfaces that are designed to be implemented by clients are explicitly flagged in the Javadoc class comment (with words like "Clients may implement."). A client may declare a subinterface of an API interface if and only if they are allowed to implement it.

- **Restricted implementors.** Do not implement an API interface that is documented as available only to certain parties unless you're one of them. In many situations, interfaces are used to hide internal implementation details from view.

Implementing public API methods

See "Overriding API methods".

Accessing fields in API classes and interfaces

Clients may read API fields, most of which are final. Certain struct−like objects may have non−final public fields, which clients may read and write unless otherwise indicated.

- **Null fields.** Do not set an API field to null unless this is explicitly allowed.

Casting objects of a known API type

An object of a known API type may only be cast to a different API type (or conditionally cast using instanceof) if this is explicitly allowed in the API.

- **Cast and instanceof.** Do not use instanceof and cast expressions to increase what is known about an object beyond what the API supports. Improper use exposes incidental implementation details not guaranteed by the API.

And, of course, casting any object to a non−API class or interface is always inappropriate.

Not following the rules

Whether done knowingly or unwittingly, there are consequences for transgressing the rules. It might be easier for all involved if there were API police that would bust you for breaking the rules. However, that is not the case. For the most part, API conformance operates as an honor system, with each client responsible for knowing the rules and adhering to them.

The contracts on the API elements delimit the behavior that is supported and sustained. As the Eclipse platform matures and evolves, it will be the API contracts that guide how this evolution happens. Outside of these contracts, everything is unsupported and subject to change, without notice, and at any time (even mid−release or between different OS platforms). Client code that oversteps the above rules might fail on different versions and patch levels of the platform; or when run on different underlying OSes; or when run with a different mix of co−resident plug−ins; or when run with a different workbench perspective; and so on. Indeed, no one is even particularly interested in speculating exactly how any particular transgression might come back to bite you. To those who choose to ignore the rules, don't say that you weren't warned. And don't expect much more than a sympathetic "Told you so."

On the other hand, client plug−in code that lives by the above rules should continue to work across different versions and patch levels of the platform, across different underlying OSes, and should peacefully co−exist with other plug−ins. If everyone plays by the rules, the Eclipse platform will provide a stable and supported base on which to build exciting new products.

Eclipse Platform Naming Conventions

Last revised June 24, 2002 – version for Eclipse project R2.0

Naming conventions and guidelines for the Eclipse Platform:

- Java packages
- Classes and interfaces
- Methods
- Variables
- Plug–ins and extension points

Java Packages

The Eclipse Platform consists of a collection of Java packages. The package namespace is managed in conformance with Sun's package naming guidelines; subpackages should not be created without prior approval from the owner of the package sub–tree. The packages for the Eclipse platform are all subpackages **`org.eclipse`**. The first package name component after **`org.eclipse`** is called the *major* package name. The following major packages of **`org.eclipse`** are assigned in the Eclipse 2.0 release:

`org.eclipse.ant[.*]` – Ant support
`org.eclipse.compare[.*]` – Compare support
`org.eclipse.core[.*]` – Platform core
`org.eclipse.debug[.*]` – Debug
`org.eclipse.help[.*]` – Help support
`org.eclipse.jdi[.*]` – Eclipse implementation of Java Debug Interface (JDI)
`org.eclipse.jdt[.*]` – Java development tools
`org.eclipse.jface[.*]` – JFace
`org.eclipse.pde[.*]` – Plug–in Development Environment
`org.eclipse.search[.*]` – Search support
`org.eclipse.swt[.*]` – Standard Widget Toolkit
`org.eclipse.team[.*]` – Team support and Version and Configuration Management
`org.eclipse.tomcat[.*]` – Apache tomcat support
`org.eclipse.ui[.*]` – Workbench
`org.eclipse.update[.*]` – Update/install
`org.eclipse.webdav[.*]` – WebDAV support

The following package name segments are reserved:

`internal` – indicates an internal implementation package that contains no API
`tests` – indicates a non–API package that contains only test suites
`examples` – indicates a non–API package that contains only examples

These name are used as qualifiers, and must only appear following the major package name. For example,

`org.eclipse.core.internal.resources` – Correct usage
`org.eclipse.internal.core.resources` – Incorrect. **`internal`** precedes major package name.

`org.eclipse.core.resources.internal` – Incorrect. **`internal`** does not immediately follow major package name.

Aside on how the Eclipse Platform is structured: The Eclipse Platform is divided up into *Core* and *UI*. Anything classified as Core is independent of the window system; applications and plug–ins that depend on the Core and not on the UI can run headless. The distinction between Core and UI does not align with the distinction between API and non–API; both Core and UI contain API. The UI portion of the Eclipse Platform is known as the Workbench. The Workbench is a high–level UI framework for building products with sophisticated UIs built from pluggable components. The Workbench is built atop JFace, SWT, and the Platform Core. SWT (Standard Widget Toolkit) is a low–level, OS–platform–independent means of talking to the native window system. JFace is a mid–level UI framework useful for building complex UI pieces such as property viewers. SWT and JFace are UI by definition. The Java tooling is a Java IDE built atop the workbench. End aside.

API Packages API packages are ones that contain classes and interfaces that must be made available to ISVs. The names of API packages need to make sense to the ISV. The number of different packages that the ISV needs to remember should be small, since a profusion of API packages can make it difficult for ISVs to know which packages they need to import. Within an API package, all public classes and interfaces are considered API. The names of API packages should not contain **`internal`**, **`tests`**, or **`examples`** to avoid confusion with the scheme for naming non–API packages.

Internal Implementation Packages All packages that are part of the platform implementation but contain no API that should be exposed to ISVs are considered internal implementation packages. All implementation packages should be flagged as **`internal`**, with the tag occurring just after the major package name. ISVs will be told that all packages marked **`internal`** are out of bounds. (A simple text search for "`.internal.`" detects suspicious reference in source files; likewise, "`/internal/`" is suspicious in .class files).

Test Suite Packages All packages containing test suites should be flagged as **`tests`**, with the tag occurring just after the major package name. Fully automated tests are the norm; so, for example, `org.eclipse.core.tests.resources` would contain automated tests for API in `org.eclipse.core.resources`. Interactive tests (ones requiring a hands–on tester) should be flagged with **`interactive`** as the *last* package name segment; so, for example, `org.eclipse.core.tests.resources.interactive` would contain the corresponding interactive tests.

Examples Packages All packages containing examples that ship to ISVs should be flagged as **`examples`**, with the tag occurring just after the major package name. For example, `org.eclipse.swt.examples` would contain examples for how to use the SWT API.

Additional rules:

- Package names should contain only lowercase ASCII alphanumerics, and avoid underscore _ or dollar sign `$` characters.

Classes and Interfaces

Sun's naming guidelines states

> Class names should be nouns, in mixed case with the first letter of each internal word capitalized. Try to keep your class names simple and descriptive. Use whole words – avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML).
>
> Examples:
> ```
> class Raster;
> class ImageSprite;
> ```
>
> Interface names should be capitalized like class names.

For interface names, we follow the "I"–for–interface convention: all interface names are prefixed with an "`I`". For example, "`IWorkspace`" or "`IIndex`". This convention aids code readability by making interface names more readily recognizable. (Microsoft COM interfaces subscribe to this convention).

Additional rules:

- The names of exception classes (subclasses of `Exception`) should follow the common practice of ending in "`Exception`".

Methods

Sun's naming guidelines states

> Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized.
>
> Examples:
> ```
> run();
> runFast();
> getBackground();
> ```

Additional rules:

- The named of methods should follow common practice for naming getters (`get`*`X`*`()`), setters (`set`*`X`*`()`), and predicates (`is`*`X`*`()`, `has`*`X`*`()`).

Variables

Sun's naming guidelines states

> Except for variables, all instance, class, and class constants are in mixed case with a lowercase first letter. Internal words start with capital letters. Variable names should not start with underscore _ or dollar sign `$` characters, even though both are allowed.
>
> Variable names should be short yet meaningful. The choice of a variable name should be mnemonic – that is, designed to indicate to the casual observer the intent of its use. One–character variable names should be avoided except for temporary "throwaway" variables. Common names for temporary variables are `i`, `j`, `k`, `m`, and `n` for integers; `c`, `d`, and `e` for characters.

Examples:

```
int i;
char c;
float myWidth;
```

(Note: we are no longer following the convention that prefixes non−constant field names with "`f`", such as "`fWidget`".)

Constants

Sun's naming guidelines states

> The names of variables declared class constants and of ANSI constants should be all uppercase with words separated by underscores ("_").

Examples:

```
static final int MIN_WIDTH = 4;
static final int MAX_WIDTH = 999;
static final int GET_THE_CPU = 1;
```

Plug−ins and Extension Points

All plug−ins, including the ones that are part of the Eclipse Platform, like the Resources and Workbench plug−ins, must have unique identifiers following the same naming pattern as Java packages. For example, workbench plug−ins are named **`org.eclipse.ui[.*]`**.

The plug−in namespace is managed hierarchically; do not create plug−in without prior approval from the owner of the enclosing namespace.

Extension points that expect multiple extensions should have plural names. For example, "`builders`" rather than "`builder`".

Glossary of terms

It can be confusing when one person refers to the same thing with different terms, and downright misleading if they refer to different things using the same term. That's why it's important to choose the right words when referring to things in comments, code, and documentation.

Eclipse

> Eclipse is the name for the overall project supporting the construction of integrated tools for developing applications. It represents a collection of related projects that include the Eclipse Platform, the Java development tools (JDT), and the Plug−in Development Environment (PDE).

Eclipse Platform

> Eclipse Platform is the name for the core frameworks and services upon which plug−in extensions are created. It provides the runtime in which plug−ins are loaded and run. In order to use the term with the right degree of (im)precision, it is useful to know some things about the Platform. First off, the Eclipse Platform itself is not really a true product that would ship by itself. The Platform's direct consumers are tool builders, or ISVs, since they add the value to the Eclipse Platform that makes it useful to people.
> The Eclipse Platform is divided up into *Core* and *UI*. Anything classified as "UI" needs a window system, whereas things classified as "Core" can run "headless". The UI portion of the Eclipse Platform is known as the Workbench. The core portion of the Eclipse Platform is simply called the Platform Core, or Core.
> So the Eclipse Platform is just the nucleus around which tool builders build tool plug−ins.

Eclipse SDK

> The Eclipse SDK is the Eclipse Platform, JDT, and PDE. In addition to the Platform, the SDK provides the development tools required to, among other things, enable Eclipse to be a development environment for itself.

Platform − Short for "Eclipse Platform".

Workbench − Short for "Eclipse Platform UI".

> The Workbench is a high−level UI framework for building products with sophisticated UIs built from pluggable components. The Workbench is built atop JFace, SWT, and the Platform Core.

Core − Short for "Eclipse Platform Core".

> All the UI−free infrastructure of the Eclipse Platform. The major divisions are: platform runtime and plug−in management, workspaces and resource management, and version and configuration management.

Runtime − Short for "Eclipse Platform Core Runtime".

The lowest level part of the Platform Core, responsible for the plug−in registry and plug−ins. Note that the Platform Core Runtime does not include workspaces and resources (they're in the Resources plug−in).

Workspace

A workspace is the general umbrella for managing resources in the Eclipse Platform. Note that workspaces and resources are an optional part of the Platform; some configurations of the Platform will not have a workspace.

UI − Short for "Eclipse Platform UI".

All−inclusive term for the UI portion of the Eclipse Platform.

JFace

JFace is the mid−level UI framework useful for building complex UI pieces such as property viewers. JFace works in conjunction with SWT.

SWT

SWT (Standard Widget Toolkit) is a small, fast widget toolkit with a portable API and a native implementation. So far, SWT has been ported to Windows, Linux (GTK and Motif window systems), AIX (Motif), Solaris (Motif), HP−UX (Motif), QNX (Photon) and Mac OS X (Carbon).

JDT

Java development tools (n.b. "development tools" in lowercase, for legal reasons) adds Java program development capability to the Eclipse Platform.

PDE

The Plug−in Development Environment adds specialized tools for developing Eclipse plug−ins.

The project description file

Description: When a project is created in the workspace, a project description file is automatically generated that describes the project. The purpose of this file is to make the project self−describing, so that a project that is zipped up or released to a server can be correctly recreated in another workspace. This file is always called ".project", and is located as a direct member of the project's content area. The name of the file is exposed through the static field `DESCRIPTION_FILE_NAME` on `org.eclipse.core.resources.IProjectDescription`.

The name, location, and content of this file are part of the workspace API. This means they are guaranteed not to change in a way that would break existing users of the file. However, the right to add additional elements and attributes to the markup is reserved for possible future additions to the file. For this reason, clients that read the description file contents should tolerate unknown elements and attributes.

Clients that modify, delete, or replace the project description file do so at their own risk. Projects with invalid or missing description files will not be generally usable. If a project with an invalid description file is discovered on workspace startup, the project is closed, and it will not be possible to open it until the project description file has been repaired. The workspace will not generally attempt to automatically repair a missing or invalid description file. One exception is that missing project description files will be regenerated during workspace save and on calls to `IProject.setDescription`.

Modifications to the project description file have mostly the same effect as changing the project description via `IProject.setDescription`. One exception is that adding or removing project natures will not trigger the corresponding nature's configure or deconfigure method. Another exception is that changes to the project name are ignored.

If a new project is created at a location that contains an existing project description file, the contents of that description file will be honoured as the project description. One exception is that the project name in the file will be ignored if it does not match the name of the project being created. If the description file on disk is invalid, the project creation will fail.

Configuration Markup:

```
   <!ELEMENT projectDescription (name, comment, projects, buildSpec,
natures, linkedResources)>

   <!ELEMENT name EMPTY>
```

- **name –** the name of the project. Ignored if it does match the name of the project using this description. Corresponds to `IProjectDescription.getName()`.

```
   <!ELEMENT comment EMPTY>
```

- **comment –** a comment for the project. The comment can have arbitrary contents that are not interpreted by the project or workspace. Corresponds to `IProjectDescription.getComment()`.

```
<!ELEMENT projects (project)*>
<!ELEMENT project EMPTY>
```

- **projects –** the names of the projects that are referenced by this project. Corresponds to `IProjectDescription.getReferencedProjects()`.

```
<!ELEMENT buildSpec (buildCommand)*>

<!ELEMENT buildCommand (name, arguments)>

<!ELEMENT name EMPTY>

<!ELEMENT arguments (dictionary?)>

<!ELEMENT dictionary (key, value)*>

<!ELEMENT key EMPTY>

<!ELEMENT value EMPTY>
```

- **buildSpec –** the ordered list of build commands for this project. Corresponds to `IProjectDescription.getBuildSpec()`.
- **buildCommand –** a single build commands for this project. Corresponds to `org.eclipse.core.resources.ICommand`.
- **name –** the symbolic name of an incremental project builder. Corresponds to `ICommand.getBuilderName()`.
- **arguments –** optional arguments that may be passed to the project builder. Corresponds to `ICommand.getArguments()`.
- **dictionary –** a list of <key, value> pairs in the argument list. Analagous to `java.util.Map`.

```
<!ELEMENT natures (nature)*>

<!ELEMENT nature EMPTY>
```

- **natures –** the names of the natures that are on this project. Corresponds to `IProjectDescription.getNatureIds()`.
- **nature –** the name of a single natures on this project.

```
<!ELEMENT linkedResources (link)*>

<!ELEMENT link (name, type, location)>

<!ELEMENT name EMPTY>

<!ELEMENT type EMPTY>

<!ELEMENT location EMPTY>
```

- **linkedResources –** the list of linked resources for this project.
- **link –** the definition of a single linked resource.
- **name –** the name of the linked resource as it appears in the workspace.

- **type –** the resource type. Value values are: "1" for a file, or "2" for a folder.
- **location –** the file system path of the target of the linked resource. Either an absolute path, or a relative path whose first segment is the name of a workspace path variable.

Examples: The following is a sample project description file. The project has a single nature and builder configured, and some project references.

```
<?xml version="1.0" encoding="UTF-8"?>
<projectDescription>
    <name>WinterProject</name>
    <comment>This is a cool project.</comment>
    <projects>
        <project>org.seasons.sdt</project>
        <project>CoolStuff</project>
    </projects>
    <buildSpec>
        <buildCommand>
            <name>org.seasons.sdt.seasonBuilder</name>
            <arguments>
                <dictionary>
                    <key>climate</key>
                    <value>cold</value>
                </dictionary>
            </arguments>
        </buildCommand>
    </buildSpec>
    <natures>
        <nature>org.seasons.sdt.seasonNature</nature>
    </natures>
</projectDescription>
```

API Information: The contents of the project description file map to the `org.eclipse.core.resources.IProjectDescription` interface. The project description file can be overwritten by the method `IProject.setDescription()`.

Message Bundles In Eclipse 3.1

Description

Standard Java ResourceBundles have quite inefficient space characteristics. Since a running Eclipse tends to have many externalized messages we have implemented a new message bundle story to be used in Eclipse. The mechanism is quite simple and completely generic – it can be used anywhere.

Summary of the new approach:

- `messages.properties` – this file is same as before except all keys need to be valid Java identifiers.
- Each message file has a corresponding Java class.
- Each key/value pair in the file has a `public static String` field whose name is the same as the message key.
- When message bundles are loaded, the values of the fields are set to be the values from the `messages.properties` files.
- The message properties files are purged from memory.

When creating a new message:

- Create a field in your `Messages.java` file.
- Create a key/value pair in your `messages.properties` file where the key name matches the field name,
- To reference the message, simply reference the field (e.g. `Messages.my_key`) rather than the standard lookup.

Example Files:

Client Code

Old Code:

```
public class MyClass {
  public void myMethod() {
    String message;
    ...
    // no args
    message = Messages.getString("key.one"); //$NON-NLS-1$
    ...
    // bind one arg
    message = MessageFormat.format(Messages.getString("key.two"), new Object[] {"example usage"})
    ...
  }
}
```

New Code:

```
public class MyClass {
  public void myMethod() {
    String message;
```

```
    ...
    // no args
    message = Messages.key_one;
    ...
    // bind one arg
    message = NLS.bind(Messages.key_two, "example usage"); //$NON-NLS-1$
    ...
  }
}
```

Messages.java

Old Code:

```
public class Messages {
  private static final String BUNDLE_NAME = "org.eclipse.core.utils.messages"; //$NON-NLS-1$
  private static final ResourceBundle bundle = ResourceBundle.getBundle(BUNDLE_NAME);

  public static String getString(String key) {
    try {
      return bundle.getString(key);
    } catch (MissingResourceException e) {
      return key;
    }
  }
}
```

New Code:

```
import org.eclipse.osgi.util.NLS;

public class Messages extends NLS {
  private static final String BUNDLE_NAME = "org.eclipse.core.utils.messages"; //$NON-NLS-1$

  public static String key_one;
  public static String key_two;
  ...
  static {
    NLS.initializeMessages(BUNDLE_NAME, Messages.class);
  }
}
```

messages.properties

Old Code:

```
key.one = Hello world.
key.two = This is an {0} of binding with one argument.
```

New Code:

```
key_one = Hello world.
key_two = This is an {0} of binding with one argument.
```

Eclipse multi−user installs

Eclipse provides a number of strategies for supporting multi−user installs. Each strategy satisfies a specific scenario. This document covers these strategies, describing when each one should be used. The intended readers are product engineers configuring an Eclipse−based product for distribution, system administrators setting up Eclipse−based products to be used through a network and developers interested in creating plug−ins that are good citizens in such setups.

Last Modified: June 17th, 2005

Basic concepts

Locations

As described in the Eclipse Runtime options article, there are three different locations that are important in the context of deploying Eclipse in a multi−user setup:

- ***install area*** − the location where the Eclipse Platform is installed. This location can be read−only, since neither the Eclipse runtime nor plug−ins are supposed to write there. However, in a single user scenario, the configuration area, which is typically writable, defaults to a directory called "configuration" stored under the install area.
- ***configuration area*** − the location where Eclipse stores essential runtime metadata (such as information about the set of plug−ins installed and the dependencies between them) and cached data in general. Eclipse cannot run without a configuration area. Plug−ins may choose to store data here that should be available regardless the workspace in use (for instance, help index files). User settings shared across workspaces are also stored under this location.
- ***instance area*** − the location where user files are stored, commonly called the *workspace*. It is optional, however most Eclipse−based products require an instance area to work. The instance area cannot be shared, although a single user might maintain multiple instance areas. Inside the instance area there is a special directory called .metadata, where plug−ins store their own workspace−specific metadata and user settings.

Configuration initialization

Before Eclipse has been run for the first time, the configuration area is basically an empty directory. This location is gradually populated by the Eclipse runtime, and other plug−ins, across Eclipse sessions. Most of the metadata kept by the Eclipse runtime (e.g. plug−in dependencies, the extension registry) is written during the shutdown of the first session. If no changes are made to the set of installed plug−ins, no data has to be written during subsequent sessions. We say the configuration is *initialized*. When the configuration is in this state, it is even possible to make the configuration area read−only. Making the configuration area read−only is useful in scenarios such as shared configurations (more on this later).

The `−initialize` command−line option allows one to initialize the configuration area without requiring an Eclipse application to run. The initialization procedure forces the creation of any metadata that is written out to the configuration location during the first Eclipse session. However, there are other files kept in the configuration area that are created only when needed. Examples are:

- Help indexes − only created for the first time the Eclipse Help system is opened. After it is created, provided the user locale does not change, no files are ever written again.

- files extracted out of plug−in JARs − plug−ins shipped as JARs often need to access embedded files as plain files on the file system. For that, plug−ins call the Eclipse Runtime API `Platform.asLocalURL(URL)`. The result is that if the URL refers to a file inside a JAR, that file will be extracted to the file system under the configuration area. Once a file is extracted, subsequent calls to `Platform.asLocalURL()` will be able to find it, so no additional extractions will occur *for that file*. A similar (actually, the original) scenario where `Platform.asLocalURL` is used that has the same consequences relates to making sure remote contents (for instance, a file accessible via a http URL) are available locally.

For these cases (and others that 3rd−party plug−ins might introduce), the initialization procedure is not sufficient to fully initialize the configuration area. There will still be need to write to the configuration area although this need tends to disappear as all the execution paths in the application that cause files to be created in the configuration area are visited. Only after that it can be said the configuration area is fully initialized and no write access to it will ever be required in order for Eclipse to run.

Scenario #1 – private install

This is actually a single−user scenario. The Eclipse install is used by a single user, and the user has full access privileges to it. The configuration area location defaults to the configuration directory under the install location.

The procedure to set up this scenario requires just making sure the user has full rights to the install location.

Scenario #2 – shared install

In this scenario, a single install area is shared by many users. The "configuration" directory under the install area is home only to the config.ini as shipped with the product (it is not initialized). Every user has their own local standalone configuration location.

The set up for this scenario requires making the install area read−only for regular users. When users start Eclipse, this causes the configuration area to automatically default to a directory under the user home dir. If this measure is not taken, all users will end up using the same location for their configuration area, which is not supported.

Scenario #3 – shared configuration

Here users share not only an install area but also a master configuration area. Users still have, by default, their own private writable configuration areas. A user's private configuration area is cascaded to the master configuration and will not contain any interesting data if the master configuration has been fully initialized and no changes to the set of plug−ins to be installed has occurred.

In this scenario, the system administrator initializes the master configuration (typically under the install location), and ensures the whole install and configuration areas are read−only to users. When users run the Eclipse−based product from the shared install location, since they do not have write access privileges to the configuration area under the install area, a local configuration area will be automatically computed and initialized.

The more fully initialized the shared configuration is, the less need there is for files to be created under the local configuration.

Setting the private configuration area location

The default location for a private configuration area is:

```
<user-home-dir>/.eclipse/<product-id>_<product-version>/configuration
```

The user home dir is determined by the `user.home` Java system property. The product id and version are obtained from the product marker file `.eclipseproduct` under the Eclipse install.

A non−default configuration area can be defined by setting the `osgi.configuration.area` system property. This property can be set by the end user, but it is more convenient to set it either in the launcher .ini file or the in the config.ini file at the base configuration location.

Updating

Shared configuration

Plug−ins can be installed in/removed from the shared configuration. Users will catch up with those changes the next time Eclipse runs. It is necessary to make sure users having the shared configuration as their master configuration are not running Eclipse.

Private configuration

Users can modify their local configuration areas by installing additional plug−ins. This does not cause any changes to the shared configuration, so other users will not see the changes. Note that plug−ins configured at the shared configuration *cannot* be removed. If they are, they will be reinstalled the next time the platform starts.

Pre−built documentation index

When user searches help contents of a product, the search is performed within a documentation index. By default, this index is created on the first invocation of help search, but can be pre−built and delivered to the user with each plug−in, since 3.1, or as a complete index for a product. This prevents indexing from occurring on the user machine and lets the user obtain first search results faster.

Building a documentation index for a plug−in.

To build an index follow the steps:

- add an `index` element to the `org.eclipse.help.toc extension` in a documentation plug−in, to specify directory where index will exist,
- create an index for a desired locale by running the `help.buildHelpIndex` ANT task for each plug−in and fragment with documentation.

Building an index for a product

Per−product index is a one aggregate index of all documentation in the product. It should be used in scenarios in which the set of documentation plug−ins is not changing. For example an info−center installation will benefit from per−product index.

To build an index follow the steps:

- build a product, including all documentation plug−ins,
- create an index for a desired locale by running this command:

  ```
  eclipse −nosplash −application org.eclipse.help.base.indexTool −vmargs −DindexOutput=outpu
  ```

  from the directory containing the product. The following arguments need to be set :
  outputDirectory − specifies path of the directory where the index is to be saved
  locale − specifies locale for which the index will be built

For example, running

```
eclipse −nosplash −application org.eclipse.help.base.indexTool −vmargs −DindexOutput=d:/build/com
```

will result in file **doc_index.zip** being saved in the **nl/en** directory that will be created under **d:/build/com.my.plugin**. The zip will contain index of contents of documents that are available to users when they run the product in the **en** locale.

Packaging and Installation of the product's pre−built index

Pre−built indices, the **doc_index.zip** files, need to be packaged as a plug−in. You can choose to use a plug−in associated with the primary feature, or choose to package the index for each language into separate fragments.

For example, if product's documentation is available in three languages, say English, German and Simplified Chinese, a plug−in com.my.plugin can have the following structure:

```
com.my.plugin/
              plugin.xml
              nl/
                 de/
                    doc_index.zip
                 en/
                    doc_index.zip
                 zh/
                    CN/
                       doc_index.zip
              other files of this plugin
```

The ID of the plug−in needs to be specified as a **productIndex** preference for org.eclipse.help.base plug−in. For plug−in in the above example, the plugin_customization.ini file needs to contain the entry

```
org.eclipse.help.base/productIndex=com.my.plugin
```

Installing the stand–alone help system

If you are creating an application that is not based on the Eclipse framework, you can still use the Eclipse help system. Your application can package and install the stand–alone help system, a very small version of Eclipse that has everything except the help system stripped out of it. Then, your application can make API calls from its Help menu, or from UI objects, to launch the help browser. The stand–alone help system has all the features of the integrated help system, except infopops and active help. When an application is not Java based, or help is required when the application is not running, it is possible to use stand–alone help from a system shell, a shell script or a desktop shortcut and provide command line options instead of calling Java APIs.

The stand–alone help system allows passing number of options that can be used to customize various aspects of the help system. The following options are supported:

- **–eclipsehome** *eclipseInstallPath* – specifies Eclipse installation directory. This directory is a parent to "plugins" directory and eclipse executable. The option must be provided, when current directory from which infocenter is launched, is not the same as Eclipse installation directory.
- **–host** *helpServerHost* – specifies host name of the interface that help server will use. It overrides host name specified the application server plugin preferences.
- **–data** *instanceArea* – specifies a path that Eclipse can use to write instance data. The value can be an absolute path of a directory, or a path relative to Eclipse installation directory. The option must be provided when Eclipse is installed in the read only location, or has been customized to override osgi.instance.area or osgi.instance.area.default properties.
- **–port** *helpServerPort* – specifies port number that help server will use. It overrides port number specified the application server plugin preferences.
- **–dir ltr** or **–dir rtl** – sets left–to right or right–to–left rendering direction of help UI in the browser.
- Additionally, most options accepted by Eclipse executable can be passed. They are especially useful during debugging and for applying customization to Eclipse. For example, passing an option

  ```
  -nl fr_FR
  ```

  will start help system in French language instead of a language specified by the machine's locale.

Installation/packaging

These steps are for the help system integrator and are not meant to address all the possible scenarios. It is assumed that all your documentation is delivered as Eclipse plug–ins and, in general, you are familiar with the eclipse help system.

1. Download the Eclipse Platform Runtime Binary driver from www.eclipse.org.
2. Install (unzip) the driver under your application directory, for example, *d:\myApp*. This will create an eclipse sub–directory, d:\myApp\eclipse that contains the code required for the Eclipse platform (which includes the help system).

How to call the help classes from Java

1. Make sure *d:\myApp\eclipse\plugins\org.eclipse.help.base_3.1.0.jar* is on your app classpath. The class you use to start, launch, and shut down the help system is org.eclipse.help.standalone.Help.
2. Create an array of String containing options that you want to pass to help system support. Typically, the eclipsehome option is needed.

```
String[] options = new String[] { "-eclipsehome", "d:\\myApp\\eclipse" };
```

3. In your application, create an instance of the Help class by passing the options. This object should be held onto until the end of your application.

```
Help helpSystem = new Help(options);
```

4. To start the help system:

```
helpSystem.start();
```

5. To invoke help when needed:

```
helpSystem.displayHelp();
```

   You can also call help on specific primary TOC files or topics:

```
helpSystem.displayHelp("/com.mycompany.mytool.doc/toc.xml");
helpSystem.displayHelp("/com.mycompany.mytool.doc/tasks/task1.htm");
```

6. To launch context sensitive help, call helpSystem.displayContext(contextId, x, y) where contextId is a fully qualified context id. The screen coordinates, x and y, are not currently used.
7. At the end of your application, to shutdown the help system:

```
helpSystem.shutdown();
```

How to call the help from command line

The org.eclipse.help.standalone.Help class has a main method that you can use to launch stand−alone help from a command line. The command line arguments syntax is:

```
-command start | shutdown | (displayHelp [href]) [-eclipsehome eclipseInstallPath] [-data instanc
```

A simple way to display help is to invoke

```
java -classpath d:\myApp\eclipse\plugins\org.eclipse.help.base_3.1.0.jar org.eclipse.help.standal
```

from within d:\myApp\eclipse directory. To display specific TOC file or topic use

```
java -classpath d:\myApp\eclipse\plugins\org.eclipse.help.base_3.1.0.jar org.eclipse.help.standal
```

The calls above to display help will cause help system to start, display help, and keep running to allow a user to continue browsing help after the command is executed. To control the life cycle of the help system, use start and shutdown commands, in addition to the displayHelp command. For example, you may call

```
java -classpath d:\myApp\eclipse\plugins\org.eclipse.help.base_3.1.0.jar org.eclipse.help.standal
```

[Optional] Installing a minimal set of plug−ins

The stand−alone help does not require the entire Eclipse Platform package. It is possible to run the stand−alone help with the following plug−ins (located in the eclipse\plugins directory):

```
org.apache.lucene
org.eclipse.core.runtime
org.eclipse.help
org.eclipse.help.appserver
```

```
org.eclipse.help.base
org.eclipse.help.webapp
org.eclipse.osgi
org.eclipse.tomcat
org.eclipse.update.configurator
```

On Windows and Linux, the following plugins and fragments are required on top of the minimal configuration to provide robust browser support (an SWT−embedded Internet Explorer on Windows and Mozilla on Linux or stand−alone system browser on Windows).

```
org.eclipse.core.expressions
org.eclipse.help.ui
org.eclipse.jface
org.eclipse.swt
org.eclipse.swt.win32 or org.eclipse.swt.gtk
org.eclipse.ui
org.eclipse.ui.workbench
```

Some documentation plug−ins may have dependencies on other plug−ins, usually by specifying required plug−ins in their plugin.xml. The dependent plug−ins need to be installed on the infocenter as well. Additionally, plug−ins that were designed for earlier than 3.0 version of Eclipse implicitly require an `org.eclipse.core.runtime.compatibility` being present plug−in to work.

See Help System Preferences for more information on customizing help system.

org.eclipse.help.base preferences

Help System Preferences

The Eclipse help system can be configured and branded to suit your product by specifying custom defaults for number of help preferences.

The Help system itself is divided up into a number of separate plug−ins. These tables shows available preferences, and which plug−in defines them. Consult Customizing a product on how to override some of these preferences.

org.eclipse.help plug−in:

Preference key	Usage	Default
baseTOCS	Toc ordering. Ordered list of help TOC's (books) as they would appear on the bookshelf. All the other TOCS will be follow these books. Non−present TOC's on this list will be ignored. Use the location of each TOC as /pluginId/path/to/toc.xml.	/org.eclipse.platform.doc.user/toc.xml, /org.eclipse.jdt.doc.user/toc.xml, /org.eclipse.platform.doc.isv/toc.xml, /org.eclipse.jdt.doc.isv/toc.xml, /org.eclipse.pde.doc.user/toc.xml
ignoredTOCS	Disabling TOCs. List of help TOC's (books) that will be ignored by the help system. The disabled TOCs will not appear in the list of books, cannot be linked. Topics defined by disabled TOCs will not be available from search. Non−present TOC's on this list will be ignored. Use the location of each TOC as /pluginId/path/to/toc.xml.	

org.eclipse.help.base plug−in:

Preference key	Usage	Default
banner	Location of the banner page to display in the top frame Example: banner=/org.eclipse.help.webapp/advanced/banner.html	
banner_height	Height of the banner frame Example: banner_height=60	
help_home	The page to show in the content area when opening help. Specify your html page as /pluginId/path/to/home.html.	/org.eclipse.help.base/doc/help
linksView	Set to true or false to control the visibility of the related links view. Note: this option has no effect in the infocenter.	true
bookmarksView	Set to true or false to control the visibility of the bookmarks view. Note: this option has no effect in the infocenter.	true
windowTitlePrefix	Set to true or false to control the title of the browser window. If true, the title will have a form "Help − <PRODUCT_NAME>", otherwise the title will be "<PRODUCT_NAME>", where <PRODUCT_NAME> is the name of Eclipse product set in the primary feature.	true

loadBookAtOnceLimit	The maximum number of topics a book can have, for the navigation to be loaded by the browser as one document. Navigation for larger books is loaded dynamically, few levels at a time. More topics are downloaded as necessary, when branches are expanded.	1000
dynamicLoadDepthsHint	Suggested number of levels in topic navigation downloaded to the browser for large books. The value needs to be greater than 0. The actual number of levels can differ for wide tree if suggested number of levels contains large number of topics.	3
imagesDirectory	Directory containing images used in the help view. Images must have the same name as those in the org.eclipse.help.webapp plug−in. Use the /pluginID/directory format.	images
advanced.toolbarBackground	CSS background for toolbars. Value is used in browsers that display advanced help UI.	ButtonFace
advanced.viewBackground	CSS background for navigation views. Value is used in browsers that display advanced help UI.	Window
advanced.toolbarFont	CSS font for toolbars. Value is used in browsers that display advanced help UI.	icon
advanced.viewFont	CSS font for navigation views. Value is used in browsers that display advanced help UI.	icon
basic.toolbarBackground	Background color for toolbars. Value is used in browsers displaying basic help UI.	#D4D0C8
basic.viewBackground	Background color for navigation views. Value is used in browsers displaying basic help UI.	#FFFFFF
locales	List of locales that infocenter will recognize and provide a customized content for; if locales (or languages) accepted by client browser are not matched with any locales in this list, the browser will be served content for default locale − the server locale, or locale specified by eclipse −nl command line option; if list is not specified, the browser will be served contents for its preferred locale; note: not providing this option may result in a large memory and disk space requirements as navigations and indexes will be created for each distinct preferred locale among browsers accessing the infocenter. Example: locales=en ja zh_CN zh_TW	
productIndex	If per−product, pre−built documentation index is provided with the product, the ID of the plug−in delivering the index must be specified to the help system here.	
always_external_browser	Use embedded when possible (on Windows or Linux), or always external. Setting to true will force use of external browser. Option has no effect if embedded browser is not available on a given platform.	false
default_browser		

	Default external browser. ID of one of the external web browsers contributed to org.eclipse.help.base.browser extension point that help system will use. The browser's adapter available() method must return true on the current system. This preference controls external browsers in stand−alone help mode. In the workbench mode, help uses browsers provided by workbench browser support.	default dynamically set based o browser available on a given sy
custom_browser_path	Executable path for custom browser This preference controls external browsers in stand−alone help mode. In the workbench mode, help uses browsers provided by workbench browser support.	C:\Program Files\Internet Explorer\IEXPLORE.EXE" % Windows, "konqueror %1" − on Linux "mozilla %1" − on other platfo
showDisabledActivityTopics	Help system filters topics from disabled capabilities. This option controls this behavior and existence of Show All Topics button. Accepted values: never, off, on, always never − topic from disabled capabilities are not shown off − user can choose to show all topics, disabled topics initially hidden on − user can choose to show all topics, all topics initially shown always − topic from disabled capabilities are shown (filtering disabled)	off
activeHelp	Allows enabling and disabling execution of active help. The option has no effect in the infocenter setup, where active help is disabled. Accepted values: true − default active help actions enabled false − active help framework disabled	true
window_infopop	Allows enabling the old−style infopops when help key is pressed in workbench windows. If false, the new dynamic help view will open instead.	false
dialog_infopop	Allows enabling the old−style infopops when help key is pressed in dialogs. If false, the new dynamic help window will open instead.	`false`

org.eclipse.help.appserver plug−in:

Preference key	Usage	Default
port	The port number on which the sever listens for http requests. If port is not given, an arbitrary port is picked by the system.	
host	The host address or name to use for connecting to the server. The default is nothing, and eclipse will pick up an available local address. Products using help in local mode (workbench or stand−alone, not infocenter), can set this preference to "127.0.0.1" to ensure help server is not exposed to other users on the	

	network.	

org.eclipse.tomcat plug−in:

Preference key	Usage	Default
acceptCount	The maximum queue length for incoming connection requests when all possible request processing threads are in use. Any requests received when the queue is full will be refused.	100
maxProcessors	The maximum number of request processing threads to be created by this Connector, which therefore determines the maximum number of simultaneous requests that can be handled.	75
minProcessors	The number of request processing threads that will be created when this Connector is first started. This attribute should be set to a value smaller than that set for maxProcessors.	5

Installing the help system as an infocenter

You can allow your users to access the help system over the Internet or an intranet, by installing the infocenter and the documentation plug−ins on a server. Clients view help by navigating to a URL, and the help system is shown in their web browser. The infocenter help system can be used both for client applications and for web applications, either of which can have their help accessed remotely. All features of help system except infopops and active help are supported.

The infocenter help system allows passing number of options that can be used to customize various aspects of the infocenter. The following options are supported:

- **−eclipsehome** *eclipseInstallPath* − specifies Eclipse installation directory. This directory is a parent to "plugins" directory and eclipse executable. The option must be provided, when current directory from which infocenter is launched, is not the same as Eclipse installation directory.
- **−data** *instanceArea* − specifies a path that Eclipse can use to write instance data. The value can be an absolute path of a directory, or a path relative to Eclipse installation directory. The option must be provided when Eclipse is installed in the read only location, or has been customized to override osgi.instance.area or osgi.instance.area.default properties.
- **−host** *helpServerHost* − specifies host name of the interface that help server will use. It overrides host name specified in the application server plugin preferences.
- **−port** *helpServerPort* − specifies port number that help server will use. It overrides port number specified in the application server plugin preferences.
- **−locales** *localeList* − specifies a list of locales that infocenter will recognize and provide a customized content for. If the option is not specified, infocenter will build navigation, and index documents for each preferred locale of the browsers accessing the infocenter. When the option is present, locales from browser requests will be matched with locales in the list. If browser preferred locale does not exist in the list, but its language part does, it will be used. Subsequently, additional browser locales in decreased order of preference will be matched against the list. If none of the browser locales (or its language part) matches any locale on the list, the client will be served content in the default locale − server locale or locale passed with −nl option. For example using options

  ```
  −nl en −locales de en es fr it ja ko pt_BR zh_CN zh_TW
  ```

  will cause infocenter operating in 10 locales. All other locales will receive content for en locale.
- **−dir ltr** or **−dir rtl** − forces left−to−right or right−to−left rendering direction of help UI in the browser for all languages. By default direction is determined based on the browser locale.
- **−noexec** − indicates that Eclipse executable should not be used. You may need to use this option when running on a platform for which Eclipse executable is not available.
- Additionally, most options accepted by Eclipse executable can be passed. They are especially useful during debugging and for applying customization to Eclipse. For example, passing options

  ```
  −vmargs −Xmx256M
  ```

  increases memory available to the infocenter and will allow serving a larger book collection.

Installation/packaging

These steps are for the help system integrator and are not meant to address all the possible scenarios. It is assumed that all your documentation is delivered as Eclipse plug−ins and, in general, you are familiar with the eclipse help system.

1. Download the Eclipse Platform Runtime Binary driver from www.eclipse.org.
2. Install (unzip) the driver in a directory, *d:\myApp*. This will create an eclipse sub–directory, d:\myApp\eclipse that contains the code required for the Eclipse platform (which includes the help system).

How to start or stop infocenter from command line

The org.eclipse.help.standalone.Infocenter class has a main method that you can use to launch infocenter from a command line. The command line arguments syntax is:

```
-command start | shutdown | [-eclipsehome eclipseInstallPath] [-data instanceArea] [-host helpSer
```

To start an infocenter on port 8081 issue a start command by running

```
java -classpath d:\myApp\eclipse\plugins\org.eclipse.help.base_3.1.0.jar org.eclipse.help.standal
```

To shut the infocenter down issue a shutdown command by running

```
java -classpath d:\myApp\eclipse\plugins\org.eclipse.help.base_3.1.0.jar org.eclipse.help.standal
```

Using the infocenter

Start the web server. Point a web browser to the path "help" web application running on a port specified when starting the infocenter. On the machine the infocenter is installed, this would be http://localhost:8081/help/.

How to start or stop infocenter from Java

When including infocenter as part of another application, it may be more convenient to start it and stop using Java APIs instead of using system commands. Follow the steps if it is the case:

1. Make sure *d:\myApp\eclipse\plugins\org.eclipse.help.base_3.1.0.jar* is on your app classpath. The class you use to start, and shut down the infocenter isorg.eclipse.help.standalone.Infocenter.
2. Create an array of String containing options that you want to pass to the infocenter. Typically, the eclipsehome and port options are needed.

   ```
   String[] options = new String[] { "-eclipsehome", "d:\\myApp\\eclipse" , "-port", "8081" }
   ```
3. In your application, create an instance of the Help class by passing the options.

   ```
   Infocenter infocenter = new Help(options);
   ```
4. To start the help system:

   ```
   helpSystem.start();
   ```
5. To shut the infocenter down:

   ```
   helpSystem.shutdown();
   ```

Making infocenter available on the web

Eclipse contains a complete infocenter and does not require other server software to run. However, in unsecure environment like Internet, it is recommended infocenter is not accessed directly by clients, but is

made available through an HTTP server or an application server. Most servers come with modules or servlets for delegating certain request to other web resources. For example, one may configure a proxy module of Apache HTTP Server to redirect requests made to *http://mycompany.com/myproduct/infocenter* to *http://internalserver:8081/help* that runs an infocenter. Adding the lines

```
LoadModule proxy_module modules/ApacheModuleProxy.dll
ProxyPass /myproduct/infocenter http://internalserver:8081/help
ProxyPassReverse /myproduct/infocenter http://internalserver:8081/help
```

to conf/httpd.conf file of Apache server running mycompany web site accomplishes this.

Some versions of Apache HTTP server, may contain AddDefaultCharset directive enabled in configuration file. Remove the directive or replace with

```
AddDefaultCharset Off
```

to have browsers display documents using correct character set.

Running multiple instance of infocenter

Multiple instances of infocenter can be run on a machine from one installation. Each started instance must use its own port and be provided with a workspace, hence `-port` and `-data` options must be specified. The instances can serve documentation from different set of plug−ins, by providing a valid platform configuration with `-configuration` option.

If `-configuration` is not used and configuration directory is shared among multiple infocenter instances, with overlapping set of locales, it must be ensured that all search indexes are created by one infocenter instance before another instance is started. Indexes are saved in the configuration directory, and write access is not synchronized across infocenter processes.

[Optional] Installing a minimal set of plug−ins

The infocenter does not require the entire Eclipse Platform package. It is possible to run the infocenter with the following plug−ins (located in the eclipse\plugins directory):

```
org.apache.lucene
org.eclipse.core.runtime
org.eclipse.help
org.eclipse.help.appserver
org.eclipse.help.base
org.eclipse.help.webapp
org.eclipse.osgi
org.eclipse.tomcat
org.eclipse.update.configurator
```

Some documentation plug−ins may have dependencies on other plug−ins, usually by specifying required plug−ins in their plugin.xml. The dependent plug−ins need to be installed on the infocenter as well. Additionally, plug−ins that were designed for earlier than 3.0 version of Eclipse implicitly require an `org.eclipse.core.runtime.compatibility` being present plug−in to work.

Infocenter plug−ins can be updated without restarting the infocenter, using commands explained in Updating a running infocenter from command line topic. To use this functionality, the minimal set of plug−ins must include `org.eclipse.update.core` plug−in.

See Help System Preferences for more information on customizing help system.

Updating a running infocenter from command line

In addition to the start and shutdown commands supported by the infocenter and documented in Installing the help system as an infocenter , infocenter supports set of command for invoking update manager operations in running eclipse. You can install, update, enable, disable features, or list installed features, or features available on an update site, or adding an extension site to the running infocenter. In effect, you can change set of running documentation plug−ins without a need for shutting the infocenter down or restarting it.

Start infocenter as explained in Installing the help system as an infocenter. If you are running minimal set of plug−ins, add org.eclipse.update.core plug−in to the list of plug−ins before launching infocenter using the start command.
Launch infocenter update commands as follows, where [ ] means optional argument and arguments in italics must be provided by the user.
After performing updates as needed, apply the changes by issuing the apply command as the last step. It reflects the changes in the current session. If you do not call apply command, the changes will take effect the next time infocenter is started.

Installing a feature from a remote site:
java −cp plugins/org.eclipse.help.base_3.1.0.jar org.eclipse.help.standalone.Infocenter
−command install
−featureId *feature_id*
−version *version*
−from *remote_site_url*
[−to *target_site_dir*]

Example: `java -cp plugins/org.eclipse.help.base_3.1.0.jar org.eclipse.help.standalone.Infocenter -command install -from http://dev.eclipse.org/viewcvs/index.cgi/%7Echeckout%7E/platform-update-home/op -featureId com.example.root -version 1.0.0`

Updating an existing feature or all features:
java −cp plugins/org.eclipse.help.base_3.1.0.jar org.eclipse.help.standalone.Infocenter
−command update
[−featureId *feature_id*]
[−version *version*]

Enabling (configuring) a specified feature:
java −cp plugins/org.eclipse.help.base_3.1.0.jar org.eclipse.help.standalone.Infocenter
−command enable
−featureId *feature_id*
−version *version*
[−to *target_site_dir*]

Disabling (unconfiguring) a specified feature:
java −cp plugins/org.eclipse.help.base_3.1.0.jar org.eclipse.help.standalone.Infocenter
−command disable
−featureId *feature_id*
−version *version*
[−to *target_site_dir*]

Uninstalling a specified feature:
java −cp plugins/org.eclipse.help.base_3.1.0.jar org.eclipse.help.standalone.Infocenter
−command uninstall
−featureId *feature_id*
−version *version*
[−to *target_site_dir*]

In all the above commands where the −to target_site_dir is specified, corresponding configured target site at given directory will be used. If it is not specified, then the default local product site is used.

If you only need to verify if the operation would succeed, in the above commands, (i.e. it satisfies the constraints), without actually performing it, then add ***−verifyOnly=true*** to the list of arguments.

Searching a remote site, listing all available features for install:
java −cp plugins/org.eclipse.help.base_3.1.0.jar org.eclipse.help.standalone.Infocenter
−command search
−from *remote_site_url*

Listing installed features:
java −cp plugins/org.eclipse.help.base_3.1.0.jar org.eclipse.help.standalone.Infocenter
−command listFeatures
−from *local_site_dir*

The features are listed as:
```
Site: site url
  Feature: id version enabled (or disabled)
```

Adding a local site with more features:
java −cp plugins/org.eclipse.help.base_3.1.0.jar org.eclipse.help.standalone.Infocenter
−command addSite
−from *local_site_dir*

Removing a local site:
java −cp plugins/org.eclipse.help.base_3.1.0.jar org.eclipse.help.standalone.Infocenter
−command removeSite
−to *local_site_dir*

Applying the changes:
java −cp plugins/org.eclipse.help.base_3.1.0.jar org.eclipse.help.standalone.Infocenter
−command apply

The platform.xml file

Since:

3.0

Description:

The primary update manager configuration information is maintained by the platform.xml file located in the configuration directory (usually, eclipse/configuration/org.eclipse.update/platform.xml). This file format is internal and can change any time, but, in practice, it should not change much, if at all.

As some people would like to take advantage of shipping an eclipse based product with a pre−defined installation configuration, this document provides some syntax and semantics info for this configuration file.

Given that the platform.xml originated from the old platform.cfg and installConfig*.xml file, there are still some deprecated elements/attributes that will not be mentioned in this doc, as well as some relatively convoluted way of specifying the configuration info. If platform.xml is to become API, then some re−work is needed (both syntax and semantics).

Configuration Markup:

<!ELEMENT extension EMPTY>

<!ATTLIST extension

point CDATA #REQUIRED

id CDATA #IMPLIED

name CDATA #IMPLIED>

<!ELEMENT config EMPTY>

<!ATTLIST config

version CDATA #IMPLIED

date CDATA #IMPLIED

transient (true | false) "false"

shared_ur CDATA #IMPLIED>

describes the current install configuration.

- **version –** this attribute may be used to indentify the version of the configuration specification
- **date –** this is an internal attribute that is generated by the update configurator code. If you manually create or modify the platform.xml file then remove this attribute, and let the update code add it when it first runs. In this scenario, update will not try to reconcile the platform.xml with what is on the disk, it takes the platform.xml as is, and slaps the date attribute.
- **transient –** a transient configuration is not modified by the update code, it is just read and used to configure the plugins to run. Its main point is to avoid doing any reconciliation with what's on the disk. This flag is a bit of a hack for PDE to run self hosting without having features in the workspace, and it is better if it is left set to false.
- **shared_ur –** points to the shared configuration location. This is particularly useful on unix/linux, when the admin sets up a shared installation and each user's configuration consists of the shared installation data (as pointed at by this attribute) and any local changes.

<!ELEMENT site EMPTY>

<!ATTLIST site

url CDATA #REQUIRED

enabled (true | false)

updateable (true | false)

policy (USER−EXCLUDE|USER−INCLUDE|MANAGED−ONLY)

linkfile CDATA #IMPLIED

list CDATA #IMPLIED>

defines an installation location, which is a location that contains features and plugins.

- **url –** location of this installation site. The url is absolute, but there is a special url for the base location (where eclipse.exe is located), of the form platform:/base/
- **enabled –** a boolean value that defines whether the plugins and features from this location will be considered at runtime.
- **updateable –** defines whether any update/install/enable/uninstall/etc. operations can be performed on features on this site.
- **policy –** defines the way the "list" attribute is interpreted:
  USER−EXCLUDE: all the plugins from the site are run, except those listed in the "list" attribute. This is what's currently used by the update code.
  USER−INCLUDE: only the plugins listed by the "list" attribute are run. This is what PDE generated configurations use.
  MANAGED−ONLY: only plugins from the features define by this site (see the <feature> element) are run (new in 3.1)
- **linkfile –** if this site has been contributed from a .link file, then this attribute points to that file.
- **list –** lists the plugins to be included or excluded (see the "policy" attribute) at runtime.

<!ELEMENT feature EMPTY>

<!ATTLIST feature

id CDATA #REQUIRED

version CDATA #REQUIRED

url CDATA #REQUIRED>

defines a feature installed on this site.

- **id –** the feature id
- **version –** the feature version
- **url –** the feature location, relative to the site. By default, the url is of the form features/some_feature_id_version.

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<config date="1119300698390" transient="false" version="3.0">
  <site enabled="true" policy="USER-EXCLUDE" updateable="true" url="platform:/base/">
    <feature id="org.eclipse.platform" url="features/org.eclipse.platform_3.1.0/" version="3.1.0"
    </feature>
    <feature id="org.eclipse.platform.source" url="features/org.eclipse.platform.source_3.1.0/" v
    </feature>
    <feature id="org.eclipse.rcp" url="features/org.eclipse.rcp_3.1.0/" version="3.1.0">
    </feature>
    <feature id="org.eclipse.jdt" url="features/org.eclipse.jdt_3.1.0/" version="3.1.0">
    </feature>
    <feature id="org.eclipse.jdt.source" url="features/org.eclipse.jdt.source_3.1.0/" version="3.
    </feature>
    <feature id="org.eclipse.pde" url="features/org.eclipse.pde_3.1.0/" version="3.1.0">
    </feature>
    <feature id="org.eclipse.sdk" url="features/org.eclipse.sdk_3.1.0/" version="3.1.0">
    </feature>
    <feature id="org.eclipse.pde.source" url="features/org.eclipse.pde.source_3.1.0/" version="3.
    </feature>
    <feature id="org.eclipse.rcp.source" url="features/org.eclipse.rcp.source_3.1.0/" version="3.
    </feature>
  </site>
  <site enabled="true" policy="USER-EXCLUDE" updateable="true" url="file:/d:/extensions/org.eclip
  </site>
</config>
```

Running update manager from command line

In addition to the install wizard and configuration dialog, it is possible to perform update manager operations by running eclipse in a command line mode. You can install, update, enable, disable features, or list installed features, or features available on an update site, or adding an extension product site to the local install, etc. You can also, mirror chosen features from an update site to a local update site location.

Launch eclipse as follows, where [ ] means optional argument and arguments in italics must be provided by the user.

Installing a feature from a remote site:
java –cp startup.jar org.eclipse.core.launcher.Main
  –application org.eclipse.update.core.standaloneUpdate
  –command install
  –featureId *feature_id*
  –version *version*
  –from *remote_site_url*
  [–to *target_site_dir*]

Example: `java –cp startup.jar org.eclipse.core.launcher.Main –application org.eclipse.update.core.standaloneUpdate –command install –from http://dev.eclipse.org/viewcvs/index.cgi/%7Echeckout%7E/platform-update-home/op –featureId com.example.root –version 1.0.0`

Updating an existing feature or all features:
java –cp startup.jar org.eclipse.core.launcher.Main
  –application org.eclipse.update.core.standaloneUpdate
  –command update
  [–featureId *feature_id*]
  [–version *version*]

Enabling (configuring) a specified feature:
java –cp startup.jar org.eclipse.core.launcher.Main
  –application org.eclipse.update.core.standaloneUpdate
  –command enable
  –featureId *feature_id*
  –version *version*
  [–to *target_site_dir*]

Disabling (unconfiguring) a specified feature:
java –cp startup.jar org.eclipse.core.launcher.Main
  –application org.eclipse.update.core.standaloneUpdate
  –command disable
  –featureId *feature_id*
  –version *version*
  [–to *target_site_dir*]

Uninstalling a specified feature:
java –cp startup.jar org.eclipse.core.launcher.Main

−application org.eclipse.update.core.standaloneUpdate
−command uninstall
−featureId *feature_id*
−version *version*
[−to *target_site_dir*]

In all the above commands where the −to target_site_dir is specified, corresponding configured target site at given directory will be used. If it is not specified, then the default local product site is used.

If you only need to verify if the operation would succeed, in the above commands, (i.e. it satisfies the constraints), without actually performing it, then add ***−verifyOnly=true*** to the list of arguments.

Searching a remote site, listing all available features for install:
java −cp startup.jar org.eclipse.core.launcher.Main
−application org.eclipse.update.core.standaloneUpdate
−command search
−from *remote_site_url*

Listing installed features:
java −cp startup.jar org.eclipse.core.launcher.Main
−application org.eclipse.update.core.standaloneUpdate
−command listFeatures
−from *local_site_dir*

The features are listed as:

```
Site: site url
  Feature: id version enabled (or disabled)
```

Adding a local site with more features:
java −cp startup.jar org.eclipse.core.launcher.Main
−application org.eclipse.update.core.standaloneUpdate
−command addSite
−from *local_site_dir*

Removing a local site:
java −cp startup.jar org.eclipse.core.launcher.Main
−application org.eclipse.update.core.standaloneUpdate
−command removeSite
−to *local_site_dir*

Mirroring feature(s) from a remote site:
java −cp startup.jar org.eclipse.core.launcher.Main
−application org.eclipse.update.core.standaloneUpdate
−command mirror
−from *remote_site_url*
−to *target_site_dir*
[−featureId *feature_id*]
[−version *version*]
[−mirrorURL *mirror_site_url*]

If −mirrorURL is specified, an update policy will be generated in <target_site_dir>/policy.xml file. The

resulting policy.xml maps all features from the mirror site to the specified URL. The policy.xml can be used as is, or its fragments can be included into custom designed policy file.

The return code for each command is either 0 (success) or 1 (failure).

Additionally, most options accepted by Eclipse executable can be passed. They are especially useful during debugging and for specifying target environment for installed features. For example, passing option `-data some_path` will set the workspace to some_path.

Bidirectional Support in the Eclipse Workbench

As of Eclipse 3.1 RC2 bidirectional support will be complete are supported in JFace and the Workbench. A bidirectional language is one that can write both right to left and left to right based on context. Eclipse will recognize Hebrew, Arabic, Farsi and Urdu as bidirectional by default.

Enabling Bidirectional Support in the SDK

The orientation of the workbench will be flipped if one of the following ways (in order of priority)

- ***–dir command line parameter***. If the –dir command line option is used this will be the default orientation. Valid values are –dir rtl or –dir ltr.
- ***system properties***. If the system property eclipse.orientation is set this will be used. The value is of the same format as –dir.
- ***–nl command line parameter***. If the –nl option is used and the language is Hebrew, Arabic, Farsi or Urdu the orientation will be right to left, otherwise it will be left to right. Note this is s

These values will be used to call org.eclipse.jface.Window#setDefaultOrientation(). All subclasses of Window and the children of those windows will get the orientation specified. The default orientation is SWT#NONE.

Views and editors will by default inherit the window orientation from their parent. Dialogs should inherit orientation by using the shellStyle of their parent when setting their own by calling super#getShellStyle.

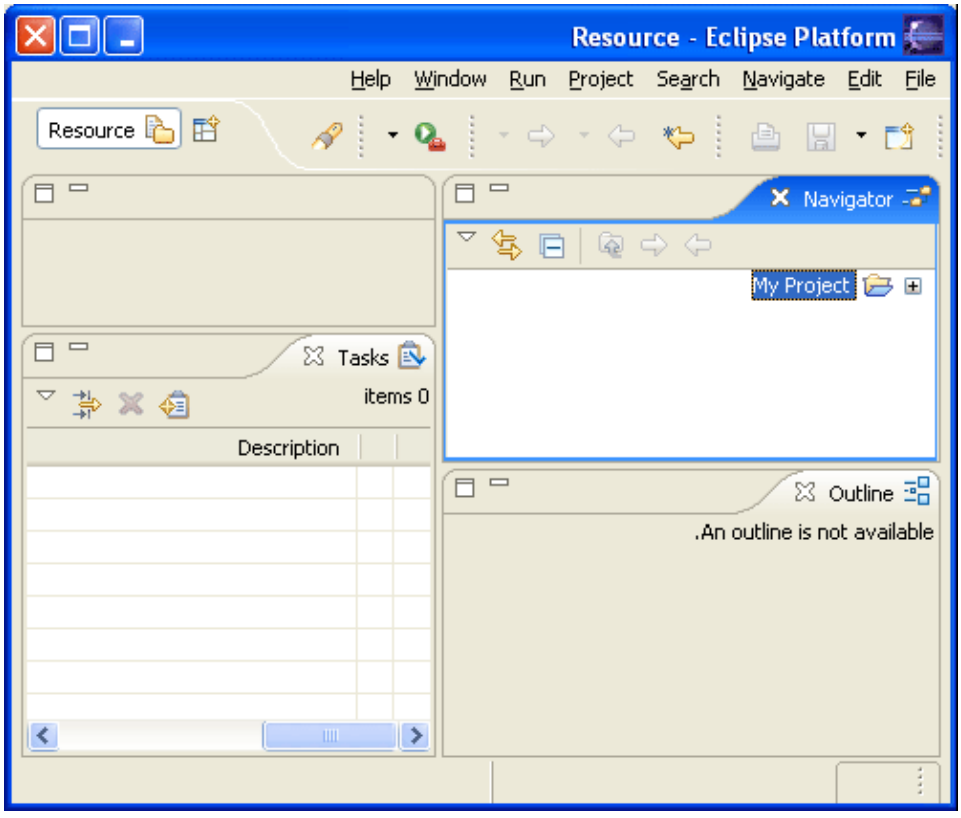


Figure 1 – Screen shot of right to left orientation of the resource perspective

Enabling your plug–in for looking up alternate icons

In many cases your icons will not make any sense in right to left mode. In particular any icon to do with editing will have this issue.To enable lookup of images in a fragment all you need to do to nl enable nl in your path and use the Platform bundle support to lookup the icons file.

For example

String iconPath = "nl/icons/myicon.gif";

URL url = Platform.find(Platform.getBundle(MyPluginId), new Path(iconPath));

Image Descriptor descriptor = ImageDescriptor.createFromURL(url);

If the icon reference is in your plugin.xml just make sure you have the nl prefix on your path. If it is an extension point defined by the workbench the lookup will be handled for you. Just make sure if you create your own extension point that it loads ImageDescriptors this way.

How to choose icons to override

There is no hard and fast rules for choosing which icons to override but in general you should focus on the icons that imply a textual direction with a horizontal arrow.

JFace Preference Stores

As of release 3.1 the ***org.eclipse.jface.preference.IPreferenceStore*** that is returned from AbstractUIPlugin#getPreferenceStore will be an instance of ***org.eclipse.ui.preferences.ScopedPreferenceStore***. The ScopedPreferenceStore uses the new core runtime API to manage preferences. In 3.0 it used the compatibility layer to interface with an instance of ***org.eclipse.core.runtime.Preferences***.

In 3.1 we have disambiguated IPreferenceStore to be more specific about the types of the values sent in preference changed events. The IPreferenceStore from AbstractUIPlugin#getPreferenceStore has the same behavior as before – all that has changed is that it has been specified more clearly.

Typing: ***org.eclipse.jface.util.IPreferenceChangeListeners*** added to an IPreferenceStore can potentially get two types of old and new values – typed or String representations. Any event generated by a call to a typed IPreferenceStore API (such as `setValue(String key, boolean value)` will generate a typed event. However it is also possible that events will be propagated from the core runtime preferences which generate an un typed event (for instance on a preference import). Preference listeners need to be prepared for both. Note also that typed events will not be propagating primitive types so a call to `setValue(String key, boolean value)` will result in an event where the oldValue and newValue are Booleans.

putValue: IPreferenceStore.putValue(String key, String value) will not generate a change event. This API is meant to be used for private preferences that no listener would want to react to.

initializeDefaultPreferences. This API was deprecated in Eclipse 3.0 as it is only fired if the compatibility layer is used. As most plug–ins rely on AbstractUIPlugin#getPreferenceStore to get their preference store this was being fired on plug–in start–up previously. If your plug–in does not access the compatibility layer itself then this method may not be fired. It is recommended that you create a ***org.eclipse.core.runtime.preferences.AbstractPreferenceInitializer*** to handle your preference initialization.

Third party libraries and classloading

Because OSGi makes use of multiple classloaders, the transparent usage of extensible / configurable third party libraries in eclipse requires the usage of an eclipse specific mechanism called "buddy loading". This mechanism allows a bundle to indicate that it needs assistance to load classes or resources when it can not find them among its prerequisites. Note that we call "extensible libraries", libraries that needs to see classes or resources provided by user code (for example log4j logger mechanism, hibernate,...).

To indicate its need for buddy loading, a bundle must modify its manifest and add the following header:

```
Eclipse-BuddyPolicy: <value>
```

<value> refers to the policy used to look for the classes. Here are the supported policies:

- registered – indicates that the buddy mechanism will consult bundles that have been registered to it. Bundle willing to be registered to a particular bundle add in their manifest: "Eclipse–RegisterBuddy: <bundleSymbolicName>";
- dependent – indicates that the classes/resources will be looked up transitively in all the dependent of the bundle;
- global – indicates that the classes/resources will be looked up in the global pool of exported package;
- app – indicates that the application classloader will be consulted;
- ext – indicates that the extensiont classloader will be consulted;
- boot – indicates that the boot classloader will be consulted.

Eclipse 3.1 Plug−in Migration Guide

This guide covers migrating Eclipse 3.0 (or earlier) plug−ins to Eclipse 3.1.

One of the goals of Eclipse 3.1 was move Eclipse forward while remaining compatible with previous versions to the greatest extent possible. That is, plug−ins written against the Eclipse 3.0 APIs should continue to work in 3.1 in spite of the API changes.

The key kinds of compatibility are API contract compatibility and binary compatibility. API contract compatibility means that valid use of 3.0 APIs remains valid for 3.1, so there is no need to revisit working code. Binary compatibility means that the API method signatures, etc. did not change in ways that would cause existing compiled ("binary") code to no longer link and run with the new 3.1 libraries.

While every effort way made to avoid breakage, there are a few areas of incompatibility. This document describes the areas where Eclipse changed in incompatible ways between 3.0 and 3.1, and provides instructions for migrating 3.0 plug−ins to 3.1.

- Eclipse 3.1 Plug−in Migration FAQ
- Incompatibilities between Eclipse 3.0 and 3.1
- Adopting 3.1 mechanisms and API

Eclipse 3.1 Plug–in Migration FAQ

1. IPreferenceStore has more explicit API
2. IWorkbenchWindow#getShell() has more explicit API

IPreferenceStore has more explicit API

Although the behavior of the IPreferenceStore provided by `AbstractUIPlugin#getPreferenceStore()` hasn't changed we have updated the specification of IPreferenceStore to explicitly define the behavior that we have provided.

Typing of PropertyChangeEvents

Any property change event from an IPreferenceStore must have an old and new value of the same type that is consistent with the setValue call that generated it.

For instance if you call `IPreferenceStore#setValue(String name, long value)` the values in the PropertyChangeEvent generated from this method will both be of type `java.lang.Long`.

putValue

Calls to `#putValue` will not generate a `PropertyChangedEvent`. Calls to the various `#setValue` methods will.

Relationship between the OSGI Preference and an IPreferenceStore

The IPreferenceStore provided by `AbstractUIPlugin#getPreferenceStore()` is an instance of `ScopedPreferenceStore` which uses `org.osgi.service.prefs.Preferences` as a back end. `org.osgi.service.prefs.Preferences` propagates change events as Strings only.

The `ScopedPreferenceStore` wrappers those OSGI events generated by `IPreferenceStore#setValue(String name, String value)` and one of it's own `PropertyChangeEvents` and forwards that event to it's listeners. For the other implementations of `IPreferenceStore#setValue` the `ScopedPreferenceStore` will create it's own events of the correct type and not propagate the events from the OSGI preferences.

Listeners to a `ScopedPreferenceStore` should be prepared for both typed and String values in thier change events as it is still possible to get an event via the OSGI preferences (during a preference import for instance). OSGI events are always of type `java.lang.String`.

IWorkbenchWindow#getShell() has more explict API

It has always been possible to get a null org.eclipse.swt.widgets.Shell from the existing IWorkbenchWindows in the Eclipse SDK. We now explictly define the conditions where this occurs, namely when the shell has not been created or when the IWorkbenchWindow has been closed.

Incompatibilities between Eclipse 3.0 and 3.1

Eclipse changed in incompatible ways between 3.0 and 3.1 in ways that affect plug−ins. The following entries describe the areas that changed and provide instructions for migrating 3.0 plug−ins to 3.1. Note that you only need to look here if you are experiencing problems running your 3.0 plug−in on 3.1.

1. Plug−in Preferences
2. Changes to IPath constraints
3. Extension registry
4. Code formatter options
5. API contract changes to AntCorePreferences
6. API contract changes to Policy class in JFace
7. API contract changes to allow a null default perspective
8. API contract changes to IViewLayout
9. API contract changes to IVMInstall
10. SelectionEnabler.SelectionClass made package−visible
11. ContributionItem.getParent() can return null
12. Changes to isPropertySet(boolean) in IPropertySource and IPropertySource2
13. handlerSubmission element deleted from the org.eclipse.ui.commands extension point
14. Static non−final API field GLOBAL_IGNORES_CHANGED in TeamUI made final
15. ClassCastException using FillLayout
16. Creating a widget with a disposed parent

---

1. Plug−in Preferences

What is affected: Plug−ins who initialize their default plug−in preference values by over−riding `Plugin#initializeDefaultPreferences` or use preference change listeners.

Description: In Eclipse 3.1 the `org.eclipse.jface.preference.IPreferenceStore` object obtained from `org.eclipse.ui.plugin.AbstractUIPlugin#getPreferenceStore` was migrated to live on top of the new 3.0 Eclipse preference framework supplied by the `org.eclipse.core.runtime` plug−in.

Action required: As a result, clients using the preference APIs should check for two possible issues:

1. The type of the objects contained in preference change events is not guarenteed; both the old value and new value in events can be null, a `String`, or a typed object. Therefore to be a good client, preference change listeners should be able to handle all three of these possible situations.
2. If your plug−in uses `org.eclipse.ui.plugin.AbstractUIPlugin#initializeDefaultPreferences` then you must be sure to include the `org.eclipse.core.runtime.compatibility` plug−in in your plug−in's list of required plug−ins as this dependancy has been removed from the `org.eclipse.ui.workbench` plug−in.

See the preference store documentation for more details.

2. Changes to IPath constraints

What is affected: Plug−ins that create, manipulate, or store `IPath` objects.

Description: In Eclipse 3.0, `IPath` had a number of restrictions on the segments of paths that were more restrictive than the restrictions of the underlying operating system. These included:

- Leading and trailing whitespace in path segments were not allowed
- The colon character (':') was reserved as a device separator character
- The backslash character ('\') was reserved as a segment separator character

These restrictions have been removed in Eclipse 3.1 when the platform's data location (workspace) is located on a file system that does not have these restrictions.

Action required: In order to enable the proper treatment of the expanded range of paths, all usage of `Path` and `IPath` within plug−ins should be reviewed and updated as described below. Most unmodified plug−ins will continue to behave exactly as in 3.0 on all paths considered legal in 3.0. However, unless these prescribed changes are made, they are likely to fail in cases involving paths considered legal in 3.1 that were illegal in 3.0.

Plug−ins that store string representations of paths in a format that needs to be readable across different platforms should migrate to the new `Path.fromPortableString` factory method. This method produces an `IPath` instance from a platform−independent format. This string representation of paths can be created using the `IPath.toPortableString` method. Examples of metadata files that are affected include files that are stored inside Eclipse workspace projects (.project, .classpath, etc), and all paths stored in the preference store (`org.eclipse.core.runtime.preferences.IPreferencesService`).

Note: `fromPortableString` will correctly read all path strings that were created using the Eclipse 3.0 `IPath.toString` method, but not the Eclipse 3.1 `toString` method. Thus in most cases no change is required to existing metadata file formats except to begin writing paths with `toPortableString` and reading paths with `fromPortableString`.

Plug−ins that were creating paths from hard−coded string literals that assumed ':' and '\' had special meaning on all platforms will need to migrate. The easiest solution is to restrict string path literals to the subset that is supported on all platforms (avoid colon and backslash characters). Path literals can support the complete set of valid Unix paths by using the portable path string format produced by `Path.toPortableString`. This format interprets the first single colon (':') as the device separator, slash ('/') as the segment separator, and double colon ("::") as a literal colon character. For example, the code `new Path("c:/temp")` will now create a relative path with two segments on Unix platforms. Similarly, `new Path("a\\b")` will now create a path with a single segment on Unix platforms, and a path with two segments on Windows.

Plug−ins constructing paths using the `IPath.append(String)` method that assumed ':' and '\' had special meaning on all platforms may need to update their code. In Eclipse 3.1, this method uses operating−system specific device and segment delimiters to interpret the provided path string. For example, calling `append("a\\b")` on Unix platforms will now append a single segment, whereas on Windows it will continue to append two segments.

Any data files read and interpreted by the platform will no longer treat ':' and '\' as special characters on all platforms. All paths stored in data files that can be read on multiple platforms must be in portable form. For example, paths of icon files and other paths in `plugin.xml` must use only '/' as the path segment separator.

3. Extension registry

What is affected: Plug−ins that manipulate or retain `IExtensionPoint`, `IExtension`, and `IConfigurationElement` objects from the Eclipse Platform's plug−in or extension registry.

Description: Prior to 3.0, all objects obtained from the extension registry (of the former plug−in registry) were good forever. Changes were made in Eclipse 3.0 that allowed plug−ins to be dynamically added or removed without having to restart Eclipse. When a plug−in is removed without restarting, its entries in the extension registry necessarily become invalid. This means that another plug−in holding on to an object obtained previously from the deleted plug−in's extension registry entry would be left holding an invalid object. The only hint that a client could get would be from listening to `IRegistryChangeEvent`. The problem has existed since Eclipse 3.0, but is rarely encountered in practice because it is highly unusual for a plug−in to be removed without restarting Eclipse.

This problem has been addressed in 3.1 by:

- Existing methods on these `IExtensionPoint`, `IExtension`, and `IConfigurationElement` now specify that `InvalidRegistryObjectException` will be thrown when the object is invalid. The exception is unchecked so that dynamic unaware clients are not forced to check it.
- A new `isValid()` method was added to these interfaces so that a client can determine whether an object is still valid.
- Recommendations were added to the specs to make it clear that hanging on to objects obtained from the extension registry is not recommended.

Action required: If your plug−in needs to be dynamic−aware (i.e. capable of dealing with the on−the−fly addition or removal of plug−ins), the code that deals with `IExtensionPoint`, `IExtension`, and `IConfigurationElement` object sourced from some other plug−in must be changed to catch `IRegistryChangeEvent`, exactly as if it were a checked exception. Catching the exception (rather than an `isValid()` pre−check) is the only surefire way to deal with the case of a plug−in being removed by a concurrent thread (which, if it happens, it almost certainly will be).

4. Code formatter options

What is affected: Plug−ins that programmatically access the Java code formatter options.

Description: In Eclipse 3.0, the values for the code formatter option `org.eclipse.jdt.core.formatter.DefaultCodeFormatterConstants#FORMATTER_TAB_CHAR` could only be `TAB` or `SPACE`. The specification made no explicit mention that the value type was an enumeration that might grow in future releases. In Eclipse 3.1, a third possible value, `MIXED`, was added to address bug 73104. The specification has been changed to include this new value, and to allow for more values being added in the future.

Action required: Clients programmatically reading or setting this code formatter option should check their code to take account of the new third value, and to ensure that it is written in a robust way that fails gracefully if it ever encounters an option value that it did not anticipate.

5. API contract changes to AntCorePreferences

What is affected: Plug−ins that subclass or instantiate `org.eclipse.ant.core.AntCorePreferences`

Description: In Eclipse 3.0, the class `org.eclipse.ant.core.AntCorePreferences` was not marked that clients may not instantiate or subclass. This was an oversight that has been addressed in Eclipse 3.1 with the class marked as not intended to be subclassed or instantiated.

Action required: Clients programmatically creating an instance of `org.eclipse.ant.core.AntCorePreferences` should migrate their code to retrieve the preferences using: `org.eclipse.ant.core.AntCorePlugin.getPreferences()`. Any subclass will need to be reworked to no longer subclass `org.eclipse.ant.core.AntCorePreferences`.

6. API contract changes to Policy class in JFace

What is affected: RCP applications that override the JFace log set by the workbench.

Description: In Eclipse 3.0, the workbench set the workbench's log as the log to use for logging JFace errors, by passing the workbench plug−in's log directly to `org.eclipse.jface.util.Policy.setLog(ILog)`. In 3.1, the dependency on `ILog` has been removed from JFace in order to enable standalone applications using SWT and JFace outside of the eclipse runtime. A new interface, `ILogger`, has been introduced to meet JFace's needs. The workbench has been changed to provide an `ILogger` wrapping the workbench `ILog`. For further details, see bug 88608.

Action required: Most RCP applications should not need to override the log set by the workbench, but if they previously called `Policy.setLog(ILog)`, they will need to be changed to pass an `ILogger` instead.

7. API contract changes to allow a null default perspective

What is affected: RCP applications expecting a non−null default perspective.

Description: In order to allow RCP applications to start with an empty window with no perspectives open (enhancement 71150), `WorkbenchAdvisor.getInitialWindowPerspectiveId()` and `IPerspectiveRegistry.getDefaultPerspective()` have been changed to allow null to be returned. In the IDE, there is always a default perspective, so `IPerspectiveRegistry.getDefaultPerspective()` will not return null. Similarly, if an existing RCP app previously returned a non−null value from `WorkbenchAdvisor.getInitialWindowPerspectiveId()`, then `IPerspectiveRegistry.getDefaultPerspective()` will still return a non−null value.

Action required: No action should be required by clients.

8. API contract changes to IViewLayout

What is affected: Plug−ins that implement `org.eclipse.ui.IViewLayout`.

Description: In Eclipse 3.0, the class `org.eclipse.ui.IViewLayout` was not marked that clients may not implement. This was an oversight that has been addressed in Eclipse 3.1 with the class marked as not

intended to be implemented by clients.

Action required: Any implementing classes will need to be reworked to no longer implement `org.eclipse.ui.IViewLayout`.

9. API contract changes to IVMInstall

What is affected: Plug−ins that implement `org.eclipse.jdt.launching.IVMInstall`.

Description: In Eclipse 3.0, the class `org.eclipse.jdt.launching.IVMInstall` was not marked that clients may not implement directly. This was an oversight that has been addressed in Eclipse 3.1 with the class marked as not intended to be implemented directly by clients. To maintain binary compatibility, we still allow clients to implement the interface directly, but strongly recommend that clients subclass `org.eclipse.jdt.launching.AbstractVMInstall` instead. Clients that implement `IVMInstall` should also implement the new optional interface `org.eclipse.jdt.launching.IVMInstall2`, which `AbstractVMInstall` now implements. It is recommended that clients implement the new interface, `IVMInstall2`, to avoid the problem noted in bug 73493. The recommended migration is to subclass `AbstractVMInstall`.

Action required: Any implementing classes that do not already subclass `org.eclipse.jdt.launching.AbstractVMInstall` should be reworked to subclass `org.eclipse.jdt.launching.AbstractVMInstall`.

10. SelectionEnabler.SelectionClass made package−visible

What is affected: Plug−ins that use `org.eclipse.ui.SelectionEnabler.SelectionClass`.

Description: In Eclipse 3.0, the nested implementation class `org.eclipse.ui.SelectionEnabler.SelectionClass` was public, with no restrictions on its usage. This was an oversight that has been addressed in Eclipse 3.1 with the class being made package−visible.

Action required: Any classes instantiating or extending `org.eclipse.ui.SelectionEnabler.SelectionClass` will need to be reworked to no longer refer to this class.

11. ContributionItem.getParent() can return null

What is affected: Plug−ins that call `getParent()` on a subclass of `org.eclipse.jface.action.ContributionItem`.

Description: In Eclipse 3.0, method `org.eclipse.jface.action.ContributionItem.getParent()` did not specify that it could return null. This was an oversight that has been addressed in Eclipse 3.1 with Javadoc for the method clarifying when it can return null. For more details, see bug 92777.

Action required: Any code calling `ContributionItem.getParent()` must ensure that it can handle a null result.

12. Changes to isPropertySet(boolean) in IPropertySource and IPropertySource2

What is affected: Plug−ins that implement `org.eclipse.ui.views.properties.IPropertySource` or `IPropertySource2.`

Description: In Eclipse 3.0, the specification for the method `org.eclipse.ui.views.properties.IPropertySource.isPropertySet(boolean)` was incorrectly changed to specify that true should be returned if the specified property did not have a meaningful default value. In previous versions, it specified that false should be returned in this case. This was an inadvertent breaking API change, although the implementation worked the same as before if the property source implemented `IPropertySource` and not `IPropertySource2`. This has been corrected in 3.1, with `IPropertySource.isPropertySet(boolean)` being reverted back to its earlier specification (that false should be returned in this case), and IPropertySource2.isPropertySet(boolean) overriding this to specify that true should be returned in this case. For more details, see bug 21756.

Action required: Any classes implementing IPropertySource or IPropertySource2, where some of the properties do not have meaningful default values, should be checked to ensure that they return the appropriate value for isPropertySource(boolean). Clients should check that the Restore Default Value button in the Properties view works as expected for their property source.

13. handlerSubmission element deleted from the org.eclipse.ui.commands extension point

What is affected: Plug−ins that used the experimental `handlerSubmission` element introduced into the `org.eclipse.ui.commands` extension point Eclipse 3.0.

Description: In Eclipse 3.0, an experimental element was introduced into the `org.eclipse.ui.commands` extension point. This element was intended as a way to register handlers through XML. Since then, a far superior mechanism, the `org.eclipse.ui.handlers` extension point, has been introduced. Since the element was marked as experimental, it has now been removed.

Action required: Any plug−ins defining a `handlerSubmission` element should migrate to the `org.eclipse.ui.commands` extension point.

14. Static non−final API field GLOBAL_IGNORES_CHANGED in TeamUI made final

What is affected: Plug−ins that were setting the GLOBAL_IGNORES_CHANGED field of TeamUI.

Description: In Eclipse 3.0, the GLOBAL_IGNORES_CHANGED field was added to the TeamUI class. This field is a constant that is used in a property change event to indicate that the list of global ignores maintained by the Team plugin has changed. This field was not marked final in 3.0 but should have been. It has been made final in 3.1.

Action required: Any plugins that were setting the above field can no longer do so.

15. ClassCastException using FillLayout

What is affected: Plug−ins that incorrectly use FillLayout.

Description: In Eclipse 3.0, no layout data was associated with the FillLayout and if an application assigned layout data to a child that was managed by a FillLayout, it was ignored. In Eclipse 3.1, support was added to FillLayout to cache size information in order to improve resize performance. The cached data is stored in a FillData object associated with each child managed by the FillLayout. If an application has incorrectly assigned layout data to a child, a ClassCastException will be thrown when computeSize is called on the parent.

Action required: Find any children in a FillLayout that have layout data assigned and stop assigning the layout data.

16. IllegalArgumentException thrown creating a widget with a disposed parent

What is affected: Plug−ins that catch exceptions while creating widgets.

Description: In Eclipse 3.0, if a widget was created with a disposed parent, no exception was thrown and the widget code failed at a later point or an SWTException with text "Widget Is Disposed" was thrown. In Eclipse 3.1, if a widget is created with a disposed parent, the constructor will throw an IllegalArgumentException with text "Argument not valid".

Action required: Any code that handles the SWTException when creating a widget will also need to handle the IllegalArgumentException.

Changes required when adopting 3.1 mechanisms and APIs

This section describes changes that are required if you are trying to change your 3.0 plug−in to adopt the 3.1 mechanisms and APIs.

Platform undo/redo support

Eclipse 3.1 provides a new infrastructure for defining **undoable operations** and a shared **operation history** that keeps track of operations that have been executed, undone, and redone. The various undo frameworks provided by add−on plug−ins should migrated over time to the platform operation support, so that clients of these frameworks can integrate more deeply with the platform and make their undoable operations available for undo in other plug−in views and editors. See the Undoable operations documentation for basic information about adding undo support to a plug−in. Plug−ins that already define undo support or use another framework can be migrated to the new undo support in a staged fashion, as described below.

Migrating plug−in specific operation (command) classes to IUndoableOperation

Plug−ins that already define classes describing their undoable operations should add an implementation for the interface **IUndoableOperation** to their operation/command classes. Plug−ins may still use older

frameworks for managing the history (command stack) if necessary, but providing an interface for **IUndoableOperation** allows a plug−in's clients to use the same operations in the platform operations history, and to mix and match undoable operations from different plug−ins. This strategy is similar to that used by the SDK text editors to migrate to the new operations framework. If a direct mapping of the interface is not possible, wrappers can be used to map **IUndoableOperation** protocol to legacy undo objects. This strategy is used by the Platform/JDT refactoring support. Migration of the operation/command classes to **IUndoableOperation** is an important step because it allows the undoable operations from different frameworks to be utilized by other plug−ins without either plug−in having to migrate completely.

Migrating command stacks with IOperationHistory

Once undoable operations or commands are expressed in terms of **IUndoableOperation**, plug−ins that define an undo history (command stack) for keeping track of the undoable and redoable operations can migrate to the platform operations history by defining an **IUndoContext** that represents their undo history. Undo histories that were previously managed locally can be merged into the common operation history by defining a unique undo context either for each part or for each model object, adding the appropriate undo context to each operation, and then adding the operation to the platform operation history. Undo histories with more global scope can be implemented by defining a unique undo context representing that undo scope, assigning that context to each operation, and then adding the operation to the platform operation history. See the Undoable operations documentation for examples of creating undo contexts, assigning them, and adding operations to the platform operation history.

Defining undoable operations global to the workbench

Plug−ins that wish their operations to be undoable from workbench views such as the Navigator or Package Explorer should assign the workbench undo context to their operations. See the Undoable operations documentation for more information about this undo context and how it can be retrieved by both workbench and headless plug−ins.

Platform undo/redo action handlers

Plug−ins that do not define an undo infrastructure or undoable operations, but wish to provide access to the platform's undo history, should consider retargeting the global undo and redo action handlers with the new common undo and redo action handlers. The action handlers should be assigned an undo context specifying which undo and redo history is to be shown. Plug−ins can use their locally defined undo contexts for showing "part−local" undo and redo history. The workbench undo context can be used for showing workbench−wide undo and redo history. Again, the Undoable operations documentation has a complete example.

Migrating text operation actions to the common action handlers

Migrating text editor undo and redo actions is a bit different than simple retargeting of the global undo/redo action handlers. The **AbstractTextEditor** framework defines common text actions using a parameterized **TextOperationAction**. These actions are stored locally in the framework and used to dispatch various commands to an editor's text operation target. For text undo to work properly, the text editor framework relies on the presence of text operation actions with the proper id's (`ITextEditorActionConstants.REDO` and `ITextEditorActionConstants.UNDO`).

AbstractTextEditor has been migrated so that it creates the common action handlers, while still assigning them to the TextOperationAction table with their legacy id's. In this way, the new undo and redo action handlers can be retrieved using the legacy techniques for retrieving the action and performing an operation.

Text editors in the **AbstractTextEditor** hierarchy will inherit this behavior.

Editors that do not inherit this behavior from **AbstractTextEditor** should consider migrating any existing undo and redo actions to use the new handlers. Editors with legacy undo and redo TextOperationActions will still have working local undo support, since the JFace Text undo manager API used by these actions is still supported. However, the undo and redo action labels will not be consistent with the new Eclipse SDK undo/redo actions, which show the name of the available undo or redo operation. To create the common undo and redo action handlers, the undo context used by the text viewer's undo manager should be used when creating the action handlers, and those handlers should be set into the editor using the appropriate `ITextEditorActionConstants` id. See **`AbstractTextEditor.createUndoRedoActions()`** and **`AbstractTextEditor.getUndoContext()`** for a detailed example. Editors that rely on an **EditorActionBarContributor** subclass to add to the action bars of their editors can use a similar technique by creating undo and redo action handlers and setting them when the active editor is set.

Help Enhancements

Information Search

Plug−ins that contribute search pages into the **Search** dialog should consider porting all their information−style searches into federated search engines. Since 3.1, all information−style search is separated from the workbench−artifact search. Information search engines are run in parallel as background jobs and their results collated in the new Help view. See Help Search for more details.

Dynamic help

The new dynamic help view will work with existing context IDs that are statically associated with widgets in workbench parts and dialogs. However, if you catch help event yourself and show help, dynamic help view will not be able to show anything useful. To fix the problem, you should adapt to the new `IContextProvider` interface as described in Dynamic Context Help document.

Eclipse 3.0 Plug−in Migration Guide

This guide covers migrating Eclipse 2.1 (or earlier) plug−ins to Eclipse 3.0.

One of the goals of Eclipse 3.0 was move Eclipse forward while remaining compatible with previous versions to the greatest extent possible. That is, plug−ins written against the Eclipse 2.1 APIs should continue to work in 3.0 in spite of the API changes.

The key kinds of compatibility are API contract compatibility and binary compatibility. API contract compatibility means that valid use of 2.1 APIs remains valid for 3.0, so there is no need to revisit working code. Binary compatibility means that the API method signatures, etc. did not change in ways that would cause existing compiled ("binary") code to no longer link and run with the new 3.0 libraries.

While every effort way made to avoid breakage, there are a few areas of incompatibility. This document describes the areas where Eclipse changed in incompatible ways between 2.1 and 3.0, and provides instructions for migrating 2.1 plug−ins to 3.0.

- Eclipse 3.0 Plug−in Migration FAQ
- Incompatibilities between Eclipse 2.1 and 3.0
- Adopting 3.0 mechanisms and API

Eclipse 3.0 Plug−in Migration FAQ

Why did Eclipse API change in incompatible ways between 2.1 and 3.0?

Eclipse 3.0 is an evolution of Eclipse 2.1. There were a few areas where we could not evolve Eclipse while maintaining perfect compatibility across the board. The four main sources of incompatibilities are:

- The Eclipse 3.0 runtime is now based on OSGi.
- The Eclipse 3.0 UI workbench is now split into RCP and IDE parts.
- The Xerces plug−in has been dropped from 3.0.
- Eclipse 3.0 performs more work in background threads.

The list of specific incompatibilities.

Will a 2.1 plug−in work in Eclipse 3.0?

Yes, except in a few cases. If a plug−in relies only on Eclipse 2.1 APIs, then it will continue to work in 3.0. The very few exceptions are places in the API where the changes between 2.1 and 3.0 could not be done in any compatible way; if a plug−in uses one of these, it will not work.

My 2.1 plug−in makes use of classes in internal packages. Will it still work in Eclipse 3.0?

If a plug−in relies on internal classes or behavior not specified in the Eclipse 2.1 API, it's impossible to make blanket claims one way or the other about whether the plug−in might work in 3.0. You'll need to try it.

How do I run my plug−in in Eclipse 3.0 without touching it?

Install your 2.1 plug−in in the eclipse/plugins/ subdirectory of an Eclipse 3.0−based product and restart Eclipse. Eclipse will recognize that the plug−in is an unconverted 2.1 plug−in (by the header on the plugin.xml) and automatically make adjustments to compensate for changes to Platform plug−in dependencies and renamed Platform extension points.

Will a 2.1 plug−ins need to be changed to compile properly in Eclipse 3.0?

Yes in all cases. There are certain differences between Eclipse 2.1 and 3.0 that necessitate changes to all plug−ins going forward. If you have a plug−in written for 2.1 and wish to recompile it, it needs to be migrated to 3.0 before it can be developed further for 3.0.

How do I migrate my plug−in to Eclipse 3.0?

Once you've loaded (or imported) your plug−in project into an Eclipse 3.0 workspace, use **PDE Tools > Migrate to 3.0** (project context menu) to convert the plug−in's manifest to the 3.0 format and automatically adjust the list of required Platform plug−ins and references to Platform extension points that were renamed. In most cases, the code for the plug−in should then compile and run successfully. The code for the plug−in should then be reviewed to make sure that it is not dependent on one of the areas of incompatible API change.

Can I trust that a plug−in will have compile errors or warnings if it relies on API that has changed incompatibly?

No. There are some areas of incompatible change that do not get flagged by the Java compiler.

Can I safely ignore warnings in the code coming from use of deprecated API?

Yes, in the short term. Wherever possible, obsolete APIs are marked as deprecated rather than being deleted outright and continue to work (albeit possibly only under limited conditions). So while there is usually no urgency to get off the deprecated API, the fact that it is now considered obsolete means that there is now a better way to do something. Plug−ins should be weaned off all usage to deprecated API at the earliest convenience.

Once I migrate my plug−in to Eclipse 3.0, can I still install and run the resulting binary plug−in in Eclipse 2.1?

No. This is not supported, and probably wouldn't work due to the renamed extension points.

What is the purpose of org.eclipse.core.runtime.compatibility?

The move in 3.0 to an OSGi−base runtime made some of the existing core runtime APIs obsolete. Wherever possible, obsolete APIs in the org.eclipse.core.runtime.* packages, along with the implementation behind it, were moved from the org.eclipse.core.runtime plug−in to a new org.eclipse.core.runtime.compatibility plug−in. By default, newly−created plug−ins depend on org.eclipse.core.runtime and are expected to use only non−deprecated runtime APIs. On the other hand, existing plug−ins migrating from 2.1 will depend by default on org.eclipse.core.runtime.compatibility and can make use of the old APIs as well (the org.eclipse.core.runtime.compatibility plug−in re−exports APIs of org.eclipse.core.runtime). While the org.eclipse.core.runtime.compatibility plug−in is likely to be included in Eclipse IDE configurations, it's dead wood that's unlikely to be included in products based on RCP configurations.

What is the purpose of org.eclipse.ui.workbench.compatibility?

org.eclipse.ui.workbench.compatibility is a plug−in fragment that provides enhanced binary compatibility for 2.1 plug−ins being run in an Eclipse 3.0−based product. In 3.0, six methods with an explicit dependence on IFile or IMarker were moved from the org.eclipse.ui.IWorkbenchPage interface in order to cleanly separate the workbench from workspace and resources. The org.eclipse.ui.workbench.compatibility fragment arranges to add back these methods so that existing 2.1 plug−ins can run without modification. Note, however, that plug−ins being migrated to 3.0 that reference the moved methods will see compile errors that can (only) be resolved by calling the replacement methods now located on org.eclipse.ui.ide.IDE.

The IWorkbenchPage methods in question are: openEditor(IFile), openEditor(IFile, String), openEditor(IFile, String, boolean), openEditor(IMarker), openEditor(IMarker, boolean), and openSystemEditor(IFile).

Incompatibilities between Eclipse 2.1 and 3.0

Eclipse changed in incompatible ways between 2.1 and 3.0 in ways that affect plug−ins. The following entries describe the areas that changed and provide instructions for migrating 2.1 plug−ins to 3.0. Note that you only need to look here if you are experiencing problems running your 2.1 plug−in on 3.0.

1. Plug−in manifest version
2. Restructuring of Platform UI plug−ins
3. Restructuring of Platform Core Runtime plug−ins
4. Removal of Xerces plug−in
5. Eclipse 3.0 is more concurrent
6. Opening editors on IFiles
7. Editor goto marker
8. Editor launcher
9. Editor registry
10. Workbench marker help registry
11. Text editor document providers
12. Text editors
13. Headless annotation support
14. Console view
15. Java breakpoint listeners
16. Clipboard access in UI thread
17. Key down events
18. Tab traversal of custom controls
19. Selection event order in SWT table and tree widgets
20. New severity level in status objects
21. Build−related resource change notifications
22. Intermediate notifications during workspace operations
23. URL stream handler extensions
24. Class load order
25. Class loader protection domain not set
26. PluginModel object casting
27. ILibrary implementation incomplete
28. Invalid assumptions regarding form of URLs
29. BootLoader methods moved/deleted
30. Plug−in export does not include the plug−in's JARs automatically
31. Re−exporting runtime API
32. Plug−in parsing methods on Platform
33. Plug−in libraries supplied by fragments
34. Changes to build scripts
35. Changes to PDE build Ant task
36. Changes to eclipse.build Ant task
37. Changes to eclipse.fetch Ant task
38. Replacement of install.ini

1. Plug−in manifest version

The header of the manifest files for plug−ins (and plug−in fragments) has changed to include a new line which identifies the appropriate plug−in manifest version. Prior to 3.0, plug−ins did not carry one of these

<?eclipse ...?> lines; after 3.0, they must always have one. This change is to allow the Eclipse runtime to reliably recognize pre−3.0 plug−ins that have not been ported to 3.0, so that it can automatically provide greater binary compatibility for such plug−ins. This is the general form of the plugin.xml file (fragment.xml is similar):

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin ...>
    ...
</plugin>
```

Plug−in manifests created by PDE 3.0 automatically have this form. ***It is strongly recommended that you use the PDE plug−in migration tool.*** It automatically inserts the indicated line into the manifest of 2.1 plug−ins and plug−in fragments and addresses many of the other changes described here.

If you do add this directive to a plugin.xml (manually or using PDE), the file must also be updated to explicitly list the plug−ins on which it depends. For example, prior to Eclipse 3.0 dependencies on org.eclipse.core.runtime and org.eclipse.core.boot were implicit. With 3.0, org.eclipse.core.boot is no longer needed and developers must choose org.eclipse.core.runtime or org.eclipse.core.runtime.compatibility (or neither) as appropriate.

Note: This is one of the incompatibilities that does **not** impact how 2.1 binary plug−ins are run by Eclipse 3.0.

2. Restructuring of Platform UI plug−ins

The org.eclipse.ui plug−in, which used to be the main Platform UI plug−in, now provides just the API and extension points for the generic (i.e., non−IDE−specific) workbench. Optional and IDE−specific API and extension points have moved to other plug−ins.

The impact of this change is two−fold: (1) the moved org.eclipse.ui extension points have new extension point ids; and (2) the list of required plug−ins has changed.

The org.eclipse.ui extension points in the following table have moved to different plug−ins, causing their extension point ids to change. If an existing plug−in contributes an extension to the moved extension points, then the reference in the "point" attribute of the <extension> element in the plug−in manifest file must be changed to refer to the corresponding new ones extension point id. The PDE plug−in migration tool makes these fix−ups.

Note: This is one of the incompatibilities that does **not** impact how 2.1 binary plug−ins are run by Eclipse 3.0. The Eclipse 3.0 runtime automatically detects pre−3.0 plug−ins (by the absence of the aforementioned <?eclipse version="3.0"?> line in the plug−in manifest) and automatically compensates for these extension point and plug−in dependency changes.

Old extension point id	New extension point id
org.eclipse.ui.markerHelp	org.eclipse.ui.*ide*.markerHelp
org.eclipse.ui.markerImageProviders	org.eclipse.ui.*ide*.markerImageProviders
org.eclipse.ui.markerResolution	org.eclipse.ui.*ide*.markerResolution
org.eclipse.ui.projectNatureImages	org.eclipse.ui.*ide*.projectNatureImages
org.eclipse.ui.resourceFilters	org.eclipse.ui.*ide*.resourceFilters

org.eclipse.ui.markerUpdaters	org.eclipse.ui.*editors*.markerUpdaters
org.eclipse.ui.documentProviders	org.eclipse.ui.*editors*.documentProviders
org.eclipse.ui.workbench.texteditor. markerAnnotationSpecification	org.eclipse.ui.*editors*.markerAnnotationSpecification

The following table lists the API packages formerly provided by the org.eclipse.ui plug−in that have been moved to different plug−ins. (The names of the API packages, classes, fields, and methods did not change.) In some cases, the API packages are now split across more than one plug−in. Since the API classes visible to any given plug−in are determined by that plug−in's list of required plug−ins, these changes may require adjusting "<requires>" elements in an existing plug−in's manifest to regain access to API class.

This change only affects plug−ins that depend on the org.eclipse.ui plug−in (that is, includes <import plugin="org.eclipse.ui"/> in the <requires> section of the plug−in manifest); all other plug−ins are unaffected. If it is affected, you *may* need to change the <import> element, or add additional <import> elements, so that all the API classes your plug−in needs are in scope. We strongly recommend that plug−ins only state dependencies on the plug−ins that they actually use. Including unnecessary dependencies reduces runtime performance because the Java class loader must search for classes in all dependents. (The PDE plug−in migration tool will fix up the dependencies, and help to determine a minimal set.)

API package	2.1 plug−in	Corresponding 3.0 plug−in(s)
org.eclipse.jface.text.*	org.eclipse.ui	org.eclipse.jface.text
org.eclipse.text.*	org.eclipse.ui	org.eclipse.jface.text
org.eclipse.ui	org.eclipse.ui	org.eclipse.ui, org.eclipse.ui.ide
org.eclipse.ui.actions	org.eclipse.ui	org.eclipse.ui, org.eclipse.ui.ide
org.eclipse.ui.dialogs	org.eclipse.ui	org.eclipse.ui, org.eclipse.ui.ide
org.eclipse.ui.editors.*	org.eclipse.ui	org.eclipse.ui.editor
org.eclipse.ui.model	org.eclipse.ui	org.eclipse.ui, org.eclipse.ui.ide
org.eclipse.ui.part	org.eclipse.ui	org.eclipse.ui, org.eclipse.ui.ide
org.eclipse.ui.texteditor	org.eclipse.ui	org.eclipse.ui.workbench.texteditor, org.eclipse.ui.editors
org.eclipse.ui.texteditor.*	org.eclipse.ui	org.eclipse.ui.workbench.texteditor
org.eclipse.ui.views.bookmarkexplorer	org.eclipse.ui	org.eclipse.ui.ide
org.eclipse.ui.views.contentoutline	org.eclipse.ui	org.eclipse.ui.views
org.eclipse.ui.views.markers	org.eclipse.ui	org.eclipse.ui.ide
org.eclipse.ui.views.navigator	org.eclipse.ui	org.eclipse.ui.ide
org.eclipse.ui.views.properties	org.eclipse.ui	org.eclipse.ui.views
org.eclipse.ui.views.tasklist	org.eclipse.ui	org.eclipse.ui.ide
org.eclipse.ui.wizards.datatransfer	org.eclipse.ui	org.eclipse.ui.ide
org.eclipse.ui.wizards.newresource	org.eclipse.ui	org.eclipse.ui.ide

3. Restructuring of Platform Core Runtime plug−ins

The Eclipse 3.0 Platform Runtime is based on OSGi, necessitating changes to the structure of the two Platform Runtime plug−ins, org.eclipse.core.runtime and org.eclipse.core.boot.

A new org.eclipse.core.runtime.compatibility plug−in provides an implementation bridge between the old and new APIs, and is the new home for many of the obsolete APIs formerly found in org.eclipse.core.runtime and

org.eclipse.core.boot. Platform Runtime extension points are unaffected by the restructuring.

When migrating the existing plug−in to 3.0, the plug−in's manifest needs to be updated to reflect the new structure of the Eclipse Platform Runtime plug−ins. The PDE plug−in manifest migration tool will add a dependency to org.eclipse.core.runtime.compatibility if required.

Note also that if you mark you plug−in as 3.0 (using <?eclipse version="3.0"?>) and your plug−in defines a Plugin class, you must either explicitly <import plugin="org.eclipse.core.runtime.compatibility"/> in the plug−in manifest or ensure that the Plugin class defines the default constructor.

Note: This is one of the incompatibilities that does **not** impact how 2.1 binary plug−ins are run by Eclipse 3.0. The Eclipse 3.0 runtime automatically detects pre−3.0 plug−ins (by the absence of the <?eclipse version="3.0"?> line in the plug−in manifest) and automatically compensates for these changes to the Platform Runtime.

4. Removal of Xerces plug−in

The org.eclipse.xerces plug−in is no longer necessary and has been deleted. XML parsing support is built in to J2SE 1.4, and the presence of the Xerces plug−in creates class loader conflicts. The javax.xml.parsers, org.w3c.dom.*, and org.xml.sax.* API packages formerly provided by the org.eclipse.xerces plug−in are now available from the J2SE libraries.

If your plug−in requires the org.eclipse.xerces plug−in, you must change your plug−in manifest to remove this stated dependency. Once that is done, the plug−in's code should compile and run without further change.

A 2.1 binary plug−ins with a stated dependency on the org.eclipse.xerces plug−in will be missing a prerequisite when run in a standard Eclipse 3.0 configuration. The plug−in will not be activated as a consequence.

5. Eclipse 3.0 is more concurrent

Prior to Eclipse 3.0, Eclipse operated mostly in a single thread. Most API methods and extension points operated either in the UI thread, or in a thread spawned from a progress dialog that blocked the UI thread. Most plug−in writers did not have to worry much about thread safety, apart from ensuring that all UI activity occurred in the UI thread. In Eclipse 3.0, there is generally much more concurrency. Many operations now occur in a background thread, where they may run concurrently with other threads, including the UI thread. All plug−ins whose code runs in a background thread must now be aware of the thread safety of their code.

In addition to plug−ins that are explicitly running operations in the background using the `org.eclipse.core.runtime.jobs` API, there are several platform API facilities and extension points that make use of background threads. Plug−ins that hook into these facilities need to ensure that their code is thread safe. The following table summarizes the API and extension points that run some or all of their code in a background thread in Eclipse 3.0:

Extension point or API class	Notes
org.eclipse.core.runtime.IRegistryChangeListener	New in Eclipse 3.0, runs in background
org.eclipse.core.resources.IResourceChangeListener	`AUTO_BUILD` events now in background
org.eclipse.core.resources.builders (ext. point)	Auto−build now in background
org.eclipse.core.resources.ISaveParticipant	`SNAPSHOT` now in background

org.eclipse.ui.workbench.texteditor.quickdiffReferenceProvider (ext. point)	New in Eclipse 3.0, runs in background
org.eclipse.ui.decorators (ext. point)	Already in background in Eclipse 2.1
org.eclipse.ui.startup (ext. point)	Already in background in Eclipse 2.1
org.eclipse.team.core.org.eclipse.team.core.repository (ext. point)	Many operations now in background
org.eclipse.team.ui.synchronizeParticipants (ext. point)	New in Eclipse 3.0, runs in background
org.eclipse.debug.core.launchConfigurationTypes (ext. point)	Now runs in background
org.eclipse.jdt.core.IElementChangedListener	`ElementChangedEvent.PRE_AUTO_BUILD` now runs in background, `POST_RECONCILE` already ran in the background

There are various strategies available for making code thread safe. A naive solution is to ensure all work occurs in the UI thread, thus ensuring serialized execution. This is a common approach for UI plug−ins that are not doing CPU−intensive processing. When doing this, be aware of the deadlock risk inherent in `Display.syncExec`. `Display.asyncExec` is generally safer as it does not introduce deadlock risk, at the expense of losing precise control over when the code is executed.

Other techniques for making thread safe code include:

- Wrapping unsafe code in semaphores or Java monitors. The new concurrency infrastructure includes one such semaphore object: `org.eclipse.core.runtime.jobs.ILock`. The advantage of `ILock` over generic locks is that they transfer automatically to the UI thread when doing a `syncExec`, and there is deadlock detection support built into their implementation that logs and then resolves deadlocks.
- Message queues. Code can be serialized by forwarding all processing to a message queue that is processed serially in a single thread. One such example is the SWT deferred event queue (`Display.asyncExec`), which is processed entirely in the UI thread.
- Immutable objects. Make data structures immutable, and make new copies on modification. This is the approach used to make data structures such as `java.lang.String` and `org.eclipse.core.runtime.IPath` thread safe. The advantage of immutable objects is extremely fast read access, at the cost of extra work on modification.

6. Opening editors on IFiles

The following methods were deleted from the org.eclipse.ui.IWorkbenchPage interface. IWorkbenchPage is declared in the generic workbench, but the methods are inherently resource−specific.

- public IEditorPart openEditor(IFile input)
- public IEditorPart openEditor(IFile input, String editorID)
- public IEditorPart openEditor(IFile input, String editorID, boolean activate)
- public IEditorPart openEditor(IMarker marker)
- public IEditorPart openEditor(IMarker marker, boolean activate)
- public void openSystemEditor(IFile input)

Clients of these IWorkbenchPage.openEditor methods should instead call the corresponding public static methods declared in the class org.eclipse.ui.ide.IDE (in the org.eclipse.ui.ide plug−in).

Clients of these IWorkbenchPage.openSystemEditor(IFile) method should convert the IFile to an IEditorInput using new FileEditorInput(IFile) and then call the openEditor(IEditorInput,String) method. In other words,

rewrite page.openSystemEditor(file) as page.openEditor(new FileEditorInput(file), IEditorRegistry.SYSTEM_EXTERNAL_EDITOR_ID). Note: clients using editor id IEditorRegistry.SYSTEM_EXTERNAL_EDITOR_ID must pass an editor input which implements org.eclipse.ui.IPathEditorInput (which FileEditorInput does).

Note: This is one of the incompatibilities that does **not** impact how 2.1 binary plug−ins are run by Eclipse 3.0. Eclipse 3.0 includes a binary runtime compatibility mechanism that ensures existing 2.1 plug−in binaries using any of the deleted openEditor and openSystemEditor methods continue to work as in 2.1 in spite of this API change. (The deleted methods are effectively "added back" by the org.eclipse.ui.workbench.compatibility fragment.)

7. Editor goto marker

The following method was deleted from the org.eclipse.ui.IEditorPart interface. IEditorPart is declared in the generic workbench, but the method is inherently resource−specific.

- public void gotoMarker(IMarker marker)

The corresponding methods were also deleted from the classes in the org.eclipse.ui.part package that implement IEditorPart, namely EditorPart, MultiEditor, MultiPageEditorPart, and MultiPageEditor.

Clients that call this method should instead test if the editor part implements or adapts to org.eclipse.ui.ide.IGotoMarker (in the org.eclipse.ui.ide plug−in) and if so, call gotoMarker(IMarker). The IDE class has a convenience method for doing so: IDE.gotoMarker(editor, marker);

Clients that implement an editor that can position itself based on IMarker information should implement or adapt to org.eclipse.ui.ide.IGotoMarker.

Since IGotoMarker's only method is gotoMarker(IMarker) and has the same signature and specification as the old IEditorPart.gotoMarker(IMarker), existing editor implementations can adapt to this change simply by including IGotoMarker in the implements clause of the class definition.

A 2.1 binary plug−ins with code that calls this method will get an class linking error exception when run in a standard Eclipse 3.0 configuration.

8. Editor launcher

The editor launcher interface org.eclipse.ui.IEditorLauncher is implemented by plug−ins that contribute external editors. The following method was removed from this interface. IEditorLauncher is declared in the generic workbench, but the method is inherently resource−specific.

- public void open(IFile file)

It was replaced by

- public void open(IPath file)

Clients that call IEditorLauncher.open(file) should instead call IEditorLauncher.open(file.getLocation()). Clients that implement this interface should replace (or augment) their implementation of open(IFile) by one for open(IPath).

A 2.1 binary plug−ins with code that calls this method will get an class linking error exception when run in a standard Eclipse 3.0 configuration.

9. Editor registry

The following methods were removed from the org.eclipse.ui.IEditorRegistry interface. IEditorRegistry is declared in the generic workbench, but the methods are inherently resource−specific.

- public IEditorDescriptor getDefaultEditor(IFile file)
- public void setDefaultEditor(IFile file, String editorId)
- public IEditorDescriptor[] getEditors(IFile file)
- public ImageDescriptor getImageDescriptor(IFile file)

Clients that call getEditors(file) or getImageDescriptor(file) should call the "String" equivalent methods:

- Rewrite registry.getEditors(file) to be registry.getEditors(file.getName())
- Rewrite registry.getImageDescriptor(file) to be registry.getImageDescriptor(file.getName())

Clients that call setDefaultEditor(IFile file, String editorId) and getDefaultEditor(IFile file) should should instead call the corresponding public static methods declared in the class org.eclipse.ui.ide.IDE (in the org.eclipse.ui.ide plug−in):

- Rewrite registry.getDefaultEditor(file) to be IDE.getDefaultEditor(file)
- Rewrite registry.setDefaultEditor(file, id) to be IDE.setDefaultEditor(file, id)

Also, the API contract for the method IEditorRegistrygetDefaultEditor() was changed. This method, which is also now deprecated, and will always return the System External Editor editor descriptor. This change impacts clients that assumed the default editor returned would be a text editor.

There are new constants that represent the system external editor and system in−place editor identifiers (SYSTEM_EXTERNAL_EDITOR_ID and SYSTEM_INPLACE_EDITOR_ID). These two editors require an editor input that implements or adapts to org.eclipse.ui.IPathEditorInput. Note that the in−place editor descriptor will not exist in Eclipse configurations that do not support in−place editing.

10. Workbench marker help registry

The following method was deleted from the org.eclipse.ui.IWorkbench interface. IWorkbench is declared in the generic workbench, but the method is inherently resource−specific.

- public IMarkerHelpRegistry getMarkerHelpRegistry()

Clients of IWorkbench.getMarkerHelpRegistry() should instead call the public static method org.eclipse.ui.ide.IDE.getMarkerHelpRegistry() (in the org.eclipse.ui.ide plug−in).

A 2.1 binary plug−ins with code that calls this method will get an exception when run in a standard Eclipse 3.0 configuration.

11. Text editor document providers

In order to make org.eclipse.ui.texteditor.AbstractTextEditor independent of IFile, org.eclipse.ui.texteditor.AbstractDocumentProvider introduces the concept of a document provider operation (DocumentProviderOperation) and a document provider operation runner (IRunnableContext). When requested to perform reset, save, or synchronize, AbstractDocumentProvider creates document provider operations and uses the operation runner to execute them. The runnable context can be provided by subclasses via the getOperationRunner method. Here is a summary of the changes that clients must adapt to:

- Added protected abstract IRunnableContext getOperationRunner(); non−abstract subclasses must implement this method in order to provide their own operation runner.
- The method resetDocument has been changed to final in order to allow the document provider to wrap the function with a document provider operation. The document provider operation calls the newly introduced doResetDocument method. AbstractDocumentProvider.doResetDocument contains the code that originally resided inside AbstractDocumentProvider.resetDocument. Subclasses must change their implementation of resetDocument to doResetDocument and any contained call of super.resetDocument to super.doResetDocument.
- The method saveDocument has been changed to final in order to allow the document provider to wrap the function with a document provider operation. The document provider operation calls the newly introduced doSaveDocument method. AbstractDocumentProvider.doSaveDocument contains the code that originally resided inside AbstractDocumentProvider.saveDocument. Subclasses must change their implementation of saveDocument to doSaveDocument and any contained call of super.saveDocument to super.doSaveDocument.
- The method synchronize has been changed to final in order to allow the document provider to wrap the function with a document provider operation. The document provider operation calls the newly introduced doSynchronize method. AbstractDocumentProvider.doSynchronize contains the code that originally resided inside AbstractDocumentProvider.synchronize. Subclasses must change their implementation of synchronize to doSynchronize and any contained call of super.synchronized to super.doSynchronize.

The AbstractDocumentProvider subclass org.eclipse.ui.editors.text.StorageDocumentProvider implements the getOperationRunner method to always returns null. This means that subclasses of StorageDocumentProvider should not be affected by this change.

The StorageDocumentProvider subclass org.eclipse.ui.editors.text.FileDocumentProvider implements the getOperationRunner method that returns an IRunnableContext for executing the given DocumentProviderOperations inside a WorkspaceModifyOperation. Other changes to FileDocumentProvider are:

- resetDocument(Object) has been replaced by doResetDocument(Object,IProgressMonitor).
- synchronize(Object) has been replaced by doSynchronize(Object,IProgressMonitor).

12. Text editors

Changes to org.eclipse.ui.texteditor.AbstractTextEditor:

- The method IEditorPart.gotoMarker(IMarker marker) was removed from the IEditorPart interface because it was resource−specific. The default implementation provided by AbstractTextEditor has been removed from AbstractTextEditor and has been moved to AbstractDecoratedTextEditor. Direct subclasses of AbstractTextEditor or StatusTextEditor that want to provide that functionality should

follow the migration instructions for IEditorPart [ref Editor goto marker].

- AbstractTextEditor no longer differentiates between implicit and explicit document providers as the concrete implementation was resource specific. This functionality has been moved to AbstractDecoratedTextEditor. Direct subclasses of AbstractTextEditor or StatusTextEditor must perform the following steps:
  - Override getDocumentProvider.
  - Override the newly introduced hook method for disposing the document provider (disposeDocumentProvider)
  - Override the newly introduced hook method setDocumentProvider(IEditorInput) that is called while updating the document provider for the new editor input. I.e., that is the method in which you can configure the appropriate implicit document provider for the given editor input.
- All internal occurrences of WorkspaceModifyOperation have been removed. The editor now calls the document provider methods directly (see described changes for document providers);
  - Removed createSaveOperation, changed performSaveOperation to performSave. Subclasses of AbstractTextEditor overriding createSaveOperation or performSaveOperation must now override performSave.
  - Removed createdRevertOperation, changed peformRevertOperation to performRevert. Sunclasses overriding createRevertOperation or performRevertOperation must now override performRevert.
  - The implementation of the method handleEditorInputChanged has been changed to not use WorkbenchModifyingOperation. Subclasses overriding handleEditorInputChange must adapt accordingly. Please use the changes applied to AbstractTextEditor.handleEditorInputChange as the blueprint.
- AbstractTextEditor.createActions no longer registers any actions under ITextEditorActionConstants.ADD_TASK and ITextEditorActionConstant.BOOKMARK as those actions are IDE−specific. The registration of these actions has been moved to AbstractDecoratedTextEditor. Direct subclasses of AbstractTextEditor or StatusTextEditor should override the createActions method and add the following lines (accordingly adapted to their circumstances):

```
ResourceAction action= new AddMarkerAction(TextEditorMessages.getResourceBundle(), "Edito
action.setHelpContextId(ITextEditorHelpContextIds.BOOKMARK_ACTION);
action.setActionDefinitionId(ITextEditorActionDefinitionIds.ADD_BOOKMARK);
setAction(IDEActionFactory.BOOKMARK.getId(), action);

action= new AddTaskAction(TextEditorMessages.getResourceBundle(), "Editor.AddTask.", thi
action.setHelpContextId(ITextEditorHelpContextIds.ADD_TASK_ACTION);
action.setActionDefinitionId(ITextEditorActionDefinitionIds.ADD_TASK);
setAction(IDEActionFactory.ADD_TASK.getId(), action);
```

The AbstractTextEditor subclass org.eclipse.ui.texteditor.StatusTextEditor provides the predicate method isErrorStatus(IStatus). Subclasses may override in order to decide whether a given status must considered an error or not.

Changes to org.eclipse.ui.editors.text.AbstractDecoratedTextEditor:

- The IEditorPart method gotoMarker(IMarker marker) has been deprecated to allow a future change of its visibility. Clients of this method must follow the migration instructions in the deprecation message.
- AbstractDecoratedTextEditor returns an adapter for IGotoMarker.
- AbstractDecoratedTextEditor implements the concept of implicit/explicit document providers previously provided by AbstractTextEditor, as discussed above.

13. Headless annotation support

As part of the introduction of headless annotation support, the following changes to Annotation were made:

- Removed static Annotation.drawImage methods, use org.eclipse.jface.text.source.ImageUtilities instead.
- Removed setLayer, getLayer, and paint methods from Annotation, annotations which want to draw themselves should implement IAnnotationPresentation.
- MarkerAnnotation now directly implements the methods removed from Annotation (setLayer, getLayer, paint).
- The following classes have been moved from the plug−in org.eclipse.jface.text to the plug−in org.eclipse.text, the package names remain the same and org.eclipse.jface.text rexports the classes from org.eclipse.jface.text, such that this change is transparent to clients:

```
org.eclipse.jface.text.source.Annotation
org.eclipse.jface.text.source.AnnotationModel
org.eclipse.jface.text.source.AnnotationModelEvent
org.eclipse.jface.text.source.IAnnotationModel
org.eclipse.jface.text.source.IAnnotationModelListener
org.eclipse.jface.text.source.IAnnotationModelListenerExtension
```

14. Console view

Eclipse 3.0 has new generic console support. The generic console is available via the Window > Show View > Basic > Console, and is used by the Eclipse debug and Ant integration.

The view id for the console has changed from org.eclipse.debug.ui.ConsoleView to org.eclipse.ui.console.ConsoleView. 2.1 plug−ins that programmatically open the console will be unsuccessful because that the old view no longer exists.

15. Java breakpoint listeners

In 3.0, the return types for the methods org.eclipse.jdt.debug.core.IJavaBreakpointListener.breakpointHit(IJavaBreakpoint, IJavaThread) and installingBreakpoing(IJavaTarget, IJavaBreakpoint, IJavaType) changed from boolean to int to allow listeners to vote "don't care". In releases prior to 3.0, listeners could only vote "suspend" or "don't suspend" when a breakpoint was hit, and "install" or "don't install" when a breakpoint was about to be installed. In 3.0, listeners can also vote "don't care" for either of these notifications. This allows clients to only make a decisive vote in situations that they care about. For "breakpoint hit" notifications, the breakpoint will suspend if any listeners vote "suspend", or all listeners vote "don't care"; and it will not suspend if at least one listener votes "don't suspend" and no listeners vote "suspend". Similarly, for "breakpoint installing" notifications, the breakpoint will be installed if any listeners vote to install, or all listeners vote "don't care"; and it will not be installed if at least one listener votes "don't install" and no listeners vote "install". In general, implementors should return DONT_CARE unless they have a strong opinion one way or the other. It is important to keep in mind, for example, that voting "suspend" will override any other listener's vote of "don't suspend".

The IJavaBreakpointListener interface is implemented by clients that create or react to breakpoints in Java code. There are likely few clients beyond JDT itself, save the one that reported the problem (bug 37760) that this change remedies. This is a breaking change for existing code that implements the IJavaBreakpointListener interface. This code needs to be modified to return an appropriate int value before it

will compile or run in 3.0.

16. Clipboard access in UI thread

Prior to 3.0, the methods on the SWT class org.eclipse.swt.dnd.Clipboard were tacitly permitted to run in threads other than the UI thread. This oversight resulted in failures on GTK where the operating system requires that all clipboard interactions be performed in the UI thread. The oversight was not revealed earlier because many applications are single−threaded and receive most of their testing on Windows. In order for the Clipboard API to be sustainable and cross−platform, in 3.0 the specification and implementation of all Clipboard API methods have been changed to throw an SWT Exception (ERROR_THREAD_INVALID_ACCESS) if invoked from a non−UI thread. Clipboard services are commonly provided automatically by Eclipse components such as the text editor, which insulate many clients from this breaking change. Existing code that does make direct use of Clipboard should ensure that the API methods are called on the correct thread, using Display.asyncExec or syncExec when appropriate to shift accesses into the UI thread.

17. Key down events

In 3.0, SWT reports key down events before the work is done in the OS. This is much earlier than it was prior to 3.0. This change was made to support key bindings in Eclipse which necessitates intercepting key events before any widget has a chance to process the character. Consequences of this change are visible to code that handles low−level org.eclipse.swt.SWT.KeyDown events directly. For example, it means that when a listener on a Text widget receives a key down event, the widget's content (getText()) will not yet include the key just typed (it would have prior to 3.0). The recommended way to get the full text from the widget including the current key is to handle the higher−level SWT.Modify or SWT.Verify events rather than the low−level SWT.KeyDown event; code that already does it this way is unaffected by this change.

18. Tab traversal of custom controls

Prior to 3.0, when the focus was in the SWT class org.eclipse.swt.widgets.Canvas or one of its subclasses (including custom widgets), typing Ctrl+Tab, Shift+Tab, Ctrl+PgUp, or Ctrl+PgDn would automatically trigger traversal to the next/previous widget without reporting a key event. This behavior was unspecified, and runs counter to the rule that Canvases see every key typed in them. The proper way to handle traversal is by registering a traverse listener. In order to properly support Eclipse key bindings in 3.0, the default behavior was changed so that Canvas now sees Ctrl+Tab, Shift+Tab, Ctrl+PgUp, and Ctrl+PgDn key events instead of traversing. If you use a raw Canvas or define a subclass of Canvas, ensure that that you register a traverse listener.

19. Selection event order in SWT table and tree widgets

Mouse selections of items in the SWT classes org.eclipse.swt.widgets.Table and Tree generate the event sequence MouseDown−Selection−MouseUp uniformly in all operating environments. Similarly, keyboard selections generate the event sequence KeyDown−Selection−KeyUp uniformly in all operating environments. Prior to 3.0, the event order was not uniform, with Motif and Photon at variance with the rest by always reporting the Selection event first; i.e., Selection−MouseDown−MouseUp or Selection−KeyDown−KeyUp. For 3.0, the event order on Motif and Photon has been changed to match the others. Existing code that was functioning correctly on {Windows, GTK} and on {Motif, Photon} is unlikely to be affected. But it is wise to check your code to ensure that it does not rely on an invalid event order.

20. New severity level in status objects

`org.eclipse.core.runtime.IStatus` has a new severity constant, `IStatus.CANCEL`, that can be used to indicate cancelation. Callers of `IStatus.getSeverity()` that rely on the set of possible severities being limited to `IStatus.OK`, `INFO`, `WARNING`, and `ERROR` are affected by this addition. Callers of `getSeverity` should update their code to include the new severity.

21. Build−related resource change notifications

In Eclipse 3.0, workspace auto−builds now occur in a background thread. This required an API contract change to `org.eclipse.core.resources.IResourceChangeEvent`. The contract of `IResourceChangeEvent` previously guaranteed the following ordering of events for all workspace changes:

1. `PRE_DELETE` or `PRE_CLOSE` event notification if applicable
2. Perform the operation
3. `PRE_AUTO_BUILD` event notification
4. If auto−build is on, perform incremental workspace build
5. `POST_AUTO_BUILD` event notification
6. `POST_CHANGE` event notification

With auto−build now running in the background, there is no longer any guarantee about the temporal relationship between the `AUTO_BUILD` events and the `POST_CHANGE` event. In Eclipse 3.0, steps 3−5 in the above structure are removed from the operation. The resulting picture looks like this:

1. `PRE_DELETE` or `PRE_CLOSE` event notification if applicable
2. Perform the operation
3. `POST_CHANGE` event notification

Periodically, the platform will perform a background workspace build operation. Note that this happens regardless of the whether auto−build is on or off. The exact timing of when this build occurs will not be specified. The structure of the build operation will look like this:

1. `PRE_BUILD` event notification (`PRE_BUILD` is the new name for `PRE_AUTO_BUILD`)
2. If auto−build is on, perform incremental workspace build
3. `POST_BUILD` event notification (`POST_BUILD` is the new name for `POST_AUTO_BUILD`)
4. `POST_CHANGE` event notification

The reference point for the deltas received by auto−build listeners will be different from post−change listeners. Build listeners will receive notification of all changes since the end of the last build operation. Post−change listeners will receive a delta describing all changes since the last post−change notification. This new structure retains three characteristics of resource change listeners that have been true since Eclipse 1.0:

- `POST_CHANGE` listeners receive notification of absolutely all resource changes that occur during the time they are registered. This includes changes made by builders, and changes made by other listeners.
- `PRE_AUTO_BUILD` listeners receive notification of all resource changes **except** changes made by builders and resource change listeners.
- `POST_AUTO_BUILD` listeners receive notification of all resource changes **except** changes made by other `POST_AUTO_BUILD` listeners.

However, there are some important differences with this approach. Prior to Eclipse 3.0, auto−build listeners were always called before `POST_CHANGE` listeners. For this reason, the delta received by auto−build listeners was always a subset of the delta received by the `POST_CHANGE` listeners. This relationship is now essentially reversed. Auto−build listeners will receive a delta that is a super−set of all deltas supplied to `POST_CHANGE` listeners since the end of the last background build. As before, auto−build listeners will be allowed to modify the workspace, and post−change listeners will not.

It will no longer be true that upon completion of a workspace changing operation, that `AUTO_BUILD` event listeners will have been notified. Client code that registers resource change listeners with `IWorkspace.addResourceChangeListener(IResourceChangeListener)` is unlikely to be affected by this change because `AUTO_BUILD` events were never reported to these listeners. However, clients that use `IWorkspace.addResourceChangeListener(IResourceChangeListener,int)` and specify an event mask that includes `AUTO_BUILD` events are likely to be broken by this change if they make assumptions about when auto−build listeners run or what thread they run in. For example, if an auto−build listener is updating a domain model to reflect changes to the workspace, then this update might not have happened when the workspace changing operation returns. It is worth noting that only UI−level code can be affected in this way. Core−level code that is called via API may be called within the scope of an `IWorkspaceRunnable`, so it can never be sure about when resource change listeners will be called. The suggested fix for this breakage is to use `POST_CHANGE` instead of build listeners if it is necessary to have notification occur before the operation completes.

22. Intermediate notifications during workspace operations

It will no longer be guaranteed that all resource changes that occur during the dynamic scope of an `IWorkspaceRunnable` will be batched in a single notification. This mechanism can still be used for batching changes to avoid unnecessary builds and notifications, but the Platform may now decide to perform notifications during the operation. This API contract change is not likely to be a breaking change for existing clients. It is equivalent to the Platform deciding to call `IWorkspace.checkpoint` periodically during a long running operations. The reason for this change is that it is now possible for multiple threads to be modifying the workspace concurrently. When one thread finishes modifying the workspace, a notification is required to prevent responsiveness problems, even if the other operation has not yet completed. This change also allows users to begin working on a set of resources before the operation completes. For example, a user can now begin browsing files in a project that is still in the process of being checked out. The new method `IWorkspace.run(IWorkspaceRunnable, ISchedulingRule, int, IProgressMonitor)` has an optional flag, `AVOID_UPDATE`, which operations can use as a hint to the platform to specify whether periodic updates are desired.

23. URL stream handler extensions

What is affected: Plug−ins that contribute extensions to the `org.eclipse.core.runtime.urlHandlers` extension point.

Description: The contract for the `org.eclipse.core.runtime.urlHandlers` extension point was changed to use the URL Stream Handler service provided by OSGi. The OSGi support is superior to the one in Eclipse 2.1, and correctly handles dynamic handlers. Because of various design issues with the base Java URL handler mechanism, URLStreamHandlers registered with the OSGi handler service must implement `org.osgi.service.url.URLStreamHandlerService`.

Action required: Formerly, the handler class had to implement `java.net.URLStreamHandler` and extend the urlHandlers extension point. The extension point is no longer supported and the handler must be

updated to implement `org.osgi.service.url.URLStreamHandlerService` interface. The OSGi framework provides an abstract base class (`org.osgi.service.url.AbstractURLStreamHandlerService`) that can be trivially subclassed to fill this role.

Instead of registering the handler using an extension point, plug−ins must now do so by registering their handler as a service. For example,

```
Hashtable properties = new Hashtable(1);
properties.put(URLConstants.URL_HANDLER_PROTOCOL, new String[] {MyHandler.PROTOCOL});
String serviceClass = URLStreamHandlerService.class.getName();
context.registerService(serviceClass, new MyHandler(), properties);
```

24. Class load order

What is affected: Plug−ins which supply packages provided which are also supplied by other plug−ins. A very limited number of plug−ins are affected by this change and some of those affected will actually benefit (see below).

Description: In Eclipse 2.x, class loaders search for classes in the following order: consult (1) parent class loader (in practice this is the Java boot class loader), then (2) its own classpath contents, and finally (3) all of its prerequisites in the order declared. OSGi offers an optimization over this model. In this approach a class loader will consult (1) parent class loader (again, effectively the Java boot classloader), then either (2a) a single prerequisite known to contribute classes in the package being queried or (2b) its own classpath entries for the desired class.

The class loader determines whether to consult self or its prerequisites based on its imported and required packages. This information is inferred from the plug−in content in the case of traditional plug−ins and directly specified in the case of plug−ins with explicit OSGi bundle manifest. In either case, it is known *a priori* which class loaders will supply the classes for which packages. This offers performance improvements as well as a solution to the vexing problem of multiple prerequisites contributing the same classes.

Take for example the case of Xerces and Xalan, both of which contain various classes from org.xml packages. Using the first approach, the Xerces plug−in would see its copy of these classes while the Xalan plug−in would see their copy. Since these plug−ins need to communicate, ClassCastExceptions occur. Using the second approach, only one of the two plug−ins contributes the duplicate classes and both plug−ins see the same copies.

Action required: The action required depends on the particulars of the usecase. Affected developers need to review their classpath and resolve any conflicts which may be happening.

25. Class loader protection domain not set

What is affected: Plug−ins that expect the protection domain of their class loader to be set at all times.

Description: In Eclipse 2.1 plug−in class loaders were java.security.SecureClassloaders and, as such, always had a protection domain set. In Eclipse 3.0, class loaders do not extend SecureClassloader and only set the protection domain if Java security is turned on (not the normal case).

Action required: The action required will depend on the scenario in which the plug−in is using the protection

domain.

26. PluginModel object casting

What is affected: Plug−ins which cast objects of type org.eclipse.core.runtime.IPlugin* to org.eclipse.core.runtime.model.Plugin*Model. Even though the relationship between these interfaces and the model classes is not specified in the Eclipse 2.1 API, we are explicitly calling out this change as we have found instances of plug−ins relying on this relationship in the 2.1 implementation.

Description: The Eclipse API provides a series of interfaces (e.g., `IPluginDescriptor`) and so−called "model" classes (e.g., `PluginDescriptorModel`) related to plug−ins and the plug−in registry. In the Eclipse 2.1 implementation it happens that the model classes implement the relevant interfaces. In the new OSGi−based runtime, the plug−in registry has been significantly reworked to allow for a separation between the class loading and prerequisite aspects of plug−ins and the extension and extension−point aspects. As such the Eclipse 3.0 runtime is unable to maintain the implementation relationship present in 2.1.

Action required: Plug−ins relying on this non−API relationship need to be reworked code according to their usecase. More information on this is given in the recommended changes section of this document and in the Javadoc for the related classes and methods.

27. ILibrary implementation incomplete

What is affected: Plug−ins that use `org.eclipse.core.runtime.ILibrary`.

Description: The new runtime maintains the classpath entries in a different and incompatible form from Eclipse. As a result, the compatibility layer is unable to correctly model the underlying OSGi structures as ILibrary objects. The runtime's compatibility support creates ILibrary objects but must assume default values for everything except the library's path.

Action required: Users of ILibrary should consider accessing the desired header values (e.g., `Bundle−Classpath`) from the appropriate Bundle (see `Bundle.getHeaders()`) and using the `ManifestElement` helper class to interpret the entries. See the class Javadoc for more details.

28 Invalid assumptions regarding form of URLs

What is affected: Plug−ins that make assumptions regarding their installation structure, location and the local file system layout.

Description: Methods such as `IPluginDescriptor.getInstallURL()` return URLs of a particular form. Despite their form being unspecified, various plug−ins are making assumptions based on the current implementation. For example, they may expect to get a `file:` URL and use URL.getFile() and use `java.io.File` manipulation on the result. To date, this has been a workable but rather fragile approach. For example, if a plug−in is installed on a web server, it is possible that an `http:` URL would be returned. The new Eclipse 3.0 runtime is even more flexible and opens more possibilities for execution configurations (e.g., maintaining whole plug−ins in JARs rather than exploded in directories). That is, while the new OSGi−based runtime does not actually break 2.1 API, it exposes more cases where assumptions made in current plug−ins are invalid.

Action required: Plug−in writers should ensure that the information to which they need access is available via `getResource()` (and is on the classpath) or use the relevant API for accessing the contents of a plug−in (e.g., `Bundle.getEntry(String)`).

29. BootLoader methods moved/deleted

What is affected: Non−plug−in code that calls certain methods from the class `org.eclipse.core.boot.BootLoader`.

Description: The static methods BootLoader.startup(), shutdown() and run() were moved to org.eclipse.core.runtime.adaptor.EclipseStarter, which is part of the OSGi framework. This API is the interface between the main() in startup.jar and the OSGi framework/Eclipse runtime. The restructuring of the runtime did not permit these methods to remain on BootLoader. The old BootLoader class is now located in the runtime compatibility layer and is deprecated, and the moved methods are stubbed to do nothing.

There is no replacement for the old BootLoader.getRunnable() as the runtime can no longer support the acquisition of individual applications. Rather, users must indicate the application of interest when they start the platform.

Action required: In general this API is used by very few people (it cannot be used from within an Eclipse plug−in). In the rare case that it is, the code must be adapted to use the corresponding methods on EclipseStarter.

30. Plug−in export does not include the plug−in's JARs automatically

What *is affected:* All plug−ins.

Description: In Eclipse 2.1, a plug−in's bin.includes line from their build.properties did not have to contain the list of JARs from their library declaration in the plugin.xml file; these JARs were added for free. In Eclipse 3.0 the list of files in the bin.includes section of the build.properties is an exhaustive list and must include all files which plug−in developers intend to be included in their plug−in when building or exporting.

Action required: Ensure that the bin.includes line from the build.properties file includes all of the JARs listed in your plugin.xml file.

31. Re−exporting runtime API

What is affected: Plug−ins that expose API that includes elements from changed runtime API.

Description: Various plug−ins expose API that includes elements from the runtime API. With the changes to the Eclipse 3.0 runtime outlined here, client plug−ins must re−evaluate their use of runtime API in their API.

Action required: This scenario is quite rare as very little of the Eclipse runtime API is changing. Depending on the scenario, clients may have to change their API or continue to rely on the the compatibility layer.

32. Plug−in parsing methods on Platform

What is affected: Plug−ins that use `org.eclipse.core.runtime.Platform.parsePlugins(..., Factory)`.

Description: The method `org.eclipse.core.runtime.Platform.parsePlugins(..., Factory)` has been moved. The API associated with the Factory argument has been moved from the org.eclipse.core.runtime plug−in up to the org.eclipse.core.runtime.compatibility plug−in (which depends on the runtime plug−in). As a result, the parsing method has been moved as well.

Action required: Users of this method should use the same method on the class `org.eclipse.core.runtime.model.PluginRegistryModel`.

33. Plug−in libraries supplied by fragments

What *is affected:* Plug−ins that specify code on their classpath but do not supply that code (i.e., the JAR is supplied by a fragment; for example, the org.eclipse.swt plug−in).

Description: The new runtime must convert plug.xml files to manifest.mf files behind the scenes. This is done through a straight mechanical transformation and an analysis of the jars listed and supplied by the plug−in. In the case where a plug−in specifies a jar on its classpath but does not supply the jar, there is no code to analyze and the plug−in convertor cannot generate a correct manifest.mf.

Action required: Providers of such plug−ins must either change to supply the appropriate jar in the plug−in itself or hand craft/maintain a META−INF/MANIFEST.MF file for their plug−in. Typically this can be done using PDE to get the initial manifest and then adding in the appropriate Provide−Package header.

34. Changes to build scripts

What is affected: Scripts (e.g., Ant build.xml files) which define classpaths containing runtime−related jars and class directories.

Description: The new runtime contains a number of new plug−ins and jars. Their introduction was mandated by the refactoring of the runtime into configurable pieces. For most runtime situations these changes are transparent. However, if you have custom build.xml (or similar) scripts which currently compile code against `org.eclipse.core.runtime`, you will need to update them before they will function correctly. A typical script contains a classpath entry in a <javac> task that references the `org.eclipse.core.runtime` plug−in as follows:

```
../org.eclipse.core.runtime/bin;../org.eclipse.core.runtime/runtime.jar
```

The runtime plug−in continues to contain much of the original runtime code. However, various parts of the runtime which are there only for compatibility purposes are contained in a compatibility plug−in (`org.eclispe.core.runtime.compatibility`). Most of the new runtime code is contained in a collection of plug−ins (`org.eclipse.osgi.*`).

Action required: Developers should add the entries below as needed to eliminate compilation errors. While the complete set of jars supplied is listed below, typical uses require only a subset on the classpath at compile time. As usual, the inclusion of the /bin directories is discretionary. The entries are given here in logical

groupings by supplying plug−in:

- ../org.eclipse.core.runtime.compatibility/bin;../org.eclipse.core.runtime.compatibility/compatibility.jar;
- ../org.eclipse.osgi/bin;../org.eclipse.osgi/osgi.jar;
- ../org.eclipse.update.configurator/bin;../org.eclipse.update.configurator/configurator.jar;
- ../org.eclipse.osgi.util/util.jar;

In addition the following jars may be required in special cases:

- ../org.eclipse.osgi/core.jar; ../org.eclipse.osgi/resolver.jar; ../org.eclipse.osgi/defaultAdaptor.jar; ../org.eclipse.osgi/eclipseAdaptor.jar; ../org.eclipse.osgi/console.jar

While updating such scripts, you should also take the opportunity to clean up (i.e., remove) references to `org.eclipse.core.boot`. This plug−in is obsolete and longer contains any code. The entries can be left on the classpath but they serve no purpose and should be removed. Look to remove:

```
../org.eclipse.core.boot/bin;../org.eclipse.core.boot/boot.jar
```

35. Changes to PDE build Ant task

What is affected: Scripts (e.g., Ant build.xml files) using the eclipse.buildScript task.

Description: PDE Build introduced a new property to the eclipse.buildScript task to control the generation of plug−ins build scripts. This was mandated by the introduction of the new OSGi−based runtime.

Action required: If you want to use Eclipse 3.0 to build a 2.1 based product, then introduce in eclipse.buildScript the property "buildingOSGi" and set it to false. For example:

```
<eclipse.buildScript ... buildingOSGi="false"/>
```

36. Changes to eclipse.build Ant task

What is affected: Scripts (e.g., Ant build.xml files) using the eclipse.buildScript task.

Description: PDE Build introduced a new property to the eclipse.buildScript task to control the generation of plug−ins build scripts. This was mandated by the introduction of the new OSGi−based runtime.

Action required: If you want to use Eclipse 3.0 to build a 2.1 based product, then introduce in eclipse.buildScript the property "buildingOSGi" and set it to false. For example:

```
<eclipse.buildScript ... buildingOSGi="false"/>
```

37. Changes to eclipse.fetch Ant task

What is affected: Scripts (e.g., Ant build.xml files) using the eclipse.buildScript task.

Description: PDE Build changed the behavior of the eclipse.fetch task to ease building eclipse in an automated build style. The elements style now only support one entry at a time and the scriptName is always ignored.

Action required: If you had a list of entries in the "elements" tag of an eclipse.fetch call, spread them out over several call to eclipse.fetch. If you use to set the scriptName, note that now the generated fetch script is always called "fetch_{elementId}". For example:

```
<eclipse.fetch elements="plugin@org.eclipse.core.runtime, feature@org.eclipse.platform" .../>
```

becomes

```
<eclipse.fetch elements="plugin@org.eclipse.core.runtime" .../>
<eclipse.fetch elements="feature@org.eclipse.platform" .../>
```

38. Replacement of install.ini

The install.ini file is no longer included. In its place is the new config.ini file in the configuration sub−directory. Products that used the install.ini file to specify a primary feature (e.g., to provide branding information) need to make changes to the config.ini file instead. In addition to the new filename, the names of the keys have changed.

The value of the feature.default.id key in 2.1 should be set as the value of the new eclipse.product key. The value of the eclipse.application should be set to "org.eclipse.ui.ide.workbench".

Finally, in 2.1 the image for the splash image was always splash.bmp in the branding plug−in's directory. In 3.0 the location of the splash image is provided explicitly by the osgi.splashPath key in the config.ini file.

Changes required when adopting 3.0 mechanisms and APIs

This section describes changes that are required if you are trying to change your 2.1 plug−ing to adopt the 3.0 mechanisms and APIs.

Getting off of org.eclipse.core.runtime.compatibility

The Eclipse 3.0 runtime is significantly different. The underlying implementation is based on the OSGi framework specification. Eclipse 3.0 runtime includes a compatibilty layer (in the org.eclipse.core.runtime.compatibility plug−in) which maintains the 2.1 APIs. Plug−in developers interested in additional performance and function should consider adopting the 3.0 APIs and removing their dependence on the compatibility layer. Compatibility code shows up in three places:

- org.eclipse.core.boot − entire plug−in is legacy
- org.eclipse.core.runtime.compatibility − entire plug−in is legacy
- org.eclipse.core.runtime − various classes and methods are legacy

The text below gives more detail on the which classes and methods are present for compatibility purposes as well as guidance on how to update your plug−in.

Plug−ins and bundles

The Eclipse runtime has been refactored into two parts; classloading and prerequisite management, and extension/extension−point management. This split allows for natural/seamless adoption of the OSGi framework specification for classloading and prerequisite management. This in turn enables a range of new capabilities in the runtime from dynamic plug−in install/update/uninstall to security and increased

configurability.

While we continue to talk about *plug−ins*, in the new runtime a plug−in is really a *bundle* plus some extensions and extension−points. The term *bundle* is defined by the OSGi framework specification and refers to a collection of types and resources and associated inter−bundle prerequisite information. The *extension registry* is the new form of the plug−in registry and details only extension and extension−point information. By−in−large the extension registry API is the same as the relevant plug−in registry API (for more information see Registries).

In the Eclipse 2.x runtime, the plug−in object has a number of roles and responsibilities:

- Lifecycle − The `Plugin` class implements method such as startup() and shutdown(). The runtime uses these methods to signal the plug−in that someone is interested in the function it provides. In response, plug−ins typically do a combination of:
  - Registration − Hook various event mechanisms (e.g., register listeners) and otherwise make their presence known in the system (e.g., start needed threads).
  - Initialization − Initialize or prime their data structures and load models so they are ready for use.
- Plug−in global data/function − While never explicitly put forth for this role, in common practice plug−in classes have become a place to hang data and function which is effectively global to the plug−in itself. In some cases this data/function is API in others it is internal. For example, the UI plug−in exposes as API methods such as `getDialogSettings()` and `getWorkbench()`.
- Context − The standard `Plugin` class provides access to various runtime−provided function such as preferences and logging.

In the Eclipse 3.0 runtime picture, these roles and responsibilities are factored into distinct objects.

Bundle
: Bundles are the OSGi unit of modularity. There is one classloader per bundle and Eclipse−like inter−bundle class loading dependency graphs can be constructed. Bundles have lifecycle for start and stop and the OSGi framework broadcasts bundle related events (e.g., install, resolve, start, stop, uninstall, ...) to interested parties. Unlike the Eclipse `Plugin` class, the OSGi `Bundle` class is not extensible. That is, developers do not have the opportunity to define their own bundle class.

BundleActivator
: BundleActivator is an interface defined by the OSGi framework. Each bundle can define a bundle activator class much like a plug−in can define its `Plugin` class. The specified class is instantiated by the framework and used to implement the `start()` and `stop()` lifecycle processing. There is a major difference however in the nature of this lifecycle processing. In Eclipse it is common (though not recommended) to have the `Plugin` classes do both initialization and registration. In OSGi activators must only do registration. Doing large amounts of initialization (or any other work) in `BundleActivator.start()` threatens the liveness of the system.

BundleContext
: BundleContexts are the OSGi mechanism for exposing general system function to individual bundles. Each bundle has a unique and ***private*** instance of BundleContext which they can use to access system function (e.g., getBundles() to discover all bundles in the system).

Plugin
: The new `Plugin` is very much like the original Eclipse `Plugin` class with the following exceptions: `Plugin` objects are no longer required or managed by the runtime and various methods have been deprecated. It is essentially a convenience mechanism providing a host of useful function and mechanisms but is no longer absolutely required. Much of the function provided there is also available on the `Platform` class in the runtime.

`Plugin` also implements `BundleActivator`. This recognizes the convenience of having one central object representing the lifecycle and semantic of a plug−in. Note that this does not however sanction the eager initialization of data structures that is common in plug−ins today. We cannot stress enough that plug−ins can be activated because a somewhat peripheral class was referenced during verification of a class in some other plug−in. That is, just because your plug−in has been activated does not necessarily mean that its function is needed. Note also that you are free to define a different `BundleActivator` class or not have a bundle activator at all.

The steps required to port a 2.x `Plugin` class to Eclipse 3.0 depends on what the class is doing. As outlined above, most startup lifecycle work falls into one of the following categories:

Initialization

Datastructure and model initialization is quite often done in `Plugin.startup()`. The natural/obvious mapping would be to do this work in a `BundleActivator.start()`, that is to leave the function on `Plugin`. **This is strongly discouraged.** As with 2.x plug−ins, 3.0 plug−ins/bundles may be started for many different reasons in many different circumstances.
An actual example from Eclipse 2.0 days illuminates this case. There was a plug−in which initialized a large model requiring the loading of some 11MB of *code* and many megabytes of data. There were quite common usecases where this plug−in was activated to discover if the project icon presented in the navigator should be decorated with a particular markup. This test did not require any of the initialization done in `startup()` but yet all users, in all usecases had to pay the memory and time penalty for this eager initialization.
The alternative approach is to do such initialization in a classic lazy style. For example, rather than having models initialized when the plug−in/bundle is activated, do it when they are actually needed (e.g., in a centralized model accessor method). For many usecases this will amount to nearly the same point in time but for other scenarios this approach will defer initialization (perhaps indefinitely). We recommend taking time while porting 2.1 plug−ins to reconsider the initialization strategy used.

Registration

Plug−in startup is a convenient time to register listeners, services etc. and start background processing threads (e.g., listening on a socket). `Plugin.start()` may be a reasonable place to do this work. It may also make sense to defer until some other trigger (e.g., the use of a particular function or data element).

Plug−in global data

Your `Plugin` class can continue to play this role. The main issue is that `Plugin` objects are no longer globally accessible via a system−managed list. In Eclipse 2.x you could discover any plug−in's `Plugin` object via the plug−in registry. This is no longer possible. In most circumstances this type of access is not required. `Plugins` accessed via the registry are more typically used as generic `Plugins` rather than calling domain−specific methods. The equivalent level of capability can be had by accessing and manipulating the corresponding Bundle objects.

Registries and the plug−in model

In the new runtime there is a separation between the information and structures needed to execute a plug−in and that related to a plug−in's extensions and extension points. The former is defined and managed by the OSGi framework specification. The latter are Eclipse−specific concepts and are added by they Eclipse runtime code. Accordingly, the original plug−in registy and related objects have been split into OSGi *bundles* and the Eclipse *extension registry*.

The parts of `IPluginRegistry` dealing with execution specification (e.g., `IPluginDescriptor`, `ILibrary`, `IPrequisite`) have been deprecated and the remaining parts related to extensions and extension point have been moved to `IExtensionRegistry`. Further, the so−called model objects related

to the plug−in registry as a whole are now deprecated. These types were presented and instantiated by the runtime primarily to support tooling such as PDE. Unfortunately, it was frequently the case that the level of information needed exceeded the runtime's capabilities or interests (e.g., remembering line numbers for plugin.xml elements) and in the end, the potential consumers of the runtime's information had to maintain their own structures anyway.

In the new runtime we have re−evaluated the facilities provided by the runtime and now provide only those which are either essential for runtime execution or are extraordinarily difficult for others to do. As mentioned above, the plug−in registry model objects have been deprecated as has the plug−in parsing API. The new extensions registry maintains the essential extension−related information. A new *state* (see `org.eclipse.osgi.service.resolver.State` and friends) structure represents and allows the manipulation of the essential execution−related information.

NL fragment structure

In Eclipse 3.0 the NL fragment structure has been updated to be more consistent. Previously the translations for files like plugin.properties were assumed to be inside of JARs supplied by fragments. Since the original files are found in the root of the relevant host plug−in, a more consistent location would have the translated files located in the root of the NL fragments. For example,

```
org.eclipse.ui.workbench.nl/
   fragment.xml
   plugin_fr.properties
   plugin_pt_BR.properties
   ...
   nl1.jar
```

Note here that the file nl1.jar previously would have contained the translations for plugin.properties. These files are now at the root of the fragment and the JAR contains translations of any translatable resources (i.e., files loaded via the classloader) in the host plug−in.

Of course, the Eclipse 2.1 NL fragment structure is still supported for 2.1 host plug−ins running in Eclipse 3.0. You cannot however use a 2.1 NL fragment on a 3.0 plug−in. The fragment must be updated to the new structure.

API changes overview

org.eclipse.core.boot (package org.eclipse.core.boot)

The entire org.eclipse.core.boot package has been deprecated. BootLoader has been merged with `org.eclipse.core.runtime.Platform` since it no longer made sense to have a split between boot and runtime. Note that in fact, the org.eclipse.core.boot plug−in has been broken up and all its code moved to either the new runtime or the compatibility layer.

`IPlatformConfiguration` has always been a type defined by and for the Eclipse Install/Update component. With the reorganization of the runtime we are able to repatriate this type to its rightful home. This class remains largely unchanged and has been repackaged as `org.eclipse.update.configurator.IPlatformConfiguration`.

IPlatformRunnable has been moved to org.eclipse.core.runtime.IPlatformRunnable.

IExtension and IExtensionPoint (package org.eclipse.core.runtime)

The `getDeclaringPlugin()` method (on both classes) gives an upward link to the plug−in which declares the extension or extension−point (respectively). The new registry model separates the execution aspects of plug−ins from the extension/extension−point aspects and no longer contains `IPluginDescriptors`. Users of this API should consider the new method `getParentIdentifier()` found on both `IExtension` and `IExtensionPoint`.

ILibrary, IPluginDescriptor, IPluginRegistry and IPrerequisite (package org.eclipse.core.runtime)

In the original runtime, the plug−in registry maintained a complete picture of the runtime configuration. In Eclipse 3.0 this picture is split over the OSGi framework and the extension registry. As such, these classes have been deprecated. The deprecation notices contain details of how you should update your code.

Platform and Plugin (package org.eclipse.core.runtime)

In the new runtime, `Plugin` objects are no longer managed by the runtime and so cannot be accessed generically via the Platform. Similarly, the plug−in registry no longer exists or gives access to plug−in descriptors. There are however suitable replacement methods available and detailed in the Javadoc of the deprecated methods in these classes.

org.eclipse.core.runtime.model (package org.eclipse.core.runtime.model)

All types in this package are now deprecated. See the discussion on registries for more information.

IWorkspaceRunnable and IWorkspace.run (package org.eclipse.core.resources)

Clients of the `IWorkspace.run(IWorkspaceRunnable,IProgressMonitor)` method should revisit their uses of this method and consider using the richer method `IWorkspace.run(IWorkspaceRunnable,ISchedulingRule,int,IProgressMonitor)`. The old `IWorkspace.run` method acquires a lock on the entire workspace for the duration of the `IWorkspaceRunnable`. This means that an operation done with this method will never be able to run concurrently with other operations that are changing the workspace. In Eclipse 3.0, many long−running operations have been moved into background threads, so the likelihood of conflicts between operations is greatly increased. If a modal foreground operation is blocked by a long running background operation, the UI becomes blocked until the background operation completes, or until one of the operations is canceled.

The suggested solution is to switch all references to old `IWorkspace.run` to use the new method with a scheduling rule parameter. The scheduling rule should be the most fine−grained rule that encompasses the rules for all changes by that operation. If the operation tries to modify resources outside of the scope of the scheduling rule, a runtime exception will occur. The precise scheduling rule required by a given workspace operation is not specified, and may change depending on the installed repository provider on a given project. The factory `IResourceRuleFactory` should be used to obtain the scheduling rule for a resource−changing operation. If desired, a `MultiRule` can be used to specify multiple resource rules, and the `MultiRule.combine` convenience method can be used to combine rules from various resource−changing operations.

If no locking is required, a scheduling rule of `null` can be used. This will allow the runnable to modify all resources in the workspace, but will not prevent other threads from also modifying the workspace concurrently. For simple changes to the workspace this is often the easiest and most concurrency−friendly

solution.

IWorkbenchPage (package org.eclipse.ui)

- The constant EDITOR_ID_ATTR is now deprecated. This is an IMarker attribute name that specifies the preferred editor id to open the IMarker resource with. This constant in now on org.eclipse.ui.ide.IDE class.

IEditorDescriptor (package org.eclipse.ui)

- There are new API methods to determine whether the editor will open internally to the workbench page (isInternal), in−place to the workbench window (isOpenInPlace), or externally to the workbench (isExternal). While this is not a breaking change, it is a good opportunity for clients that are illegally down−casting IEditorDescriptor to org.eclipse.ui.internal.model.EditorDescriptor to call isInternal to bring there code back into line.

ISharedImages (package org.eclipse.ui)

- The following fields were removed (deprecated) from this interface because they were IDE−specific:
    - String IMG_OBJ_PROJECT
    - String IMG_OBJ_PROJECT_CLOSED
    - String IMG_OPEN_MARKER
    - String IMG_OBJS_TASK_TSK
    - String IMG_OBJS_BKMRK_TSK
- Existing clients should instead use the fields of the same names declared on IDE.SharedImages in the org.eclipse.ui.ide package of the org.eclipse.ui.ide plug−in.

IWorkbenchActionConstants (package org.eclipse.ui)

- The following fields were removed (deprecated) from this interface; they are subsumed by the new ActionFactory class:
    - String ABOUT
    - String BACK
    - String CLOSE
    - String CLOSE_ALL
    - String COPY
    - String CUT
    - String DELETE
    - String EXPORT
    - String FIND
    - String FORWARD
    - String IMPORT
    - String MOVE
    - String NEW
    - String NEXT
    - String PASTE
    - String PREVIOUS
    - String PRINT
    - String PROPERTIES
    - String QUIT
    - String REDO

- String REFRESH
- String RENAME
- String REVERT
- String SAVE
- String SAVE_ALL
- String SAVE_AS
- String SELECT_ALL
- String UNDO
- String UP

- Clients should instead call getID() on the fields of the same names declared on ActionFactory in the org.eclipse.ui.actions package (org.eclipse.ui plug−in). For example, change IWorkbenchActionConstants.CUT to ActionFactory.CUT.getId().
- The following fields were removed (deprecated) from this interface because they were IDE−specific.
  - String ADD_TASK
  - String BOOKMARK
  - String BUILD
  - String BUILD_PROJECT
  - String CLOSE_PROJECT
  - String FIND
  - String OPEN_PROJECT
  - String REBUILD_ALL
  - String REBUILD_PROJECT
- Clients should instead call getID() on the fields of the same names declared on IDEActionFactory in the org.eclipse.ui.ide package (org.eclipse.ui.ide plug−in). For example, change IWorkbenchActionConstants.BUILD to IDEActionFactory.BUILD.getId().

IWorkbenchPreferenceConstants (package org.eclipse.ui)

- The following fields were removed (deprecated) from this interface because they were IDE−specific:
  - String PROJECT_OPEN_NEW_PERSPECTIVE
- Clients should instead use the fields of the same names declared on IDE.Preferences in the org.eclipse.ui.ide package.

IExportWizard (package org.eclipse.ui)

- Prior to 3.0, the selection passed to IWorkbenchWizard.init(IWorkbench, IStructuredSelection) for an export wizard was preprocessed. If any of the selections were IResources, or adaptable to IResource, then the selection consisted only of these resources. In 3.0, the generic export wizard does not do any preprocessing.
- The selection passed to the wizard is generally used to prime the particular wizard page with contextually appropriate values.
- Client that implement IExportWizard and requires this resource−specific selection transformation should add the following to their init(IWorkbench, IStructuredSelection selection) method to computer `filteredSelection` from `selection`:
  - ```
    IStructuredSelection filteredSelection = selection;
    List selectedResources =
    IDE.computeSelectedResources(currentSelection);
    if (!selectedResources.isEmpty()) {
        filteredSelection = new
    StructuredSelection(selectedResources);
    }
    ```
    ```

IImportWizard (package org.eclipse.ui)

- Prior to 3.0, the selection passed to IWorkbenchWizard.init(IWorkbench, IStructuredSelection) for an import wizard was preprocessed. If any of the selections were IResources, or adaptable to IResource, then the selection consisted only of these resources. In 3.0, the generic import wizard does not do any preprocessing.
- The selection passed to the wizard is generally used to prime the particular wizard page with contextually appropriate values.
- Client that implement IImportWizard and requires this resource−specific selection transformation should add the following to their init(IWorkbench, IStructuredSelection selection) method to compute a filtered selection from the selection passed in:
    - ```
      IStructuredSelection filteredSelection = selection;
      List selectedResources =
      IDE.computeSelectedResources(currentSelection);
      if (!selectedResources.isEmpty()) {
          filteredSelection = new
      StructuredSelection(selectedResources);
      }
      ```

INewWizard (package org.eclipse.ui)

- Prior to 3.0, the selection passed to IWorkbenchWizard.init(IWorkbench, IStructuredSelection) for a new wizard was preprocessed. If there was no structured selection at the time the wizard was invoked, but the active workbench window had an active editor open on an IFile, then the selection passed in would consist of that IFile. In 3.0, the generic new wizard does not do any preprocessing, and an empty selection will be passed when there is no structured selection.
- The selection passed to the wizard is generally used to prime the particular wizard page with contextually appropriate values.
- Client that implement INewWizard and requires this capability should add the following to their init(IWorkbench, IStructuredSelection selection) method to compute a selection from the active editor's input:

```
if (selection.isEmpty()) {
   IWorkbenchWindow window = PlatformUI.getWorkbench().getActiveWorkbenchWindow();
   if (window != null) {
      IWorkbenchPart part = window.getPartService().getActivePart();
      if (part instanceof IEditorPart) {
         IEditorInput input = ((IEditorPart) part).getEditorInput();
         if (input instanceof IFileEditorInput) {
            selection = new StructuredSelection(((IFileEditorInput) input).getFile());
         }
      }
  }
}
```

WorkbenchHelp (package org.eclipse.ui.help)

- The following WorkbenchHelp method was removed (deprecated) from this class because its result (IHelp) was removed (deprecated):
    - public static IHelp getHelpSupport()
- Clients that called this method to obtain an IHelp should instead call the static methods on HelpSystem or WorkbenchHelp.
```

IHelp (package org.eclipse.help)

- This interface has been removed (deprecated). WorkbenchHelp.getHelpsupport() was the only way to get hold of an IHelp object. This method has also been removed (deprecated).
- The following IHelp methods now appear as static methods on a new HelpSystem class in the same package:
  - public IToc[] getTocs()
  - public IContext getContext(String contextId)
- The rest of the IHelp methods now appear as static method on WorkbenchHelp.
- This interface was formerly mentioned in the contract for the org.eclipse.help.support extension point. This extension point has been renamed "org.eclipse.ui.helpSupport", and the contract simplified so that the implementer only needs supply the display methods. For these purposes, IHelp has been replaced by AbstractHelpUI (in the org.eclipse.ui.help package).
- There should be no clients implementing this interface beyond the Platform which supplied the sole implementation of this interface.

ITextEditorActionConstants (package org.eclipse.ui.texteditor)

- This interface additionally includes newly defined constants that redefine deprecated constants inherited from org.eclipse.ui.IWorkbenchActionConstants. This change allows clients to free their code from deprecation warnings. The constants ADD_TASK and BOOKMARK have not been redefined as they are IDE specific. When using these two constants in your code, please follow the instructions given in the deprecation message.

IAbstractTextEditorHelpContextIds (package org.eclipse.ui.texteditor)

- BOOKMARK_ACTION and ADD_TASK_ACTION have been deprecated because they are IDE specific. Use the constants defined in org.eclipse.ui.editors.text.ITextEditorHelpContextIds instead.

BasicTextEditorActionContributor (package org.eclipse.ui.texteditor)

- BasicTextEditorActionContributor no longer assigns any editor action as global action for org.eclipse.ui.IWorkbenchActionConstants.ADD_TASK and org.eclipse.ui.IWorkbenchActionConstants.BOOKMARK because these actions are IDE specific. This is now done by the org.eclipse.ui.editors.text.TextEditorActionContributor. If your editors are not configured to used TextEditorActionContributor but uses a contributor that is a subclass of BasicTextEditorActionContributor, this contributor has to be extended to also assign global action handlers for ADD_TASK and BOOKMARK. This can be done by adding the following lines to the setActiveEditor method of the editor action contributor:
- ```
  IActionBars actionBars= getActionBars();
  if (actionBars != null) {
    actionBars.setGlobalActionHandler(IDEActionFactory.ADD_TASK.getId(), getAction(textEdit
    actionBars.setGlobalActionHandler(IDEActionFactory.BOOKMARK.getId(), getAction(textEdit
  }
  ```

TextEditorActionContributor (package org.eclipse.ui.editors.text)

- TextEditorActionContributor assigns global action handlers for IWorkbenchActionConstants.ADD_TASK (now IDEActionFactory.ADD_TASK.getId()) and IWorkbenchActionConstants.BOOKMARK (now IDEActionFactory.BOOKMARK.getId()). These action handlers have previously been registered by BasicTextEditorActionContributor. See migration notes for BasicTextEditorActionContributor.
  ```

annotationTypes extension point (plug−in org.eclipse.ui.editors)

There is now the explicit notion of an annotation type. See Annotation.getType() and Annotation.setType(). The type of an annotation can change over it's lifetime. A new extension point has been added for the declaration of annotation types: "org.eclipse.ui.editors.annotationTypes". An annotation type has a name and can be declared as being a subtype of another declared annotation type. An annotation type declaration may also use the attributes "markerType" and "markerSeverity" in order to specify that markers of a given type and a given severity should be represented in text editors as annotations of a particular annotation type. The attributes "markerType" and "markerSeverity" in the "org.eclipse.ui.editors.markerAnnotationSpecification" should no longer be used. Marker annotation specifications are thus becoming independent from markers and the name thus misleading. However, the name is kept in order to ensure backward compatibility.

Instances of subclasses of AbstractMarkerAnnotationModel automatically detect and set the correct annotation types for annotations they create from markers. In order to programmatically retrieve the annotation type for a given marker or a given pair of markerType and markerSeverity use org.eclipse.ui.texteditor.AnnotationTypeLookup.

Access to the hierarchy of annotation types is provided by IAnnotationAccessExtension. For a given annotation type you can get the chain of super types and check whether an annotation type is a subtype of another annotation type. DefaultMarkerAnnotationAccess implements this interface.

markerAnnotationSpecification extension point (plug−in org.eclipse.ui.editors)

The annotation type is the key with which to find the associated marker annotation specification. As annotation types can extend other annotation types, there is an implicit relation between marker annotation specifications as well. Therefore a marker annotation specification for a given annotation type is completed by the marker annotation specifications given for the super types of the given annotation type. Therefore, marker annotation specification do not have to be complete as this was required before. Marker annotation specifications are retrieved by AnnotationPreferences. By using org.eclipse.ui.texteditor.AnnotationPreferenceLookup, you can retrieve an annotation preference for a given annotation type that transparently performs the completion of the preference along the annotation super type chain.

Marker annotation specification has been extended with three additional attributes in order to allow the definition of custom appearances of a given annotation type in the vertical ruler. These attributes are: "icon", "symbolicIcon", and "annotationImageProvider". The value for "icon" is the path to a file containing the icon image. The value of "symbolicIcon" can be one of "error", "warning", "info", "task", "bookmark". The attribute "symbolicIcon" is used to tell the platform that annotation should be depicted with the same images that are used by the platform to present errors, warnings, infos, tasks, and bookmarks respectively. The value of "annotationImageProvider" is a class implementing org.eclipse.ui.texteditor.IAnnotationImageProvider that allows for a full custom annotation presentation.

The vertical ruler uses it's associated IAnnotationAccess/IAnnotationAccessExtension to draw annotations. The vertical ruler does not call Annotation.paint any longer. In general, Annotations are no longer supposed to draw themselves. The "paint" and "getLayer" methods have been deprecated in order to make annotation eventually UI independent. DefaultMarkerAnnotationAccess serves as default implementation of IAnnotationAccess/IAnnotationAccessExtension. DefaultMarkerAnnotationAccess implements the following strategy for painting annotations: If an annotation implements IAnnotationPresentation, IAnnotationPresentation.paint is called. If not, the annotation image provider is looked up in the annotation preference. The annotation image provider is only available if specified and if the plug−in defining the enclosing marker annotation specification has already been loaded. If there is an annotation image provider,

the call is forwarded to it. If not, the specified "icon" is looked up. "symbolicIcon" is used as the final fallback. For drawing annotations, the annotation presentation layer is relevant. DefaultMarkerAnnotationAccess looks up the presentation layer using the following strategy: If the annotation preference specifies a presentation layer, the specified layer is used. If there is no layer and the annotation implements IAnnotationPresentation, IAnnotationPresentation.getLayer is used otherwise the default presentation layer (which is 0) is returned.

Migration to annotationTypes extension point (plug−in org.eclipse.ui.editors)

The following annotation types are declared by the org.eclipse.ui.editors plug−in:

```
<extension point="org.eclipse.ui.editors.annotationTypes">
   <type
      name="org.eclipse.ui.workbench.texteditor.error"
      markerType="org.eclipse.core.resources.problemmarker"
      markerSeverity="2">
   </type>
   <type
      name="org.eclipse.ui.workbench.texteditor.warning"
      markerType="org.eclipse.core.resources.problemmarker"
      markerSeverity="1">
   </type>
   <type
      name="org.eclipse.ui.workbench.texteditor.info"
      markerType="org.eclipse.core.resources.problemmarker"
      markerSeverity="0">
   </type>
   <type
      name="org.eclipse.ui.workbench.texteditor.task"
      markerType="org.eclipse.core.resources.taskmarker">
   </type>
   <type
      name="org.eclipse.ui.workbench.texteditor.bookmark"
      markerType="org.eclipse.core.resources.bookmark">
   </type>
</extension>
```

The defined markerAnnotationSpecification extension no longer provide "markerType" and "markerSeverity" attributes. They define the "symbolicIcon" attribute with the according value. Thus, MarkerAnnotation.paint and MarkerAnnotation.getLayer are not called any longer, i.e. overriding these methods does not have any effect. Affected clients should implement IAnnotationPresentation.

ILaunchConfigurationType (package org.eclipse.debug.core)

With the introduction of extensible launch modes in 3.0, more than one launch delegate can exist for a launch configuration type. Releases prior to 3.0 only supported one launch delegate per launch configuration type. The method `ILaunchConfigurationType.getDelegate()` is now deprecated. The method `getDelegate(String mode)` should be used in its place to retrieve the launch delegate for a specific launch mode. The deprecated method has been changed to return the launch delegate for the `run` mode.

ILaunchConfigurationTab and ILaunchConfigurationTabGroup (package org.eclipse.debug.ui)

Launch tab groups and launch tabs are no longer notified when a launch completes. The method `launched(ILaunch)` in the interfaces `ILaunchConfigurationTab` and

`ILaunchConfigurationTabGroup` has been deprecated and is no longer called. Relying on this method for launch function was always problematic, since tabs only exist when launching is performed from the launch dialog. Also, with the introduction of background launching, this method can no longer be called, as the launch dialog is be closed before the resulting launch object exists.

ILaunchConfigurationTab and AbstractLaunchConfigurationTab (package org.eclipse.debug.ui)

Two methods have been added to the `ILaunchConfigurationTab` interface – activated and deactivated. These new life cycle methods are called when a tab is entered and exited respectively. Existing implementations of `ILaunchConfigurationTab` that subclass the abstract class provided by the debug plug–in (`AbstractLaunchConfigurationTab`) are binary compatible since the methods are implemented in the abstract class.

In prior releases, a tab was sent the message `initializeFrom` when it was activated, and `performApply` when it was deactivated. In this way, the launch configuration tab framework provided inter–tab communication via a launch configuration (by updating the configuration with current attribute values when a tab is exited, and updating the newly entered tab). However, since many tabs do not perform inter–tab communication, this can be inefficient. As well, there was no way to distinguish between a tab being activated, and a tab displaying a selected launch configuration for the first time. The newly added methods allow tabs to distinguish between activation and initialization, and deactivation and saving current values.

The default implementation of `activated`, provided by the abstract tab, calls `initializeFrom`. And, the default implementation of `deactivated` calls `performApply`. Tabs wishing to take advantage of the new API should override these methods as required. Generally, for tabs that do not perform inter–tab communication, the recommended approach is to re–implement these methods to do nothing.

launchConfigurationTabGroup extension point Type (package org.eclipse.debug.ui)

In prior releases, perspective switching was specified on a launch configuration, via the launch configuration attributes `ATTR_TARGET_DEBUG_PERSPECTIVE` and `ATTR_TARGET_RUN_PERSPECTIVE`. With the addition of extensible launch modes in 3.0, this approach no longer scales. Perspective switching is now specified on launch configuration type basis, per launch mode that a launch configuration type supports. API has been added to `DebugUITools` to set and get the perspective associated with a launch configuration type for a specific launch mode.

An additional, optional, `launchMode` element has been added to the `launchConfigurationTabGroup` extension point, allowing a contributed tab group to specify a default perspective for a launch configuration type and mode.

From the Eclipse user interface, users can edit the perspective associated with a launch configuration type by opening the launch configuration dialog, and selecting a launch configuration type node in the tree (rather than an individual configuration). A tab is displayed allowing the user to set a perspective with each supported launch mode.

[JDT only] IVMRunner (package org.eclipse.jdt.launching)

Two methods have been added to the `VMRunnerConfiguration` class to support the setting and retrieving of environment variables. Implementors of `IVMRunner` should call `VMRunnerConfiguration.getEnvironment()` and pass that environment into the executed JVM. Clients who use `DebugPlugin.exec(String[] cmdLine, File workingDirectory)` can do

this by calling `DebugPlugin.exec(String[] cmdLine, File workingDirectory, String[] envp)` instead. Simply passing in the result from `getEnvironment()` is sufficient.

[JDT only] VMRunnerConfiguration and Bootstrap Classes (package org.eclipse.jdt.launching)

In prior releases, the `VMRunnerConfiguration` had one attribute to describe a boot path. The attribute is a collection of `Strings` to be specified in the `-Xbootclasspath` argument. Three new attributes have been added to the VMRunnerConfiguration to support JVMs that allow for prepending and appending to the boot path. The new methods/attributes added are:

- `getPrependBootClassPath()` – returns a collection of entries to be prepended to the boot path (the `-Xbootclasspath/p` argument)
- `getMainBootClassPath()` – returns a collection of entries to be placed on the boot path (the `-Xbootclasspath` argument)
- `getAppendBootClassPath()` – returns a collection of entries to be appended to the boot path (the `-Xbootclasspath/a` argument)

The old attribute, `getBootClassPath()`, still exists and contains a complete path equivalent to that of the three new attributes. However, `VMRunners` that support the new boot path options should take advantage of the new attributes.

[JDT only] Improved support for working copies (package org.eclipse.jdt.core)

The Java model working copy facility has been reworked in 3.0 to provide greatly increased functionality. Prior to 3.0, the Java model allowed creation of individual working copies of compilation units. Changes could be made to the working copy and later committed. There was support for limited analysis of a working copy in the context of the rest of the Java model. However, there was no way these these analyses could ever take into account more than one of the working copies at a time.

The changes in 3.0 make it possible to create and manage sets of working copies of compilation units, and to perform analyses in the presence of all working copies in a set. For example, it is now possible for a client like JDT refactoring to create working copies for one or more compilation units that it is considering modifying and then to resolve type references between the working copies. Formerly this was only possible after the changes to the compilation unit working copies had been committed.

The Java model API changes in 2 ways to add this improved support:

(1) The functionality formerly found on `IWorkingCopy` and inherited by `ICompilationUnit` has been consolidated into `ICompilationUnit`. The `IWorkingCopy` interface was only used in this one place, and was gratuitously more general that in needed to be. This change simplifies the API. `IWorkingCopy` has been deprecated. Other places in the API where `IWorkingCopy` is used as a parameter or result type have been deprecated as well; the replacement API methods mention `ICompilationUnit` instead of `IWorkingCopy`.

(2) The interface `IBufferFactory` has been replaced by `WorkingCopyOwner`. The improved support for working copies requires that there be an object to own the working copies. Although `IBufferFactory` is in the right place, the name does not adequately convey how the new working copy mechanism works. `WorkingCopyOwner` is much more suggestive. In addition, `WorkingCopyOwner` is declared as an abstract class, rather than as an interface, to allow the notion of working copy owner to evolve in the future. The one method on `IBufferFactory` moves to `WorkingCopyOwner` unaffected.

`WorkingCopyOwner` does not implement `IBufferFactory` to make it clear that `IBufferFactory` is a thing of the past. `IBufferFactory` has been deprecated. Other places in the API where `IBufferFactory` appears as a parameter or result type have been deprecated as well; the replacement API methods mention `WorkingCopyOwner` instead of `IBufferFactory`.

These changes do not break binary compatibility.

When migrating, all references to the type `IWorkingCopy` should instead reference `ICompilationUnit`. The sole implementation of `IWorkingCopy` implements `ICompilationUnit` as well, meaning objects of type `IWorkingCopy` can be safely cast to `ICompilationUnit`.

A class that implements `IBufferFactory` will need to replaced by a subclass of `WorkingCopyOwner`. Although `WorkingCopyOwner` does not implement `IBufferFactory` itself, it would be possible to declare the subclass of `WorkingCopyOwner` that implements `IBufferFactory` thereby creating a bridge between old and new (`IBufferFactory` declares `createBuffer(IOpenable)` whereas `WorkingCopyOwner` declares `createBuffer(ICompilationUnit)`; `ICompilationUnit` extends `IOpenable`).

Because the changes involving `IWorkingCopy` and `IBufferFactory` are interwined, we recommend dealing with both at the same time. The details of the deprecations are as follows:

- `IWorkingCopy` (package `org.eclipse.jdt.core`)
  - `public void commit(boolean, IProgressMonitor)` has been deprecated.
    - The equivalent functionality is now provided on `ICompilationUnit` directly:
      - `public void commitWorkingCopy(boolean, IProgressMonitor)`
    - Rewrite `wc.commit(b,monitor)` as `((ICompilationUnit) wc).commitWorkingCopy(b,monitor)`
  - `public void destroy()` has been deprecated.
    - The equivalent functionality is now provided on `ICompilationUnit` directly:
      - `public void discardWorkingCopy(boolean, IProgressMonitor)`
    - Rewrite `wc.destroy()` as `((ICompilationUnit) wc).discardWorkingCopy()`
  - `public IJavaElement findSharedWorkingCopy(IBufferFactory)` has been deprecated.
    - The equivalent functionality is now provided on `ICompilationUnit` directly:
      - `public ICompilationUnit findWorkingCopy(WorkingCopyOwner)`
    - Note: `WorkingCopyOwner` substitutes for `IBufferFactory`.
  - `public IJavaElement getOriginal(IJavaElement)` has been deprecated.
    - The equivalent functionality is now provided on `IJavaElement`:
      - `public IJavaElement getPrimaryElement()`
    - Rewrite `wc.getOriginal(elt)` as `elt.getPrimaryElement()`
    - Note: Unlike `IWorkingCopy.getOriginal`, `IJavaElement.getPrimaryElement` does not return `null` if the receiver is not a working copy.
  - `public IJavaElement getOriginalElement()` has been deprecated.
    - The equivalent functionality is now provided on `ICompilationUnit` directly:
      - `public ICompilationUnit getPrimary()`

- ◊ Rewrite `wc.getOriginalElement()` as `((ICompilationUnit) wc).getPrimary()`
- ◊ Note: Unlike `IWorkingCopy.getOriginalElement`, `IWorkingCopy.getPrimary` does not return `null` if the receiver is not a working copy.

♦ `public IJavaElement[] findElements(IJavaElement)` has been deprecated.
- ◊ The method is now declared on `ICompilationUnit` directly.
- ◊ Rewrite `wc.findElements(elts)` as `((ICompilationUnit) wc).findElements(elts)`

♦ `public IType findPrimaryType()` has been deprecated.
- ◊ The method is now declared on `ICompilationUnit` directly.
- ◊ Rewrite `wc.findPrimaryType()` as `((ICompilationUnit) wc).findPrimaryType()`

♦ `public IJavaElement getSharedWorkingCopy(IProgressMonitor, IBufferFactory, IProblemRequestor)` has been deprecated.
- ◊ The equivalent functionality is now provided on `ICompilationUnit` directly:
  - `public ICompilationUnit getWorkingCopy(WorkingCopyOwner, IProblemRequestor, IProgressMonitor)`
- ◊ Note: the parameter order has changed, and `WorkingCopyOwner` substitutes for `IBufferFactory`.

♦ `public IJavaElement getWorkingCopy()` has been deprecated.
- ◊ The equivalent functionality is now provided on `ICompilationUnit` directly:
  - `public ICompilationUnit getWorkingCopy(IProgressMonitor)`
- ◊ Rewrite `wc.getWorkingCopy()` as `((ICompilationUnit) wc).getWorkingCopy(null)`

♦ `public IJavaElement getWorkingCopy(IProgressMonitor, IBufferFactory, IProblemRequestor)` has been deprecated.
- ◊ The equivalent functionality is now provided on `ICompilationUnit` directly:
  - `public ICompilationUnit getWorkingCopy(WorkingCopyOwner, IProblemRequestor, IProgressMonitor)`
- ◊ Note: the parameter order has changed, and `WorkingCopyOwner` substitutes for `IBufferFactory`.

♦ `public boolean isBasedOn(IResource)` has been deprecated.
- ◊ The equivalent functionality is now provided on `ICompilationUnit` directly:
  - `public boolean hasResourceChanged()`
- ◊ Rewrite `wc.isBasesOn(res)` as `((ICompilationUnit) wc).hasResourceChanged()`

♦ `public boolean isWorkingCopy()` has been deprecated.
- ◊ The method is now declared on `ICompilationUnit` directly.
- ◊ Rewrite `wc.isWorkingCopy()` as `((ICompilationUnit) wc).isWorkingCopy()`

♦ `public IMarker[]  reconcile()` has been deprecated.
- ◊ The equivalent functionality is now provided on `ICompilationUnit` directly:
  - `public void reconcile(boolean,IProgressMonitor)`
- ◊ Rewrite `wc.reconcile()` as `((ICompilationUnit) wc).reconcile(false, null)`
- ◊ Note: The former method always returned `null`; the replacement method does not return a result.

- ♦ `public void reconcile(boolean, IProgressMonitor)` has been deprecated.
  - ◊ The method is now declared on `ICompilationUnit` directly.
  - ◊ Rewrite `wc.reconcile(b,monitor)` as `((ICompilationUnit) wc).reconcile(b.monitor)`
- ♦ `public void restore()` has been deprecated.
  - ◊ The method is now declared on `ICompilationUnit` directly.
  - ◊ Rewrite `wc.restore()` as `((ICompilationUnit) wc).restore()`

• `IType` (package `org.eclipse.jdt.core`)

- ♦ `public ITypeHierarchy newSupertypeHierarchy(IWorkingCopy[], IProgressMonitor)` has been deprecated.
  - ◊ The replacement method is provided on the same class:
    - · `public ITypeHierarchy newSupertypeHierarchy(c, IProgressMonitor)`
  - ◊ Note: The Java language rules for array types preclude casting `IWorkingCopy[]` to `ICompilationUnit[]`.
- ♦ `public ITypeHierarchy newTypeHierarchy(IWorkingCopy[], IProgressMonitor)` has been deprecated.
  - ◊ The replacement method is provided on the same class:
    - · `public ITypeHierarchy newTypeHierarchy(ICompilationUnit[], IProgressMonitor)`
  - ◊ Note: The Java language rules for array types preclude casting `IWorkingCopy[]` to `ICompilationUnit[]`.

• `IClassFile` (package `org.eclipse.jdt.core`)

- ♦ `public IJavaElement getWorkingCopy(IProgressMonitor, IBufferFactory)` has been deprecated.
  - ◊ The replacement method is provided on the same class:
    - · `public ICompilationUnit getWorkingCopy(WorkingCopyOwner, IProgressMonitor)`
  - ◊ Note: the parameter order has changed, and `WorkingCopyOwner` substitutes for `IBufferFactory`.

• `JavaCore` (package `org.eclipse.jdt.core`)

- ♦ `public IWorkingCopy[] getSharedWorkingCopies(IBufferFactory)` has been deprecated.
  - ◊ The replacement method is provided on the same class:
    - · `public ICompilationUnit[] getWorkingCopies(WorkingCopyOwner)`
  - ◊ Note: `WorkingCopyOwner` substitutes for `IBufferFactory`.
  - ◊ Note: The Java language rules for array types preclude casting `ICompilationUnit[]` to `IWorkingCopy[]`.

• `SearchEngine` (package `org.eclipse.jdt.core.search`)

- ♦ `public SearchEngine(IWorkingCopy[])` has been deprecated.
  - ◊ The replacement constructor is provided on the same class:
    - · `public SearchEngine(ICompilationUnit[])`
  - ◊ Note: The Java language rules for array types preclude casting `IWorkingCopy[]` to `ICompilationUnit[]`.

Restructuring of org.eclipse.help plug−in

The org.eclipse.help plug−in, which used to hold APIs and extension points for contributing to and extending help system, as well as displaying help, now contains just APIs and extension points for contributing and

accessing help resources. A portion of default help UI implementation contained in that plug−in has been moved to a new plug−in org.eclipse.help.base together with APIs for extending the implementation. The APIs and extension point for contributing Help UI and displaying help have been moved to org.eclipse.ui plug−in. This restructuring allows applications greater flexibility with regard to the help system; the new structure allows applications based on the generic workbench to provide their own Help UI and/or Help implementation, or to omit the help system entirely.

Because the extension points and API packages affected are intended only for use by the help system itself, it is unlikely that existing plug−ins are affected by this change. They are included here only for the sake of completeness:

- API packages org.eclipse.ui.help.browser and org.eclipse.ui.help.standalone, formerly provided by the org.eclipse.help plug−in, have been moved to the org.eclipse.help.base plug−in.
- Extension points org.eclipse.help.browser, org.eclipse.help.luceneAnalyzer, and org.eclipse.help.webapp, formerly defined by the org.eclipse.help plug−in, have been moved to the org.eclipse.help.base plug−in, with a corresponding change in extension point id.
- Extension point org.eclipse.help.support has been replaced by the org.eclipse.ui.helpSupport extension point. The contract for this extension point changed as well (see entry for IHelp).

New Search UI API

A new API for implmenting custom searches has been added in 3.0. The original API is deprecated in 3.0 and we recommend that cllients port to the new API in the packages org.eclipse.search.ui and org.eclipse.search.ui.text.

Clients will have to create implementations of `ISearchQuery`, `ISearchResult` and `ISearchResultPage`. The `ISearchResultPage` implementation must then be contributed into the new `org.eclipse.search.searchResultViewPages` extension point.

Default implementations for `ISearchResult` and `ISearchResultPage` are provided in the package `org.eclipse.search.ui.text`.

null messages in MessageBox and DirectoryDialog (package org.eclipse.swt.widgets)

Prior to 3.0, calling SWT's DirectoryDialog.setMessage(String string) or MessageBox.setMessage(String string) with a null value for string would result in a dialog with no text in the title. This behavior was unspecified (passing null has never been permitted) and creates problems with getMessage which is not permitted to return null. In 3.0, passing null now results in an IllegalArgumentException exception being thrown, and the specifications have been changed to state this, bringing it into line with the method on their superclass Dialog.setMessage. If you use Dialog.setMessage, ensure that that the string passed in is never null. Simply pass an empty string if you want a dialog with no text in the title.

Improving modal progress feedback

Supporting concurrent operations requires more sophisticated ways to show modal progress. As part of the responsiveness effort additional progress support was implemented in the class IProgressService. The existing way to show progress with the ProgressMonitorDialog is still working. However, to improve the user experience we recommend migrating to the new IProgressService.

The document Showing Modal Progress in Eclipse 3.0 describes how to migrate to the new IProgressService.

Debug Action Groups removed

The Debug Action Groups extension point (org.eclipse.debug.ui.debugActionGroups) has been removed. In Eclipse 3.0, the workbench introduced support for Activities via the org.eclipse.platform.ui.activities extension point. This support provides everything that Debug Action Groups provided and is also easier to use (it supports patterns instead of specifying all actions exhaustively) and has a programmatic API to support it. Failing to remove references to the old extension point won't cause any failures. References to the extension point will simply be ignored. Product vendors are encouraged to use the workbench Activities support to associate language−specific debugger actions with language−specific activities (for example, C++ debugging actions might be associated with an activity called "Developing C++").

BreakpointManager can be disabled

IBreakpointManager now defines the methods setEnabled(boolean) and isEnabled(). When the breakpoint manager is disabled, debuggers should ignore all registered breakpoints. The debug platform also provides a new listener mechanism, IBreakpointManagerListener which allows clients to register with the breakpoint manager to be notified when its enablement changes. The Breakpoints view calls this API from a new toggle action that allows the user to "Skip All Breakpoints." Debuggers which do not honor the breakpoint manager's enablement will thus appear somewhat broken if the user tries to use this feature.

[JDT only] Java search participants (package org.eclipse.jdt.core.search)

Languages close to Java (such as JSP, SQLJ, JWS, etc.) should be able to participate in Java searching. In particular, implementors of such languages should be able to:

- index their source by converting it into Java equivalent source, and feeding it to the Java indexer
- index their source by parsing it themselves, but record Java index entries
- locate matches in their source by converting it into Java equivalent source, and feeding it to the Java match locator
- locate matches in their source by matching themselves, and return Java matches

Such an implementor is called a search participant. It extends the SearchParticipant class. Search participants are passed to search queries (see SearchEngine.search(SearchPattern, SearchParticipant[], IJavaSearchScope, SearchRequestor, IProgressMonitor)).

For either indexing or locating matches, a search participant needs to define a subclass of SearchDocument that can retrieve the contents of the document by overriding either getByteContents() or getCharContents(). An instance of this subclass is returned in getDocument(String).

A search participant wishing to index some document will use SearchParticipant.scheduleDocumentIndexing(SearchDocument, IPath) to schedule the indexing of the given document in the given index. Once the document is ready to be indexed, the underlying framework calls SearchParticipant.indexDocument(SearchDocument, IPath). The search participant then gets the document's content, parses it and adds index entries using SearchDocument.addIndexEntry(char[], char[]).

Once indexing is done, one can then query the indexes and locate matches using SearchEngine.search(SearchPattern, SearchParticipant[], IJavaSearchScope, SearchRequestor, IProgressMonitor). This first asks each search participant for the indexes needed by this query using SearchParticipant.selectIndexes(SearchPattern, IJavaSearchScope). For each index entry that matches the given pattern, a search document is created by asking the search participant (see getDocument(String)). All these documents are passed to the search participant so that it can locate matches using

locateMatches(SearchDocument[], SearchPattern, IJavaSearchScope, SearchRequestor, IProgressMonitor). The search participant notifies the SearchRequestor of search matches using acceptSearchMatch(SearchMatch) and passing an instance of a subclass of SearchMatch.

A search participant can delegate part of its work to the default Java search participant. An instance of this default participant is obtained using SearchEngine.getDefaultSearchParticipant(). For example when asked to locate matches, an SQLJ participant can create documents .java documents from its .sqlj documents and delegate the work to the default participant passing it the .java documents.

SWT Example Launcher

The Example Launcher is used to launch SWT examples, which can either be Workbench views or standalone applications.

- Workbench views are examples that are integrated into Eclipse. When the launcher starts a Workbench view, it is opened in the currently active perspective.
- Standalone applications are launched in a separate window.

For information on how to run the standalone examples without the SWT Example Launcher, refer to SWT standalone examples setup.

The SWT Workbench view examples can also be launched directly without using the SWT Example Launcher. SWT Workbench view examples can be found under the ***SWT Examples*** category of the ***Show Views*** dialog.

Running the Example Launcher

From Eclipse's ***Window*** menu, select ***Show View > Other***. In the ***Show View*** dialog, expand ***SWT Examples*** and select the ***SWT Example Launcher*** view. A view containing a list of examples will appear in your current perspective. When you select an example from the list a brief description of the example is displayed. Click on the ***Run*** button to launch the example.

SWT example – Browser

The Browser Example is a simple demonstration of the SWT Browser widget. It consists of a composite containing a Browser widget to render HTML and some additional widgets to implement actions commonly found on browsers (toolbar with back, forward, refresh and stop buttons, status bar etc.).

Running the example

Follow the SWT standalone examples setup instructions to install and run the example from your workspace.

The "Main" class is `org.eclipse.swt.examples.browserexample.BrowserExample`.

This example can also be run using the Example Launcher. Select the ***Browser*** item from the ***Workbench Views*** category and click ***Run***.

SWT example – Controls

The Controls Example is a simple demonstration of common SWT controls. It consists of a tab folder where each tab in the folder allows the user to interact with a different control. The user can change styles and settings and view how this affects each control.

Running the example

Follow the SWT standalone examples setup instructions to install and run the example from your workspace.

The "Main" class is `org.eclipse.swt.examples.controlexample.ControlExample`.

This example can also be run using the Example Launcher. Select the ***Controls*** item from the ***Workbench Views*** category and click ***Run***.

SWT example – Custom Controls

The Custom Controls example is a simple demonstration of emulated SWT controls. It consists of a tab folder where each tab in the folder allows the user to interact with a different emulated control. The user can change styles and settings and view how this affects each control.

Running the example

Follow the SWT standalone examples setup instructions to install and run the example from your workspace.

The "Main" class is
`org.eclipse.swt.examples.controlexample.CustomControlExample`.

This example can also be run using the Example Launcher. Select the ***Custom Controls*** item from the ***Workbench Views*** category and click ***Run***.

SWT example – Layouts

This example is a simple demonstration of common SWT layouts. It consists of a tab folder where each tab in the folder allows the user to interact with a different SWT layout. The user can insert widgets into a layout and set the values of the layout data using a property sheet. When the user has a suitable arrangement, the underlying code can be generated by clicking on the ***Code*** button.

Running the example

Follow the SWT standalone examples setup instructions to install and run the example from your workspace.

The "Main" class is `org.eclipse.swt.examples.layoutexample.LayoutExample`.

This example can also be run using the Example Launcher. Select the ***Layouts*** item from the ***Workbench Views*** category and click ***Run***.

SWT example – OLE Web Browser

This example shows how to embed an Active X control into an SWT application or an Eclipse view.

When the view is opened, it will create an instance of the Windows Internet Explorer control. The OLE web browser has ***Back*** and ***Forward*** buttons to take you to recently visited pages. The browser also contains a ***Home*** button to take you to the web browser's home page, a ***Stop*** button which stops the current transfer, a ***Search*** button which will search for text typed in the ***Address*** text field, and a ***Refresh*** button which re−draws the contents of the currently displayed page. There is also a ***Go*** button which, when clicked, will attempt to load the page specified in the ***Address*** field.

Running the example

From Eclipse's ***Window*** menu select ***Show View > Other***. In the ***Show View*** dialog, expand ***SWT Examples*** and select the view named ***OLE Web Browser (win32)***. A view containing a web browser will appear.

This example can also be run using the Example Launcher. Select the ***OLE Web Browser*** item from the ***Win32 only*** category and click ***Run***.

SWT example – Paint Tool

This example demonstrates the use of SWT graphics operations in the form of a rudimentary bitmap painting program. The Paint Tool implementation also demonstrates a mechanism for managing timed GUI operations in the background that are triggered by user input.

Select a tool with which to draw in the drawing area. There are a number of tools to choose from on the toolbar. To change the color selection, click on a color in the palette below the drawing area; left−click to set the foreground color, right−click to set the background color.

Running the example

From Eclipse's ***Window*** menu select ***Show View > Other***. In the ***Show View*** dialog, expand ***SWT Examples*** and select the view named ***Paint***. A view containing the paint program will appear.

This example can also be run using the Example Launcher. Select the ***Paint*** item from the ***Workbench Views*** category and click ***Run***.

Example – Java Editor

Introduction

The Java Editor example demonstrates the standard features available for custom text editors. It also shows how to register an editor for a file extension (in this case .jav) and how to define a custom Document provider for use by that editor. This example is only for demonstration purposes. Java editing support is provided by the Eclipse Java Tooling.

The code for this example is in the `org.eclipse.ui.examples.javaeditor` plug−in. To explore the code it is recommended to import that plug−in into your workspace.

Features demonstrated in the example editor

- syntax highlighting
- content type sensitive content assist (Javadoc and Java code)
- communication between content outliner and editor, i.e. content outliner selection feeds the highlight range of the editor
- two different presentation modes
    - ♦ marking the highlight range using a visual range indicator
    - ♦ confining the editor to show only text within the highlight range (e.g. show a single method of a Java class)
- marker handling
- document sharing

Features not demonstrated

- content formatting
- dynamic reconciling of content outline page

Running the example Java editor

1. Create a project
2. Create a file with the file extension ".jav" in the newly created project. The Java example editor opens automatically.
3. Insert Java code. The Java code is dynamically colored. The example editor presents the following language elements in different colors: multi−line comments, single line comments, Java language reserved words, string and character constants, regular Java code, as well as multi−line comments following the Javadoc guidelines. Inside those Javadoc comments, Javadoc keywords (green) and HTML tags (gray) are differently colored.
4. Open a new Java multi−line comment by inserting "/*" outside a Java comment. All the text between the inserted "/*" and the first occurrence of "*/" or the end of the text changes its color to red. Append another "*". The red range changes color to black as the regular multi−line comment now is considered containing Javadoc. Invoke code assist using CTRL−SPACE. The function of content assist is to support the user in writing code. So on invocation, content assist should list all possible valid completions at the invocation location. Inside Javadoc, the example editor always proposes all Javadoc keywords.
5. Outside a Java comment invoke content tip using CTRL+SHIFT+SPACE. Five proposals are listed. Select one and press ENTER. A small floating red window appears above the current line displaying the selected proposal. Content tips are used to let the user express her intention (e.g. entering a method call), and then to present contextual information which guides the user in doing so. In the example editor, the proposal is considered valid five characters around the initial invocation location. While the content tip is visible, invoke content assist using CTRL+SPACE. Content assist invoked in this situation should help the user to accomplish her stated intention, which is still visible in the content tip. Inside regular Java code, the example editor always proposes all Java keywords.
6. Save the Java code. Saving updates the content outliner. The content outliner contains ten entries each of them representing one of ten equally sized segments of the Java code in the editor. This style of content outline has been chosen to show that the semantics of highlight ranges can arbitrarily be defined. (See next steps.)
7. Select one of the entries in the content outliner. The corresponding lines are marked with a blue bar in the editor's left vertical ruler.

8. Now switch to the segmented presentation mode of the Java editor. For that make sure that the editor has the focus and press that button in the desktop's toolbar whose hover help says "Enable/Disable segmented source viewer". This functionality is used for single method views and similar functionality.
9. Select a different entry in the content outliner. Now the editor only shows the selected segment. By deselecting the entry in the content outliner, the complete Java code is shown again.
10. Select an entry in the content outliner, select a fraction of the visible text, and add a task for the selection. The task shows up in the task list. Modify the visible code. In the task list, select the previously created task and press the "Go to file" button. The task is selected in the visible area, correctly taking the previously applied modifications into account.
11. Select another entry in the content outliner. Reveal the previously added task from the task list. The editor's highlight range is automatically enlarged to enclose the range of the revealed task.
12. Open a new workspace. In the new workspace, open a Java editor for the same file as in the original workspace. Modify the editor content. Switch back to the original workspace. The editor shows the changes made in the other workspace. The two editors showing the same file are linked.

Principles for creating custom text editors

The following steps are usually necessary do develop a custom text editor.

1. Create a document provider. A document provider (see IDocumentProvider) produces and manages documents (see IDocument) containing a textual representation of editor input elements. It is important to decide how the translation between element and textual representation should be performed and whether the document provider should be shared between multiple editors or not. See the class FileDocumentProvider in the Java example editor.
2. Create a document partitioner. A document partitioner (see IDocumentPartitioner) divides a document into disjoint regions. The partitioner assigns each region one content type out of a set of content types predefined by the partitioner. On each document change the document's partitioning must be updated. See the class JavaPartitioner in the Java example editor. The JavaPartitioner determines regions of types "multi−line comments", "Javadoc comments", and "everything else". It must be ensured that the document provider is set on each document produced by the document provider.
3. Determine which of the source viewer plug−ins should be provided. Among other supported plug−ins are auto indent strategies, double click strategies, content formatter, and text presentation reconciler. The subsequent description will be restricted to the text presentation reconciler (see IPresentationReconciler). In the Java example editor, the text presentation reconciler is utilized to implement syntax highlighting.
4. Create for all source viewer plug−ins the appropriate extensions for each supported content type. As seen above, the document partitioner defines the supported content types. The default implementation of IPresentationReconciler supports IPresentationDamagers and IPresentationRepairers as extensions. Those extensions are considered to be specific for a particular content type. Thus, for a custom editor, the user must first select a subset of the supported content types. Regions of a type in the selected subset will, e.g., be syntax highlighted. For each of those types the extensions must be implemented. See JavaDamagerRepairer and JavaDocDamagerRepairer in the example editor.
5. Build a source viewer configuration using the previously created plug−ins and extensions. See JavaSourceViewerConfiguration in the example editor.
6. Customize the class TextEditor or AbstractTextEditor with the developed document partitioner and source viewer configuration. Add or replace actions and adapt the construction of the editor's context menu. In the actual version, this customization must be done in a subclass. See JavaEditor in the example editor.
7. Set up an appropriate action bar contributor who contributes editor−related actions to the desktop's

toolbar and menus. See JavaActionContributor in the example editor.
8. Extend the XML configuration file of the editor's plug−in, so that the editor registers at the predefined editor extension point for a specific set of file extensions. Also set up the action bar contributor in the XML file. See plugin.xml of this example.

Code organization of the example

The Java editor example code is organized in four packages:

- `org.eclipse.ui.examples.javaeditor` contains all the editor specific classes.
- `org.eclipse.ui.examples.javaeditor.java` contains all Java specific source viewer plug−ins such as the JavaDamagerRepairer as well as the Java specific document partitioner.
- `org.eclipse.ui.examples.javaeditor.javadoc` contains all Javadoc specific source viewer plug−ins such as the JavaDocDamagerRepairer.

`org.eclipse.ui.examples.javaeditor.util` contains convenience classes shared by the three other packages.

Example – Template Editor

Introduction

The Template Editor example demonstrates how to add template support to an editor. The example is based on the PDE example editor project that can be created using the new project wizard. The editor is a simple XML editor; it is only for demonstration purposes.

The code for this example is in the `org.eclipse.ui.examples.javaeditor` plug−in. To explore the code it is recommended to import that plug−in into your workspace.

Features demonstrated in the template editor

- creating a template context for an editor
- setting up a content assist processor that will propose template completions
- contributing a context type and variable resolvers to an editor via plugin.xml
- contributing templates to a context type via plugin.xml
- adding a preference page for handling templates, both contributed and user−added

Features not demonstrated

- template formatting

Running the example Template editor

1. Create a project
2. Create a file with the file extension ".xml" in the newly created project.
3. Close the editor that opened automatically
4. Select the new file in the Navigator and from the context menu select ***Open With > Template Editor*** to open the Template example editor .

Code organization of the template editor example

The example code is organized in three packages:

- `org.eclipse.ui.examples.templateeditor.editors` contains all the editor specific classes. See `XMLConfiguration` to see how the `TemplateCompletionProcessor` is added in the `getContentAssistant` method.
- `org.eclipse.ui.examples.templateeditor.preferences` contains the contributed template preference page.
- `org.eclipse.ui.examples.templateeditor.template` contains the example context type, the completion processor and the variable resolver that is contributed via plugin.xml.

Example – Multi–page Editor

Introduction

The Multi Page Editor Example adds an editor for files with the .mpe extension. It shows how to create an editor that uses multiple pages by defining both an editor and a page contributor that can add entries to an action bar.

Running the example

To start using the Multi–page Editor Example, create a new file with .mpe extension. Select the file, bring up the popup menu, select the Open With menu and from the sub–menu select the Multi Page Editor Example menu item. You can then select the different editors by selecting the different tabs.

Creating a new mpe file

Create a new file with file extension .mpe. From the File menu, select New and then select Other... from the sub–menu. Click on Simple in the wizard, then select
File in the list on the left. Click on Next to supply the file name (make sure the extension is .mpe) and the folder in which the file should be contained.

Details

The Multi Page Editor Example demonstrates how to create an multi page editor with a custom page contributor.

The Multi Page Editor Example is constructed with two extensions – a document contributor (MultiPageContributor) and an editor (MultiPageEditor). The document contributor is a subclass of `org.eclipse.ui.part.MultiPageEditorActionBarContributor` and defines the actions that are added to an editor when the `setActivePage` method is called. The editor is an `org.eclipse.ui.part.MultiPageEditorPart` that creates 3 pages whose activation is handled by the `pageChange` method that in turn sends `setActivePage` to the contributor. These two classes do not refer to each other directly – the contributor for the editor is set using the contributorClass tag in the

plugin.xml

Example – Property Sheet

Introduction

The Property Sheet Example adds an editor for files with the .usr extension and also demonstrates how to add properties and outline views on a file.

Running the example

To start using the Property Sheet Example, create a file with extension .usr. Open the file and the Outline and Properties views to see the example in action. Click on items in the Outline view and look in the Properties view for details of the selected item in the Outline view. The people items in the list show the most detail in the Properties view.

Creating a new usr file

Create a new file with file extension .usr. From the File menu, select New and then select Other... from the sub−menu. Click on Simple in the wizard, then select
File in the list on the left. Click on Next to supply the file name (make sure the extension is .usr) and the folder in which the file should be contained.

Details

The Property Sheet Example demonstrates how to provide properties to the Property Sheet View.

The tree viewer of the Content Outline View contains and presents OrganizationElements. When an element is selected in the Content Outline View the Workspace will invoke `getAdapter` on the selected OrganizationElement. If an adapter is requested for IPropertySource, the OrganizationElement returns itself as it impements IPropertySource The Property Sheet View then use the Organization Element as it source.

Example – Undo

Introduction

The Undo Example adds two views to the workbench. The **Box View** is a rudimentary view that allows the user to create boxes by clicking into empty space and dragging the mouse to form a box. Boxes can be moved by selecting a box and dragging it around. The user may undo and redo any operations performed in the box view. The **Undo History View** shows the undo history maintained by the workbench operations history.

Features demonstrated in the example

- Creating an `IUndoableOperation` inside an action (Delete all boxes) to perform the action's work .

- Creating an `IUndoableOperation` based on an operation implied by a gesture (Adding and moving boxes).
- Using a local `IUndoContext` to keep undo operations local to a particular view.
- Using the platform undo and redo action handlers to provide undo and redo in a view.
- Providing a user preference for setting the undo limit for a particular undo context.
- Using an `IOperationApprover` to install additional policy (Prompt before undo) in the operation history.
- Using `IOperationHistory` protocol to show undo history for different undo contexts (Undo History View).

Features not demonstrated

- Since the example is focused on simple undo, the BoxView code is kept to a minimum. Therefore, it does not provide the expected graphical editor features such as selection, resize and selection handles, color and line style attributes, etc. For the same reason, advanced features in the undo framework are not shown.
- There is no example of assigning multiple undo contexts in order to share operations between views or editors.
- There is no example of using composite undo operations.

Running the example

From Eclipse's ***Window*** menu select ***Show View > Other...***. In the ***Show View*** dialog, expand ***Undo Examples*** and select the view named ***Box View***. The box view will appear.

Likewise, from the ***Window*** menu select ***Show View > Other...***. In the ***Show View*** dialog, expand ***Undo Examples*** and select the view named ***Undo History View***. A view containing the undo history will appear. This view can be used alongside the Box View to watch the undo history as boxes are added or moved. It can also be used independently of the Box View to follow the undo history for any view or editor that uses the workbench operations history to track undoable operations.

Details

Box View

Click in the box view and drag the mouse to create a box that follows the mouse. Clicking inside an existing box and dragging the mouse will move the box to a new location. Note the operations that appear in the ***Undo*** and ***Redo*** menus as boxes are added and moved. The box view can be cleared by selecting ***Delete all boxes*** from the context menu or the view's local menu and toolbar. This operation can also be undone.

Undo History View

The Undo History View shows the operations available for undo in all undo contexts. To view the history in a particular undo context, select ***Filter on...*** from the view's context menu. This will filter the undo history on a particular undo context. The view can be used to view the undo history for the Box View, for SDK text editors, and for undoable operations that affect the workspace, such as refactoring operations. Undo and redo can be performed from the Undo History View's menus. **Undo selected** will attempt to undo the selected operation in the undo history, regardless of its position in the operation history. Depending on the operation in question, this may or may not be allowed. For example, the Box View allows all add and move operations to be undone or redone even if they aren't the most recently performed operation. Text editors will prompt if an

attempt is made to undo a typing operation that is not the most recent.

Example Preferences

Preferences are provided that affect the operation of both views. From Eclipse's ***Window*** menu select ***Preferences > Undo Preferences***.

- ***Undo history limit*** controls how many undoable operations are kept in the history for the Box View.
- ***Show debug labels in undo history view*** controls whether the simple label shown in the undo menu is used for displaying operations, or a debug label that includes information such as the assigned undo context(s).
- ***Confirm all undo operations*** controls whether prompting occurs before undoing or redoing an operation. This preference is used by the operation approver installed by the example.

Help

Introduction

This example illustrates the use of the help system online documentation.

Running the example

To run the Help Example, pull down the Help menu. Select Help Contents menu to launch the help view, and select the Online Help Sample from the list of books. The main topic will provide detailed information on how the help sample plugin was created.

Details

The Help Example illustrates the following tasks that need to be done in order to create a simple, proper help topics contribution:

- using the help toc extension point
- help documentation in doc.zip
- defining topics and navigation structure for help documents

The name of the plugin directory that contains the help examples is **org.eclipse.help.examples.ex1**.

See the documentation for the **org.eclipse.help.toc** extension point for more detail on how to contribute help content.

Team – File System Repository Provider Example

Introduction

The File System and Pessimistic Repository Providers examples shows how to define your own repository provider. In particular you this example shows how to:

- Extend the RepositoryProvider class and register a new repository provider.
- Register a sharing wizard that will appear in the Team > Share... wizard.

- Add resource actions to the Team menu.
- Implement synchronization support that shows up the Synchronize View.
- Use decorators to show the repository state of the local resources.
- How to implement a file modification validator.
- Adding a repository provider to a capability.

The example includes two separate repository providers, the basic file system and the pessimistic file system. The basic provider illustrates the synchronization support whereas the pessimistic provider is more focused to allowing you to test the workbench behavior with pessimistic repository providers. There is a preference page for the pessimistic provider that allows configuring the behavior of the file modification validator.

Running the example

To start using this example create a project and select **Team > Share Project...** from the project's popup menu. Click the show all wizards button. This will show both the file system provider and the pessimistic provider.

- **Basic file system provider**: you will have to enter the location in the local file system where you would like to connect the project to. The Get and Put operations in the Team menu will now transfer to and from the selected location. And if you open the Synchronize View you can browse the synchronization between the local workspace and the remote file system location the files are stored in. If you edit a file and then create a Synchronization you can browse changes between the local and the remote.
- **Pessimistic file system provider**: the sharing wizard next page doesn't actually require any user input. The pessimistic provider doesn't actually copy the local files anywhere, and instead simply simulates a check in/check out by flipping the read–only bit on files. Once a project is shared with the pessimistic provider you can add files to control and the check in and check out.

Team – Local History Synchronize Participant Example

Introduction

The Local History Synchronize Participant example illustrates how to intergate a participant into the synchronize view. It covers such things as:

- Creating a simple subscriber for accessing the local history
- Creating a synchronize particpant
- Adding a cusotm action to a particpant
- Showing custom label decorations
- Add a synchronize wizard

Running the example

To start using this example, open the ***Team Synchronizing*** perspective, click on the global Synchronize toolbar action and choose ***Synchronize with Latest from Local History***. A more detailed look at this example is available in the Local History Synchronization Example guide section.

Compare Example – Structural Compare for Key/Value Pairs

Introduction

This example demonstrates how to support structural compare for files consisting of key/value pairs. It shows how to implement and register a custom structure creator that parses key/value pairs into a tree structure that is used as the input to the structural compare framework provided by the Compare plugin. In addition, it registers a standard text viewer for the individual key/value pairs.

This example is only for demonstration purposes. Structural compare support for Java property files (another key/value format) is provided by the Eclipse Java Tooling.

Running the example

1. Create a project (not necessarily a Java project)
2. Create a key/value pair file f1.kv
3. Open ***Window > Preferences > Workbench > File Association*** and associate the default text editor with the file extension "kv"
4. Open f1.kv with the editor and enter this contents

```
lastname=Doe
firstname=John
city=Chicago
state=IL
```

5. Make a copy of this file and rename it f2.kv
6. Open f2.kv and change the firstname "John" to "Mary"
7. Add another key/value pair "country=US" to f2.kv
8. Select both files f1.kv and f2.kv
9. From the context menu select ***Compare With > Each Other***
10. A new compare editor opens that shows the structural differences of both files in its top pane. Selecting one of the properties "firstname" or "country" feeds the text of the corresponding key/value pair into the standard text compare viewer in the bottom pane.

Code organization of the example

The example code is organized in a single package `org.eclipse.compare.examples.structurecreator`:

- `KeyValuePairStructureCreator`
  is the structure creator that parses the contents of a stream into a tree of `IStructureComparator`s.
- `TextMergeViewerCreator`
  is a factory for TextMergeViewers. It is registered for the type "kvtxt" which is the type of an individual key/value pair.
- `Util`
  provides utility methods for NLS support and for reading an InputStream as a String.

Eclipse Platform XML Compare

The XML Compare plugin allows you to perform a structural compare of two XML documents. It returns a difference tree which indicates which XML elements have been added or removed and – for modified XML elements – what differences there are with respect to attributes or body text.

Installing the plugin

- Copy the folder `org.eclipse.compare.examples.xml` to the `plugins` subfolder of Eclipse.

Using the plugin

The plugin is automatically used when comparing files with the extension .xml.
By default, the XML compare uses the *Unordered* compare method, which ignores the order in which the XML elements appear in the document and matches them so that elements which are most similar are matched. There is also an *Ordered* compare method, which simply compares the XML elements exactly in the order in which they appear in the document. In most cases, this compare method will not be of much use.
The compare method can be changed from a drop–down list in the toolbar of the structure view.

When an XML document contains elements that can uniquely be identified by an attribute or the text of a child element, it is recommended that an ID Mapping Scheme be created for this type of XML document.

See Tutorial and Examples for more information on using the plugin.

ID Mapping Schemes

An ID Mapping Scheme specifies for XML elements an attribute or the text of a child element that uniquely identifies this element. This assures that – in the compare process – the right elements will be matched and therefore compared with each other. If for an XML element no ID Mapping is specified, a general matching algorithm is be used. However, this general matching algorithm does not always return the desired result. The reason for this is that the general matching algorithm looks for a matching of the nodes of the two parsed trees to compare that minimizes the differences. The effect is that often two XML elements are matched with are structurally similar but represent two completely different entities of information.

Ordered entries

When using the default Unordered compare (with or without id mappings) it is sometimes desired to specify that the children of certain elements be compared in ordered fashion intead of the usual unordered method. For example, when comparing ANT files the order of appearance of the children of `target` elements is important.
In such cases one can create an *Ordered entry*. An Ordered entry specifies that the direct children of an xml element, identified by its path, will be compared in ordered fashion (attributes however are still compared in unordered way). The children of these children will continue to be compared in unordered way, unless otherwise specified.

Defining ID Mapping Schemes and Ordered entries

ID Mapping Schemes can be created in three different ways:

1. By extending the extension point *idMapping*
2. Using the XML Compare Preference page.
3. Using the Create new Id Map Scheme button in the toolbar and the context menu

Method 1 creates a so−called *internal* mapping scheme. An internal ID Mapping Scheme cannot be edited at runtime. However, using the *Edit Copy* button in the Preference Page, an editable copy of the internal ID Mapping Scheme can be created.
Methods 2 and 3 create so−called *user* mapping schemes. These are created by the user at runtime and can be modified anytime in the Preference Page.

Internal and user mapping schemes can be associated with a file extension. As a result, when comparing two XML files with this file extension, the particular ID Mapping scheme with this extension is automatically used.

When creating or editing the ID mapping for a particular XML element, four items must be specified (see example):

1. The element name.
2. The element path. This is the path of the element from the root of the XML document to the element's parent.
3. The name of the id which will identify the element
4. Whether the id name in point 3 is the name of an attribute of the element or the name of one of its children (in which case the text of this child element will be used as id).

Extension Points

Only one extension point is available in the XML Compare plugin. It is used to create internal ID Mapping Schemes:

- org.eclipse.compare.examples.xml.idMapping

Tutorial and Examples

General Matching vs. ID Mapping Schemes: How to create an ID Mapping Scheme to improve compare results

Consider an example XML file in two slightly different versions, left and right. Assume that the element `extension-point` is uniquely identified by the attribute `id`. The textual differences are shown in bold.

```
<?xml version="1.0" encoding="UTF−8"?>

<plugin
  name="%Plugin.name"
  id="org.eclipse.ui"
  version="1.0"
  provider−name="Object Technology International, Inc."
  class="org.eclipse.ui.internal.WorkbenchPlugin">

<extension−point name="%ExtPoint.editorMenus "
id="editorActions"/>
<extension−point name="%ExtPoint.popupMenus "
id="popupMenus"/>
<extension−point
name="%ExtPoint.importWizards"
id="importWizards"/>

</plugin>
```

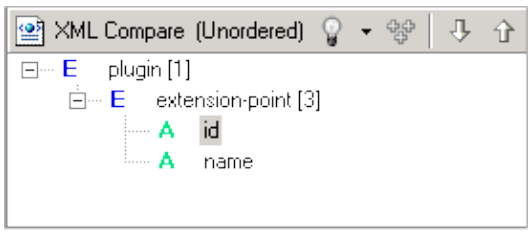
```
<?xml version="1.0" encoding="UTF−8"?>

<plugin
  name="%Plugin.name"
  id="org.eclipse.ui"
  version="1.0"
  provider−name="Object Technology International, Inc."
  class="org.eclipse.ui.internal.WorkbenchPlugin">

<extension−point name="%ExtPoint.editorMenus "
id="editorActions"/>
<extension−point name="%ExtPoint.popupMenus "
id="popupMenus"/>
<extension−point
name="%ExtPoint.exportWizards"
id="exportWizards"/>

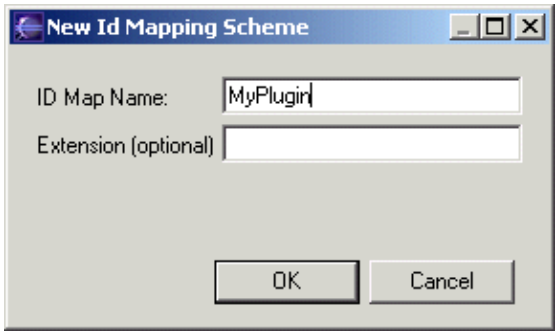
</plugin>
```

Assume that the order of the elements should be ignored. The structural difference between the two documents consists in the `extension-point` element on the left with `id="importWizards"` being replaced on the right with a new `extension-point` with `id="exportWizards"`. Using the general matching algorithm called *Unordered*, because it ignores the order in which the XML elements appear in the document, we obtain the following tree of differences.

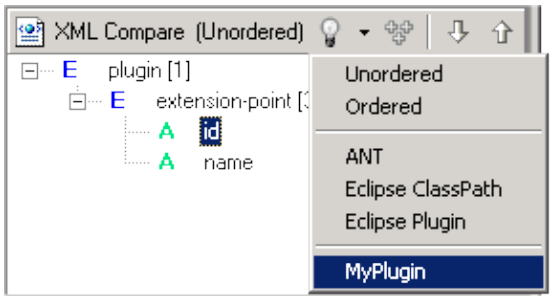


The first two `extension-point` elements are identical and are therefore matched and are not shown in the difference tree. There remains the third `extension-point` element on both sides which, having the same element name, are also matched. The difference tree then shows the differences between the third `extension-point` element left and the third `extension-point` element right. These differences

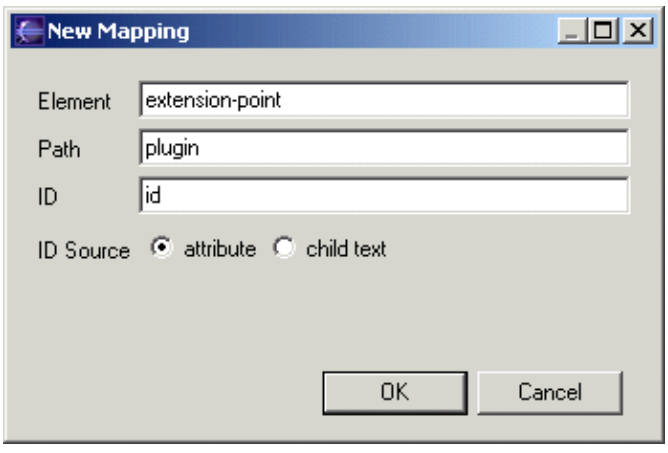
consist in the values of the attributes `id` and `name`.
However, this is not what we would like to see. We would like the difference tree to show us that an `extension-point` element was removed from the left side and a new `extension-point` element was added on the right side.
To achieve this, we create a new ID Mapping Scheme. We can do this by using the appropriate button on toolbar.

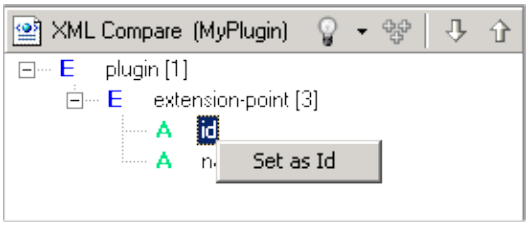


Assume we call the ID Mapping Scheme *MyPlugin*. We now select the ID Mapping Scheme MyPlugin from the drop–down list in the Toolbar

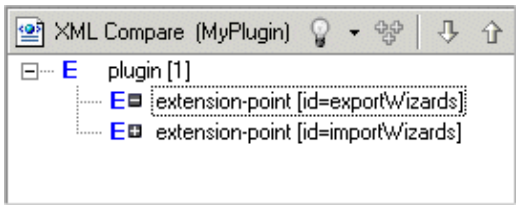


and add to it the following Mapping:





This can be done from the preference page (left) or from the context menu in the structure view (right).
The difference tree now becomes:
(To refresh the structure view, click on the button of the drop–down list in the toolbar.)



This is the compare result that we wanted and that we achieved by created an ID Mapping Scheme.

The XML Compare Plugin already comes with a ID Mapping Scheme for Plugin files, which can be customized for particular Plugin files.

Warning:
If an ID Mapping is created, it is assumed that the id is unique, i.e. there are no two XML elements with the same name and path that have the same id. Should this not be the case, the ID Mapping Scheme may not deliver a sensible difference tree.
When an id can appear more than once, one should rely on the general algorithm.

Also, when an ID Mapping Scheme is used and there are elements with no id mapping specified, the *Unordered* compare method will be used, i.e. elements are matched by their similarity and not by the order in which they appear in the document. To specify that the children of an element should be compared in order of appearance. See the next section on Ordered entries.

Adding Ordered entries

Ordered entries are used to specify that the direct children (excluding attributes) of an xml element – identified by its path – should be compared in ordered way instead of the default unordered method.
As an example consider the following ANT file in two slightly different versions:

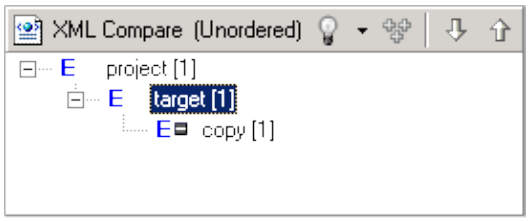
```
<?xml version="1.0" encoding="UTF−8"?>

<project name="org.junit.wizards" default="export"
basedir="..">
  <target name="export" depends="build">
    <mkdir dir="${destdir}" />
    <delete dir="${dest}" />
    <mkdir dir="${dest}" />
    <jar
      jarfile="${dest}/JUnitWizard.jar"
      basedir="bin"
    />
</project>
```

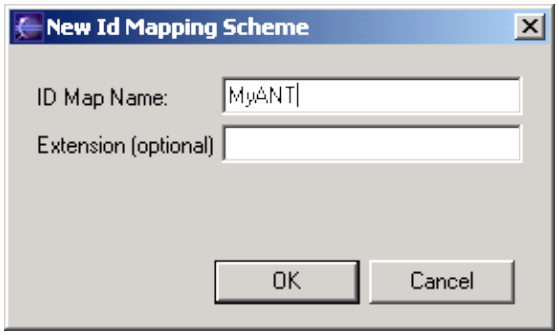
```
<?xml version="1.0" encoding="UTF−8"?>

<project name="org.junit.wizards"
default="export" basedir="..">
  <target name="export" depends="build">
    <mkdir dir="${destdir}"/>
    <mkdir dir="${dest}"/>
    <delete dir="${dest}"/>
    <jar
      jarfile="${dest}/JUnitWizard.jar"
      basedir="bin"
    />
    <copy file="plugin.xml"
todir="${dest}"/>
</project>
```

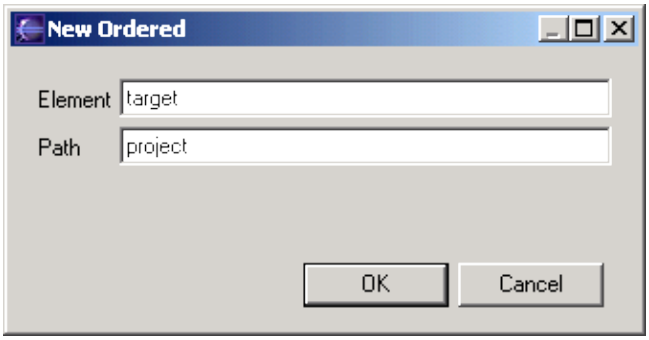
The differences between the two documents are shown in bold. Two elements have been swapped (`<mkdir dir="${dest}"/>` and `<delete dir="${dest}"/>`) and a new element (`<copy ... />`) has been appended to the target element.
Performing an unordered compare will result in the following tree of differences:

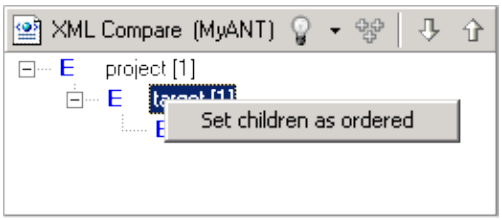


The fact that two elements have been swapped is not shown since the order of elements is ignored.
However, from an ANT point of view, the two documents cause very different behaviour, because the order of the elements inside a `target` is important. We therefore want to create an *ordered entry* for target to instruct the compare engine to compare the direct children of target in ordered fashion.
We do so by first creating a new ID Mapping Scheme. This can be done using the appropriate button in the toolbar.

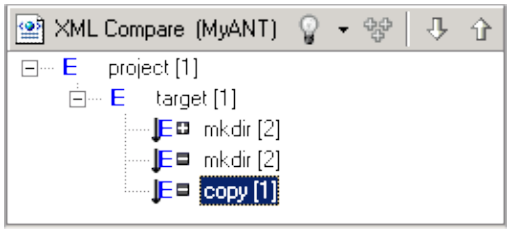


Assume we call the ID Mapping Scheme *MyANT*.
We now select the ID Mapping Scheme MyANT from the drop–down list in the Toolbar and add to it the following Ordered Entry:





This can be done from the preference page (left) or from the context menu in the structure view (right).
The difference tree now becomes:
(To refresh the structure view, click on the button of the drop–down list in the toolbar.)



This is the compare result that we wanted and that we achieved by creating an Ordered Entry.

Additionally, Id Mappings (see previous section) can be used to uniquely identify ordered children. Especially when there are many changes, this will improve compare results.

The XML Compare Plugin already comes with a ID Mapping Scheme for ANT files, which can be customized for particular ANT files.

idMapping

Identifier: org.eclipse.compare.examples.xml.idMapping

Description: This extension point allows to define internal XML ID Mapping schemes using the *mapping* element. These schemes can then be used when performing an XML compare to uniquely identify XML elements by the value of an attribute or the text in a child element.
Additionally, *ordered* elements can be used to specify that the direct children of an element should be compared in ordered fashion instead of the default unordered way.

Configuration Markup:

```
<!ELEMENT idmap (mapping*)>
<!ATTLIST idmap
   name       CDATA #REQUIRED
   extension  CDATA
>
```

- **name –** the name of the ID Mapping scheme. Should be unique.
- **extension –** (optional) a file extension associated with this ID Mapping Scheme. When comparing files with this extension, the current ID Mapping Scheme will automatically be used.
  If an extension is specified, then the extension should also be added in the plugin.xml file of the XML Compare Plugin. For example, if we create an ID Mapping Scheme with extension cd, the plugin.xml of the XML Compare Plugin is updated as follows (update shown in bold):

  <extension
      point="org.eclipse.compare.structureMergeViewers">
    <structureMergeViewer
        extensions="xml,classpath,**cd**"
        class="org.eclipse.compare.xml.XMLStructureViewerCreator">
    </structureMergeViewer>

  Warning: If an extension is associated more than once, only the first association will be considered. Also, internal ID Mapping schemes have priority over user ID Mapping schemes when duplicate extensions are defined.

  Warning: When an internal ID Mapping scheme with extension association is removed (from a `plugin.xml`), the XML Compare plugin has to be reloaded to disassociate itself from the extension. If not, on the first compare of files with this extension, the XML Compare plugin will be used with the default *Unordered* algorithm. (However, at this point the extension will have been disassociated from the XML Plugin, as the plugin has been loaded.)

```
<!ELEMENT mapping EMPTY>
<!ATTLIST mapping
   signature  CDATA #REQUIRED
   id         CDATA #REQUIRED
   id-source  CDATA
>
```

- **signature –** the XML path from the root to the current element (see examples below).
- **id –** the attribute that identifies this element or the name of the child element whose text identifies this element.
- **id–source –** (optional) if *id* is the name of a child element, then this attribute must have the value *body*. If *id–source* is left out, it is assumed that *id* is an attribute.

```
<!ELEMENT ordered EMPTY>
<!ATTLIST ordered
   signature  CDATA #REQUIRED
>
```

- **signature –** the XML path from the root to the element whose direct children will be compared in ordered fashion instead of the default unordered way.

Examples:
The following is an example ID Mappings Scheme for ANT files.
project elements are identified by an attribute *name*. *target* elements (which are children of *project*) are also identified by an attribute *name*. Also, the children of *target* will be compared in the order in which they appear in the document.

```
<idmap name="ANT">
   <mapping signature="project" id="name"/>
   <mapping signature="project>target" id="name"/>
   <ordered signature="project>target"/>
<idmap>
```

The following example illustrates a case where the text of a child element is used as id:

```
<idmap name="Book Catalog" extension="book">
   <mapping signature="catalog>book" id="isbn" id–source="body"/>
<idmap>
```

Supplied Implementation:
The XML Compare plugin defines an ID Mapping scheme for Eclipse `plugin.xml` files, one for `.classpath` files and one for ANT files.

GIF89a” ÷)1 A B B BB9BBBB J J J! J)!J1)J R R !R! R! R! R) R) R) R1 R1)R91RJ
\˜Œ²Ê)ŸÌ… (H ™Ì–‘ÀÙg8*f*¦™
¥¥ „ ?»¶Ú|²Á&ÛÑ´ ‘„mJ ñ oP 'ÜÔÂE1µÕV#‡õt\Kçu ØmÇÝØÝq Fzì - Úê…q |pÃMŸÜeä7† (
þ ö-à €Ÿ!x*f* :H¡á Z¨¡†r$>â‡ “x¢)º^âŠuX yæ7v¾c |ð¡G BþXz ¨û ú’¬;¹d”¯sÒ‰”²×>{'¸Ë¾åî`,)&)Z i&›d²9ç›
Oø Ì ‹ae ¡%ºÊðö å ðuïQâ£”D1z>ô *}ëS_ûHZ¿UÑO~õk þhå«ýÕª¥ü_
(, °YËZ ¥ N0,HÖ–¹ÄQ®qqðÉ ¤—»F ¯x 0 ÷º²–ß Ö zy†6 ³ Çü° ñÌDÔ þ\ 1qd%+
[(Ã _èÂ r `€
‹yLc #Ù2 ÑŒf$AKB“3 Àq³£©c %mGÚä±6·YÚm ¹ BN- ÁQ rDÍHG^ç‘’L5w¦ Lºú;äÑÂ'‡K P’Ò>£ ¥~V
^ h@¯T d¹ [ûp âå/“ Ì yHrÅ,¯2™Ù"fÂ^½Ò´‘¶E _nj3HHB’“Â NÚM
›@ç:y×Îv–‰Lñ¬'=á„&}âÓNüì§ GÐ~û" e¨À9 ÑG ÜRÀ¸”¥0zÑN
¤ ÿ¨‹Iõ¾W•ô¤&e©-RêÒ˜â*¦ÿC Mþ‡uS’ ‹§;¥–PW TlUðåâ(¸,H®¦®
ªN}WU©Z¯«š°«\õ×W FÖ± ý` K«[—þÖ¹VŒ^EôGCÜÂW‘9ñ
FA0þÑ FØÁ _^Ã a / %)‘¬eo†Ù”äl4›UMgã84£ A´¨-Mn2m àôæïÁ)$Õ¬¦ Ùf

Q°Îm§ ¶Iævl–´äÙ€+Ü´¹-=r‹ }îc7¼ùzo úuà¤û· ™¾B¢ƒâ'ÍÝ e(¼Ä$ï d¿"óf ½œÃ'îE :ÞŸ.¾«S êì»$Mè÷¾T¢ÝíÊ
¤ ˜€Öâ@E†dGFAM–.é¢AèÐAç"BQFe%"eûÒUêÐBíðB^&03TC8",aÖCððCgf1i– k¦ Qƒ Ñfùðf WÇ „€ þà
s
‰o p Op ð2" 'vd,,hh,,)qY †3m"…§!G"&Gw4 zÔG¦´4 H€ xRS ‡" ¤Ö ¦†j' I"4Iá ¯æ ›ÔIi³‡ê J¦TJ¸¶J
° lþÄfzµ ! ,K°"] ÒKÝÕ] "LÄtLÇT9ÕÖLË´{Ñ MÔ :Ü†MÞ¶Màfl š $äFNš`%ý•nY,%^RN»ãn¶Ø&ò–‹óô&öÆ`ùö
€2P Õo ¥('p‡B ‡ QÎ8) •p
Çp U åQ R¦BR +*Ec+Åq–å?0E,½b Çbr%§SGIF89aÿ Ë÷)1 A B B BB9BBBB J J J! J)!J1)
y)9{B |E8|,,,,!B{!B,,)B{)B,,)BŒ)J,,)JŒ1B,,1J,,1JŒ1RŒ9JŒ9RŒFF RG†ÿZ1"Z1œJ9"RB"RJŒRJ"BJ¥1R"9R"BR"J
z›çíß¼{(Sª\É²¥Ë—0cÊœI³¦Í›8sêÜÉ³§ÏŸ@ƒ J´(LBÞJ¢ÇÛ6 ßÀƒ t ^íÜ´
1>® ùÒÅMQ šN}zcæØo7ß Ý¹wíÚ»þƒÿ.¾|øóäÑ }®˜{÷çV Ÿ ürzï‡1ß_/=érÈ,,àc ! ¬äØu, è , ¶¤ }
F á,,úIX!… ^¨a† :èa,þ â^"–Èà‰ vH"Š †Èâ†.ÞÓ^|(Íø!}ô¥dÜ‹ ºGÈJ>z¨Ü‹ YdŠ hâbú)7
>J 8 Å¥D%•;Þƒ%–•M À—U"(¥}y ©_™b¢te˜T†i¦™\Zö¦•Z†)gš]âyg |îéçcsö ¨ ƒ
Š¦¡d&z¦¢ þy¨eÅÐ U{ â8ßi…ZÖ ?,,ÄwÒ¦ fšß¢Q –O1J"ú !¨&: tþ"))Ä˜ó G^ÖæRkFÖæ® îZç£H®XìJ¿B ì±N
¥úE
âÔÝã.¼Ö]7¤…Ê Z`† :Ù*,,LbæäAù¤j •'ýŠ°–Ñf™0JYŠ¨Y£™V|h´Åµ™—ÃŠÖi§,Ä&»1£,, k2ÉªR|2©*£ì¨Ë Ã¼rË‹Î
aºÂA ²«Åð£ >ó¼*¯ÐD }Ù¨ ;•ï¿ :°¾b
à?³ÒZë? Wø°Æ`fŒ1fÎz©ð•[~É¥Úb‹ ' Ñ>\˜Æ±MOØjŸ-fÞÚmÞÍ¶Çv,Ëìà, ¾–»^ ~¸âÝ N®¼Q™;1ÏÁ-+8u¯Z‡y¼R
u=aßs·îñÈ¯W¹k¯ÉÎ ¬Ç¶gú(Æwï-&à´Ói;Æ½âþgÌ,—,óòÉ§¬ló/C "S÷~c9+–ð3ñ R¼ª UG~F² ûJ—
«¥dtþ Q KlwÁhìvU `ð\ÇÄ%¶°P á ñ€W@&þ pW¦ iÖW=çy1z] £õÆøÅé'QŒŸÚ øu §3r;IÄ. *I%jHœ¹Çèò
j ZW"[yõÀIJ ' „¿H(® –ò"Ü2e*Q™8 ¾${6Û ÜøÄ =j óš¡™ô˜˜sé' ?´àö¢x½~Ê- ß«A‹bö³ ý± ã
ä—4¶Æ3 /;Ã
áj±±¶±JJu¦Lå*[ùÊXÎ²–·Ìå.{ùË` ³˜ÇLæ2›ùÌhN³þš×Ìæ6‡9)þ HœåLç ÌùÎu 3 ÷Lg ê ð7"â€ â@ èŠ÷· L ïð €
†›7› y†¢Wz¸©†©÷z± ‡ é µW‡¢°{"©‡|È‡ ˜œÌ·œÍW^…˜^Õw} è}à ‰àW~Ø© —þ¨~Ù°‰âÀ
H ¢¨ ÿwŠ ˜Š ø Gqm3's È /'s ¨G È ^–»Ø jY Â¸nLGnÆ¸r$Ø–%X—vy xi % Ë0 Ã
dð O€ /^ ð,0 á¸uŒ
vdGv;^vjÇvò(r „ù šM(x© RHš Y£ çxpÐš_ø… y zzÐ × ‰z• z ñ†²G{À‰{vÈ{¢@'«0
«`'® ® '–Ù|Ð×¥Ó7"ÐÀ^6ù}áw ã·"ç—~@¹
ß ‰òg"â äp KÉ vê (€õ Ÿ] · àãL ¸•þsõ ôi ô saÉ‹ŒÊŸAçŸã —Ãxn ú–êV©t‰ ñŒJ Ëà û
kÀ dP : p˜ !ƒÿÐu6ø ª8¨ƒjG™JP Dh,, qwV0 © !xZ RH…‹w…Y(yn0 ^Ø ²I†gX† 'z©×†I
‡r 's˜‡x¸‡Ü:lÊ)^Í }†^^ÕW}Ø7 Ê Ö)‰ØY‰ì—‰ï÷~òç‰ã‰ ¢HŠç‰Š©¸ŠËaZ?ä f ¨Òá•ö W
3@m'q«x‹‹ús Ë ¿ØŸ nÆ8©D' Éx© —s©© ñŒXÐ ºà ™@ "
‹p ¦j àþ "@ áø ^çªèHvê8«îX«ÿ€¢q7 +JwT@ ú˜ ' €]0 SX ™ 7y•× – '³ÙyxÀ z¦g'
«—¤®Ç''÷'vX‡x(¥£ ®Ð‡¬€'*É'. "3)"6 7Y¦:É"‰‰í ¯ð×‰õ Š÷ª¿¨ ë¹L ¥¨Í² «À¬U7 c ‹¹ ŠÉ›üÐ ,
š,¨@ Ö` §@ M ³ ¦
à²(, ±Â4ëÂ Ñþ 4¼³8À pÂ Z»\g d‡»¸¼ kW qg :Ü™üøÃ Û a aÛA Ôv` T¬ Vì ÑÔ®ÇÅÆ½½a Î
þ8¸1aG ?:nüøO¤Æ $/† É eÉ—0]flIpeL–'gÊÔÉ³§ÏŸ@ƒ J´¨Ñ£H"*]Ê"
¾§P£J JµªÕ«X³jÅjo+¾®QÁJ ;•,×«fÍ õz¶ªZ¶OßÂ K·®Ý»xóêŸÉ·¯ß¿€ _Ý÷ Ÿ¿ ‡ &þ·¸1âÇŠ!3–ì8²åÉ—+cÞ¬Y3g
¥_¯¾=ûô, ÅŸ/¿>ýûöóÚ ÿöû¥ à~ h ‡0²_, 2r "
> !"‹Pha… ^ha# $È¡‡ $Âaˆ#Š8â‰(¦ØH$:Âb#.6òˆ‹ Ì I " ‹YÅê#
M‡ s"c …" :Ì »×© Œ$$¹µ JÚn\íà »0i®NöN"™Œ×:Dy D ^ ~(&?Œ Sc Jæ€6v ' Â™ r …&„2
-iš‡€Y6]–Í™yÓDàòfÎv&#GðŒF=BçÐ„FB – iJƒšÔš5y¾°KVþ³'
¿ÆO}ò –¸ œ :Ð9Å©mmBèA…87MÌ–PS„hÞ& 7¿Y´P€ \àºÈÑJ–î£)ä&GÒR- sá°\JãÈR[™ãs/
8FGSe!Ë¦6e–M[·ºFúÔZµ ªî†Ê;uýîwÃkGRKÉÔ{9¯yÌ" T¯G ¬ Â•#˜,Óã Œà
½°…/ˆá '0á0 @ œ`ˆùáÏ~p XàÚ¿BÔU€xÕX ÷ŠÍ¾>0A ]` +HØ
VE ä`ŒÚIÂ Ð„ uÑ$T˜Â N †˜}R•®d¥)Y lûÜ¡hÉ D '^¨¥S WûD&Bñ¢ ½þ¢ µH[Iu´R_tTnÁÈ©`ŒÑ·m,Õ¨
£ÅÝŸZë' /$»¥ p W"Ÿì]'3™ÞQ¾Ë½òjª½Pé Sºk•O¹G\ð¡ß¯ …B¢,
S˜ÙjSB†Uò¬> ,¶"óó´hrrjU+'NÈÉµèvbº—øZ)ÊVQ‰ºâ ç–fJ‰Q·½Í÷– Ëïàö[¹pŒU›ieÇçÒ* ä
Ö éÌpëÚÙ ^4ä³ôL–I†wZåÝVÆÑ Þ ºã¡l/)þ =/ûÒ øå/>&} ISz_ p à„_Ø! p€ƒ ¾ð….†à ®Ð Õà wÐ
Ï÷|« v` ñ"}öAtð8 Ýg1Óg}v…~ "ayh ¢~ÿ ‡¹†@ Økñ AõWX÷‡AÇÖAû· 8ö Î €ÐVY Y€Õµ„€{g&
8&=äd¦ sB H4 š°n ÈZ çnñö' Rþ[\$,¼…oûæ[79Fl´,(x\÷"»Xp3^p
']>x"t–H ÷ƒÇhqD8^ g^æ',, ×^!'/Ohh&Wr)§_.—…[hiÅ×&ð õ À _ð >ç ð 1×— Ô7uƒpL_Qun8

u˜— ©—âw V~#¦‡ Xb» v/3veÇb ¤vç4clç^6 ‰:&wód‰>–™x7d}×O|‡dÜö%Yâ&e£ l"'¤(e
5D§Ø'ª˜ŠxÃŠ åŠ ‹…#‹fVy&E9¨'y¹¸R»¨9 Wœ g8Ez
wzÇ^ŒìÐg}¦ŒFÕŒå¢^´w Ñx{Óþ¸h×8 i*çr[sÅç \€ _ lÙ =ð À ð ª ø`? & R!‡°¶ŸãW~Z§~þ
¢k°æ~ ™ óWXÄ¶
úç ýG‘ YÒv¡—…‘0t€T,mÛÆm Y& ¨&— D$YD(9 +¹¢´ [0é Y "!Xo#È[%è[e´)¬B\Ø]@8,„â%I W^•„ ëÂIX°
µ =° ;ð ^ tnu— SLw¹aî±LÒ1 bþ
su^w˜€øu(¶¨ tˆfx˜^˜" ©NŒH™>3¡'x$šÍvg‰S"‰Cæw›h&^ó¡ x&aC¢ õd†7etC7[¦e²Ù¢µ
f„ƒ›†£›´8Ri†f¨ÂfºŒ¶Ò‹¿(gu&ŒÈrzÅØœ×'Œ-‡;ÌøzŸôŒÔš Æ³ Éc Òƒ ûÕ_öÀ ®ä Þ N@9 À à' Ø `
Ð Ã W…ÀjpU?ú*küJ µVk d ùu X Ä öç *Ò É 'ý×l'%'Ó6± C^RCW P8"%G ¢
‰&#z¢ ˆ¢o¢'&«ní 80‰E2 þ£3I£5Y"&˜"7É,>ÚF7ûo1¸³2("ÐE"Ó¥¤:h]‰ä¤Mù"Qz„Û²IZÚ´[j•ð µ_:…\Y… Ê Ë„U
¶ À p —nµ§l‹1W ¨~*@ÐD¨%S! b·^ ˆÛ4ˆÝT3~ ©.&©Š8¸'i©•‰c™Ú©™ı™ ú©˜ø™7ô™4Ô%£
x ŠPVx¬‰Z¯)›±YQ^æ7 f¸z)ºº›¿Éo¼ê*Á)¬Ä
S°‹œÇšSÇš"ÆˆgÌú:Î* &-ÑÊ ÐX{ÖZ/¹§äÿLÛý ãÇ¼åå–Þø=âÜ¡ÏH^Ø# Ûl!æjnÌú àþU Ó¸ÝÎi!àW àð
Ýåý Þ >à ^årmÞ^.ç =á Mß‰nè –åvýÑñ é„ é ^Îg–Ý âšNà í× îÔ
Ñ&ÎÙ Þá¶mß–NÖ ã ®Üµîè íç˜ýé4½â}>ç"ÍÞ > »ÝÙØ}Ù^ æQN ÎÎìÏ âÒNê,~Þ¯Îâø]åP> í7ÎÛ ë^ýåŒÎ ¬
à™íþãw½×« Û\þìô®ÙêÝ€ýâ1oÝ _â Ý>>âp ôoáÝ ã þñQ^éßÝê¨îó?nÎ
^ì ØËGIF89aZ å÷)1 A B B BB9BBBB J J J! J)!J1)J R R !R! R! R! R) R) R) R1 R1)F
ä B Iä'F) Xmñó– PF)å"^¼eÂ W Â \ Ñ×
F aÄ c–yÄ™‰5 D N°É¦ m:A… VPaE wæéY W„1† JÆ ¦•AF ^ªñÚ¢jÄÑh qÈ i¤sÈa‡¥rÜQ‡o¿uŠG xðÁ:
+¿ÚrË°¹Ü' .Ã ‹,1ó1ë,1ÊØ·Œ1Ë0³L3×6 3Ûn;M4Ô€ ®4ÚT£Í6Ôls.7 n# „á^ã 8ãÐ;Î8ët˜ 9é¬CÎ:ì
0N9 šÓ¡9ë„ p‰'šX°Â°Ë #=õØXãÄ;î˜±G!ñ.RH ¸ä»I½'°»ðÈ ßã'a¹%, („ X H XDÑÂ \àE oa9Â
ã"ð¥ `¦`æ˜@7!´ÐL8æ&œvÒi§ x^¡YŸ|†á T+"¡Ê ð5 ê ¤ ()IÕáRwÈÔ &HAáàa Ÿ*Õ¨F•‰T-ªU²' ÞFx+¾õm
‡ÅBbÝ"YÃ F.dø,f ãqÖ¢ ¶$§–n
H\þ— ½ÊEÄm°Ës {W7ÄÑ z9± øÒ—9Ôa €YÑD c ÛO¶ ä oYN^ "bƒ",,Ä%Îë›÷Ú éÏ 8Æ?ú 8 â
o $ 2P% |$KÿàÒùÀ$&2y"}BûÈšâô 85 sªS ð"?þõi
¡ ¡ …¨C%ŠQ |Ô¤ X);ÊPœ2 uÃ²Zk‡=„F4š o Q ä2—¹ÔeÄv)1 M¤—½¢˜¯¾ÞËŠ ;Ñ ¦0†‰±a «ÇíÌx±z ô
m H?';²8J Ùç?÷QÇ'=¯ …ð„?H H@Â Š^¨ dP% Œä.ÿHþ$GyæH ¥ #1ÚH&Ã[¦YÆiŸ CÔ¦65A ê4Y{
Õ(¯Õ&lÐ ›n2%Ë´Y—m¥z[.åæ ^Þ–oÚ &UG 8¬"³…‡3\â gCh> ‡"óÏ€
t9)è¾›{‡A÷MyAQœ§3gêÐé°uÆî ´³ ÒHOË²1x'= f¿b

Platform questions index

Getting started

- How do I install the examples?

Core runtime

- What is an extension point? an extension?
- What is the formal definition for the manifest file?
- Where do I put translated files for NLS?

Resources

- What is a workspace?
- What is a resource?
- What are resource properties?
- How can I save my plug−in's information when the workspace is saved?
- How do I know what changes are being made to resources in the workspace?
- How do I use markers?
- How do I define my own marker?
- What is a project nature?
- How do I set up an incremental project builder?

New file types in the UI

- How do I set up an editor for a particular file extension?
- How do I add a properties page for a resource?
- How do I add a new resource wizard to the workbench?
- How do I provide special icons for my new file type?

Workbench UI

- What is a view? an editor? a page?
- What is a perspective? How do I define one?
- When do I want to build a view vs. an editor?
- How do I customize a content outliner for an editor?
- How do I add a page to the preferences dialog?
- Where should I start learning how to extend the workbench UI?
- What extension points are defined by the workbench UI?
- How do I add a menu to the workbench?
- How do I specify menu and toolbar paths?
- How do I implement common actions like cut/copy/paste?
- How should I manage the icons in my plug−in?
- How do I build an error or message dialog?
- How do I implement a simple dialog of my own?
- How should I remember the user settings in a dialog?
- Why would I want a viewer instead of an SWT widget?
- How should I start learning about the text framework?

- How do I launch a wizard if I'm not adding it as a workbench extension?
- How do I build a wizard with multiple pages?
- What widgets are available in SWT?
- How do I implement an SWT layout?
- How do I implement an SWT native widget?
- How do I implement a custom widget in SWT?
- How do I draw my own graphics?

Installation and upgrade

- How do I configure my own splash screens and product name?
- What is a feature?
- How do I package a feature so it will work with the update manager?
- How do I configure a custom welcome (intro) page?
- How do I install an Eclipse based product?

Notices

About This Content

February 24, 2005

License