Programmer's Guide

Contents

1. Welcome to Eclipse

1.1. Who needs a platform?

1.2. The challenge

1.3. What is Eclipse?

1.4. Go to eclipse.org

2. Platform architecture

2.1. Platform SDK roadmap

3. Simple plug-in example

3.1. A minimal plug-in

3.2. Creating the plug-in project

3.3. The Hello World view

3.4. The Hello World manifests

3.5. Running the plug-in

3.6. Beyond the basics

4. Runtime overview

4.1. The runtime plug-in model

4.1.1. Plug-ins and bundles

4.1.2. Extension points and the registry

4.2. Runtime components

4.3. Runtime preferences

4.4. Runtime application model

4.5. Content types

4.5.1. Using content types

4.5.2. Contributing content types

4.6. Concurrency infrastructure

4.6.1. Reporting progress

4.6.2. Job scheduling

4.6.3. Scheduling rules

4.6.4. Locks

4.7. Network support

4.8. Security

4.8.1. Secure storage

4.8.1.1. Secure storage architecture

4.8.1.2. Developing with secure storage

4.8.1.3. Creating password providers

4.8.2. Signed bundles and protecting against malicious code

4.8.3. Login based on JAAS

4.9. Console Shell

4.10. Structured text

4.10.1. Terminology and design

4.10.2. Supported text types

5. Plugging into the workbench

5.1. Workbench under the covers

5.2. Basic workbench extension points using commands

5.2.1. org.eclipse.ui.views

5.2.2. org.eclipse.ui.editors

5.2.3. org.eclipse.ui.commands

5.2.4. org.eclipse.ui.menus

5.2.5. org.eclipse.ui.handlers

5.2.6. org.eclipse.ui.bindings

5.2.7. Workbench Core Expressions

5.3. The plug-in class

5.4. Preference pages

5.4.1. Contributing a preference page

5.4.2. Implementing a preference page

5.4.3. Field editors

6. Dialogs and wizards

6.1. Standard dialogs

6.2. Application dialogs

6.3. Dialog settings

6.4. Wizards

6.5. Workbench wizard extension points

6.5.1. org.eclipse.ui.newWizards

6.5.2. org.eclipse.ui.importWizards

6.5.3. org.eclipse.ui.exportWizards

6.6. Wizard dialogs

6.7. Multi-page wizards

6.8. Filtered items selection dialog

6.8.1. Creating a custom filtered items selection dialog

6.8.2. Advanced use of the filtered items selection dialog

7. JFace UI framework

7.1. Viewers

7.2. Actions and contributions (Deprecated)

7.3. User interface resources

7.4. Long-running operations

7.5. Field assist

8. Standard Widget Toolkit

8.1. Widgets

8.1.1. Controls

8.1.2. Events

8.1.3. Custom widgets

8.2. Layouts

8.2.1. Custom Layouts

8.3. Threading issues

8.4. Error handling

8.5. Graphics

8.6. Examples

9. UI Forms

9.1. Controls

9.1.1. Form

9.1.2. Hyperlink

9.1.3. Section

9.1.4. FormText

9.1.4.1. XML Markup

9.2. Layouts

9.2.1. TableWrapLayout

9.2.2. ColumnLayout

9.3. Advanced Topics

9.3.1. Color and font management

9.3.2. Managed forms

9.3.3. Master/Details block

9.4. Multi-page form editors

10. Common Navigator Framework

10.1. Configuring the Common Navigator

10.2. Operational Topics

10.3. Step-by-step Instructions

10.3.1. Creation of Common Navigator View

10.3.2. Content and Action Binding

10.3.3. Migrating from the ResourceNavigator

10.4. Troubleshooting

11. Resources overview

11.1. Resources and the workspace

11.2. Resources and the file system

11.3. Resource properties

11.4. Project-scoped preferences

11.5. File encoding and content types

11.6. Linked resources

11.7. Virtual folders

11.8. Resources filters

11.9. Resource markers

11.10. Modifying the workspace

11.10.1. Batching resource changes

11.10.2. Tracking resource changes

11.10.3. Concurrency and the workspace

12. Advanced resource concepts

12.1. Project natures

12.2. Incremental project builders

12.2.1. Project Build Configurations

12.3. Derived resources

12.4. Workspace save participation

12.5. Resource modification hooks

12.6. Refresh providers

12.7. Alternate file systems

12.7.1. Using the file system API

12.7.2. Working with resources in other file systems

12.7.3. User interface support in other file systems

12.7.4. File system providers

12.8. Dynamic path variables

13. Advanced workbench concepts

13.1. Workbench menu contributions

13.2. Undoable operations

13.2.1. Undo and the IDE Workbench

13.3. Perspectives

13.3.1. org.eclipse.ui.perspectives

13.3.2. org.eclipse.ui.perspectiveExtensions

13.4. Decorators

13.5. Workbench key bindings

13.5.1. Bindings

13.5.2. Schemes

13.5.3. Contexts and key bindings

13.6. Element factories

13.7. Accessible user interfaces

13.8. Honoring single click support

13.9. Working sets

13.10. Filtering large user interfaces

13.10.1. Activities (Not currently in 4.1)

13.10.2. Contexts

13.11. Workbench concurrency support

13.12. Workbench resource support

13.12.1. Contributing a property page

13.12.2. Implementing a property page

13.12.3. Support for displaying markers

13.12.4. Marker help and resolution

13.12.4.1. Contributing marker help

13.12.4.2. Contributing marker resolution

13.12.5. Contributing resource filters

13.12.6. Text file encoding

13.13. Status handling

13.13.1. Defining a status handler

13.14. Services

14. Editors

14.1. Workbench editors

14.2. Text editors and platform text

14.3. Documents and partitions

14.4. Source viewers and annotations

14.5. Configuring a source viewer

14.6. Text and ruler hover

14.7. Syntax coloring

14.8. Content assist

14.9. Registering editor actions

14.10. Other text editor responsibilities

14.11. Content outliners

15. Search support

15.1. Contributing a search page

15.2. Contributing a search result page

16. Compare support

16.1. Merging multiple files

16.2. Implementing a content viewer

16.3. Implementing a structure viewer

16.4. Advanced compare techniques

17. Team support

17.1. Repository providers

17.2. Resource management

17.3. Synchronization Support

17.3.1. Local History Synchronization Example

17.3.2. Beyond the basics

17.4. Support for Logical Model Integration

17.4.1. Repository Roadmap for Logical Model Integration

17.4.2. Model Roadmap for Logical Model Integration

17.5. Rich Team Integration

17.5.1. Adding team actions

17.5.2. Team decorators

17.5.3. Adding preferences and properties

17.5.4. Project sets

17.5.5. Team and linked resources

17.5.6. Team private resources

17.5.7. File types

17.5.8. Adding History support

18. Program debug and launch support

18.1. Launching a program

18.1.1. Adding launchers to the platform

18.1.1.1. Obtaining a program's source code

18.1.1.2. Comparing launch configurations

18.1.1.3. Process factories

18.1.1.4. Launching Java applications

18.1.2. Handling errors from a launched program

18.1.3. Launch configuration dialog

18.1.4. Launch configuration type images

18.1.5. Launch shortcuts

18.1.6. Launch groups

18.2. Debugging a program

18.2.1. Platform debug model

18.2.2. Breakpoints

18.2.3. Expressions

18.2.4. Debug model presentation

18.2.5. Debug UI utility classes

19. Platform Ant support

19.1. Running Ant buildfiles programmatically

19.2. Ant tasks provided by the platform

19.3. Contributing tasks and types

19.4. Developing Ant tasks and types within Eclipse

19.5. Expanding the Ant classpath

20. User assistance support

20.1. Welcome

20.1.1. Defining an intro part

20.1.1.1. Contributing a HelloWorld intro Part

20.1.2. Using the CustomizableIntroPart

20.1.2.1. Defining an intro config

20.1.2.2. Defining intro content

20.1.2.3. Defining a minimal intro configuration

20.1.2.4. Using XHTML as intro content

20.1.2.5. Displaying static HTML content in the CustomizableIntroPart

20.1.2.6. Extending an intro config

20.1.2.6.1. Extending the content of an intro config

20.1.2.6.2. Defining intro themes

20.1.2.6.3. Contributing a standby content part

20.1.2.6.4. Defining a custom IntroURL action

20.1.2.6.5. Intro Content File XML Format

20.1.2.6.6. SWT Properties for Intro

20.1.3. Universal intro

20.1.3.1. Contributing to universal intro

20.1.3.2. Adding the preference page

20.1.3.3. Extending the universal intro

20.1.3.4. Configuring product defaults

20.1.3.5. Managing links

20.2. Help

20.2.1. Configuration/setup

20.2.1.1. Rich Client Platform (RCP) help

20.2.1.2. Standalone help

20.2.1.3. Information center

20.2.1.4. Help system customization

20.2.1.4.1. Help data XML format

20.2.1.4.2. Generated navigation topics

20.2.1.4.3. The Help Menu

20.2.1.4.4. Information center customization

20.2.1.4.5. Using about.html to debug information centers

20.2.1.4.6. Using AbstractHelpScope to filter an information center

20.2.1.5. Pre-indexing documentation

20.2.2. Help content

20.2.2.1. Table of contents (toc) files

20.2.2.2. Help server and file locations

20.2.2.3. Completing the plug-in manifest

20.2.2.4. Adding criteria to help content

20.2.2.5. Building nested documentation structures

20.2.2.6. Contributing XHTML help documents

20.2.2.7. Remote Help

20.2.2.8. Active help

20.2.2.8.1. Writing the help action

20.2.2.8.2. Invoking the action from HTML

20.2.2.8.3. Tips for debugging active help

20.2.2.9. Embedding commands in help

20.2.2.9.1. Authoring a command link

20.2.2.10. Processing Help Content

20.2.2.11. Adding child links to help topics

20.2.2.12. Help Placeholders

20.2.3. Context-sensitive help

20.2.3.1. Declaring a context id

20.2.3.2. Describing and packaging context-sensitive help

20.2.3.3. Dynamic context help

20.2.3.4. Infopops

20.2.4. Help search

20.2.4.1. Plugging in search engines

20.2.5. Deploying the information center as a Web Archive

20.3. Cheat sheets

20.3.1. Simple cheat sheets

20.3.2. Cheat sheet content file XML format

20.3.3. Composite cheat sheets

20.3.4. Composite cheat sheet content file XML format

20.3.5. Cheat sheet authoring guidelines

20.4. Status handling

20.4.1. Defining a product status handler

20.4.2. Contributing a sample handler

20.5. Dynamic content

20.5.1. Filters

20.5.2. Includes

20.5.3. Extensions

21. Provisioning platform, p2

21.1. API Overview

21.2. Metadata management

21.2.1. Publishing metadata

21.2.2. Ant tasks for publishing metadata

21.2.3. Customizing metadata

21.2.4. Provisioning Actions and Touchpoints

21.3. Installation management

21.4. Repository management

21.4.1. Mirroring repositories

21.4.2. Composite repositories

21.4.3. Content categorization

21.4.4. Ant tasks for managing repositories

21.5. Customizing the p2 UI

21.5.1. Reusing parts of the UI

21.5.2. Configuring the UI Policy

21.5.3. Forced update on startup

22. Packaging and delivering Eclipse based products

22.1. Defining a Product

22.1.1. The products extension point

22.1.2. Customizing a product

22.2. Features

22.3. Plug-ins and fragments

22.4. Locale specific files

22.5. Product installation guidelines

22.6. Product extensions

22.7. Updating a product or extension

22.8. Deploying eclipse based application with Java Web Start

22.9. Associate and open files with an eclipse based application

23. Building a Rich Client Platform application

24. Http Service and JSP Support

1. 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.

1.1. 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.

1.2. The challenge

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.

1.3. 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 functionality 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 functionality for navigating and manipulating resources.

Help system

Defines extension points for plug-ins to provide help or other documentation as browsable books.

Team support

Defines a team programming model for managing and versioning resources.

Debug support

Defines a language independent debug model and UI classes for building debuggers and launchers.

Other utilities

Other utility plug-ins supply functionality 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.

2. 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 functionality to the system. Functionality 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.

2.1. 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 functionality 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.

3. 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.

3.1. 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...

3.2. 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.

Open the New Project... wizard ( File > New > Project...) and choose Plug-in Project from the Plug-in Development category and click Next.

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.

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.

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.

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.

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.

When asked if you would like to switch to the Plug-in Development perspective, answer Yes.

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.

3.3. 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.

3.4. 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="3.2"?>

<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.

3.5. 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.

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.

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.

3.6. 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.

4. 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.

4.1. 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 installed plug-ins and the functionality 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.

4.1.1. 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 functionality 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 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 plug-in 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.

4.1.2. 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.

4.2. Runtime components

The Eclipse runtime itself is factored into several bundles:

org.eclipse.core.contenttype - Content type mechanism

org.eclipse.core.jobs - Concurrency infrastructure

org.eclipse.core.net - Network and proxy management infrastructure

org.eclipse.equinox.app - Application Model

org.eclipse.equinox.common - Common basic functionality

org.eclipse.equinox.preferences - Runtime preferences

org.eclipse.equinox.registry - Extension registry

org.eclipse.equinox.security - Security infrastructure

org.eclipse.core.runtime - Aggregates the Eclipse runtime bundles into a single capability that can be depended on

If your plug-in has a dependency on the org.eclipse.core.runtime bundle, then it will continue to run unchanged. However, if you would like to minimize the number of dependencies, then you are able to pick only the runtime bundles that you use. Dependencies on the runtime bundles may be declared using either the Import-Package or Require-Bundle

If you would like to use Import-Package header (rather than Require-Bundle) then you should be aware that the org.eclipse.core.runtime package is split across several bundles. The bundles that export a portion of the split package org.eclipse.core.runtime use a mandatory directive. The mandatory directive requires importers to specify additional matching attributes in order to successfully wire to the exported package.

For example, to import only the content of the package provided by the org.eclipse.equinox.common bundle, use this line in the manifest file:

Import-Package: org.eclipse.core.runtime; common="split"

To import the content of the package from the org.eclipse.equinox.registry and org.eclipse.equinox.common bundles, use:

Import-Package: org.eclipse.core.runtime; registry="split"

The above examples use the mandatory matching attributes common and registry respectively. This allows the bundle to get resolved to the desired part of the split package. To import the complete package then do not use any matching attributes:

Import-Package: org.eclipse.core.runtime

This will resolve to the complete package as exported by the org.eclipse.core.runtime aggregate bundle.

4.3. 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", true, 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.

4.4. Runtime application model

The platform runtime provides an application container for controlling and executing applications. The runtime application container implements the Application Admin service specification included in the OSGi R4 specification. The application container is responsible for discovering all available applications and registering an ApplicationDescriptor OSGi service for each application that is available. An ApplicationDescriptor service can be used to launch an application. When an application is launched an ApplicationHandle OSGi service is registered to represent the instance of the running application. An ApplicationHandle service can be used to shutdown an application. See the org.osgi.service.application package and the OSGi R4.2 specification for more information.

The default application

A given Eclipse configuration may contain many products and applications. An Eclipse configuration specifies a default application which is launched by the application container as soon as the platform runtime is up and running. The default application can be specified by one of the following configuration options

eclipse.product - identifies the product to launch the platform runtime with. A product supplies branding information (window icons, title bar text etc.) as well as defining the default application to run.

eclipse.application - identifies the application to launch the platform runtime with. This option overrides default application defined by the product.

Defining an application

Plug-ins may define their own applications using the org.eclipse.core.runtime.applications extension. In this extension, the plug-in defines the name and ID of the application, as well a class that implements the application. The class which implements the application is used to launch and shutdown application instances.

4.5. 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.

4.5.1. 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 IContentTypes are related. The method IContentType.getBaseType() allows determining the base type of a given IContentType.

4.5.2. 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.AntBuildfileContentDescriber">

</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 tools (JDT) (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.

4.6. 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 Job API:

IJobManager jobMan = Job.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 = Job.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.

Completing jobs before shutdown

Since jobs run concurrently, it is possible for your jobs to be still running when the platform begins to shutdown. This is a dangerous situation, since after your plug-in stops it may not behave correctly or be able to load classes. For this reason, it is important for you to ensure all your jobs are canceled and completed in your plug-in's stop method. As in the previous example, you can use a job family to ensure all jobs scheduled by your plug-in are canceled and joined before your plug-in stops running.

4.6.1. 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 = Job.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.

4.6.2. 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.

4.6.3. 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 = Job.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 ((FileLock)rule).path.startsWith(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.

4.6.4. 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 = Job.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.

4.7. Network support

The networking support provided as part of the Platform includes:

Proxy support provided by the org.eclipse.core.net.proxy package of the org.eclipse.core.net plug-in.

SSH2 support provided by the com.jcraft.jsch plug-in. Clients should make use of the API in the org.eclipse.jsch.core package to ensure that the settings specified in the preference page are respected.

4.8.1. Secure storage

You can use secure storage to persist sensitive data (such as login credentials) in an encrypted form.

Secure storage takes care of a chicken-and-egg problem of "needing to get a password to encrypt a password" and organizes data in a hierarchical form. It provides a central secure repository for the application.

Secure storage is intended to store data that is relatively small in size. There is no hard cap limit at this time on the data size, but it is advisable to keep size of an entry under a kilobyte. If you do need to encrypt a large size object, consider creating a random key and using it with Java PBE encryption to encrypt your data. You can then store the key used to encrypt the object in secure storage. Later you can obtain the key from secure storage to decrypt the data.

The secure storage can be extended by supplying custom password provider modules and custom encryption algorithms. Multiple secure storages can be created and maintained at any time using different customizations.

This collection of topics describes how your code can utilize the secure storage. It assumes familiarity with the concepts described in the user documentation.

4.8.1.1. Secure Storage Architecture

From a logical perspective secure storage represents a hybrid of OSGi's Preferences and a keyring.

The front end is modeled in a fashion very similar to the OSGi Preferences Service specification. Secure storage is represented as a tree of nodes. Nodes provide context. For instance, a bundle "com.abc" could use the node "abc" under the node "com" under the root node. As in Preference's, the path to such node can be described as "/com/abc".

Picture 1. How the data is organized.

The data is stored under the node as a key-value pair. The data can be stored in an encrypted form or as a clear text. The ability to store clear-text data is provided so that logically related information (such as non-encrypted user name and encrypted password) can be stored using the same mechanism.

The code to store a password associated with "com.abc" could look like this:

...

ISecurePreferences root = SecurePreferencesFactory.getDefault();

ISecurePreferences node = root.node("/com/abc");

node.put("password", "12345", true /\*encrypt\*/);

...

and then to retrieve the data:

...

String password = node.get("password", null /\*default\*/);

...

If secure storage was modified, it will be saved when the application is closed. Alternatively, a save can be triggered programmatically by calling ISecurePreferences#flush().

On the back end, secure storage can be thought of as a keyring. Each entry is associated with a password provider; only this password provider can be used in the future to decrypt the entry.

Picture 2. Relationship between entries and password providers.

Secure storage has provisions to obtain the master password from various sources described by password providers. Even more importantly for developers, the set of password providers is open and easily customizable.

4.8.1.2. Developing with Secure Storage

Central Classes and Interfaces

The functionality of secure storage is exposed via APIs of the org.eclipse.equinox.security bundle. The org.eclipse.equinox.security.storage package contains several classes and interfaces that are likely going to serve as a starting point.

The SecurePreferencesFactory can be used to obtain the root node of the secure preferences. The SecurePreferencesFactory#getDefault() method will return default secure preference root node. If you prefer your secure storage to be isolated from the rest of the application, you can use SecurePreferencesFactory#open(URL location, Map options) method to open a secure storage corresponding to the location of your choice. This method also takes a map of optional hints that can be used to influence the behavior of this instance of secure storage.

The restrictions put on the node names are similar to the OSGi Preferences: no double forward slashes, no trailing forward slash. In addition, node names can only contain printable ASCII characters. If you need to use a string that does not fit those restrictions, the EncodingUtils class can be used to convert the string into a compliant form.

The functionality provided by the nodes is described by the ISecurePreferences interface. You'll find this interface is very similar to the OSGi Preferences with a few minor differences. One of the differences is that many methods use StorageException to provide more detailed description of exceptions.

Debugging

The contents of the default secure storage can be seen in the General > Security > Secure Storage preferences page. To enable modifications of the contents of secure storage, enable debug options of the org.eclipse.equinox.security.ui bundle. Enabling debug options will add context menus to the preferences page tree and to the values table. The context menus allow removal and addition of nodes and values to be performed directly from the preferences page.

You'll notice that context menus are disabled for the "org.eclipse.equinox.secure.storage" node. This node and its children are reserved for secure storage use and should not be modified directly.

4.8.1.3. Creating Password Providers

Extending Password Providers

When storing information (such as a CVS password) in the secure storage, the information gets encrypted with a master password. The interesting question about this process is how to store the master password?

The answer is in the password provider implementations. In a logical sense, password provider modules represent a trusted 3rd party, an external black box, which we trust to store our master password. That black box can be an operating system, some piece of hardware, or the user himself.

Several password provider modules are supplied with the SDK. They provide a basic "fallback" support and an improved support for some specific operating systems. For the most part you won't need to create a password provider - unless you'd like to add (or improve) integration with an operating system or add interaction with some specific hardware. In this case the set of password providers can be easily extended.

Declaring a Password Provider

The password provider modules have to be described as an extension to the extension point org.eclipse.equinox.security.secureStorage. The extension must specify an ID; the qualified ID of the extension will be used as an ID of the provider module and must not be changed between releases. The optional name and description of the password provider show up in the General > Security > Secure Storage preferences page and are visible to the end user.

Provider declaration includes the fully qualified class name that implements the provider, the numerical priority value, and a set of hints.

The priority is used to select a password provider when new data is added to the secure storage. The enabled provider with the highest priority is used by default. (Note that this can be overridden programmatically using IProviderHints#REQUIRED\_MODULE\_ID or by using the runtime option "-eclipse.password".)

The priority is an integer number from 0 to 10, with 10 being the highest priority. Consider adding your providers in the mid-range of priorities, 2 to 7, reserving top and bottom values.

An end user can disable a password provider via Secure Storage preferences page. Disabled providers are not considered when new data is added to the secure storage, but can be called on data retrieval if the data was originally encrypted with this provider.

Provider declaration can also specify a set of hints to help optimize interaction between the secure storage framework and the provider. The set of hints is open and can be extended in future. At the time of this writing the only expected hint is "AutomaticPasswordGeneration". This hint informs the framework that the password provider creates a master password without user interaction.

Implementing a Password Provider

The class implementing the password provider must extend the PasswordProvider class. The central point of that class is the getPassword() method that returns the master password for the provider for the current user. The method receives hints when a new password is expected to be created (as opposing to retrieving previously generated master password).

If your password provider has user interaction or has alternative paths to obtain the master password, you might consider overriding the PasswordProvider#retryOnError() method to inform the secure storage that the provider might be able to obtain a "better" password. For example, consider a password provider with the master password entered by the user. If the password entered was invalid, then the PasswordProvider#retryOnError() method will be consulted to see if the password provider can obtain a correct password.

4.8.2. Signed bundles and protecting against malicious code

Overview

The Eclipse platform faces a problem common to any program that can be extended by executable code: the potential that the extension code may be malicious. To address this concern, Eclipse embraces and extends the facilities built into the Java runtime in order to do two things:

Authenticate the source of executable code in bundles

Authorize the installation and runtime behaviour of code

Authenticating code packaged in a bundle

The first step towards protecting the platform from malicious code is 'authentication', or proving the identity of the person who is distributing the code in a bundle. This is facilitated by the use of a Public Key Infrastructure (PKI), where users are issued public certificates and private keys that uniquely identify them. Another common and well know use of certificates is the SSL protocol by which secured web sites are accessed.

The mechanism by which the authentication step is enabled is through the use of 'digital signatures' created by the code distributor, using a private key and a process called 'signing'. When consuming code and validating the identity of the code signer, the digital signatures are input - along with the signers public key - to an inverse process called 'verifying'.

In the Java platform, digital signatures are built into the Jar packaging format. A signed jar contains additional files alongside the META-INF\MANIFEST.MF that contain the information required to verify the signature, specifically a signature file (<SIGNERNAME>.SF) containing cryptographic digests of the resources contained in the Jar, and a block file (<SIGNERNAME>.RSA or <SIGNERNAME>.DSA) containing the signature data and associated certificates.

Using this information, one can verify that specific resources in a Jar file have not been tampered with and also establish the identity of the signer. The runtime then determines the 'trustedness' of the signer, the default behaviour being to check for the presence of the signer certificate (or one of its ancestor certificates) in the JRE's 'cacerts' file.

For more general information on PKI, Certificates, Digests, and Signatures:

http://en.wikipedia.org/wiki/Public\_key

http://en.wikipedia.org/wiki/Digital\_signature

http://en.wikipedia.org/wiki/Hash\_algorithm

For more specific information on signing in the Java platform:

http://download.oracle.com/javase/tutorial/deployment/jar/signing.html

http://www.ibm.com/developerworks/library/j-jar/index.html

http://www.onjava.com/pub/a/onjava/2001/04/12/signing\_jar.html?page=1

Signing code packaged in a bundle

In order to sign a bundle, an Eclipse developer has a handful of options:

The Java command line tool, 'jarsigner'

Jarsigner is a tool that ships with the Java SDK, and can be used to generate signatures on Jar files. For more information see the SDK documentation at:

http://download.oracle.com/javase/7/docs/technotes/tools/windows/jarsigner.html

A custom ant task, <signjar>

The 'Ant' build tool has a built-in task for automating the signing of code. For more information, see the related Ant documentation at:

http://ant.apache.org/manual/Tasks/signjar.html

The Eclipse 'export deployable plug-ins and fragments' functionality

Eclipse ships with the ability to export Eclipse plug-ins and sign them at the same time. When exporting deployable plug-ins and fragments, there is a JAR Signing tab available. In the tab, the exporter can specify a KeyStore (a password-protected key storage) containing a private key to use for the signing.

Verifying signed code packaged in a bundle

As with generation of signed bundles, verification of signed code can be done by an Eclipse developer in several ways:

The command line, 'jarsigner'

The Jarsigner tool is also capable of verifying signatures when passed a -verify option. For more information, consult the Java SDK documentation.

Programmatically via the core Java APIs.

There are several APIs in the Java API which allow interaction with signed Jars. For example, the java.util.JarFile API can be passed a parameter which enables verification of signatures. Several consumers of this API, including the URLClassLoader, pass this parameter to cause verification to occur. Unfortunately, little detail beyond the success or failure of verification is available through the core APIs.

Programmatically via the Equinox 'signedcontent' APIs

In 3.4, several interfaces for inspecting the signer details of a bundle have been introduced into the Equinox bundle (org.eclipse.osgi). Starting with the SignedContentFactory, a developer can programmatically inspect the entries in a bundle to see signers, timestamps and whether a code signer is trusted or not. For more information see the javadoc for the org.eclipse.osgi.signedcontent package of the org.eclipse.osgi bundle.

Authorization of signed code

Once the signer(s) of code packaged in a bundle is established via authentication, the next step to perform is 'authorization'. Authorization is the process by which the system decides whether a piece of code should be able to perform a specified action. Although the Java runtime supports a fine grained runtime permission model, Eclipse also supports two additional enforcement points with respect to signed code packaged in bundles. This gives Eclipse platform deployers a range of security solutions that allow tradeoffs to be made between flexibility and complexity versus manageability and performance. In Eclipse 3.4 or later, authorization based on signatures can be performed:

When code is installed by the provisioning system

For several previous releases and in the new P2 provisioning framework, Eclipse has the ability to check signatures as bundles are provisioned into the system. As the provisioning system encounters bundles, it automatically performs authentication of the code signer and will prompt if a signer is not trusted according to the system configuration. The end user will be presented with a list of untrusted signers, and choosing to trust will allow the bundles to be installed into the platform.

When code is loaded by the runtime

Since 3.4, the Equinox runtime has had the ability to check the signature of code as it is loaded. The benefit to this feature beyond checking signatures during provisioning is the ability to dynamically remove trust and disable code should an exploit be exposed in deployed code. In order to enable signature-based authorization at load time, the following VM argument must be passed:

-Dosgi.signedcontent.support=authority

See the runtime options page for more information about the osgi.signedcontent.support runtime variable.

For more information on security topics related to Eclipse and Equinox, see the site:

http://www.eclipse.org/equinox/security

4.8.3. Login based on JAAS

Overview

The Java Authentication and Authorization Service (or JAAS) is a login framework available in the Java runtime since version 1.4. It allows developers to write security-agnostic code that allows for pluggable backends for authentication (consider LDAP vs Smartcard vs Operating system, etc). There are several implementations of login backends that ship with the various JREs.

For more information on JAAS:

http://download.oracle.com/javase/7/docs/technotes/guides/security/jaas/JAASRefGuide.html

Extension-point based contribution

JAAS has specific expectations of where the classes that are used to build a LoginContext should be found - typically the extension classloader. In order to get around these limitations and bridge into the Eclipse environment, there are several extension points available for bundles to contribute JAAS artifacts into the system. Specifically, the following artifacts have corresponding extension points in the org.eclipse.equinox.security bundle:

javax.security.auth.Configuration

javax.security.auth.spi.LoginModule

javax.security.auth.callback.CallbackHandler

In order to evolve the API but still embrace the model, there is an org.eclipse.equinox.security.auth.ILoginContext interface which maps to the LoginContext class in JAAS. Additional features are available on this interface, but most functionality still speaks in terms of core JAAS artifacts (Subjects, Principals, etc).

Declarative wiring via LoginContextFactory

Another limitation of the JAAS framework is that there is a strong coupling between the backend login Configuration and the CallbackHandler (UI) which will service the login execution. There is an extension to create the mapping from a configuration name to a specific CallbackHandler, and a factory (org.eclipse.equinox.security.auth.LoginContextFactory) which can generate an ILoginContext without knowledge required of specific CallbackHandlers.

Notifications through ILoginContextListener

Another use case which is implemented is the ability to monitor the execution of an applications login lifecycle via an event listener model. This functionality is available through the org.eclipse.equinox.security.auth.ILoginContextListener interface.

Future work

Feedback surrounding this functionality is greatly appreciated. Potential future direction involves integrating login into the RCP lifecycle, Jobs framework, etc and using the user context to affect the behaviour of the runtime system (filter, etc).

4.9. Console Shell

General features

The Equinox OSGi console is based on Apache Felix Gogo shell. The console provides:

support for the ssh protocol (in addition to the telnet connectivity)

JAAS-based user authentication

improved tab completion

The default authentication is password-based, but key-based authentication is also supported.

Installation and configuration of the console

Running without SSH

Place the necessary bundles in a folder. The bundles are:

org.apache.felix.gogo.command\_<version>.jar

org.apache.felix.gogo.runtime\_<version>.jar

org.apache.felix.gogo.shell\_<version>.jar

org.eclipse.equinox.console\_<version>.jar

org.eclipse.osgi.jar

Create a configuration subfolder and a config.ini file in it.

Add the following entries in the config.ini file:

osgi.bundles=org.apache.felix.gogo.runtime,\

org.apache.felix.gogo.command,\

org.apache.felix.gogo.shell,\

org.eclipse.equinox.console

osgi.console=<port>

Start the Equinox framework with the following command line: java -jar org.eclipse.osgi.jar

The property osgi.console specifies the port on which the console should listen for telnet connections. Instead of specifying this property, you can start the framework with the -console <port> option. If you do not specify the port in either of these ways, the console will not listen for telnet connections. If you start with the option -console and do not specify the port, you will be able to interact with the console through the standard input.

Running with SSH

If you want to access the console through SSH, you need more bundles and configurations.

Place the necessary bundles in a folder. The bundles are:

org.apache.felix.gogo.command\_<version>.jar

org.apache.felix.gogo.runtime\_<version>.jar

org.apache.felix.gogo.shell\_<version>.jar

org.eclipse.equinox.console\_<version>.jar

org.eclipse.equinox.console.ssh\_<version>.jar

org.eclipse.equinox.console.jaas.fragment\_<version>.jar

org.eclipse.osgi.jar

org.apache.mina.core\_<version>.jar

org.apache.sshd.core\_<version>.jar

slf4j-api

some slf4j-api implementation

Create a configuration subfolder and a config.ini file in it.

Add the following entries in the config.ini file:

osgi.bundles=org.apache.felix.gogo.runtime,\

org.apache.felix.gogo.command,\

org.apache.felix.gogo.shell,\

org.eclipse.equinox.console,\

slf4j-api,\

<slf4j-api\_impl>,\

org.apache.mina.core,\

org.apache.sshd.core,\

org.eclipse.equinox.console.ssh@start,\

org.eclipse.equinox.console.jaas.fragment

osgi.console.ssh=<port>

osgi.console.ssh.useDefaultSecureStorage=true

Create a file with name org.eclipse.equinox.console.authentication.config in the configuration subfolder.

Add the following entry to org.eclipse.equinox.console.authentication.config file:

equinox\_console {

org.eclipse.equinox.console.jaas.SecureStorageLoginModule REQUIRED;

};

Create logging configuration file. This depends on the particular implementation of slf4j-api, which you have chosen.

Start the Equinox framework with the following command line: java -Djava.util.logging.config.file=configuration/logging.properties \

-Dssh.server.keystore=configuration/hostkey.ser \

-Dorg.eclipse.equinox.console.jaas.file=configuration/store \

-Djava.security.auth.login.config=configuration/org.eclipse.equinox.console.authentication.config \

-jar org.eclipse.osgi.jar

Notes:

For both properties, specifying the telnet and ssh ports, you can also specify the host:

osgi.console=<host>:<port>

osgi.console.ssh=<host>:<port>

where <host> is either localhost or 127.0.0.1, so that the console listens for telnet connections only from the localhost.

You can use both telnet and ssh. For this you have to specify both properties.

The version of slf4j-api must be in the range [1.5,2]

The property osgi.console.ssh.useDefaultSecureStorage is necessary only if ssh is used and the default login mechanism is used (i. e., a custom login module is not provided)

The configuration file org.eclipse.equinox.console.authentication.config may have a different name, but its name must be given as a value of the property -Djava.security.auth.login.config accordingly.

The bundles org.eclipse.equinox.console\_<version>.jar, org.eclipse.equinox.console.ssh\_<version>.jar and org.eclipse.equinox.console.jaas.fragment\_<version>.jar, as well as the three Gogo bundles are available in Equinox SDK, which can be downloaded from the Equinox download page

You can download the bundles org.apache.mina.core\_<version>.jar and org.apache.sshd.core\_<version>.jar Eclipse Orbit repository.

You can download slf4j-api and an slf4j implementation from here.

For logging you can use instead org.slf4j.api from Eclipse Orbit repository. In this case you should take from this repository the logback slf4j implementation. It consists of three bundles - ch.qos.logback.classic, ch.qos.logback.core and ch.qos.logback.slf4j.

All configuration files, excluding config.ini, may be placed in a folder, different from the configuration subfolder. In this case the VM properties must be changed respectively to point to the correct folder.

JAAS Authentication

The Equinox console uses JAAS authentication when the SSH is used. The default JAAS login module, implemented in the console, uses custom store file, where it stores users and passwords. The passwords are one-way encrypted. The console also provides shell commands for administering users:

add user

delete user

list users

reset user password

change user password

add user roles

remove user roles

There is a default user equinox with password equinox, which is dynamically created when the ssh starts if no other user exists in the user store. Upon first login, the console prompts the user to create a new user, and automatically deletes the default user. The password must be at least 8 symbols long. The username may contain alphanumerical characters, plus underscore and dot. Currently only authentication is implemented in the console. All authenticated users have similar rights. Implementing authorization may be a future enhancement. That is why roles are introduced, although they are not currently used. If the default JAAS login module is be used, then the property osgi.console.ssh.useDefaultSecureStorage must be set to true. The file, where the users are stored, is specified through the property org.eclipse.equinox.console.jaas.file

Using Custom JAAS Authentication Login Module

Custom JAAS authentication login modules could be used with the ssh instead of the default one. If a custom login module is used, then the property osgi.console.ssh.useDefaultSecureStorage must not be set at all. The custom module must be specified in the file org.eclipse.equinox.console.authentication.config instead of the default entry there, or in a different file. If a different file is used, then its name must be specified as the value of the property -Djava.security.auth.login.config instead of org.eclipse.equinox.console.authentication.config. Also, it will be necessary to create a fragment to sshd-core bundle, with which to import the package of the custom login module. This is necessary for the sshd to be able to load the module class.

Key-based Authentication

Instead of using JAAS authentication, the console can use key-based user authentication with ssh. It is possible to provide a file containing a list of keys that are allowed to connect to the SSH console. The file will be read every time a connection is made so that it can be modified dynamically at runtime. The file is specified with the property ssh.server.authorized\_keys. If you use key-based authentication you do not need to specify the JAAS-related properties java.security.auth.login.config and osgi.console.ssh.useDefaultSecureStorage.

The console supports also custom public keys authentication. If no specific authorized\_keys file is configured (via system property) the OSGi service registry will be searched for available authenticators. To enable this feature you have to set the property ssh.custom.publickeys.auth to true.

Configuring of telnet and ssh host and port through Configuration Admin service

The console provides the option to configure the telnet and ssh host and port through Configuration Admin instead of through the properties osgi.console=<host>:<port> and osgi.console.ssh=<host>:<port>. This could be helpful in more complex scenarios, for example when you want to run different instances of the console in different subsystems of the framework. In this case if the port is configured through a system property, it is one and the same for all console instances and only one will be able to bind to the socket. If this feature is used, then you should:

set the property osgi.console.useConfigAdmin to true

install and start, along with the console bundles, the org.eclipse.equinox.cm bundle and its dependencies

write a bundle to provide the configuration with the telnet and ssh properties

The configuration PID for the telnet configuration is osgi.console.telnet and for the ssh configuration is osgi.console.ssh. Both configurations have the following properties:

host

port

enabled

All must have values of type String. The enabled property determines if telnet/ssh is to be started at all. If the value is true, telnet/ssh is started. If the value is false, or the property enabled is absent, the telnet/ssh is not started.

Using Commands

The Gogo shell, on which the Equinox console is based, supports different type of commands from these currently supported in Equinox. Currently in Equinox, commands are provided by a class implementing the CommandProvider interface. The Equinox console provides an adapter from this type of commands to the new type of commands, used in Gogo. They commands are arbitrary classes, registered as services, with two special properties:

osgi.command.scope - specifies the scope of the command

osgi.command.function - specifies the commands provided by this service; this is a list of the names of public methods in the service class, which must be available for execution as commands

The commands in the Equinox console use the notion of command scope. The scope can be used, for example, to differentiate between commands with one and the same name, but provided by different providers. Then they should have different scopes.

The scope is a prefix of the command name, separated from it with a column. When writing the command in the console, the scope may or may not be specified. If the scope is not specified, then the command with this name from the default scope is used. If in the default scope there is no command with such name, all commands are searched.

If there is more than one command with the specified name in different scopes, it is not guaranteed which one will be actually executed. Therefore, if you know that there are commands with the same name but different scopes, you must explicitly prefix the command name with the scope to ensure that you will get executed exactly the command you want.

The legacy commands, adapted by the Equinox console, have scope equinox.

Getting Help for Commands

The default help command in Gogo lists the names of the registered commands, and if it is called with a command name as an argument, it displays help message for the specified command. The default help command does not provide help for the legacy Equinox commands, which are adapted by the Equinox console. For this reason the Equinox console provides its own help command, which calls the default help, but could also provide help for the legacy commands.

If no argument is specified the names of all new commands are displayed.

If a command name is specified, the help message for this command is displayed. If the command is legacy and the CommandProvider, which provides this command, does not provide separate help for each command (this is a new feature, introduced in Equinox 3.7), then the help messages for all commands in this CommandProvider are displayed.

The help command supports -scope argument, which should be used as follows: help -scope <command\_scope>. Thus the help command will display help only for the commands with the specified scope.

There is also a man command. It accepts the same arguments as the help command and has the same semantics.

Writing Gogo Commands

Writing Gogo commands is easy. You have to do the following:

Write your class with all commands you want implemented as public methods. Methods may have arbitrary arguments. Also, unlike Equinox command providers, in RFC 147 there is no interface that the class of the command provider should implement.

Register the command provider as a service with the two properties, described above.

Something new for Gogo commands are the converters and formatters.

The converter is a class which converts the arguments, passed on the command line, to the actual arguments, which the command accepts. For example, the command may have one argument of type Bundle. There may be a converter, which from a long finds the bundle with this id. Then the command may be called with the ID of the bundle as an argument, the converter will convert it to the corresponding Bundle object and the command method will be called with this object as an argument.

The formatter is a class which displays the result, returned by the command method.

For more information on Gogo commands, you can check the Gogo documentation.

Tab Completion

The console provides tab completion. This feature is available only when connecting through telnet or ssh.

Completion is available for:

command names

variable names - these must be previously defined in the session; variable names may be passed as command arguments

file names - passed as command arguments

When the tab key is typed, all possible candidates for completion for the current word are searched. If there is only one possible completion, the current word is automatically completed. If there are more than one options, all are displayed in a column. The possible options can be iterated by hitting tab multiple times, until the desired completion candidate is reached.

If longest common prefix of all available completion candidates is longer than the current word, then the current word is completed automatically to this prefix, before choosing the final completion. For example, if we have the following available completions for the word bun:

bundles

bundle

bundlelevel

the current word is completed automatically to bundle, and then by hitting tab you can choose which completion you like, or may type it for yourself.

The console allows custom command completers to be provided. A custom completer should implement the interface org.eclipse.equinox.console.common.Completer, provided by the bundle of the Equinox console. It has the single method getCandidates, which get as a parameter the whole command line and the current cursor position in it, and returns a map with all completion candidates, and the position in the command line, on which the completion begins. The keys in the map are the candidates, and the values - the start position for the completion.

Command Line Editing

Both with telnet and ssh command line editing is available in addition to tab completion. The following functions are supported:

backspace - deletes the character to the left of the cursor

delete - deletes the character on the cursor position

home/end - moves the cursor to the beginning/end of the command line

left/right arrow - moves the cursor one character left/right

up/down arrow - moves backward/forward one entry in the command history

pageup/pagedown - moves to the first/last entry of the command history

These functions are supported also when the the console is run standalone with Equinox. In PDE these features are not supported.

Closing Telnet or Ssh Session

Both telnet and ssh sessions are closed with the disconnect command.

Running Non-interactive Shell

In some cases you may wish to run the console without an interactive session. In this case, the console can be connected from telnet and ssh, but is not available on standard in and out. For this the following property should be specified: -Dgosh.args=--nointeractive

4.10. Structured Text Overview

Languages like Arabic and Hebrew are generally written from right to left, but included numbers and phrases in English must be written from left to right. This is the origin of the term "bidirectional" which qualifies these languages.

In most computer environments, the text is stored in logical order (the order the text is read) but is reordered into visual order for presentation. For plain text, the Unicode Bidirectional Algorithm (UBA) generally specifies satisfactorily how to reorder bidirectional text for display. This algorithm, or close to it, is implemented in the presentation systems of a number of platforms, giving them a good handle on bidirectional support.

However, all bidirectional text is not necessarily plain text. There are also instances of text structured to follow a given syntax, which should be reflected in the display order. The general algorithm, which has no awareness of these special cases, often gives incorrect results when displaying such structured text.

This document describes various examples of this issue, and proposes a methodology to solve the related problems. The types of structured text treated in this document are all excerpted from actual products, including Eclipse.

For a general introduction to bidirectional concepts, the reader is kindly referred to the following technical article: "Bidirectional script support: a primer" available at http://www-128.ibm.com/developerworks/websphere/library/techarticles/bidi/bidigen.html.

Working with Structured Text

Eclipse provides support for correct presentation of structured text. The base functionality can be found in the package "org.eclipse.equinox.bidi". The methods process() and deprocess() and their variants in class STextProcessor will be a good starting point.

Consumers who would like to adjust Bidi environment settings, process bidi text in portions, or directly manage directional formatting will find lower level functionality available in the "org.eclipse.equinox.bidi.advanced" package.

Developers wishing to create handlers for types of structured text not currently supported by Eclipse "out of the box" will create extensions for class STextTypeHandler in package "org.eclipse.equinox.bidi.advanced". Such new type handlers should register with the "org.eclipse.equinox.bidi.bidiTypes" extension point.

4.10.1. Design Overview

Terms and Abbreviations

Bidi

Bidirectional

LTR

Left to Right

RTL

Right to Left

LRM

Left-to-Right Mark

RLM

Right-to-Left Mark

LRE

Left-to-Right Embedding

RLE

Right-to-Left Embedding

PDF

Pop Directional Formatting

General Definitions, Terminology and Conventions

Every instance of bidi text has a base text direction. Bidi text in Arabic or Hebrew has a RTL base direction, even if it includes numbers or Latin phrases which are written from left to right. Bidi text in English or Greek has a LTR base direction, even if it includes Arabic or Hebrew phrases which are written from right to left.

Structured expressions also have a base text direction, which is often determined by the type of structured expression, but may also be affected by the content of the expression (whether it contains Arabic or Hebrew words).

This document addresses two groups of problematic cases:

Expressions with simple internal structure: this category regroups cases in which strings are concatenated together in simple ways using known separators. For example: variable names, "name = value" specifications, file path, etc...

Expressions with complex internal structure: this category regroups structured text like regular expressions, XPath expressions and Java code. This category differs from the previous one since the expressions belonging to it have a unique syntax which cannot be described by concatenation of string segments using separators.

We will see that the same algorithms can handle both groups, with some adaptations in the details.

In the examples appearing in this document, upper case Latin letters represent Arabic or Hebrew text, lower case Latin letters represent English text.

"@" represents an LRM, "&" represents an RLM.

Notations like LRE+LRM represent character LRE immediately followed by character LRM.

Bidirectional Control Characters

When there are problems of wrong display of bidi text, it is often possible to cure them by adding some bidi control characters at appropriate locations in the text. There are 7 bidi control characters: LRM, RLM, LRE, RLE, LRO, RLO and PDF. Since this design has no use for LRO and RLO (Left-to-Right and Right-to-Left Override, respectively), the following paragraphs will describe the effect of the 5 other characters.

LRM (Left-to-Right Mark): LRM is an invisible character which behaves like a letter in a Left to Right script such as Latin or Greek. It can be used when a segment of LTR text starts or ends with characters which are not intrinsically LTR and is displayed in a component with a RTL orientation.

Example: assume in memory the string "\\myserver\myshare(mydirectory)". We want it displayed identically, but within a component with RTL orientation it would be displayed as "(myserver\myshare(mydirectory\\". Adding one LRM character at the beginning of the string will cause the leading backslashes to be displayed on the left side, and adding one LRM character at the end of the string will cause the trailing parenthesis to be displayed on the right side.

RLM (Right-to-Left Mark): RLM is an invisible character which behaves like a letter in a Right to Left script like Hebrew. It can be used when a segment of RTL text starts or ends with characters which are not intrinsically RTL and is displayed in a component with a LTR orientation.

Example: assume in memory the string "HELLO WORLD !". We want it displayed as "! DLROW OLLEH", but within a component with a LTR orientation it would be displayed as "DLROW OLLEH !" (exclamation mark on the right side). Adding one RLM character at the end of the string will cause the trailing exclamation mark to be displayed on the left side.

LRE (Left-to-Right Embedding): LRE can be used to give a base LTR direction to a piece of text. It is most useful for mixed text which contains both LTR and RTL segments.

Example: assume in memory the string "i love RACHEL and LEA" which should be displayed as "i love LEHCAR and AEL". However, within a component with RTL orientation, it would be displayed as "AEL and LEHCAR i love". Adding one LRE character at the beginning of the string and one PDF (see below) character at the end of the string will cause proper display.

RLE (Right-to-Left Embedding): RLE can be used to give a base RTL direction to a piece of text. It is most useful for mixed text which contains both LTR and RTL segments.

Example: assume in memory the string "I LOVE london AND paris" which should be displayed as "paris DNA london EVOL I". However, within a component with LTR orientation, it would be displayed as "EVOL I london DNA paris". Adding one RLE character at the beginning of the string and adding one PDF (see below) character at the end of the string will cause proper display.

PDF (Pop Directional Formatting): PDF may be used to limit the effect of a preceding LRE or RLE. It may be omitted if not followed by any text.

Note that pieces of text bracketed between LRE/PDF or RLE/PDF can be contained within larger pieces of text themselves bracketed between LRE/PDF or RLE/PDF. This is why the "E" of LRE and RLE means "embedding". This could happen if we have for instance a Hebrew sentence containing an English phrase itself containing an Arabic segment. In practice, such complex cases should be avoided if possible. The present design does not use more than one level of LRE/PDF or RLE/PDF, except possibly in regular expressions.

Bidi Classification

Characters can be classified according to their bidi type as described in the Unicode Standard (see Bidirectional\_Character\_Types for a full description of the bidi types). For our purpose, we will distinguish the following types of characters:

"Strong" characters: those with a bidi type of L, R or AL (letters in LTR or RTL scripts);

Numbers: European Numbers (type EN) or Arabic Numbers (type AN);

Neutrals: all the rest.

Text Analysis

In all the structured expressions that we are addressing, we can see characters with a special syntactical role that we will call "separators", and pieces of text between separators that we will call "tokens". The separators vary according to the type of structured expression. Often they are punctuation signs like colon (:), backslash (\) and full stop (.), or mathematical signs like Plus (+) or Equal (=).

Our objective is that the relative progression of the tokens and separators for display should always follow the base text direction of the text, while each token will go LTR or RTL depending on its content and according to the UBA.

For this to happen, the following must be done:

Parse the expression to locate the separators and the tokens.

While parsing, note the bidi classification of characters parsed.

Depending on the bidi types of the characters before a token and in that token, a LRM or a RLM may have to be added. The algorithm for this is detailed below.

If the expression has a LTR base direction and the component where it is displayed has a RTL orientation, add LRE+LRM at the beginning of the expression and LRM+PDF at its end.

If the expression has a RTL base direction and the component where it is displayed has a LTR orientation, add RLE+RLM at the beginning of the expression and RLM+PDF at its end.

The original structured expression, before addition of directional formatting characters, is called lean text.

The processed expression, after addition of directional formatting characters, is called full text.

LRM Addition (structured text with LTR base text direction)

A LRM will be added before a token if the following conditions are satisfied:

The last strong character before the token has a bidi type equal to R or AL and the first non-neutral character in the token itself has a bidi type equal to R, AL, EN or AN.

Examples (strings in logical order where "@" represents where an LRM should be added):

HEBREW @= ARABIC

HEBREW @= 123

OR

The last non-neutral character before the token has a bidi type equal to AN and the first non-neutral character in the token has a bidi type equal to R, AL or AN.

Examples (strings in logical order where "@" represents where an LRM should be added):

ARABIC NUMBER 123 @< MAX

ARABIC NUMBER 123 @< 456

RLM Addition (structured text with RTL base text direction)

A RLM will be added before a token if the following conditions are satisfied:

The last strong character before the token has a bidi type equal to L and the first non-neutral character in the token itself has a bidi type equal to L or EN.

Example (string in logical order where "&" represents where an RLM should be added):

my\_pet &= dog

4.10.2. Supported Text Types

Out-of-the-box we include support for the following text types:

Comma-delimited List

Full Path - Relative Path - File Name

Java Code

Regular Expression

Compound Name with Underscores

URL, URI, IRI

XPath

Developers can add custom text types by contributing to the org.eclipse.equinox.bidi.bidiTypes extension point.

Unless specified otherwise, we assume that the relative progression of the tokens and separators for display should always be from left to right, while the text of each token will go LTR or RTL depending on its content and according to the Unicode Bidirectional Algorithm.

(In the examples, "@" represents an LRM, "&" represents an RLM.)

Comma-delimited List (comma)

Pattern

[first list item] , [second list item] , . . . , [last list item]

Detailed Design

The general algorithm applies, with the following adaptations:

There is only one separator, the comma(,).

This design can easily be adapted to accomodate a different separator, like a semicolon (;) or a tab character, etc...

Example:

Logical order (without LRM): ABC,DE,FGH

Display (without LRM): HGF,ED,CBA

Logical order (with LRM): ABC@,DE@,FGH

Display (without LRM): CBA,ED,HGF

Name or Path of File or Directory

Patterns

Windows full path: [drive letter]:\ [sub-path] \ . . . \ [sub-path]

Windows relative path: [sub-path] \ . . . \ [sub-path]

Windows full file path: [drive letter]:\ [sub-path] \ . . . \ [sub-path] \ [file name] . [extension]

Windows relative file path: [sub-path] \ . . . \ [sub-path] \ [file name] . [extension]

Linux full path: / [sub-path] / . . . / [sub-path]

Linux relative path: [sub-path] / . . . / [sub-path]

Linux full file path: / [sub-path] / . . . / [sub-path] / [file name] . [extension]

Linux relative file path: [sub-path] / . . . / [sub-path] / [file name] . [extension]

Detailed Design

The general algorithm applies, with the following adaptation:

The separators are colon (:), backslash (\) and full stop (.) for Windows, slash (/) and full stop (.) for Linux.

Example:

Logical order (without LRM): c:\DIR1\DIR2\MYFILE.ext

Display (without LRM): c:\ELIFYM\2RID\1RID.ext

Logical order (with LRM): c:\DIR1@\DIR2@\MYFILE.ext

Display (without LRM): c:\1RID\2RID\ELIFYM.ext

Java Code

Requirement

We can classify elements of a Java program as:

white space

operators

String literals: they start with a double quote and end with a double quote which is not escaped (not preceded by a backslash).

comments: they start with /\* and end with \*/ or start with // and end at the end of the line.

tokens: anything delimited by the previous items.

The requirement is to make the relative order of elements left-to-right, while each element by itself will be presented according to the Unicode Bidirectional Algorithm.

Detailed Design

The general algorithm applies, with the following adaptations:

Each String literal or comment is considered as one token.

The separators are all the characters used as operators and separators in the Java language: plus (+), minus (-), asterisk (\*), slash (/), percent (%), less-than (<), greater-than (>), ampersand (&), vertical bar (|), circumflex (^), tilde (~), left and right parentheses ( ( ) ), left and right square brackets ([ ]), left and right curly brackets ( { } ), comma (,), full stop (.), semicolon (;), exclamation mark (!), question mark (?), colon (:), spaces which are not part of a String literal or a comment.

If a String literal or a comment includes LRE or RLE characters but do not include the proper number of matching PDF characters, missing PDF characters must be added at the end of the literal or comment.

Example:

Logical order (without LRM): A = /\*B+C\*/ D;

Display (without LRM): D /\*C+B\*/ = A;

Logical order (with LRM): A@ = /\*B+C@\*/ D;

Display (without LRM): A = /\*C+B\*/ D;

Regular Expression

Requirement

Preserve the relative order of the regular expression components identical to the order in which they appear when exclusively Latin characters are used.

Detailed Design

The general algorithm applies, with the following adaptations:

Regular expressions consist of operators, pattern characters, and " in most implementations of extended syntax " named identifiers.

Since the syntax of regular expression is not standardized, the list of operators should be adapted to the specific implementation at hand.

Common operators include: question mark (?), circumflex (^), dollar ($), plus (+), minus (-), asterisk (\*), vertical bar (|), tilde (~), left and right parentheses ( ( ) ), left and right square brackets ([ ]), left and right curly brackets ( { } ), commercial at (@), number sign (#), ampersand (&), backslash (\).

The separators will be the characters used as operators for regular expressions.

Characters which are not operators are pattern characters. If an operator is immediately preceded by a backslash, both the backslash and the operator must be handled as pattern characters.

Each pattern character is a separate token, so pattern characters will always be ordered according to the base text direction of the expression.

Identifiers appear in certain syntactic constructs, and are treated as tokens. For example, the strings "digit" and "number" in the expression "total: (?<number>[:digit:]+)\s" are identifiers, whereas "total" is just a sequence of 5 pattern characters.

The following constructs must be recognized as delimiting tokens (note: this list should be adapted to the specific syntax of regular expressions in a given environment):

(?<name>

(?'name'

(?(<name>)

(?('name')

(?(name)

(?&name)

(?P<name>

\k<name>

\k'name'

\k{name}

(?P=name)

\g{name}

\g<name>

\g'name'

(?(R&name)

[:class:]

Comments of the form (?# . . . ) must be handled as individual tokens.

Quoted sequences of the form \Q . . . \E must be handled as individual tokens.

Numbers used as quantifiers (numbers of occurrences) or as group references must be handled as individual tokens.

If the first strong directional character in a regular expression is an Arabic letter, the base direction of the expression must be RTL.

If the first strong directional character in a regular expression is a Hebrew letter or a LTR letter, the base direction of the expression must be LTR.

If the regular expression contains no strong directional character, its base direction must be LTR for Hebrew users. For Arabic users, its base direction should follow the user interface direction (RTL if mirrored, LTR otherwise).

Example (Hebrew):

Logical order (without LRM): ABC(?'DEF'GHI

Display (without LRM): IHG'FED'?(CBA

Logical order (with LRM): A@B@C@(?'DEF'@G@H@I

Display (without LRM): ABC(?'FED'GHI

Example (Arabic):

Logical order (without LRM): ABC(?'DEF'GHI

Display (without LRM): IHG'FED'?(CBA

Logical order (with LRM): ABC(?'DEF'GHI

Display (without LRM): IHG'FED'?(CBA

Compound Name with Underscores

Pattern

[first part] \_ [second part] \_ [third part]

Note: name parts must not include underscores.

Detailed Design

The general algorithm applies, with the following adaptation:

There is only one separator, the underscore (\_).

Example:

Logical order (without LRM): MYPACKAGE\_MYPROGRAM

Display (without LRM): MARGORPYM\_EGAKCAPYM

Logical order (with LRM): MYPACKAGE@\_MYPROGRAM

Display (without LRM): EGAKCAPYM\_MARGORPYM

URL, URI, IRI

Patterns

http:// [domain label] . . . . . [domain label]

http:// [domain label] . . . . . [domain label] / [sub-path] / . . . / [sub-path] / [file name] . [extension]

http:// [domain label] . . . . . [domain label] / [sub-path] / . . . / [sub-path] / [file name] . [extension] # [local reference]

http:// [domain label] . . . . . [domain label] / [sub-path] / . . . / [sub-path] / [file name] . [extension] ? [key1] = [value1] & [key2] = [value2]

Detailed Design

The general algorithm applies, with the following adaptations:

The detailed syntax of URLs, URIs, IRIs is described in RFC 3986 and RFC 3987. A rigorous analysis to identify tokens and separators is not simple.

For most practical cases, it is sufficient to consider the following separators: colon (:), question mark (?), number sign (#), slash (/), commercial at (@), full stop (.), left bracket ([), right bracket (]).

Example:

Logical order (without LRM): www.DOC.MYDOMAIN.com\HEB\LESSON1.html

Display (without LRM): www.NIAMODYM.COD.com\1NOSSEL\BEH.html

Logical order (with LRM): www.DOC@.MYDOMAIN.com\HEB@\LESSON1.html

Display (without LRM): www.COD.NIAMODYM.com\BEH\1NOSSEL.html

XPath

Patterns

/ book / chapter / paragraph

/ year / month [@name = "April"]

Detailed Design

The general algorithm applies, with the following adaptations:

Strings

Strings are started by a quotation mark which can be a double-quote (") or an apostrophe ('), and are closed by the same character.

Double-quotes may appear within a string limited by apostrophes and vice versa, and must be handled as characters internal to the string.

A string started on one line is not necessarily closed on the same line.

Whitespace (e.g. blanks and tab characters) appearing outside of strings constitutes a delimiter for tokens.

Each occurrence of a string must be handled as one token.

After isolating strings, the following characters are separators: white space, slash (/), square brackets ( [ and ] ), less-than (<), greater-than (>), equal sign (=), exclamation mark (!), colon (:), at sign (@), period (.), vertical bar (|), parentheses ( ( and ) ), plus (+), minus (-), asterisk (\*).

Some operators are words like "and", "or", "div", "mod". For our purpose, they can be handled as tokens.

Some operators are represented by a pair of symbols like "not equal" (!=), "descendant-or-self" (//), "parent" (..). For our purpose, they can be handled as 2 successive operators represented by one symbol each.

Example:

Logical order (without LRM): DEF!GHI 'A!B'=JK

Display (without LRM): KJ='B!A' IHG!FED

Logical order (with LRM): DEF@!GHI@ 'A!B'@=JK

Display (without LRM): FED!IHG 'B!A'=KJ

5. 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 functionality 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 project explorer 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

5.1. 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 project explorer 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().

5.2. Basic workbench extension points using commands

Commands and handlers have been provided by the workbench in one form or another since 3.0. In 3.2 the commands, handlers, and keybindings portion of the Command Framework became mature. In 3.3 the menu contributions portion of the Command Framework became available.

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'll use the Info example from the contributions plugin. It provides a view that lists people and an editor for updating their names.

We are going to take a look at how commands, handlers, menu contributions, and keybindings can be used to contribute behaviour to the workbench, to views, and to editors.

You can show any view in the workbench by choosing Window > Show View > Other... and selecting the view from the Show View list. Use this to select the Info View from the View Contributions category.

The Info example is located in the org.eclipse.ui.examples.contributions package. To follow along, you will need to make sure that you have installed the platform examples. (See the Examples Guide for more information.)

5.2.1. 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.contributions.viewCategory"

name="%contributions.viewCategory.name">

</category>

<view

category="org.eclipse.ui.examples.contributions.viewCategory"

class="org.eclipse.ui.examples.contributions.view.InfoView"

id="org.eclipse.ui.examples.contributions.view"

name="%contributions.view.name">

</view>

</extension>

This should look pretty familiar. We see that a new view, InfoView, is contributed to the workbench. The view id, name, and category are specified as we've seen before.

Let's look at the InfoView. 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 InfoView, a view with a list in it pops up. The list is pre-populated with some data.

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);

viewer.setContentProvider(new ContentProvider());

viewer.setLabelProvider(new LabelProvider());

viewer.addDoubleClickListener(new IDoubleClickListener() {

public void doubleClick(DoubleClickEvent event) {

editSelection();

}

});

// A service will be providing our input

IPersonService service = (IPersonService) getSite().getService(

IPersonService.class);

viewerInput = new ArrayList(service.getPeople());

viewer.setInput(viewerInput);

...

// register myself as a selection provider for this view

getSite().setSelectionProvider(viewer);

}

The view creates and stores a ListViewer and sets the content and label provides. It also registers itself as the selection provider for this view. (The concept of selection provider, label provider, and content provider come from JFace viewers.) It obtains the service from its IViewSite, which contains information about the view's context, such as its workbench window, its containing page, its local services, and its plug-in.

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.

5.2.2. 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 editor input can also be a path to an in memory model, as in the InfoEditor example.

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 also use org.eclipse.ui.commands and org.eclipse.ui.menus to contribute to the workbench menus and toolbars when that 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 editor can contribute 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

class="org.eclipse.ui.examples.contributions.editor.InfoEditor"

icon="icons/editor.gif"

id="org.eclipse.ui.examples.contributions.editor"

name="%contributions.editor.name">

</editor>

</extension>

We see the familiar configuration markup for id, name, icon (which must be specified when specifying class), and class. You could use the extensions attribute describes the file types that the editor understands, like extensions="person", although this example doesn't need it. (You could also specify filenames if you need to be more specific.) The class implements the editor.

Editor menus and editor toolbars are placed in the main menu and main toolbar. See org.eclipse.ui.menus for how to use the locationURI to place the commands correctly.

These menu and tool bar items can be shown only when the editor is active using core expressions. To define a re-usable core expression for your editor, use org.eclipse.core.expressions.definitions

<extension

point = "org.eclipse.core.expressions.definitions">

<definition id="org.eclipse.ui.examples.contributions.view.activeEditor">

<with variable="activeEditorId">

<equals value="org.eclipse.ui.examples.contributions.editor"/>

</with>

</definition>

</extension>

Editors and content outliners

Although the Info Editor does not, 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.

5.2.3. Commands

A command is the declaration of a behaviour by id. Commands are used to declare semantic behaviour so that action implementations defined elsewhere by handlers. The separation of the command from the behaviour implementation allows multiple plug-ins to define implementations 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 implementations with these commands where it makes sense. In this way, semantically similar behaviour 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 Info example markup:

<extension

point="org.eclipse.ui.commands">

...

<command

categoryId="org.eclipse.ui.examples.contributions.commands.category"

id="org.eclipse.ui.examples.contributions.view.count"

description="%contributions.view.count.desc"

name="%contributions.view.count.name">

</command>

...

The command definition specifies a name, description, and id for the behaviour. 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="%contributions.commands.category.name"

description="%contributions.commands.category.desc"

id="org.eclipse.ui.examples.contributions.commands.category">

</category>

...

Note that there is no implementation specified for a command. A command only becomes concrete when a plug-in associates its handler or action with the command id. We'll talk about the different ways to associate handler implementations with commands in the org.eclipse.ui.handlers section. We'll talk about binding key sequences to commands in the org.eclipse.ui.bindings section.

5.2.4. org.eclipse.ui.menus

Commands can be implemented using org.eclipse.ui.handlers and bound to keys using org.eclipse.ui.bindings. With the org.eclipse.ui.menus extension point they can be placed in the main menu, view dropdown menus, context menus. They can also be added to the main toolbar, view toolbars, and various trim locations.

Contribution location

The org.eclipse.ui.menus extension point requires the id of a menu, toolbar, or trim area and an insertion point. This is the locationURI of a <menuContribution/> element. Some examples of locationURIs:

menu:org.eclipse.ui.main.menu?after=window - insert this contribution in the main menu after the Window menu.

menu:file?after=additions - insert this contribution in the File menu after the additions group. Equivalent to the old menubarPath="file/additions".

menu:org.eclipse.ui.views.ContentOutline?after=additions - insert this contribution in the Content Outline view dropdown menu after the additions group.

toolbar:org.eclipse.ui.views.ContentOutline?after=additions - insert this contribution in the Content Outline view toolbar, after the additions group.

popup:org.eclipse.ui.examples.propertysheet.outline?after=additions - insert this contribution in the Property Sheet outline page context menu after the additions group.

A word about popup: locationURIs. In popup:id, the id refers to the id that comes from registering the context menu when the registerContextMenu(\*) method is called. If an no id is specified in the call, it defaults to the id of the view or editor registering the context menu.

The workbench defines all of its group slot names in the classes IWorkbenchActionConstants and IIDEActionConstants. These ids, like the file example above, are available to menu contributions.

After the locationURI specifies the insertion point, the <menuContribution/> elements are turned into IContributionItems and inserted in order. Another difference between menu contributions and action contributions is that the menu contributions can be defined in the XML in the same kind of containing relationship that one would would see in a menu or toolbar.

Let's start with a simple example from our Info example, adding some commands to the InfoView.

<extension

point="org.eclipse.ui.menus">

<menuContribution

locationURI="menu:org.eclipse.ui.examples.contributions.view?after=additions">

<command

commandId="org.eclipse.ui.examples.contributions.view.count"

mnemonic="%contributions.view.count.mnemonic">

</command>

<command

commandId="org.eclipse.ui.examples.contributions.view.edit"

mnemonic="%contributions.view.edit.mnemonic">

</command>

<command

commandId="org.eclipse.ui.file.refresh"

mnemonic="%contributions.view.refresh.mnemonic">

</command>

</menuContribution>

...

Our locationURI marks this contribution for the org.eclipse.ui.examples.contributions.view view. Here we are adding 3 commands to the InfoView dropdown menu: Count Entries, Edit, and Refresh.

Although you can specify a label in the menu contribution <command/> element, if you don't we will use the name attribute from the org.eclipse.ui.commands command definition. Now clicking on the menu element will execute that command.

We've also placed a command in the InfoView toolbar:

<menuContribution

locationURI="toolbar:org.eclipse.ui.examples.contributions.view?after=additions">

<command

commandId="org.eclipse.ui.examples.contributions.view.swap"

label="%contributions.view.swap.name"

tooltip="%contributions.view.swap.tooltip">

</command>

</menuContribution>

Note: this command will appear in the view's toolbar. To place the same command in the main toolbar, you have to create the toolbar as well by including a <toolbar/> element in the <menuContribution/> to contain the command.

The "Swap Entries" button on the toolbar will be disabled until 2 Person entries are selected in the info view because we defined an <enabledWhen> expression in the handler definition:

<enabledWhen>

<count

value="2">

</count>

</enabledWhen>

Contribution visibility

A command's enabled state is controlled by a combination of the command is handled and if so, the handler's enabled state. Menu contributions can use core expressions to control the command's visibility in menus and toolbars.

As an example, menu contributions can still be tied to an existing action set while being contributed to the main menu or main toolbar, as action sets are converted into contexts. In this example we place our command in the main menu and main toolbar.

<extension

point="org.eclipse.ui.actionSets">

<actionSet

id="org.eclipse.ui.examples.contributions.globalActionSet"

label="%contributions.globalActionSet.label"

visible="false">

</actionSet>

</extension>

<extension

point="org.eclipse.core.expressions.definitions">

<definition

id="org.eclipse.ui.examples.contributions.inGlobalActionSet">

<with

variable="activeContexts">

<iterate

operator="or">

<equals

value="org.eclipse.ui.examples.contributions.globalActionSet">

</equals>

</iterate>

</with>

</definition>

...

The above XML defines the action set to use, and a definition for a core expression that checks contexts to see if the action set is active. See the org.eclipse.ui.handlers section for other examples of core expressions.

Now we can add our command to the main menu:

<menuContribution

locationURI="menu:org.eclipse.ui.main.menu?after=additions">

<menu

label="%contributions.menus.globalMenu.label"

mnemonic="%contributions.menus.globalMenu.label"

id="org.eclipse.ui.examples.contributions.menus.globalMenu">

<command

commandId="org.eclipse.ui.examples.contributions.commands.globalCommand"

mnemonic="%contributions.menus.globalCommand.mnemonic"

id="org.eclipse.ui.examples.contributions.menus.globalCommand">

<visibleWhen>

<reference

definitionId="org.eclipse.ui.examples.contributions.inGlobalActionSet">

</reference>

</visibleWhen>

</command>

<separator

name="additions"

visible="false">

</separator>

</menu>

</menuContribution>

<menuContribution

locationURI="toolbar:org.eclipse.ui.main.toolbar?after=additions">

<toolbar

id="org.eclipse.ui.examples.contributions.toolbars.sampleToolbar">

<command

commandId="org.eclipse.ui.examples.contributions.commands.globalCommand"

icon="icons/sample.gif"

tooltip="%contributions.toolbars.globalCommand.tooltip"

id="org.eclipse.ui.examples.contributions.toolbars.globalCommand">

<visibleWhen>

<reference

definitionId="org.eclipse.ui.examples.contributions.inGlobalActionSet">

</reference>

</visibleWhen>

</command>

<separator

name="additions"

visible="false">

</separator>

</toolbar>

</menuContribution>

...

In the above XML, we are adding the menu "Global Menu" to the main menu, and then placing the "Global Command" in it. The <visibleWhen/> element will evaluate the body of the previously defined inGlobalActionSet core expression. The <separator/> element adds and additions group that can be used by other contributions. We are also creating a toolbar in the main coolbar (org.eclipse.ui.main.toolbar) and placing a our command in it with the sample.gif icon.

Other contributions can now contribute to the "Global Menu" menu by specifying its id as a contribution locationURI: menu:org.eclipse.ui.examples.contributions.menus.globalMenu?after=additions.

Currently, commands contributed to action sets don't show up in the Window > Customize Perspective... dialog.

You can add menu contributions that work similar to org.eclipse.ui.editorActions (Deprecated). First you define your editor command and handler, like Reset. Then you can add them in an editor menu like "Info" to the main menu:

... org.eclipse.core.expressions.definitions

<definition

id="org.eclipse.ui.examples.contributions.view.activeEditor">

<with

variable="activeEditorId">

<equals

value="org.eclipse.ui.examples.contributions.editor">

</equals>

</with>

</definition>

... org.eclipse.ui.menus

<menuContribution

locationURI="menu:org.eclipse.ui.main.menu?after=additions">

<menu

id="org.eclipse.ui.examples.contributions.editor.menu"

label="%contributions.editor.menu.label"

mnemonic="%contributions.editor.menu.mnemonic">

<command

commandId="org.eclipse.ui.examples.contributions.editor.reset"

mnemonic="%contributions.editor.reset.mnemonic">

<visibleWhen>

<reference

definitionId="org.eclipse.ui.examples.contributions.view.activeEditor">

</reference>

</visibleWhen>

</command>

</menu>

</menuContribution>

This is similar to adding to the main menu with our global action example. Here our core expression will make this element visible as long as the active editor id variable matches our editor. You can check out org.eclipse.ui.ISources for a list of supported variables names.

Note: updating the main menu and especially the main toolbar are expensive operations. You generally want to confine them to actionSet equivalent contexts and active editor type. Although you can update the main toolbar on each selection change using the default variable or a <with variable="selection"/> expression, it's not a good idea. The common practice is to leave your command visibility at the action set or active editor level, and have your handler enabled state track the current selection.

Contributing to popup menus

Commands can be contributed to a specific context menu by the context menu's id, or to any context menu where it can satisfy its <visibleWhen> clause. For example, we can add our Refresh command to the Info View popup as a convenience. Because we didn't call registerContextMenu(\*) with a specific id it defaults to the view id.

<menuContribution

locationURI="popup:org.eclipse.ui.examples.contributions.view?after=additions">

<command

commandId="org.eclipse.ui.file.refresh"

mnemonic="%contributions.view.refresh.mnemonic">

</command>

</menuContribution>

To contribute a command to a popup if its selection matches a particular object type you can use the default variable, or for behaviour closest to org.eclipse.ui.popupMenus target a specific popup selection variable. Here's an example using the context menu selection provider. This just affects the popup menu visibility, not the command enabled state.

<menuContribution

locationURI="popup:org.eclipse.ui.popup.any?after=additions">

<command

commandId="org.eclipse.ui.examples.contributions.view.edit"

mnemonic="%contributions.view.edit.mnemonic">

<visibleWhen>

<with

variable="activeMenuSelection">

<iterate>

<adapt

type="org.eclipse.ui.examples.contributions.model.Person">

</adapt>

</iterate>

</with>

</visibleWhen>

</command>

</menuContribution>

Using <iterate><adapt type="Person"/></iterate> is the core expression equivalent of the old objectClass attribute.

Adding toolbars to trim areas

A 'trim' widget is a control that gets sited into a location (called a 'Trim Area') on the outer boundary of the Workbench Window. The most common example is the generic 'status line' which almost all GUI's place along the bottom of the window. The extension point org.eclipse.ui.menus allows plug-ins to add elements to the workbench trim by contributing toolbars into trim areas and populating the toolbar with one or more controls (using the same mechanism as contributing controls into the main toolbar).

Controls contributed to the trim must be a subclass of WorkbenchWindowControlContribution. This class will manage the life-cycle of the contribution, disposing and re-creating the contribution as necessary (such as when the user moves the control to another trim area). Note that the getCurSide() and getOrientation() methods allow the implementation of createControl(parent) to adjust the created control to its current location in the trim.

For this example we've contributed a simple trim widget that simply displays a string and an indication of which side the trim is currently docked on.

Let's take a look at the extension point definition used to contribute this piece of trim:

<menuContribution

locationURI="toolbar:org.eclipse.ui.trim.status">

<toolbar

id="org.eclipse.ui.examples.contributions.contributedTrim">

<command

commandId="org.eclipse.ui.examples.contributions.item2"

icon="icons/editor.gif"

id="contributions.trimItem"

label="%Trim.item"

tooltip="%TrimItem.toolTip">

</command>

<control

class="org.eclipse.ui.examples.contributions.ExampleControlContribution"

id="contributions.controlContrib1">

</control>

<command

commandId="org.eclipse.ui.examples.contributions.item2"

icon="icons/editor.gif"

id="contributions.trimItem2"

label="%Trim2.item"

tooltip="%TrimItem2.toolTip">

</command>

</toolbar>

</menuContribution>

This extension defines the 'locationURI' for this extension; identifying it as being at the start of the 'status' trim area (i.e. at the beginning of the bottom). We then define a toolbar contribution at that location, adding a control contribution bracketed by two command contributions directly into its definition.

One thing that is not reflected in this sample's code is the reliance of the trim layout manager on the trim control's proper implementation of the widget control's computeSize method. The widget must be capable of calculating and returning its 'preferred' size since this is used throughout the layout management implementation to determine, for example, how much space is needed for a particular trim area.

For example, if you want to contribute a Text input control to the trim (i.e. a 'Search Bar') and you want to control the width it has you'll need to wrap the Text control in a Composite whose Layout's 'computeSize' returns the correct values.

See the SWT documentation for notes on how to correctly implement 'computeSize' correctly.

5.2.5. Handlers

A handler is the implementation of a command's behaviour. Any plugin can contribute a handler implementation for any command. The workbench uses core expressions and programmatic scoping rules to determine which handler is active at any time. There can either be one handler active for the command, or no handlers active for the command (the command is effectively disabled). When the command has an active handler, we say the command is handled.

Associating a default handler with a command

While there is a shortcut for a default handler, most handlers are associated with their command using the org.eclipse.ui.handlers extension point.

A command that has default behaviour (it works as long as the workbench is up) can have the default implementation associated in the command definition. The Info example "Global Command" does this:

<command

categoryId="org.eclipse.ui.examples.contributions.commands.category"

defaultHandler="org.eclipse.ui.examples.contributions.handlers.GlobalMenuHandler"

id="org.eclipse.ui.examples.contributions.commands.globalCommand"

name="%contributions.commands.globalCommand.name">

</command>

defaultHandler points to the IHandler class that implements the default behaviour for this command. The default handler will always be available unless overriden by a handler association in a more specific scope. This is a shortcut for the equivalent org.eclipse.ui.handlers entry:

...

<handler

class="org.eclipse.ui.examples.contributions.handlers.GlobalMenuHandler"

commandId="org.eclipse.ui.examples.contributions.commands.globalCommand">

</handler>

...

Associating a handler with a command while a part type is active

The <activeWhen/> expressions in the plugin.xml and programmatic core expressions are used to help determine the scope of a handlers activation. For example, a specific window, a specific Shell, an active part type or active part.

Here is an example where we are adding some commands to the Info view, org.eclipse.ui.examples.contributions.view. Count to count the number of model elements in the view and swap to swap the 2 selected elements. The command definitions are in the global space as always:

<extension

point="org.eclipse.ui.commands">

<command

categoryId="org.eclipse.ui.examples.contributions.commands.category"

id="org.eclipse.ui.examples.contributions.view.count"

description="%contributions.view.count.desc"

name="%contributions.view.count.name">

</command>

<command

categoryId="org.eclipse.ui.examples.contributions.commands.category"

id="org.eclipse.ui.examples.contributions.view.swap"

name="%contributions.view.swap.name">

</command>

...

We declaratively associate the swap command with a handler that is active while a view with the correct ID is active. When declaring a handler, you can also provide a core expression for declarative enablement. The extension point description for org.eclipse.ui.handlers lists valid elements for the core expression.

<extension

point="org.eclipse.core.expressions.definitions">

...

<definition

id="org.eclipse.ui.examples.contributions.view.inView">

<with

variable="activePartId">

<equals

value="org.eclipse.ui.examples.contributions.view">

</equals>

</with>

</definition>

...

</extension>

<extension

point="org.eclipse.ui.handlers">

...

<handler

class="org.eclipse.ui.examples.contributions.view.SwapInfoHandler"

commandId="org.eclipse.ui.examples.contributions.view.swap">

<activeWhen>

<reference

definitionId="org.eclipse.ui.examples.contributions.view.inView">

</reference>

</activeWhen>

<enabledWhen>

<count

value="2">

</count>

</enabledWhen>

</handler>

...

Here we are using another extension point org.eclipse.core.expressions.definitions so we can reuse the same expression in multiple places. The definition's id is org.eclipse.ui.examples.contributions.view.inView and the expression it defines is <with variable="activePartId">...</with>. Whenever another core expression uses <reference definitionId="org.eclipse.ui.examples.contributions.view.inView"/> the <with...> expression will be evaluated.

The <activeWhen/> clause here says the this handler will be active when the activePartId equals our Info view id. The priorities defined in org.eclipse.ui.ISources can give you an idea of the relative importance of different variables, although the priorities alone do not determine a variable's relative importance.

Our handler definition also includes an <enabledWhen> clause. In this case the handler will be enabled when the default variable (the current selection converted into a java.util.Collection of objects) has 2 elements. A core expression that does not include a <with/> element is evaluated against the default variable.

Associating a handler programmically with a command while a specific part is active

Sometimes it is desirable to instantiate your handlers when your part is created. You can use the org.eclipse.ui.handlers.IHandlerService to activate your handler for your part. Here is the code that activates a handler for the count command.

private static final String VIEW\_COUNT\_ID = "org.eclipse.ui.examples.contributions.view.count"; //$NON-NLS-1$

...

/\*\*

\* Instantiate any handlers specific to this view and activate them.

\*/

private void createHandlers() {

// 1 - get the handler service from the view site

IHandlerService handlerService = (IHandlerService) getSite()

.getService(IHandlerService.class);

// 2 - create the handler instance

countHandler = new AbstractHandler() {

public Object execute(ExecutionEvent event)

throws ExecutionException {

// viewer is an instance variable of InfoView

List elements = (List) viewer.getInput();

MessageDialog.openInformation(getSite().getShell(),

ContributionMessages.SampleHandler\_plugin\_name,

NLS.bind(ContributionMessages.InfoView\_countElements,

new Integer(elements.size())));

return null;

}

};

// 3 - activate this handler instance for the count command

handlerService.activateHandler(VIEW\_COUNT\_ID, countHandler);

}

In the InfoView, createHandlers() is called from the end of the createPartControl(Composite) method. org.eclipse.ui.handlers.IHandlerService and org.eclipse.ui.context.IContextService provide scoping of their activations depending on where you get the service.

IHandlerService from the workbench is the global handler service. it provides no special activation scoping or lifecycle.

IHandlerService from the workbench window is the window handler service. Any handlers activated through the window handler service will be active when that window is active. Any listeners added to the window handler service will be removed when the window is disposed, and any active handlers will be deactivated (but not disposed).

IHandlerService from the workbench part site is the part handler service. Any handlers activated through the part handlers service will only be active when that part is active. Any listeners added to the part handler service will be removed when the part is disposed, and any active handlers will be deactivated (but not disposed).

Implementing the handler

A handler must implement org.eclipse.core.commands.IHandler although in most cases it is easier to subclass org.eclipse.core.commands.AbstractHandler.

The bulk of the work is done in the execute(ExecutionEvent) method. From the org.eclipse.core.commands.ExecutionEvent you can get any parameters from the calling command object as well as the application context the command was executed in.

Object object = event.getApplicationContext();

if (object instanceof IEvaluationContext) {

IEvaluationContext appContext = (IEvaluationContext) object;

...

}

The application context provides access to much of the workbench current state. For example, the active workbench window, active shell, active part, active editor, and current selection to name a few. See org.eclipse.ui.handlers.HandlerUtil for an example of extracting variables from the application context and org.eclipse.ui.ISources for a list of variables that are currently supported.

The GlobalMenuHandler is an example of a simple handler that just opens an information popup with a message. The example execute:

public Object execute(ExecutionEvent event) throws ExecutionException {

IWorkbenchWindow window = HandlerUtil

.getActiveWorkbenchWindowChecked(event);

MessageDialog.openInformation(window.getShell(),

ContributionMessages.SampleHandler\_plugin\_name,

ContributionMessages.SampleHandler\_hello\_msg);

return null;

}

Handlers subclassing org.eclipse.core.commands.AbstractHandler or implementing org.eclipse.core.commands.IHandler2 can implement setEnabled(Object evaluationContext). Before the workbench framework asks a handler for its enabled state, it will call setEnabled with an org.eclipse.core.expressions.IEvaluationContext to allow the handler to update its state.

Without getting into to many details, this uses a HandlerUtil convenience method to extract the active workbench window. Then it opens an information dialog with a "hello world" message. ContributionMessages is an org.eclipse.osgi.util.NLS subclass to help externalize our message strings.

5.2.6. 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. From our InfoView example:

<extension

point="org.eclipse.ui.bindings">

<key

commandId="org.eclipse.ui.examples.contributions.view.edit"

contextId="org.eclipse.ui.examples.contributions.view.context"

sequence="M1+O"

schemeId="org.eclipse.ui.examples.contributions.scheme">

</key>

</extension>

The sequence attribute for a key binding defines the key combination that is used to invoke a command. As long as our InfoView context is active, we will try and execute the InfoView edit command when the user chooses CTRL+O and the scheme is set to org.eclipse.ui.examples.contributions.scheme. The mapping of M1 to CTRL is described in the extension point description for org.eclipse.ui.bindings.

The contextId indicates what active context this key binding will be active in. More information on defining contexts declaratively and activating them programmatically can be found in Contexts. In this case, this key binding will be active when the org.eclipse.ui.examples.contributions.view.context context is active.

In our InfoView we have code called from the end of createPartControl(Composite):

private static final String VIEW\_CONTEXT\_ID = "org.eclipse.ui.examples.contributions.view.context"; //$NON-NLS-1$

...

/\*\*

\* Activate a context that this view uses. It will be tied to this view

\* activation events and will be removed when the view is disposed.

\*/

private void activateContext() {

IContextService contextService = (IContextService) getSite()

.getService(IContextService.class);

contextService.activateContext(VIEW\_CONTEXT\_ID);

}

As mentioned in org.eclipse.ui.handlers this context will be active only when the InfoView part is active. It will also be deactivated when the InfoView part is disposed.

5.2.7. Workbench Core Expressions

The workbench uses core expressions (See org.eclipse.core.expressions.definitions for a description) for enabledWhen and activeWhen for handlers, programmatic activation of contexts, and for visibleWhen for menu contributions. Core expressions are also used as conditionals for certain property page contribution and object contribution expressions, using the enablement element. In most cases, the workbench provides the IEvaluationContext that command core expressions are evaluate against.

The IEvaluationService provides methods that allow other clients and extension point builders to hook into the workbench core expressions.

Using Core Expressions in the Workbench

Core expressions are used declaratively in the plugin.xml files, and programmatically with some of the services provided in the workbench.

Note: Elements from workbench extension points that use core expressions, like enabledWhen, take one child core expression element. They do not and their elements together.

Declarative Expression Examples

Basic IStructuredSelection

IResources and the Package or Project Explorer

Active Contexts

Active Views and Editor

ActionSets and Contexts

Basic IStructuredSelection

Most of the tree or table like viewers return an IStructuredSelection. For example, the Project Explorer and Package Explorer.

When using the default variable you must treat it as an java.util.Collection. That means using <count> or <iterate>. The expression below returns true if all of the selected items are Person.

<enabledWhen>

<iterate ifEmpty="false">

<instanceof value="org.eclipse.ui.examples.contributions.model.Person"/>

</iterate>

</enabledWhen>

This is equivalent to:

<enabledWhen>

<with variable="selection">

<iterate ifEmpty="false">

<instanceof value="org.eclipse.ui.examples.contributions.model.Person"/>

</iterate>

</with>

</enabledWhen>

The behaviour of iterate is:

iterate ands the results of evaluating its child expressions for each element in the collection, unless you set the operator attribute.

iterate has the same semantics for its children as <and>

iterate with an operator of and returns true if the collection is empty, unless you set the ifEmpty attribute.

If you want to be enabled if only one Person is selected, you can include a count element.

<enabledWhen>

<with variable="selection">

<count value="1"/>

<iterate ifEmpty="false">

<instanceof value="org.eclipse.ui.examples.contributions.model.Person"/>

</iterate>

</with>

</enabledWhen>

The same expression using the default variable:

<enabledWhen>

<and>

<count value="1"/>

<iterate ifEmpty="false">

<instanceof value="org.eclipse.ui.examples.contributions.model.Person"/>

</iterate>

</and>

</enabledWhen>

IResources and the Package or Project Explorer

The Package Explorer is a mixture of org.eclipse.core.resources.IResource, org.eclipse.jdt.core.IJavaElement and other classes. If you are trying to find all of the "\*.java" files, you would need to:

Iterate through the default variable.

Adapt the selection elements to your class, in this case org.eclipse.core.resources.IResource.

Use one of the org.eclipse.core.resources property testers to test the org.eclipse.core.resources.IResource property.

<enabledWhen>

<with variable="selection">

<iterate ifEmpty="false">

<adapt type="org.eclipse.core.resources.IResource">

<test property="org.eclipse.core.resources.name"

value="\*.java"/>

</adapt>

</iterate>

</with>

</enabledWhen>

When working with the current selection and property testers it does not always make sense to use adapt. In these situations it is a good practice to check that the object under test is a valid type for that property tester:

<enabledWhen>

<with variable="selection">

<iterate ifEmpty="false">

<instanceof type="org.eclipse.core.resources.IResource"/>

<test property="org.eclipse.core.resources.name"

value="\*.java"/>

</iterate>

</with>

</enabledWhen>

Active Contexts

If your handler should be enabled when your view or editor activates a context, you can use the activeContexts variable. Contexts are defined in the org.eclipse.ui.contexts extension point and activated programmatically using the org.eclipse.ui.contexts.IContextService.

<enabledWhen>

<with variable="activeContexts">

<iterate ifEmpty="false" operator="or">

<equals value="org.example.view.context"/>

</iterate>

</with>

</enabledWhen>

Active Views and Editor

For handlers that are to be contributed to a specific view or editor, you can use activeEditorId and activePartId in the activeWhen clause. This is an example for a handler that should be active in text editors:

<activeWhen>

<with variable="activeEditorId">

<equals value="org.eclipse.ui.DefaultTextEditor"/>

</with>

</activeWhen>

The following clause is for a handler that's active while in the Project Explorer:

<activeWhen>

<with variable="activePartId">

<equals value="org.eclipse.ui.navigator.ProjectExplorer"/>

</with>

</activeWhen>

ActionSets and Contexts

As of 3.3 all org.eclipse.ui.actionSets (Deprecated) generate a context with a parent of org.eclipse.ui.contexts.actionSet. Contexts with this parent are filtered from the General > Keys preference page.

Showing an actionSet activates the matching context. This allows contributed commands to "join" actionSets, like the debug launch actionSet.

<enabledWhen>

<with variable="activeContexts">

<iterate ifEmpty="false" operator="or">

<equals value="org.eclipse.debug.ui.launchActionSet"/>

</iterate>

</with>

</enabledWhen>

Note: commands that are enabled or visible with actionSets are not currently displayed in the Customize Perspective Dialog.

Variables Provided in the Workbench

The IEvaluationService provides the global selection as the default variable (in a java.util.Collection) for expression evaluation. It can either be empty, have one entry (if the ISelection was something like an ITextSelection), or have the contents of an IStructuredSelection.

The workbench publishes variables in ISources that can be used in the <with/> element and can be retrieved from the IEvaluationContext. Some of the variables may not be set, depending on the current application context when they are evaluated. The following table explains some of the more commonly used ones:

Name

Type

Description

Since

activeContexts

A java.util.Collection of java.lang.String

This is a collection of the active context IDs as strings. Most commonly used with <iterate/>, <count/>, and <test/> with a combined org.eclipse.core.expressions.PropertyTester. In 3.3 action sets are mirrored by contexts whose parent is org.eclipse.ui.actionSet, and the active action sets show up in the list of active contexts.

3.2

activeShell

org.eclipse.swt.widgets.Shell

The currently active shell. It can be a dialog or workbench window shell.

3.2

activeWorkbenchWindowShell

org.eclipse.swt.widgets.Shell

The active workbench window shell.

3.2

activeWorkbenchWindow

org.eclipse.ui.IWorkbenchWindow

The active workbench window.

3.2

activeWorkbenchWindow.isCoolbarVisible

java.lang.Boolean

Reports coolbar visibility for the currently active workbench window.

3.3

activeWorkbenchWindow.isPerspectiveBarVisible

java.lang.Boolean

Reports perspective bar visibility for the currently active workbench window.

3.3

activeWorkbenchWindow.activePerspective

java.lang.String

Reports the name of the current perspective of the active workbench window.

3.4

activeEditor

org.eclipse.ui.IEditorPart

The currently active editor. This is remembered even if the editor is not the currently active part.

3.2

activeEditorId

java.lang.String

The ID of the currently active editor. This can be used for expressions on the editor type.

3.2

activeEditorInput

org.eclipse.ui.IEditorInput

The input of the currently active editor. This is useful for property testers.

3.5

activePart

org.eclipse.ui.IWorkbenchPart

The active part, which can be the same as the active editor.

3.2

activePartId

java.lang.String

The ID of the currently active part.

3.2

activeSite

org.eclipse.ui.IWorkbenchPartSite

The site of the currently active part.

3.2

selection

org.eclipse.jface.viewers.ISelection

The current global selection. It is often used with <test/> elements with org.eclipse.core.expressions.PropertyTester, in programmatic core expressions, and in 3.3 with <iterate/> and <count/> elements.

3.2

activeMenu

A java.util.Collection of java.lang.String

This is the list of IDs of the showing context menu. Examples are like #TextEditorRuler or a part ID. Most commonly used with <iterate/>, <count/>, and <test/> with a combined org.eclipse.core.expressions.PropertyTester.

3.2

activeMenuSelection

org.eclipse.jface.viewers.ISelection

This is a selection that is available while a context menu is showing. It is the selection from the selection provider used to register the context menu, usually from getSite().registerContextMenu(\*). It is usually the same as the selectionvariable, but not always.

3.3

activeMenuEditorInput

org.eclipse.jface.viewers.ISelection

This is a selection that is available while a context menu is showing. It is the selection from the editor input, usually if includeEditorInput was set to true during getEditorSite().registerContextMenu(\*).

3.3

activeFocusControl

org.eclipse.swt.widgets.Control

A control that has focus and has been registered with the IFocusService.

3.3

activeFocusControlId

java.lang.String

The ID of a control that has focus and has been registered with the IFocusService.

3.3

org.eclipse.core.runtime.Platform

org.eclipse.core.runtime.Platform

The runtime Platform class is available, for use with property testers like org.eclipse.core.runtime.isBundleInstalled and org.eclipse.core.runtime.product.

3.3

Property Testers Provided in the Workbench

The workbench provides a couple of property testers that can be used in core expressions. The expression defines a property attribute and then takes a combination of 'args' and a 'value' that is tester implementation dependent. The property attribute is the combination of the namespace and property name. For example, to test an IResource name the property would be org.eclipse.core.resources.name.

Namespace

Type

Implementation

org.eclipse.core.runtime

org.eclipse.core.runtime.Platform

org.eclipse.core.internal.expressions.propertytester.PlatformPropertyTester

Property

Description

product

Test the ID of the currently active product.

isBundleInstalled

Test if a given bundle is installed in the running environment. Use the 'args' attribute to pass in the bundle ID.

bundleState

Test the state of the bundle in the running environment. Use the 'args' attribute to pass in the bundle ID and the 'value' attribute to pass the state as defined in Bundle, e.g. "ACTIVE".

Namespace

Type

Implementation

org.eclipse.core.resources

org.eclipse.core.resources.IResource

org.eclipse.core.internal.propertytester.ResourcePropertyTester

Property

Description

name

A property indicating the file name (value "name"). "\*" and "?" wild cards are supported.

path

A property indicating the file path (value "path"). "\*" and "?" wild cards are supported.

extension

A property indicating the file extension (value "extension"). "\*" and "?" wild cards are supported.

readOnly

A property indicating whether the file is read only (value "readOnly").

projectNature

A property indicating the project nature (value "projectNature").

persistentProperty

A property indicating a persistent property on the selected resource (value "persistentProperty"). If two arguments are given, this treats the first as the property name, and the second as the expected property value. If only one argument (or just the expected value) is given, this treats it as the property name, and simply tests for existence of the property on the resource.

projectPersistentProperty

A property indicating a persistent property on the selected resource's project. (value "projectPersistentProperty"). If two arguments are given, this treats the first as the property name, and the second as the expected property value. If only one argument (or just the expected value) is given, this treats it as the property name, and simply tests for existence of the property on the resource.

sessionProperty

A property indicating a session property on the selected resource (value "sessionProperty"). If two arguments are given, this treats the first as the property name, and the second as the expected property value. If only one argument (or just the expected value) is given, this treats it as the property name, and simply tests for existence of the property on the resource.

projectSessionProperty

A property indicating a session property on the selected resource's project. (value "projectSessionProperty"). If two arguments are given, this treats the first as the property name, and the second as the expected property value. If only one argument (or just the expected value) is given, this treats it as the property name, and simply tests for existence of the property on the resource.

Namespace

Type

Implementation

org.eclipse.core.resources

org.eclipse.core.resources.IFile

org.eclipse.core.internal.propertytester.FilePropertyTester

Property

Description

contentTypeId

A property indicating that we are looking to verify that the file matches the content type matching the given identifier. The identifier is provided as the expected value.

Namespace

Type

Implementation

org.eclipse.core.resources

org.eclipse.core.resources.IProject

org.eclipse.core.internal.propertytester.ProjectPropertyTester

Property

Description

open

A property indicating whether the project is open (value "open").

Namespace

Type

Implementation

org.eclipse.core.resources

org.eclipse.core.resources.mapping.ResourceMapping

org.eclipse.core.internal.propertytester.ResourceMappingPropertyTester

Property

Description

projectPersistentProperty

A property indicating a persistent property on the selected resource's project. (value "projectPersistentProperty"). If two arguments are given, this treats the first as the property name, and the second as the expected property value. If only one argument (or just the expected value) is given, this treats it as the property name, and simply tests for existence of the property on the resource.

Namespace

Type

Implementation

org.eclipse.ui

org.eclipse.ui.IWorkbench (not currently available)

org.eclipse.ui.internal.activities.ActivityPropertyTester

Property

Description

isActivityEnabled

Test if the activity in 'args' is enabled.

isCategoryEnabled

Test if the category in 'args' is enabled.

Namespace

Type

Implementation

org.eclipse.ui.workbenchWindow

org.eclipse.ui.IWorkbenchWindow

org.eclipse.ui.internal.OpenPerspectivePropertyTester

Property

Description

isPerspectiveOpen

Tests if any perspective is open.

5.3. The plug-in class

So far, we've been looking at the different extensions that are provided by the contributions plug-in. Let's look at the general definition of the contributions plug-in.

Plug-in definition

The readme tool plug-in is defined in the MANIFEST.MF file.

Manifest-Version: 1.0

Bundle-ManifestVersion: 2

Bundle-Name: %contributions.Activator.name

Bundle-SymbolicName: org.eclipse.ui.examples.contributions; singleton:=true

Bundle-Version: 3.3.100.qualifier

Bundle-Activator: org.eclipse.ui.examples.contributions.Activator

Require-Bundle: org.eclipse.ui;bundle-version="[3.3.0,4.0.0)",

org.eclipse.core.runtime,

org.eclipse.core.expressions;bundle-version="[3.3.0,4.0.0)"

Bundle-Vendor: %contributions.Activator.providerName

Bundle-Localization: plugin

Bundle-RequiredExecutionEnvironment: J2SE-1.4

Bundle-ActivationPolicy: lazy

The plug-in definition includes the Bundle-Name, Bundle-SymbolicName (plug-in id), Bundle-Version, and Bundle-Vendor of the plug-in. We saw most of these parameters before in our hello world plug-in. The contribution plug-in also defines a specialized plug-in class, org.eclipse.ui.examples.contributions.Activator.

The Require-Bundle element informs the platform of the contribution plug-in's dependencies. The workbench UI plug-ins are listed as required plug-ins, along with the various core and expression plug-ins.

AbstractUIPlugin

The Activator class represents the contribution 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 Activator.

AbstractUIPlugin uses the generic startup and shutdown methods to manage images, dialog settings, and a preference store during the lifetime of the plug-in.

5.4. 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.

5.4.1. 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).

5.4.2. 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\_Field"));

textField = createTextField(composite\_textField);

pushButton\_textField = createPushButton(composite\_textField, MessageUtil.getString("Change"));

//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\_button\_1"));

radioButton2 = createRadioButton(composite\_radioButton, MessageUtil.getString("Radio\_button\_2"));

radioButton3 = createRadioButton(composite\_radioButton, MessageUtil.getString("Radio\_button\_3"));

//composite\_tab2 << parent

Composite composite\_tab2 = createComposite(parent, 2);

Label label2 = createLabel(composite\_tab2, MessageUtil.getString("Check\_Box\_Options")); //$NON-NLS-1$

//

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 Restore Defaults.

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();

}

Note: If there are no preferences saved anywhere for a plug-in, the plug-in will get an empty preference store.

The preference store is initialized with default values in ReadmePreferenceInitializer. The preference initializer is contributed to the preferences service using the org.eclipse.core.runtime.preferences extension point. These values are used the first time the preference page is shown or when the user presses Restore Defaults in the preferences page.

public void initializeDefaultPreferences() {

// These settings will show up when the Readme preference page

// is shown 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")); //$NON-NLS-1$

}

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 OK or Apply 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 Restore Defaults, the current values of the controls on the preference page should be reset to the default values in the preference store. The default values are defined using a preference initializer, ReadmePreferenceInitializer. The ReadmePreferencePage implements this logic in a separate method, 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));

...

}

5.4.3. 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\_Debug\_2"), //$NON-NLS-1$

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.

6. 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.

6.1. 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");

6.2. 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.

Popup dialogs

In some cases, you might want to show information about something in a dialog, but in a way that is more "lightweight" than launching a regular dialog. For example, a dialog may be intended to provide transient information that can be easily dismissed, without taking the focus away from the user's work. If this is the case, you can use the PopupDialog class to implement the dialog. The look and feel of a PopupDialog is different than a regular Dialog in several ways. It does not have any buttons across the bottom, it does not have the standard window title bar, and its borders, spacing, and fonts are smaller and more compact.

Although a PopupDialog looks very different than a regular dialog, the code in your plug-in's subclass that defines the dialog content is almost the same. You still implement the createDialogArea method to create the SWT controls for the dialog. The main difference in the application code is that the constructor that creates the dialog has many more parameters than the regular Dialog class. For example, the SectionsDialog could be turned into a PopupDialog by simply changing the superclass of the dialog and configuring the dialog in the constructor:

public class SectionsDialog extends PopupDialog {

protected IAdaptable input;

/\*\*

\* Creates a new SectionsDialog.

\*/

public SectionsDialog(Shell parentShell, IAdaptable input) {

super(parentShell, SWT.DEFAULT, false, // do not take focus when opened

false, // do not persist the bounds

false, // do not show a resize menu

false, // do not show a menu item for persisting bounds

null, // no title

null); // no info text

this.input = input;

}

...

6.3. 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();

}

Dialog bounds settings

In general, the definition and interpretation of dialog settings are the responsibility of your plug-in. However, there are some specific dialog settings keys defined inside the JFace dialog framework that are used to remember the last size and position of a dialog, so that the dialog can be opened to that size and position on its next invocation. The framework will do the work to query and store the dialog's size and position, but you must implement a method that supplies the IDialogSettings instance that should be used to store the dialog bounds information. The following snippet shows how the SectionsDialog could take advantage of this feature.

protected IDialogSettings getDialogBoundsSettings() {

return ReadmePlugin.getDefault().getDialogSettings();

}

By implementing this method, the size and position of the SectionsDialog will be stored in predefined keys within the plug-in's dialog settings, causing the dialog to open at its previous location.

6.4. 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.)

6.5. 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.

6.5.1. 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:

6.5.2. 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.

6.5.3. 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.

6.6. 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.

6.7. 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);

}

6.8. Filtered item selection dialog

FilteredItemsSelectionDialog is a powerful dialog for displaying a set of items to be selected by the user. This dialog is used as the basis of the "Open Resource" and "Open Type" dialogs in the Eclipse SDK. You can create your own subclass of this dialog to quickly implement a similar dialog for your domain objects. Some of the dialog's key features include:

Support for multiple selections

Details field that shows information about the selected element

Uses JFace viewers, content and label providers

Background population, filtering, sorting, and refreshing of the item list

History of previously selected elements persisted across invocations

Supports for regular expression pattern matching and camel-case matching. Filtering is implemented using SearchPattern

For examples illustrating how to create you own custom implementations of this dialog, see the following:

Creating a custom filtered item selection dialog

Advanced use of the filtered item selection dialog

6.8.1. Creating a custom filtered items selection dialog

In this example, we will contribute a basic search dialog to illustrate the steps needed to create a custom subclass of FilteredItemsSelectionDialog.

Create a new Plug-in Project using Hello, world template.

Create a class extending org.eclipse.ui.dialogs.FilteredItemsSelectionDialog. Let's name it FilteredResourcesSelectionDialogExample.

Choose a source of the resources that will be used during filtering. In our example we will generate our own set of random strings as follows:

private static ArrayList resources = new ArrayList();

static {

generateRescourcesTestCases('A', 'C', 8, ""); //$NON-NLS-1$

generateRescourcesTestCases('a', 'c', 4, ""); //$NON-NLS-1$

}

private static void generateRescourcesTestCases(char startChar, char endChar, int length, String resource){

for (char ch = startChar; ch <= endChar; ch++) {

String res = resource + String.valueOf(ch);

if (length == res.length())

resources.add(res);

else if ((res.trim().length() % 2) == 0)

generateRescourcesTestCases(Character.toUpperCase((char)(startChar + 1)), Character.toUpperCase((char)(endChar + 1)), length, res);

else

generateRescourcesTestCases(Character.toLowerCase((char)(startChar + 1)), Character.toLowerCase((char)(endChar + 1)), length, res);

}

}

Now, let's implement abstract methods from the FilteredItemsSelectionDialog class.

createExtendedContentArea(Composite): This method creates an extra content area located above the details. For now, we will just return null because we don't need any extra fields for this simple example:

protected Control createExtendedContentArea(Composite parent) {

return null;

}

createFilter(): Creates a new instance of a filter. In the simplest implementation you should also extend FilteredItemsSelectionDialog.ItemsFilter and implement the abstract methods. Eg.:

protected ItemsFilter createFilter() {

return new ItemsFilter() {

public boolean matchItem(Object item) {

return matches(item.toString());

}

public boolean isConsistentItem(Object item) {

return true;

}

};

}

fillContentProvider(FilteredItemsSelectionDialog.AbstractContentProvider, FilteredItemsSelectionDialog.ItemsFilter, org.eclipse.core.runtime.IProgressMonitor): Fills the content provider with matching items. Eg.:

protected void fillContentProvider(AbstractContentProvider contentProvider,

ItemsFilter itemsFilter, IProgressMonitor progressMonitor)

throws CoreException {

progressMonitor.beginTask("Searching", resources.size()); //$NON-NLS-1$

for (Iterator iter = resources.iterator(); iter.hasNext();) {

contentProvider.add(iter.next(), itemsFilter);

progressMonitor.worked(1);

}

progressMonitor.done();

}

getDialogSettings(): Returns the settings object that stores information about how the dialog information is persisted. This method can't return null, so we'll just return a simple settings object:

private static final String DIALOG\_SETTINGS = "FilteredResourcesSelectionDialogExampleSettings";

protected IDialogSettings getDialogSettings() {

IDialogSettings settings = Activator.getDefault().getDialogSettings()

.getSection(DIALOG\_SETTINGS);

if (settings == null) {

settings = Activator.getDefault().getDialogSettings()

.addNewSection(DIALOG\_SETTINGS);

}

return settings;

}

getElementName(Object): Returns a name for the given object. This is used to check duplicates.

public String getElementName(Object item) {

return item.toString();

}

getItemsComparator(): Returns a comparator used to sort items. In our example we will just use standard string comparison:

protected Comparator getItemsComparator() {

return new Comparator() {

public int compare(Object arg0, Object arg1) {

return arg0.toString().compareTo(arg1.toString());

}

};

}

validateItem(Object): Validates that the item is a valid selection. In our example we just return an OK status, because all items are valid:

protected IStatus validateItem(Object item) {

return Status.OK\_STATUS;

}

Add title of dialog and set simple implementation of SelectionHistory on dialog:

public FilteredResourcesSelectionDialogExample(Shell shell, boolean multi) {

super(shell, multi);

setTitle("Filtered Resources Selection Dialog Example");

setSelectionHistory(new ResourceSelectionHistory());

}

private class ResourceSelectionHistory extends SelectionHistory {

protected Object restoreItemFromMemento(IMemento element) {

return null;

}

protected void storeItemToMemento(Object item, IMemento element) {

}

}

Change run(IAction) method from SimpleAction to:

public void run(IAction action) {

Shell shell = new Shell();

FilteredItemsSelectionDialog dialog = new FilteredResourcesSelectionDialogExample(shell, true);

dialog.setInitialPattern("a");

dialog.open();

}

Change tooltip of SimpleAction from "Hello, Eclipse world" to "Filtered Items Selection Dialog Example".

Run Eclipse with created plug-in.

The resulting dialog looks as follows:

6.8.2. Advanced use of the filtered items selection dialog

In the previous example, we saw how to create a simple subclass of FilteredItemsSelectionDialog. Now let's explore some of the advanced capabilities provided by this dialog:

Adding selected items history

Adding custom filters

Adding an extra check-box to the dialog

Adding selected items history

The dialog can be configured to save and restore the history of items that have been selected.

In the previous part we created a subclass of FilteredItemsSelectionDialog.SelectionHistory but it do nothing. Now, we should fill out methods responsible for saving and loading objects:

private class ResourceSelectionHistory extends SelectionHistory {

/\*

\* @see org.eclipse.ui.dialogs.FilteredItemsSelectionDialog.SelectionHistory#restoreItemFromMemento(org.eclipse.ui.IMemento)

\*/

protected Object restoreItemFromMemento(IMemento element) {

return element.getString("resource"); //$NON-NLS-1$

}

/\*

\* @see org.eclipse.ui.dialogs.FilteredItemsSelectionDialog.SelectionHistory#storeItemToMemento(java.lang.Object,

\* org.eclipse.ui.IMemento)

\*/

protected void storeItemToMemento(Object item, IMemento element) {

element.putString("resource", item.toString()); //$NON-NLS-1$

}

}

Open the dialog.

Select listed elements and click OK:

AbCdEfGh

AbCdEfGi

AbCdEfGj

AbCdEfHh

AbCdEfHi

AbCdEfHj

abCd

abCe

abCf

abDd

abDe

Our example dialog now looks like this:

Adding custom filters

Next we will expand the filter to hide all strings that start with a lower case character.

Create a subclass of FilteredItemsSelectionDialog.ItemsFilter, implement the necessary abstract methods and override equalsFilter(ItemsFilter) and isSubFilter(ItemsFilter). These two methods are used to optimize filtering of the view. If a new filter is added that is equal to the existing filter, no further filtering is required. If the new filter is a sub-filter of the existing filter, then filtering is only done on the items that matched the existing filter.

private boolean onlyLowerCase = true;

private class ResourceFilter extends ItemsFilter {

public final boolean onlyLowerCase = FilteredResourcesSelectionDialogExample.this.onlyLowerCase;

public boolean matchItem(Object item) {

String resource = item.toString();

if (onlyLowerCase && Character.isUpperCase(resource.charAt(0)))

return false;

return matches(resource);

}

public boolean equalsFilter(ItemsFilter filter) {

ResourceFilter resourceFilter = (ResourceFilter) filter;

if (onlyLowerCase != resourceFilter.onlyLowerCase)

return false;

return super.equalsFilter(filter);

}

public boolean isSubFilter(ItemsFilter filter) {

ResourceFilter resourceFilter = (ResourceFilter) filter;

if (onlyLowerCase != resourceFilter.onlyLowerCase)

return false;

return super.isSubFilter(filter);

}

public boolean isConsistentItem(Object item) {

return true;

}

}

On your subclass of FilteredItemsSelectionDialog, override the createFilter() method:

protected ItemsFilter createFilter() {

return new ResourceFilter();

}

Now the dialog will only display strings that start with a lower case letter.

Adding an extra check-box to the dialog

We will use a check-box and a menu action to indicate whether to filter strings that start with a lower case character.

Extend the dialog's content area by implementing createExtendedContentArea(Composite) method:

private Button checkButton;

protected Control createExtendedContentArea(Composite parent) {

checkButton = new Button(parent, SWT.CHECK);

checkButton.setText("Only Lower Case Strings"); //$NON-NLS-1$

checkButton.addSelectionListener(new SelectionListener() {

public void widgetDefaultSelected(SelectionEvent e) {

}

public void widgetSelected(SelectionEvent e) {

if (onlyLowerCase != ((Button) e.widget).getSelection()) {

onlyLowerCase = ((Button) e.widget).getSelection();

applyFilter();

}

}

});

return checkButton;

}

Next, create a new action and add it to the menu by overriding fillViewMenu(IMenuManager). Eg.:

private Action showOnlyLowerCaseStringsAction = new ShowOnlyLowerCaseStringsAction();

private class ShowOnlyLowerCaseStringsAction extends Action {

/\*\*

\* Creates a new instance of the action.

\*/

public ShowOnlyLowerCaseStringsAction() {

super("Only Lower Case String", //$NON-NLS-1$

IAction.AS\_CHECK\_BOX);

}

public void run() {

if (onlyLowerCase != isChecked()) {

onlyLowerCase = isChecked();

applyFilter();

}

}

}

protected void fillViewMenu(IMenuManager menuManager) {

super.fillViewMenu(menuManager);

menuManager.add(showOnlyLowerCaseStringsAction);

}

At the end override applyFilter() as follows:

protected void applyFilter() {

super.applyFilter();

checkButton.setSelection(onlyLowerCase);

showOnlyLowerCaseStringsAction.setChecked(onlyLowerCase);

}

Now open the dialog:

7. The JFace UI framework

We've seen that the workbench defines extension points for plug-ins to contribute UI functionality 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 includes classes for handling common UI programming tasks:

Viewers handle the drudgery of populating, sorting, filtering, and updating widgets.

Actions and contributions (Deprecated) 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.

Field assist provides classes that help guide the user in choosing appropriate content for fields in dialogs, wizards, or forms.

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 functionality 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.

7.1. 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 extra functionality. The following classes are supported by the TableViewer, AbstractTreeViewer and TableTreeViewer.

IColorProvider. Make your label provider implement this interface to set a different foreground or background color for an item. Unless system colors are used, these colors should be cached to minimize the amount of system resources used. Be sure to dispose of any colors you create when your label provider is disposed.

IFontProvider . Implement this interface in your label provider to set the font of items in the view. Fonts should also be cached, and disposed when your label provider is disposed. Control.getFont() should be called as little as possible because it 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 it.

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 by calling IWorkbench.getDecoratorManager().

IColorDecorator An IColorDecorator is an object that can support decorating foreground and background colors.

IFontDecorator An IFontDecorator is an object that can support decorating fonts.

It is possible to affect the color of items in a view either from within the view's label provider, or via a decorator. Generally it is better to use the color and font support in label providers, since decorators affect every view that shows a particular type. If you do use a color or font decorator make sure its 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.

7.2. Actions and contributions (Deprecated)

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.

7.3. 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 by referring to it via its 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.

7.4. 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.

7.5. Field assist

Often, a user is expected to provide textual information in a simple field such as a text field or combo box. Although the application code that populates these fields is generally much simpler than code that populates a complex widget such as a table or tree, these "simple" fields usually place more burden on the user. The user must identify which fields require content, whether a field contains valid content, and what choices are expected. The JFace field assist support provides classes that help guide the user through input tasks.

The org.eclipse.jface.fieldassist package provides assistance in two ways. Control decorations allow you to place image decorations adjacent to a control in order to cue the user about the status of a particular field. Content proposal support allows you to provide a content assist popup that provides content choices for the user.

Control decorations

A control decoration is a rendered image that can be placed adjacent to a field in a window or dialog. Decorations may be placed adjacent to a control in one of six positions (top, center, or bottom to the left or right of the control). One or more control decorations may be defined for a control. The API for ControlDecoration allows you to hide and show the decoration, assign descriptive text to the decoration, and listen to events associated with the decoration.

Creating a control decoration

Creating a control decoration is straightforward. Clients simply specify the control to be decorated and SWT constants describing the position of the decoration relative to the control. Consider this snippet, in which an application creates a text control inside one of its dialogs:

...

// Create a text field

Text text = new Text(parent, SWT.BORDER);

text.setText("some text");

...

A decoration can then be created for the control.

...

// Create a control decoration for the control.

ControlDecoration dec = new ControlDecoration(text, SWT.TOP | SWT.LEFT);

...

Once the decoration is created, its image and text can be specified.

...

// Specify the decoration image and description

Image image = JFaceResources.getImage("myplugin.specialimage");

dec.setImage(image);

dec.setDescriptionText("This field is special");

Clients can use the setShowOnlyOnFocus method to specify whether the decoration should be shown only when the control has focus, or whether it should be shown at all times. The setShowHover method allows clients to configure whether the description text is shown in a hover when the mouse hovers over the decoration.

Laying out decorated controls

When adding decorations to controls that appear inside a dialog or window, you should make sure there is enough space adjacent to the control to render the decoration without overlapping other controls. For example, when using a grid layout in a dialog, there should be enough margin space between the cells in the grid so that a decoration can be shown adjacent to controls in the cells.

...

// Set the layout data to ensure there is enough space for the decoration

GridData data = new GridData(IDialogConstants.ENTRY\_FIELD\_WIDTH, SWT.DEFAULT);

data.horizontalIndent = image.getBounds().width;

text.setLayoutData(data);

...

The width of a decoration is simply the width of its image. However, layout can get more complicated if you are using decorations with different widths. If this is the case, you can simplify things by first creating field decorations to represent all of your control decorations.

Field decorations

A FieldDecoration is simply a data object that combines the image and text used to show a decoration. Once created, these field decorations can be registered in the FieldDecorationRegistry.

Field decoration registry

The field decoration registry allows you to register and access field decorations using a string id. This provides a convenient way for you to refer to decorations used throughout your application. You may choose to define API that exposes your decoration ids if you wish to make them available to other plug-ins. Note that registering a decoration does not manage the life-cycle of the images inside those decorations. Your application can decide how to manage these images. For example, the JFace image registry may be used to register and manage the image's life-cycle. Alternatively, your application may wish to create the image on demand and dispose of it when it is no longer needed. The javadoc for the registration methods in FieldDecorationRegistry explains the different ways that images can be specified when registering a decoration. To determine the margin width needed for decorations, you can use the FieldDecorationRegistry protocol to access the width of the largest decoration and create the necessary indent.

...

// Set the layout data

GridData data = new GridData(IDialogConstants.ENTRY\_FIELD\_WIDTH, SWT.DEFAULT);

data.horizontalIndent = FieldDecorationRegistry.getDefault().getMaximumDecorationWidth();

text.setLayoutData(data);

...

Although the field assist support does not dictate how decorations should be used, the registry does define standard decorations that can be used by applications to show certain states for a field. For example, the following snippet uses a standard decoration for indicating an error in a field:

...

// Create a control decoration to indicate an error.

ControlDecoration dec = new ControlDecoration(text, SWT.TOP | SWT.LEFT);

FieldDecoration errorFieldIndicator = FieldDecorationRegistry.getDefault().

getFieldDecoration(FieldDecorationRegistry.DEC\_ERROR);

dec.setImage(errorFieldIndicator.getImage());

dec.setDescriptionText(errorFieldIndicator.getDescription());

...

Content proposals

In addition to annotating fields with decorations, applications may provide a content proposal assistant that activates a proposal popup for a field. You may install a ContentProposalAdapter on an arbitrary control in order to provide this behavior. The following snippet installs a content proposal adapter on a text control. Note that this text control could be a control created directly by the application or one obtained from a decorated field.

...

autoActivationCharacters = new char[] { '#', '(' };

keyStroke = KeyStroke.getInstance("Ctrl+Space");

// assume that myTextControl has already been created in some way

ContentProposalAdapter adapter = new ContentProposalAdapter(

myTextControl, new TextContentAdapter(),

new SimpleContentProposalProvider(new String [] {"ProposalOne", "ProposalTwo", "ProposalThree"}),

keyStroke, autoActivationCharacters);

In order to get and set the content of the control when the user chooses a proposal in the popup, the adapter must be supplied with an instance of IControlContentAdapter, which can retrieve and set the contents of a particular kind of control. For text fields, you can use the class TextContentAdapter. However, you have the flexibility to implement IControlContentAdapter to use the content proposal adapter with any other kind of control.

When creating a content proposal adapter, you must also specify an instance of IContentProposalProvider, from which the proposals themselves are retrieved. This provider is responsible for returning an array of content proposals. The proposals themselves are specified as instances of IContentProposal, from which the label and content of the proposal can be obtained, in addition to other information, such as a detailed description of the proposal.

In the example above, the SimpleContentProposalProvider is used. This provider is defined by specifying a simple array of Strings as the content proposals. The simple provider implements the necessary protocol to map each string into the expected IContentProposal. The flexibility of IContentProposalProvider allows you to implement a proposal provider with advanced features, such as filtering the proposals based on the control's content, providing explanatory labels in the popup instead of the actual content that will be inserted, and specifying the expected cursor position after a proposal is inserted. See the Field Assist Example and search for implementors of IContentProposalProvider for advanced usage.

Configuring a content proposal adapter

We've seen that the basic definition for a content proposal adapter includes the control for which the proposals are provided, the content adapter used to alter the content of the control, and the proposal provider that defines the list of proposals in the popup. In addition to these basics, there are many ways that the content proposal adapter can be configured:

setAutoActivationCharacters allows you to specify an array of characters that will automatically trigger the content proposal popup when typed into the control.

setAutoActivationDelay defines the delay in milliseconds between the time the user types an autoactivation character and the popup is opened.

setFilterStyle allows you to control whether any automatic filtering of proposals is done, based on the keystrokes typed while the popup is active.

setLabelProvider allows you to specify a label provider which will provide an image and string for each proposal.

setPopupSize allows you to specify the desired size of the proposal popup when it is activated.

setPropagateKeys controls whether keys typed into an open proposal popup should also be propagated back to the control.

setProposalAcceptanceStyle allows you to control how an accepted proposal affects the content of the control (insertion, replacement, custom implementation).

the keyStroke parameter in the constructor can be used to activate the content proposal popup using an explicit key sequence that does not affect the content of the control.

The Field Assist Example allows you to configure these various options in the example preferences and try out the different combinations. For example, the adapter can be configured so that it is invoked explicitly with a keystroke and inserts the proposal content into the control, causing it to behave much like the text editor content assist. See the javadoc for more specifics about each of these methods and how they interact with each other.

Auto complete fields

The content proposal adapter methods can be used to configure an adapter so that it behaves more like the type-ahead field completion used in web browser URL or search fields. AutoCompleteField can be used when this style of interaction is desired. Clients need only specify the list of completions when defining an auto complete field. Configuration of the content proposal adapter and proposal provider will be handled internally.

Workbench field assist

Field assist support at the JFace level gives your application a lot of flexibility in determining how to decorate fields and show proposals for field content. This is desirable for stand-alone JFace applications or stand-alone rich client applications. However, if your application is intended to integrate with other plug-ins, such as the Eclipse SDK or third-party plug-ins, you will probably want to use the field assist support in a way that is consistent with other plug-ins. The workbench defines utility classes that use field assist for specific kinds of interactions.

For example, the class ContentAssistCommandAdapter configures a content proposal adapter for content-assist style insertion. It provides a handler for the workbench-level content assist command, so that the content proposal popup is opened when the user invokes the keystroke or trigger sequence that has been specified in the workbench key bindings. It can optionally provide a control decoration with the content assist light bulb image. See the org.eclipse.ui.fieldassist package for more detail about workbench-level field assist.

This package is expected to evolve as the workbench expands its use of field assist and standardizes the use of decorations for certain field states.

8. 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.

8.1. 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 Shells 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.

8.1.1. 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.

DateTime

Selectable user interface object that allows the user to enter and modify date or time values.

ExpandBar

Composite control that groups pages that can be shown or hidden by the user with labeled headers.

ExpandItem

Selectable user interface object corresponding to a header for a page in an ExpandBar.

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.

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.

8.1.2. 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 (XyzListener), a class that provides event information (XyzEvent), and an API method to add the listener (addXyzListener). If there is more than one method defined in the listener interface then there is an adapter (XyzAdapter) 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, Deactivate

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.

EraseItem

A TableItem or TreeItem is about to have its background drawn.

Expand, Collapse

An item in a Tree is expanded or collapsed.

Gesture

The user has used a touch-based input source to perform a gesture over the control.

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.

ImeComposition

Allows custom text editors to implement in-line editing of international text.

MeasureItem

The size of a custom drawn TableItem or TreeItem is being requested.

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.

Movement

An updated caret offset is needed in response to a user action in a StyledText.

OpenDocument

The operating system has requested that a document be opened.

OrientationChange

The orientation of a Text control is changing.

PaintItem

A TableItem or TreeItem is about to have its foreground drawn.

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.

Settings

An operating system property, such as a system font or color, has been changed.

Show, Hide

A control's visibility has changed.

Skin

A control needs to be skinned.

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.

MouseHorizontalWheel, MouseVerticalWheel, MouseWheel

The mouse wheel has been rotated.

Paint

The control has been damaged and requires repainting.

Touch

The user has touched a touch-based input source over the control.

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);

8.1.3. 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.

8.2. 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.

8.2.1. 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.

8.3. 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.

8.4. 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.

8.5. 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 ();

}

});

8.6. SWT Examples

This page lists places where you can find SWT examples and more information on SWT.

Examples Guide

Within this Platform Plug-in Developer Guide is an Examples Guide that describes how to install and run the Eclipse platform examples. SWT has contributed many of these examples, some as Eclipse plug-ins, and others as standalone SWT examples.

SWT Snippets

A great collection of SWT examples can be found on the SWT Snippets page: http://www.eclipse.org/swt/snippets/. Here, you will find hundreds of simple stand-alone SWT examples.

More Information

For further information on SWT, multi-platform screenshots, FAQ, links to books, articles, and tutorials, and more, please visit: http://www.eclipse.org/swt/.

9. 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)

9.1. 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

9.1.1. 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.

9.1.2. 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.

9.1.3. 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.

9.1.4. 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 fly

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, false, 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.

9.1.4.1. 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. Soft line breaks can be added using <br/> tag. The text is parsed as XML, this means that there must be an end tag for each element so you cannot use open <br> as in HTML). The XML entity &amp; is recognized and replaced with &, and it is required that ampersand (&) characters which are not part of an entity declaration be represented as &amp;. Numeric entities may also be used in formatted text.

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.

9.2. 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).

9.2.1. TableWrapLayout

TableWrapLayout is a grid-based layout very similar to SWT's versatile GridLayout. It differs from GridLayout in that it uses a layout algorithm that works more like HTML tables. It tries to respect the provided client area width and grows 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 of TableWrapLayouts from the form body to each text control that needs to wrap.

9.2.2. 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).

9.3.1. 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.

9.3.2. 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 'getForm()' 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).

9.3.3. 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

9.4. 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:

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.

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, functionality 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).

10. Common Navigator Framework

A JFace Viewer provides the user with a view of objects using a single content provider, label provider, sorter and filter. The Common Navigator Framework (CNF) extends this idea by allowing a single view to dynamically use multiple and unrelated sets of content providers, label providers, sorters and filters. These can be activated in the view depending on declarative expressions or using API calls. The CNF implemented by the org.eclipse.ui.navigator plugin.

The CNF uses the idea of Navigator Content Extensions (NCE) which can refer to a content provider, label provider, sorter, or drag adaptor (note that filters are configured separately). An NCE has associated expressions that tell it when it is active. NCEs are also presented to the user in the view context menu so that user can turn them on or off in order to show the view in different ways. Examples of NCEs are a resource content extension that controls how resources are presented and Java content extension that shows Java projects. In the IDE's Project Explorer, you can turn off the Java content extension getting a pure resource view of the workspace.

By using the NCEs it's possible to include different model objects directly in the view. These model objects might be related in a cascading fashion. For example a resource object (an IFile) might refer to a Java class object. This Java class object can be presented directly in the view by defining an NCE. The Java class object can have children (representing methods, fields, etc) all of which appear in the view. You can then add another NCE to display model objects related to the Java (or resource object) that display content related to a Java Server Page object. This can all be done without the "lower level" NCEs (resources, Java) being aware of the existence of the "outer" NCEs (JSP). Further, it's possible for outer NCEs to override by suppressing processing associated with lower level NCEs according to their requirements. And even further, it's possible (using the pipelining mechanism) for even level in this cascaded set of NCEs to arrange the presentation of the objects according to their specifications, moving, adding, or suppressing objects as necessary.

The CNF has depends only on org.eclipse.ui and is therefore suitable for use both within the IDE and in an RCP application.

The CNF is highly configurable and many of its components can be used separately (though this is not the typical case). For example, it's possible use the service that manages content extensions with a viewer other than the standard CommonViewer.

The CNF documentation has the following sections:

Configuration Overview - A high level discussion of the configuration.

Operational Topics - A detailed discussion of various topics in the operation of the CNF

Step-by-step Instructions - Provides configuration instructions for various scenarios.

Troubleshooting - Some suggestions for troubleshooting.

Major Components of the Common Navigator Framework

The CNF has the following major parts:

View Part - CommonNavigator .

Viewer -CommonViewer the implementation of a viewer that is contained in the above view part. You can also use the CNF with your own viewer that can be displayed anywhere.

Navigator Content Extensions - org.eclipse.ui.navigator.navigatorContent This extension point is where you configure the collections of content you want to make visible.

Navigator Viewer - org.eclipse.ui.navigator.navigatorContent This extension point is used to declare the relationship between the NCEs and a particular viewer. Regular expressions may be used here to allow a viewer to dynamically detect NCEs that it does not know about beforehand. visible.

Resource Support - Resources are one of the main types of objects managed by the CNF. To support this the org.eclipse.ui.navigator.resources plug-in provides all of the necessary definitions. The Project Explorer is configured using these definitions.

Navigators - Where the CNF Fits

There are currently 3 major navigators in the Eclipse IDE.

Project Explorer - This is an instance of the CommonNavigator that is provided by the org.eclipse.ui.navigator.resources plugin. It provides a view of the workspace and has a large number of NCEs contributed for resources, Java, C, Web Tools, Data Tools, PHP, etc.

Package Explorer - Provided by the Java Development Tools (JDT) UI project, this provides a view of Java classes for the workspace. Generally speaking, the presentation of the Package Explorer and Project Explorer for Java objects is substantially similar. The JDT is not used for projects beyond Java.

Navigator - This view is an implementation of the now deprecated (as of 3.5) org.eclipse.ui.views.navigator ResourceNavigator class provided in the org.eclipse.ui.ide plugin. It shows only the workspace resources and does not support extensibility to show other content. In a future release, the Navigator view will be provided by an implementation of the CommonNavigator.

We also provide instructions for migration from the ResourceNavigator to the CNF.

The only navigator intended for general purpose client use if the CNF, which can be used either by adding NCEs visable in the Project Explorer (if developing an IDE plugin), or using a separate instance of the CommonNavigator as a view.

Navigator Content Extensions (NCE)

The CNF allows you to include not only resources but any type of object. One typical use case is that of a resource (consider a Java file for example) which has several subordinate objects presented as tree nodes representing aspects of the class like imports, methods, etc. The CommonViewer (portion of the Common Navigator) will include the IResource object representing the Java file, and then as its children an object for each import, method, etc. The mechanism for defining this is a navigator content extension.

Each content extension also provides a means of associating objects with appropriate icons, labels and menu items.

Content extensions are invoked based on an enablement expression which is defined using core expressions in the extension point. Enablement expressions are commonly defined by sensing the class of an object.

A priority may be associated with a content extension which helps to determine the order of invocation of the NCEs. It's possible that more than one content extension may be enabled for a given object and situation. Priorities are defined with words like "high" or "normal". A high priority would be associated with an extension that defines the primary UI for frequently manipulating an object. A low priority would be given to relatively infrequent utility sorts of UI.

Content extensions may be activated and deactivated for a given view part in the UI using the Filters and Customization menu item in the header menu associated with the view part. Common filters are also may also be activated and deactivated the same way.

Common Filters

Filters allow the user to specify which resources or objects to exclude. Like content extensions, filters may be specified centrally and shared.

Common Wizards

References to wizards for new/import/export may be defined. These are included in the appropriate popup menu based on the enablement.

Action Providers

Though the CNF provides a mechanism to work with actions, it is recommended that you instead use the Workbench Commands instead of actions.

Action providers provide a means to configure the retargetable actions and programmatically configure the popup menu in a CNF viewer. These are useful for when you must perform a computation to determine which items are added to the menu, or to adjust the retargetable actions to ensure that the user keystrokes are handled properly (like for Cut/Copy/Paste).

Drag and Drop Support

A drop assistant may be associated with a content extension to provide handling of additional (non-standard) transfer types or validation in the event of a drop.

Link with Editor Support

The link helper extension point provides a flexible mechanism of determining the appropriate selection in the navigator when an editor is activated. Conversely, it also provides the appropriate editor to be activated when the selection changes.

10.1. Configuring the Common Navigator

This section defines the configuration at a conceptual level; some level of detail is omitted for clarity and conciseness. For the complete details see the extension point documentation or the Operational Topics section.

org.eclipse.ui.navigator.viewer

org.eclipse.ui.navigator.navigatorContent

org.eclipse.ui.navigator.linkHelper

Viewer Configuration

As the CommonNavigator is a View, it is added using the org.eclipse.ui.views extension point. The CNF configuration aspects of the view instance are specified with a corresponding org.eclipse.ui.navigator.viewer extension point.

It is possible to use the CNF facilities for an arbitrary TreeViewer (??? could this be any StructuredViewer). In this case you still need to use this extension point, but it is used only to bind the CNF view information with the NCEs. You then programmatically (??? how) bind your Viewer with the NavigatorContentService.

The view may be associated with Navigator Content Extensions that define how its content is found and rendered. The entries here are used to look up NCEs, common filters and common wizards.

The view may also be associated with Action Providers that define code for programmatic updating and provision of actions or retargetable actions. The entries here are used to look up action providers.

The view may also be associated with Link Helpers that define the relationship between the selection in the view and the active editor.

Content extensions, common wizards, common filters, action providers and link helpers are bound to the view using an include/exclude mechanism and with the capability of pattern matching. This allows the actual content extensions (defined below) to be specified in a granular fashion such that views can select only those they actually need. The point of the exclude part of the mechanism is to exclude undesired items that were specified in the include statement. For example, the include statement could specify "com.mycompany.content.\*" and exclude could remove the test content extensions by saying "com.mycompany.content.test.\*".

A view is always associated with a popup menu. One important part of the popup menu configuration at this level is the insertion points, which are the locations where menu items can be added depending on when and where the menu is shown. By default, a standard set of insertion points is provided (described in the extension point documentation). However, you can can define your insertion points directly using popupMenus.

You can also direct the view to use a specific popup menu by id using popupMenuId, and you can indicate whether to ignore platform action contributions to the menu.

In the startup case, it is necessary to call some content extensions on the root node of the tree of the view in order to get the initial set of children. This set of content extensions is specified by setting the isRoot() property on the viewerContentBinding.

This defines:

viewerContentBinding - bind a Content Extension or Common Filter specified with the org.eclipse.ui.navigator.navigatorContent/navigatorContent extension point;

viewerActionBinding - bind action providers specified with the org.eclipse.ui.navigator.navigatorContent/actionProvider extension point;

popupMenus - defines the insertion points (standard places where menu items can be added) for the popup menu associated with the viewer. Mutually exclusive with popupMenuId.

popupMenuId - causes the view to use the menu defined by the org.eclipse.ui.popupMenus extension point. Mutually exclusive with popupMenus.

dragAssistent - points to code that may provide extra transfer types to be used when starting a drag from the viewer.

options - specifies options used to control the presentation of the view. For example these allow hiding of menus and buttons shown at the top of the view. Theses options are defined by INavigatorViewerDescriptor

Navigator Content Extension (NCE)

A navigator content extension, specified with org.eclipse.ui.navigator.navigatorContent/navigatorContent defines a named collection of attributes (content provider class, label provider class, icon, etc) to be enabled under certain conditions (typically in response to selection in the navigator). An example of a content extension is that for a resource. Content extensions are useful to describe other model objects that might be included in the view.

The definition of content extensions is separated from their association with a particular view instance allowing them to be shared and reused. Each content extension has an id, which is used to bind it with a view, and a display name which is presented to the user to allow the content extensions to be activated or deactivated by the user interface associated with the each view.

Content extensions may be active or not. This can be controlled by the user using the Filters and Customization menu item of the navigator. Inactive content extensions are not considered when processing the view.

The content extension defines the following:

triggerPoints - specifies the conditions when this content extension is enabled based on a given object. You can specify tests with core expressions on the object for which the content extension should be enabled.

possibleChildren - like triggerPoints, but used for the case where the parent of the desired content extension is known, like in the drop case where we need to determine which content extension is required to handle the drop based on the drop target which is the parent of the eventual object that will be added to the tree.

enablement - specifies an enablement that is both a triggerPoints and possibleChildren.

labelProvider - an ILabelProvider provides the text to be shown in the view.

contentProvider - anITreeContentProvider provides the means of getting the parent and child objects for the viewer. Other interfaces are possible, see the extension point documentation for these.

descriptionProvider - IDescriptionProvider provides a description that is shown in the status bar.

activeByDefault - indicates this content extension should be made active in the default configuration of the workbench (e.g. a new workspace).

priority - used to determine which content extension to use in the event that multiple extensions are enabled (based on their enablement conditions).

icon - used to associate the object with a specific icon.

providesSavables - indicates the content provider provides saveables. If true the content provider must adapt to SavablesProvider.

Common Filter

A common filter, specified with org.eclipse.ui.navigator.navigatorContent/commonFilter defines a filter that is controllable by the user in the view. The filter definition contains an id, a description that describes the filter, a description that describes what is filtered out, and the conditions (using core expressions) that identify objects the filter suppresses.

The common filter is bound to the view using the viewerContentBinding element of the viewer configuration.

Common Wizard

A common filter, specified with org.eclipse.ui.navigator.navigatorContent/commonWizard defines wizard that is to be shown in the new, import, or export menu.

The common wizard is bound to the view using the viewerContentBinding element of the viewer configuration.

Action Providers

An action provider, specified with org.eclipse.ui.navigator.navigatorContent/actionProvider allows you to specify a class, subclassing org.eclipse.ui.navigator.CommonActionProvider that is invoked at right-click and selection time to allow you to contribute to the popup menu or to the action bars.

Action providers may be defined at the top level of the extension point, not associated with any content extension. These action providers are named and then bound to the CommonViewer using the org.eclipse.ui.navigator.viewer/viewerActionBinding.

Action providers may also be associated with a content extension, in which case they are active with the content extension. This is done by including the action provider inside of the org.eclipse.ui.navigator.navigatorContent/navigatorContent extension point.

Link Helpers

A link helper, specified with org.eclipse.ui.navigator.linkHelper allows you to control how the "Link with Editor" matches the selection in the navigator with an editor and matches an active editor with the selection in the navigator.

10.2. Operational Topics

This section covers the operation of the CNF in detail.

Content Provider selection

Selecting the content provider is done by finding one or more NCEs associated with an object. The CNF is registered as a content provider a viewer and thus gets called at the content provider APIs in response to actions by the user on the viewer. In general, if the user is navigating by expanding in the viewer, the getElements() or getChildren() methods are called. However if the user has selected some object (like in an editor) and wishes to show it in the viewer, the getParent() method is used because the viewer needs to be able to figure out the part of the tree between the object and the content currently visible in the viewer.

When selecting an NCE in reaction to the getElements() or getChildren() call on the viewer, the triggerPoints (or enablement if specified) expression is evaluated against the object. The content provider associated with that NCE is invoked. If there are multiple content providers enabled by their triggerPoints (or enablement) expressions, they are all invoked in order according to priority and their results are concatenated.

When selecting an NCE in reaction to the getParent() call on the viewer, the possibleChildren (or enablement if specified) expression is evaluated against the object. The content provider associated with that NCE is invoked. If there are multiple content providers enabled by their possibleChildren (or enablement) expressions, the content providers are invoked in priority order. The first non-null parent returned by the content provider is used.

The overrides element allows an NCE to be overridden by another. In this case the content provider associated with the suppressed NCE will not be invoked to contribute.

During the initialization of the viewer, the it gets the initial input by invocation of the CommonNavigator.getInitialInput(). The default implementation of this method is to return the input of the current page. Thus, the initial input becomes the first object that is evaluated for the selection of NCEs.

10.3. Procedures and Examples for Using the CNF

The sections below define the set of steps for using the CNF in various scenarios. Some of the sets of below have a plugin in the UI examples that shows the end result.

Creation of a Common Navigator View

Content and Action Binding

Migrating from the ResourceNavigator

10.3.1. Creation of Common Navigator View

In order to use the CNF-based view in the application several steps are required which are discussed below. The created view can be configured by binding specific content and actions to it, which is discussed in the Content and Action Binding.

The example plugin org.eclipse.ui.examples.navigators.resources shows this.

Add the following as dependent plug-ins:

org.eclipse.ui

org.eclipse.ui.navigator

Add a View extension (org.eclipse.ui.views) and provide org.eclipse.ui.navigator.CommonNavigator as the value of class attribute. Please note, that the id of the view will be required in later steps for the content and action bindings.

<extension

point="org.eclipse.ui.views">

<view

name="View"

class="org.eclipse.ui.navigator.CommonNavigator"

id="example.view">

</view>

</extension>

Please note, that if you are defining your own perspective in which the Common Navigator is used, the showTitle parameter must be set to true in order the viewer renders correctly (see bug 235171).

public void createInitialLayout(IPageLayout layout) {

String editorArea = layout.getEditorArea();

layout.setEditorAreaVisible(false);

layout.setFixed(true);

layout.addStandaloneView("example.view", true /\* show title \*/, IPageLayout.LEFT, 1.0f, editorArea);

}

10.3.2. Content and Action Binding

The sections below define the set of steps for usage of resource content and definition of own content for the CNF-based viewer. Finally, the steps needed for Action Binding are described. As described previously, in order to render content the viewer selects the corresponding NCEs based on the evaluation of the expressions on the selected objects.

Usage of Resource Content

One of the use cases of usage of the Common Navigator is the manipulation of the workspace resources. The resources-related content is defined by the org.eclipse.ui.navigator.resources plugin. The example of the resource usage of CNF can be found in org.eclipse.ui.examples.navigator.resources.

Perform the general steps described in Creation of Common Navigator Viewer

Add the following as dependent plug-ins:

org.eclipse.ui.navigator.resources

Add a org.eclipse.ui.navigator.viewer extension that has:

viewerContentBinding, point this to Id of your view (example.view)

includes of:

org.eclipse.ui.navigator.resourceContent

org.eclipse.ui.navigator.resources.\*

<extension

point="org.eclipse.ui.navigator.viewer">

<viewerContentBinding

viewerId="example.view">

<includes>

<contentExtension pattern="org.eclipse.ui.navigator.resourceContent" />

<contentExtension pattern="org.eclipse.ui.navigator.resources.filters.\*"/>

</includes>

</viewerContentBinding>

</extension>

Please note, that using CNF inside of your own RCP requires three additional steps.

Add the org.eclipse.ui.ide as dependent plug-in

To get the resource workspace root (IWorkspaceRoot) as default page input, override the WorkbenchAdvisor.getDefaultPageInput() method in your WorkbenchAdvisor:

public IAdaptable getDefaultPageInput()

{

return ResourcesPlugin.getWorkspace().getRoot();

}

Hook the workbench adapters correctly before they are used, so add this code to the WorkbenchAdvisor.initialize(IWorkbenchConfigurer) method:

public void initialize(IWorkbenchConfigurer configurer)

{

IDE.registerAdapters();

}

Own Content Definition

Along with resource content provided by the org.eclipse.ui.navigator.resources plugin, application-specific content can be defined and presented in the CNF-based viewer. In order to define and use application specific you need to:

Use extension org.eclipse.ui.navigator.navigatorContent and declare the NCE, with it providers and trigger conditions for the content:

<extension

point="org.eclipse.ui.navigator.navigatorContent">

<navigatorContent

id="org.eclipse.ui.examples.navigator.propertiesContent"

name="Properties File Contents"

contentProvider="org.eclipse.ui.examples.navigator.PropertiesContentProvider"

labelProvider="org.eclipse.ui.examples.navigator.PropertiesLabelProvider"

activeByDefault="true"

icon="icons/prop\_ps.gif"

priority="normal" >

<triggerPoints>

<or>

<and>

<instanceof value="org.eclipse.core.resources.IResource"/>

<test

forcePluginActivation="true"

property="org.eclipse.core.resources.extension"

value="properties"/>

</and>

<instanceof value="org.eclipse.ui.examples.navigator.PropertiesTreeData"/>

</or>

</triggerPoints>

<possibleChildren>

<or>

<instanceof value="org.eclipse.ui.examples.navigator.PropertiesTreeData"/>

</or>

</possibleChildren>

</navigatorContent>

</extension>

For more details, please consult the CNF Configuration and Operational Topics.

Bind the content to the viewer using the org.eclipse.ui.navigator.viewer extension.

<extension

point="org.eclipse.ui.navigator.viewer">

<viewerContentBinding viewerId="example.view">

<includes>

<contentExtension pattern="org.eclipse.ui.examples.navigator.propertiesContent"/>

</includes>

</viewerContentBinding>

</extension>

Note, that instead of using the exact Id of the content, the usage of the regular expression is allowed in the pattern attribute, so org.eclipse.ui.examples.navigator.\* would also work.

Action Binding

The usage of actions in the viewer follows the same pattern as the content binding. For usage of existing actions on resources defined in the org.eclipse.ui.navigator.resources plugin:

Add the following as dependent plug-ins:

org.eclipse.ui.navigator.resources

Add a org.eclipse.ui.navigator.viewer extension that has:

actionContentBinding, point this to Id of your view (example.view)

includes of:

org.eclipse.ui.navigator.resources.\*

<extension

point="org.eclipse.ui.navigator.viewer">

<viewerActionBinding

viewerId="example.view">

<includes>

<actionExtension pattern="org.eclipse.ui.navigator.resources.\*" />

</includes>

</viewerActionBinding>

</extension>

10.3.3. Migrating from the ResourceNavigator

First follow the steps to add the Common Navigator and check that the steps required for usage of resource content are applied.

The following steps discuss the migration component for each part of the ResourceNavigator

org.eclipse.ui.ide.resourceFilters

Add the corresponding org.eclipse.ui.navigator.navigatorContent/commonFilter extensions.

FrameList support get/setFrameList()

The FrameList support is used for the Go Into functionality. Support for this is included in the ProjectExplorer subclass of CommonNavigator. You should subclass ProjectExplorer instead of CommonNavigator.

ResourcePatternFilter support get/setPatternFilter()

Configure the ResourcePatternFilter using the org.eclipse.ui.navigator.navigatorContent/commonFilter extension. Then you can access the instance of the filter using the following code (where yourViewer is the instance of the CommonViewer and yourFilterId is the string of the Id of your common filter configured above):

INavigatorContentService contentService = yourViewer.getNavigatorContentService();

INavigatorFilterService filterService = contentService.getFilterService();

ICommonFilterDescriptor[] fds = filterService.getVisibleFilterDescriptors();

for (int i = i; i < fds.length; i++) {

if (fds[i].getId().equals(yourFilterId))

return filterService.getViewerFilter(filterDescriptor);

}

IWorkingSet support get/setWorkingSet()

Working sets are not directly supported by the Common Navigator. The Project Explorer however does support working sets. There is however no current API in the Project Explorer to manipulate working sets. The Project Explorer provides a UI to work with working sets. If this sort of API is required, please file an enhancement request explaining the requirement.

ResourceSorter support get/setResourceSorter()

Use the sorting facility in the Common Navigator.

ResourceComparator support get/setResourceComparator()

Use ResourceSorter instead for now as the Common Navigator does not directly support ResourceComparators.

10.4. Troubleshooting

Check Your Ids

This is the most common problem in configuring the CNF. Make sure the Ids match exactly what you expect them to.

Is it a Wildcard or Regular Expression?

Be sure that the includes and excludes Ids for the navigator content extensions or filters are actually regular expressions and not wild cards. So if you are looking for com.mycompany.myplugin\* (this is a wildcard), make sure you use com.mycompany.myplugin.\* (which is a matching regular expression). The former will find things like com.mycompany.mypluginnnnn, but won't find com.mycompany.myplugin.function1.

Tracing

The CNF provides tracing that can be helpful to resolve problems. To enable the tracing go to the Tracing tab in the Launch Configuration Dialog and select Enable Tracing. Then check the org.eclipse.ui.navigator plugin, and on the right check debug and also one of the following:

debug/setup - Shows the processing of the configuration.

debug/dnd - Shows all aspects of drag and drop processing.

debug/resolution - Shows the resolution of an object to its navigator content extension.

debug/viewermap - Shows the low-level map that associates the Navigator Content Extensions with active elements in the viewer.

If you require support, include the tracing when you report your problem.

11. 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 file system

Resource properties

Project-scoped preferences

File encoding and content types

Linked resources

Virtual folders

Resources filters

Resource markers

Modifying the workspace

Batching resource changes

Tracking resource changes

Concurrency and the workspace

If you still haven't found what you need, venture on to the Advanced resource concepts section.

11.1. 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 in some file system.

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 in a file system 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.

11.2. Resources and the 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 one or more file systems. 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 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 started the platform SDK, you were prompted to provide a workspace directory. This is the directory where various plug-ins store interesting metadata that is unique to a particular instance of the platform. By default, the resources plug-in stores each project in a sub-directory of the workspace directory. Within these subdirectories are the folders and files that each project contains.

Let's say your chose the directory c:\MySDK\workspace for your workspace. Inside this 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 there is an option to periodically refresh 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 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 mapped to any arbitrary directory in some backing file system, perhaps even on a different machine.

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.getLocationURI. 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.findFilesForLocationURI or IWorkspaceRoot.findContainersForLocationURI.

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.

11.3. 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.

11.4. 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="preferences">

<scope name="project" class="org.eclipse.core.internal.resources.ProjectPreferences"/>

</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.getNode("com.example.myplugin");

if (projectNode != null) {

value = projectNode.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();

11.5. 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.

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.

11.6. 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 a file system outside of the project's location.

Linked resources can be located virtually anywhere in a 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 operations 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) or (String -> URI) mapping that defines a shortcut for a location in a 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.getRawLocationURI 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 getRawLocationURI method acts exactly like the getLocationURI 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>

<options 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"

11.7. Virtual Folders

Virtual folders are folders that do not exist in the file system. They are shown to the user with a unique folder icon.

The method IResource.isLinked returns true for virtual folders, and IResource.getLocation/IResource.getLocationURI always returns null.

Virtual folders can be created by passing the IResource.VIRTUAL flag to the IResource.create method.

By design, virtual folders can only have linked resources or other virtual folders as children.

11.8. Resource Filters

Resource filters allow the user to configure which files and folders are included automatically in a project resource hierarchy when refresh is performed.

By adding resource filters to a project or folder, the user can systematically prevent some file system entries to be displayed in the resource tree.

Resource filters can be created using the IContainer.createFilter method.

The FileInfoMatcherDescription object has to be initialized with the ID of a filterMatchers extension point. The org.eclipse.core.resources plugin specifiy one default filter matcher, of ID org.eclipse.core.resources.regexFilterMatcher, which accepts regular expressions Strings as argument. Other plugins distributed with the workbench (such as the org.eclipse.ui.ide plugin) define other reusable filter matchers.

Filter matchers are extensible through the org.eclipse.core.resources.filterMatchers extension point.

11.9. 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);

}

11.10. 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.

11.10.1. 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. When a non-null scheduling rule is used, any attempt to access a resource outside the scope of the scheduling rule will trigger an exception.

There are two special scheduling rules that are important to consider. First, if you use an instance of IWorkspaceRoot as the scheduling rule, it means your thread is blocking access to all resources in the tree. While a thread holds the root rule, no other thread is allowed to modify the workspace. Conversely, a rule of null indicates that the thread will block access to no resources in the tree. While a thread with the null rule is free to modify the workspace itself, other threads will not be prevented from performing their own modifications. The IWorkspaceRoot and null scheduling rules occupy opposite ends of the concurrency spectrum. With IWorkspaceRoot there is no concurrency and only one thread is modifying the workspace at a time. With a null rule, there is maximum concurrency as all threads can modify the workspace concurrently.

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, null);

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.

11.10.2. 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\_BUILD

Notifies listeners before any building occurs. This event is broadcast when an explicit build is requested, or when the platform detects a 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 end of the last POST\_BUILD event.

POST\_BUILD

Notifies listeners after any building occurs. This event is broadcast after an explicit build is requested, or when the platform detects a 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 end of the last POST\_BUILD event.

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 before any PRE\_BUILD and after any POST\_BUILD 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\_BUILD

| IResourceChangeEvent.POST\_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\_BUILD:

System.out.println("Build about to run.");

event.getDelta().accept(new DeltaPrinter());

break;

case IResourceChangeEvent.POST\_BUILD:

System.out.println("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.

11.10.3. 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.

12. Advanced resource concepts

In Resources overview we took a look at the basic services and APIs provided by the org.eclipse.core.resources plug-in. This and other related plug-ins also contain many advanced features that are useful for plug-ins that require a deeper level of integration. For details on these advanced features, take a gander at some of these topics:

Project natures

Incremental project builders

Project Build Configurations

Derived resources

Workspace save participation

Resource modification hooks

Refresh providers

Alternate file systems

Using the file system API

Working with resources in other file systems

File system providers

User interface support for alternative file systems

Dynamic path variables

12.1. 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(newNatures);

// 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.myOtherNature");

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

IProjectNatureDescriptor[] descriptors = workspace.getNatureDescriptors();

12.2. 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. Resources created by a builder are typically marked as derived resources.

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.

Projects support multiple build configurations which allow Incremental Project Builders to build a project in more than one way while providing a resource delta for each build configuration.

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. No delta is available.

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 or workspace auto-build have been requested, a resource delta is provided to describe the changes in the resources since the last build.

NOTE: If no delta is available, the build kind will always be FULL\_BUILD. See Advanced Build Kinds for more on build kinds.

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.

Build Locking

By default, builders run with the Workspace Root scheduling rule. This prevents other threads from modifying the Eclipse workspace while the build is in progress. Long running builders, or builders which do not care about concurrent changes in parts of the resource tree, may wish to allow modification while the build is in progress. A builder may relax the scheduling rule its run with by overriding the #getRule method:

public ISchedulingRule getRule(int kind, Map<String,String> args) {

// Allow any resource to be modified concurrently with this buidler's invocation.

return null;

}

Discovering what's being built

It's sometimes useful for a builder to discover why its being built. Builders have access to a build context by calling

IBuildContext buildContext = getContext();

When building a project and its references, projects may be implicitly built if they are reachable in the reference graph of the top-level project the user requested built. With this API a builder can discover which build configurations have been built before it (IBuildContext#getAllReferencedBuildConfigs()), will be built after it (IBuildContext#getAllReferencingBuildConfigs()), as well as which build configurations the user asked to be built (IBuildContext#getRequestedConfigs()).

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. A common way to associate a builder with a project is by configuring a project nature.

Advanced Build Kinds

Triggers vs. Kinds

When a build is invoked on an IProject or IWorkspace, we call the passed in build kind the build trigger. This trigger is usually passed through to the IncrementalProjectBuilder as the build kind argument when IncrementalProjectBuilder#build is invoked.

The major exception to this is where no delta exists. In this case the build trigger is always promoted to FULL\_BUILD.

The mapping between triggers and kinds looks like:

Trigger

Kind

FULL\_BUILD

FULL\_BUILD

INCREMENTAL\_BUILD

INCREMENTAL\_BUILD or FULL\_BUILD

AUTO\_BUILD

AUTO\_BUILD or FULL\_BUILD

CLEAN\_BUILD

IncrementalProjectBuilder#clean(...)

Ignoring Build Kinds

The platform provides API for configuring which build kinds a builder responds to. If the builder is marked as 'isConfigurable' in its extension point it can be configured at runtime to respond to, or ignore, certain build triggers.

For example a long running builder which doesn't want to be called during workspace AUTO\_BUILD, can use:

ICommand#setBuilding(AUTO\_BUILD, false);

It is important to note that:

if INCREMENTAL\_BUILD (or AUTO\_BUILD) is promoted to FULL\_BUILD, a builder that responds to INCREMENTAL\_BUILD (or AUTO\_BUILD) will be called,

if INCREMENTAL\_BUILD is promoted to FULL\_BUILD, a builder that responds to FULL\_BUILD will be called,

if AUTO\_BUILD is promoted to FULL\_BUILD, only builders that respond to AUTO\_BUILD will be called. This is to allow long running builders to ignore AUTO\_BUILD requests completely.

See ICommand for more information.

12.2.1. Project Build Configurations

The platform provides Build Configurations to allow integrations to build a project in multiple orthogonal ways. All the functionality provided to an Incremental Project Builder is provided per Build Configuration.

Build configurations are very useful when a project's files can be built in more than one way. In statically compiled C/C++, for example, the preprocessor is used to re-write or conditionally compile a single source file to multiple object files. Users can build a project in a 'debug' configuration: including asserts, debug information, etc; or in a 'release' configuration which is optimised at compile time. When building these two (or more) configurations, the project builder needs to know which source files have changed since it was last invoked on the specific configuration.

Build configurations provide a mechanism to represent the different variants of the build. Build commands are run in the order defined on the Project, and the build resource delta is maintained between builds of the configuration. Build configurations allow the Incremental Project Builder to build only what needs to be built (based on the delta) when the builder is invoked.

Builders which don't care about build configurations work as before.

Build Configurations are simply a tuple: (project, build-config name). Names are domain/application specific strings. They should be human-readable as they may be displayed by the UI, for example: "Debug". Integrations should respect configurations created by other platform integrations. By default each project has one build configuration.

See IBuildConfiguration for more information.

API is provided:

to create build configuration(s) on a Project

set the active configuration on the Project

to build particular build configuration(s)

to create reference between build configurations

for a builder to discover which build configuration is currently being built

We distinguish between client API and builder API.

Client API

Creating and setting build configurations

Each project has one default build configuration with name equal to the empty string. Build configurations are set on the Project description:

IProject project = ...

IProjectDescription projDesc = project.getDescription();

// Get the build configurations for a project

IBuildConfiguration[] buildConfigs = project.getBuildConfigs();

...

// Creating a build configuration

ResourcesPlugin().getWorkspace().newBuildConfig(project.getName(), "myNewBuildConfig");

...

// Set new build configurations on a project

projDesc.setBuildConfigs(buildConfigs);

// Set the description

project.setDescription(projDesc, null);

As each project must have at least one build configuration, it's not possible to remove all configurations from a project. Doing so automatically re-adds the default build configuration, with name: "".

Active configuration

By default one configuration on the project is defined as 'active'. The active configuration is the one that is built when a build configurations isn't specified (e.g. when: IProject#build(kind, monitor) is called on the project). To set the active build configuration:

projDesc.setActiveBuildConfig(String buildConfigName);

Creating references between build configurations

Build Configurations can reference each other. Much like project level references, this can be used to ensure that prerequisite build configurations are built before referencing build configurations.

projDesc.setBuildConfigReferences(String configName, IBuildConfiguration[] references);

causes the named build configuration to reference the passed in build configurations.

Note: A referenced build configurations may have a null name. Such references map to the active build configuration of the referenced project when the project is built. Therefore a project level reference can be thought of as a build configuration reference to the active build configuration.

Building Project Build Configurations

New API exists on the workspace to allow building a build configuration and its references (if required):

ResourcesPlugin.getWorkspace().build(IBuildConfiguration[] buildConfigs, int kind, boolean buildReferences, IProgressMonitor monitor);

API also exists on IProject to build a specific build configuration:

project.build(IBuildConfiguration config, int kind, IProgressMonitor monitor);

Builder API

From the point of view of an Incremental Project Builder the platform now builds build configurations, rather than projects. In simple cases the build configuration will be the default configuration, and the project will only have one configuration.

Which build configuration is being built

A builder can discover which configuration is being built via:

protected IProject[] build(int kind, Map<String,String> args, IProgressMonitor monitor) throws CoreException {

IBuildConfiguration thisBuildConfig = getBuildConfig();

...

}

Note builders needn't necessarily care about build configurations. IncrementalProjectBuilder#getDelta(IProject) will continue to return the resource changes since the builder was last run for the current build configuration.

12.3. 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.

12.4. 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.

12.5. 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.FileModificationValidatorManager"/>

</extension>

The supplied class must implement FileModificationValidator, 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.

12.6. 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.

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.

12.7. Alternate file systems

Most of the time, resources in the workspace are stored in a local file system that is accessible via java.io.File. However, it is also possible to create resources that are stored in other kinds of file systems, such as network file systems or a database.

The next few sections cover how to contribute alternative file systems, and how to work with resources stored in different file systems.

12.7.1. Using the file system API

The org.eclipse.core.filesystem plug-in provides a generic API for interacting with an arbitrary file system. This API is similar to java.io.File, with a few key differences:

Plug-ins can install providers for different types of file systems.

All methods integrate support for reporting progress and responding to cancelation, making it easier to integrate into a graphical user interface.

There is support for some additional functionality not available in java.io.File, such as getting and setting file permissions.

There are more convenience methods, such as copy, move, and recursive deletion.

In the file system API, the path for any given file is represented as a hierarchical java.net.URI. The URI scheme represents the kind of file system, and the URI path component represents the location of the file within the file system tree. Thus any given hierarchical URI represents a potential file or directory in some arbitrary file system.

The API for working with files and file systems is found in the org.eclipse.core.filesystem) package. The central API type is IFileStore Each instance of IFileStore represents a single file in the file system. As with IResource, the existence of an IFileStore instance does not mean that such a file exists on disk. You can use an IFileStore instance to create, delete, copy, move, or open streams on files. For a given URI, you can get your hands on an IFileStore instance using the static method EFS.getStore(URI)

The IFileSystem interface can be used to find out things about the file system as a whole. Each IFileSystem instance represents a single URI scheme, such as "file:", "ftp:", etc. You can use this type to ask questions such as what file attributes are supported, or whether the file system is case-sensitive. You can also use this type to obtain an IFileStore for a given URI.

Most methods on IFileStore have a flag parameter that allows extra options to be supplied. The flag values can be found in the EFS class. For example, to open an output stream for appending to a file, use:

IFileStore store = ...//some file store

store.openOutputStream(EFS.APPEND, null);

If you want the default behaviour for a method, use EFS.NONE.

The IFileInfo interface represents the state of a file at a particular moment in time. In particular, you can find out if the file exists, whether it is a directory, what its attributes are, and so on. This information can be modified and then set back into the file. For example, here is a snippet that sets the read only attribute on a directory:

IFileStore store = ...//some file store

IFileInfo info = store.fetchInfo();

if (info.exists() && info.isDirectory()) {

info.setAttribute(EFS.ATTRIBUTE\_READ\_ONLY, true);

store.putInfo(info, EFS.SET\_ATTRIBUTES, null);

}

This style of API allows you to obtain and change file information with a single call to the file system. In the above example, there is only one file system call to fetch the info, and then you can perform as many operations as you want on the IFileInfo object without hitting the disk again.

The EFS class has static factory methods for obtaining IFileStore and IFileSystem instances, as well as various option constants and error codes.

12.7.2. Working with resources in other file systems

Most of the API in the IResource hierarchy works the same way regardless of what kind of file system the resources are stored in. However, there are some particular APIs that you need to avoid when working with resources in other file systems, because they are only designed to work with the local file system.

In particular, the IResource.getLocation method is specified to return the local file system path of the resource. If the resource is in some other file system, then this method is not applicable and will return null. It is better to use the method getLocationURI instead, which works regardless of what kind of file system the resource is stored in.

Similarly, the method IProjectDescription getLocation and setLocation methods should be avoided because they are only effective in the local file system. The URI-based location methods should be used instead.

Local caching

Say you are working with resources that are not in the local file system, but you really need a local file. For example, you may be using a library that has a dependency on java.io.File). In this case you can use the IFileStore.toLocalFile method to obtain a local copy of the file. Note that this will only cause one file or directory to be cached locally, rather than an entire directory tree. Here is an example of using local caching to open a zip file on an IFileStore:

IFileStore store = ...;//some file store

java.io.File file = store.toLocalFile(EFS.NONE, null);

if (file == null) {

//we are not a local file store, so we need to cache a local copy

file = store.toLocalFile(EFS.CACHE, null);

}

java.util.zip.ZipFile zip = new java.util.ZipFile(file);

Linking to other file systems

You can use linked resources to create projects that draw together resources from multiple file systems. Simply use the method IFile.createLink(URI, int, IProgressMonitor) or IFolder.createLink(URI, int, IProgressMonitor) to create a resource in an existing project whose contents are stored in another location in an arbitrary file system. You can even create links below other linked resources to create arbitrary resource trees drawing from many different file systems. Here is a simple example that creates a sibling linked file that shares the same file system location as the source:

IFile source = ...;//some source file

IFile link = source.getParent().getFile(new Path(source.getName() + ".link"));

link.createLink(source.getLocationURI(), IResource.NONE, null);

12.7.3. User interface support for alternative file systems

In environments where there is more than one file system there are several places in the user interface where the user can select a file system.

The UI support for additional file systems can be added via the org.eclipse.ui.ide.fileSystemSupport extension point. The class attribute of these schema must be a org.eclipse.ui.ide.fileSystem.FileSystemContributor which is used to supply validation and browsing of the other file systems.

Figure 1. New Project wizard with support for selecting alternative file systems

Above is the new Project wizard showing the extra support for multiple file systems. When more than one org.eclipse.ui.ide.fileSystem.FileSystemContributor is registered with Eclipse this selection will become available in the file, folder and simple project wizards.

12.7.4. File system providers

By default, the org.eclipse.core.filesystem plug-in only includes a file system implementation for the local file system. Plug-ins can contribute implementations for other file systems using the org.eclipse.core.filesystem.filesystems extension point. File system providers must provide subclasses of FileStore and FileSystem. Minimally, all abstract methods must be implemented, but other methods can be overridden to provide more efficient implementations of methods such as copy, move, delete, etc. Refer to the javadoc on these classes for more details on what the default implementations do, and for advice on how and when to override each method.

12.8. Dynamic path variables

Each linked resource in the workspace can be specified relative to a path variable.

Each IResource has a dedicated path variable manager, accessible from IResource.getPathVariableManager(). The path variable manager allows reading, creating and changing path variables, along with resolving URIs that may be relative to a path variable, as opposed to be defined as an absolute path.

Currently, each project has a separate list of path variables, so all resources under a given project have path variable managers that contain the same list of path variables, although that could change in the future. Still variables values can be different between two path variable managers of two different resources. For example, the variable "PARENT\_LOC" contains the location of the workspace parent of the current resource for which the path variable manager was created.

The default path variable list can be extended through the org.eclipse.core.resources.variablesResolvers extension point.

Because of limitations and backward compatibilities in the legacy IPathVariableManager, the path variable values cannot contain parent path lexical element (".."). Instead, the methods IPathVariableManager.convertToUserEditableFormat and IPathVariableManager.convertFromUserEditableFormat can be used to convert a path that contains the ".." element into an internal format that can be stored in the URI of a variable. The UI automatically converts user provided strings into the internal format and vice versa.

13. 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 contributions plug-in 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.

13.1. 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

menus

Commands can be placed in the main menu and toolbar. Command can be placed in part context menus, as well as view menus and view toolbars. The placement is controlled by the locationURI and the visibility is controlled by the visibleWhen core expression.

The extension point should be used to place commands in menus and toolbars so they can be executed by the user.

This replaces the old action extension points, actionSets, editorActions, viewActions, popupMenus, and actionSetPartAssociations.

13.2. Undoable operations

We've looked at many different ways to contribute commands to the workbench, but we haven't focused on the implementation of a command's execute(ExecutionEvent) method. The mechanics of the method depend on the specific command in question, but structuring the code as an undoable operation allows the command 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 execute(ExecutionEvent) method to create an IUndoableOperation, the operation can be made available for undo and redo. Converting an command or 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 example in org.eclipse.ui.examples.contributions.editor.DeltaInfoHandler. It builds a string and then opens a dialog.

public Object execute(ExecutionEvent event) throws ExecutionException {

// Build the string buffer "buf"

MessageDialog.openInformation(editor.getSite().getShell(),

ContributionMessages.DeltaInfoHandler\_shellTitle, buf

.toString());

return null;

}

Using operations, the execute method is responsible for creating an operation that does the work formerly done in the execute method, and requesting that an operations history execute the operation, so that it can be remembered for undo and redo.

public Object execute(ExecutionEvent event) throws ExecutionException {

IUndoableOperation operation = new DeltaInfoOperation(

editor.getSite().getShell());

operationHistory.execute(operation, null, null);

return null;

}

The operation encapsulates the old behavior from the run method, as well as the undo and redo for the operation.

class DeltaInfoOperation extends AbstractOperation {

Shell shell;

public DeltaInfoOperation(Shell shell) {

super("Delta Operation");

this.shell = shell;

}

public IStatus execute(IProgressMonitor monitor, IAdaptable info) {

// Build the string buffer "buf"

MessageDialog.openInformation(shell,

ContributionMessages.DeltaInfoHandler\_shellTitle, buf

.toString());

return Status.OK\_STATUS;

}

public IStatus undo(IProgressMonitor monitor, IAdaptable info) {

// Build the string buffer "buf"

MessageDialog.openInformation(shell,

ContributionMessages.DeltaInfoHandler\_shellTitle,

"Undoing delta calculation");

return Status.OK\_STATUS;

}

public IStatus redo(IProgressMonitor monitor, IAdaptable info) {

// Build the string buffer "buf"

// simply re-calculate the delta

MessageDialog.openInformation(shell,

ContributionMessages.DeltaInfoHandler\_shellTitle, buf

.toString());

return Status.OK\_STATUS;

}

}

For simple commands, 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 handler classes into a single handler class that is parameterized. The handler would simply execute the supplied operation when it is time to execute. This is largely an application design decision.

When a command 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 an execute 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 Object execute(ExecutionEvent event) throws ExecutionException {

IUndoableOperation operation = new DeltaInfoOperation(

editor.getSite().getShell());

...

operationHistory.execute(operation, null, null);

return 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 = editor.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 execute method could assign a workbench-wide context to its operations.

public Object execute(ExecutionEvent event) throws ExecutionException {

IUndoableOperation operation = new DeltaInfoOperation(

editor.getSite().getShell());

...

IWorkbench workbench = editor.getSite().getWorkbenchWindow().getWorkbench();

IOperationHistory operationHistory = workbench.getOperationSupport().getOperationHistory();

IUndoContext undoContext = workbench.getOperationSupport().getUndoContext();

operation.addContext(undoContext);

operationHistory.execute(operation, null, null);

return 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\_CONTEXT);

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 execute method 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 Object execute(ExecutionEvent event) throws ExecutionException {

IUndoableOperation operation = new DeltaInfoOperation(

editor.getSite().getShell());

...

operationHistory.execute(operation, null, infoAdapter);

return null;

}

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) {

// Build the string buffer "buf"

MessageDialog.openInformation(shell,

ContributionMessages.DeltaInfoHandler\_shellTitle, buf

.toString());

return Status.OK\_STATUS;

}

}

// do something else...

}

Undo and redo action handlers (Deprecated)

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 operation

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 recent.

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. In your plug-in, it may be appropriate to seek approval for the original execution of an operation, in addition to the undo and redo of the operation. If this is the case, your operation approver should also implement IOperationApprover2, which approves the execution of the operation. When asked to execute an operation, the platform operation history will seek approval from any operation approver that implements this interface.

13.2.1. 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(IUndoContext.class);

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.

Undoable Workspace Operations

Undoable operations are provided by the IDE plug-in that allow you to execute, undo, and redo common workspace manipulations. Operations are provided for creating, moving, copying, and deleting projects, folders, and files. The IDE plug-in uses these operations to provide undo support in the Resource Navigator. These classes can also be used by clients. They are available in the package org.eclipse.ui.ide.undo.

Using the workspace undoable operations is similar to the examples shown for using any IUndoableOperation. The primary difference is that some of the details, such as assigning the appropriate undo context, are handled internally by the operation. The following snippet shows how to use an undoable operation for creating a new file.

IFile newFileHandle = IDEWorkbenchPlugin.getPluginWorkspace().getRoot().

getFile(myNewFilePath);

CreateFileOperation op = new CreateFileOperation(newFileHandle, null,

initialContentStream, "Create New File");

The operation is configured with a file handle for the file to be created, an input stream with the initial contents, and a label describing the operation. Internally, the operation will assign the workspace undo context and handle other details such as keeping track of which resources the operation manipulates. To execute the operation, the caller uses the standard IOperationHistory protocol.

try {

PlatformUI.getWorkbench().getOperationSupport().getOperationHistory()

.execute(op, myProgressMonitor,

WorkspaceUndoUtil.getUIInfoAdapter(getShell()));

} catch (final ExecutionException e) {

// handle exceptions

}

Note that the caller is responsible for providing a progress monitor if desired, and an IAdaptable that can provide the shell that should be used for any prompting. The utility class WorkspaceUndoUtil provides a utility method for creating the necessary adapter from a shell.

Callers are also responsible for determining whether to execute the operation in the UI thread, the background, or a workbench job. If the operation should be executed in the background, then the caller should create the appropriate progress monitor and run the snippet shown above in the appropriate runnable.

The undo and redo of the operation will be performed by the undo and redo action handlers when the user initiates undo or redo. The standard workbench action handlers use IAdvancedUndoableOperation2 protocol to determine how to perform the undo and redo of operations. All of the workspace operations implement protocol that instructs clients to run the undo or redo of the operation in the background if possible.

13.3. 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 Project Explorer 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 Project Explorer. If the user shows the Bookmarks view, it will appear in the same folder with the Project Explorer, 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 Project Explorer.

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 by using the showInPart element of the Perspective Extensions extension point. Recall that a "Show In" navigation may open a view that is not already present in the perspective.

13.3.1. 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.

13.3.2. 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.

13.4. 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 package explorer. 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.

13.5. 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.

See the org.eclipse.ui.bindings section for simple binding scenarios and the Basic workbench extension points using commands section for using the new command framework.

13.5.1. 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 commandId="org.eclipse.ui.file.save"

contextId="org.eclipse.ui.contexts.window"

schemeId="org.eclipse.ui.examples.contributions.scheme"

sequence="CTRL+S">

</key>

...

There is our friend org.eclipse.ui.file.save. In Eclipse, the workbench registers the SaveAction as a handler for this command id in every Workbench Window.

The sequence attribute for a key binding defines the key combination that is used to invoke a command. When the workbench SaveAction is active, the key combination CTRL+S will invoke it, since the workbench uses the same command id for its SaveAction.

13.5.2. 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 > Keys preference page.

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

13.5.3. 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:

<extension

point="org.eclipse.ui.bindings">

<key

sequence="Ctrl+S"

commandId="org.eclipse.ui.file.save"

contextId="org.eclipse.ui.textEditorScope"

schemeId="org.eclipse.ui.defaultAcceleratorConfiguration">

</key>

</extension>

...

(See Contexts) for a more detailed discussion of contexts and how they are defined.

13.6. 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>

13.7. 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.

13.8. 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.

13.9. 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.

13.10. 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 functionality 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 functionality while the user is performing a specific task. They influence what commands are available to the user at any given moment.

13.10.1. Activities

An activity is a logical grouping of functionality 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. Activities can be used to implement progressive disclosure of UI elements; when used for this purpose, they are called capabilities in the UI. The second use for activities is to filter available UI elements based on other criteria such as the current user's access permissions as defined bythe application.

In the following text, we will be using the following terms to distinguish between the two uses of activities:

Activities that are used to declutter the user desktop from unnecessary UI elements will be called conventional activities.

Activities that filter out UI elements which are not supposed to be shown to the user (e.g. based on their access rights) will be called expression-based activities.

Conventional activities

Conventional activities are exposed to the user under the name capabilities, although not in a way that is apparent to a new user. When an activity is enabled in the platform, the UI contributions associated with that activity are shown. When a activity is disabled in the platform, its UI contributions are not shown. Users can enable and disable these conventional activities as needed using the General > Capabilities preference page.

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 functionality 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.

There are certain places in the UI where the user can ask to see all contributions - even the ones filtered by conventional activities, for example in the New... wizard. UI elements that are filtered by conventional activities can also still be used programmatically using the Eclipse API.

Expression-based activities

Expression-based activities differ from conventional activities in that:

they are solely controlled by expressions (see "org.eclipse.core.expressions.definitions"). All other declarations related to such an activity, such as "categories", "default enabled activities", and "requirement bindings" will be ignored.

they move an UI contribution completely out of reach for users and programmers. The UI contributions cannot be accessed programmatically using API calls, and they do not show up when the user asks to see all contributions to, for example, the New... wizard.

Conventional Activities vs. perspectives

We have 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 functions are 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 activities

Activities are defined using the org.eclipse.ui.activities extension point.

Let's look at first at a simplified version of how the Eclipse SDK plug-in defines two conventional 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. This name and description can be shown to the user whenever the they are enabling and disabling conventional 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, for conventional activities, it is possible to declare 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 conventional 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 conventional activities together. This category is shown to the user when the user manually configures conventional activities. Note that expression-based activities can be also added to categories, but they are ignored when the user changes states of these categories.

Binding activities to UI contributions

Activities can be associated with UI contributions by referring to them by id, or by 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>

It is also possible to refer to a single UI contribution using its id without having to use regular expression syntax if the attribute isEqualityPattern is set to true.

The following XML shows the previous example with the isEqualityPattern set to true.

<activityPatternBinding

activityId="org.eclipse.javaDevelopment"

pattern="org.eclipse.jdt.core/javanature"

isEqualityPattern="true">

</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 with an activity. The following contributions are affected by activities:

Views and editors

Perspectives

Preference and property pages

Menus and toolbars

New wizard

Common Navigator Action Providers

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 all defined activities (to some extent also expression-based activities) and changing the enabled state (only for conventional activities). Most plug-ins need not be concerned with this API, but it is useful when implementing functions that allow the user to work with activities, or for implementing the trigger points that enable a particular conventional 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 conventional 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.

Note that the API methods will silently ignore attempts to enable expression-based activities, or similar requests that do not apply to expression-based activities.

Using expression-based activities

To filter a UI element using an expression-based activity, create an activity like the following:

<activity

id="forbiddenViewActivityId" name="Forbidden View Activity">

<enabledWhen>

<with variable="rightsVariable">

<iterate ifEmpty="false" operator="or">

<equals value="grantShowForbidden" />

</iterate>

</with>

</enabledWhen>

</activity>

Then, bind this activity to a UI element, for example a view:

<activityPatternBinding

activityId="forbiddenViewActivityId"

<!-- Switches the interpretation of the pattern as regular expression off -->

isEqualityPattern="true"

pattern="DemoRCP/demorcp.views.ForbiddenView">

</activityPatternBinding>

The following code snippets show how to control the variable "rightsVariable" that appears in the activity's "enabledWhen" expression. New variables can be added through the org.eclipse.ui.services extension point as subclasses of AbstractSourceProvider.

import java.util.HashMap;

...

import org.eclipse.ui.AbstractSourceProvider;

import org.eclipse.ui.PlatformUI;

import org.eclipse.ui.handlers.IHandlerService;

import org.eclipse.ui.services.IEvaluationService;

public class RightsSourceProvider extends AbstractSourceProvider {

public final static String RIGHT\_FORBIDDEN = "grantShowForbidden";

public final static String RIGHTS\_VARIABLE = "rightsVariable";

private final static String[] PROVIDED\_SOURCE\_NAMES = new String[] { RIGHTS\_VARIABLE };

private final static Map<String, List<String>> stateMap = new HashMap<String, List<String>>();

public Map getCurrentState() {

/\* "YourRightsHandler" is here just an example for a static class

\* which returns the list of rights as a list of strings. \*/

stateMap.put(RIGHTS\_VARIABLE, YourRightsHandler.getUserRights());

return stateMap;

}

public String[] getProvidedSourceNames() {

return PROVIDED\_SOURCE\_NAMES;

}

/\* This triggers an update of the rights variable state, and will update also all

\* listeners to the evaluation service. So that every menu point, which is also

\* expression controlled, gets updated too. \*/

public void updateRights() {

fireSourceChanged(0, getCurrentState());

}

// ...

}

13.10.2. 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 functionality 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:

<extension

point="org.eclipse.ui.bindings">

<key

sequence="M1+X"

contextId="org.eclipse.ui.contexts.dialogAndWindow"

commandId="org.eclipse.ui.edit.cut"

schemeId="org.eclipse.ui.defaultAcceleratorConfiguration"/>

</extension>

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 IContextService. Plug-ins can obtain the global context support instance from the workbench.

IContextService contextService = (IContextService)PlatformUI.getWorkbench()

.getService(IContextService.class);

Services like IContextService, IHandlerService, and IBindingService can be retrieved using an IServiceLocator. IWorkbench, IWorkbenchWindow, and IWorkbenchSite are all IServiceLocator.

IContextService defines protocol for getting all defined or enabled context ids, and for getting the associated Context 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.core.commands.contexts for more information.

Contexts can be enabled programmatically:

IContextActivation activation = contextService.activateContext("org.eclipse.ui.textEditorScope");

The IContextActivation is a token that can be used to deactivate an active context. You should ensure that you only activate defined Contexts.

If you are activating a more specific Context within your part (either View or Editor) you can use the part site service locator to active your Context. The part's IContextService will take care of activating and deactivating the Context as your part is activated or deactivated. It will also dispose the Context when the part is disposed.

13.11. 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.class);

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 dialog

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.

13.12. 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.

13.12.1. 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.

13.12.2. 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());

...

}

13.12.3. Support for displaying markers

The org.eclipse.ui.ide.markerSupport extension point provides facilities for controlling how markers are displayed and organized in the UI. This extension point handles filters and grouping in views such as the Problems view as well as definition of custom marker views.

Marker Content Generators

Custom marker views can be created by specifying the view using a markerContentsGenerator. A markerContentsGenerator specifies the types used, the configurations available and the grouping to be shown in the view. The problems view, tasks view and bookmarks view are defined using markerContentsGenerator as of version 3.4. A reusable superclass org.eclipse.ui.views.markers.AbstractMarkersView has been supplied for use by custom markerContentsGenerators.

Commands

The menus defined in the markersViews now all use the org.eclipse.ui.menus extension point so they can be extended by clients at any desired point. The pop-up menus in the supplied markers views are available to all subclasses of org.eclipse.ui.views.markers.AbstractMarkersView but pull-down entries must be added for the new views explicitly. It is recommended that all new handlers for commands to be added to the markersView subclass org.eclipse.ui.views.markers.MarkerViewHandler so as to get access to the org.eclipse.core.resources.IMarkers currently selected.

Configurations

The Problems view supports multiple configurations at once. When multiple configurations are enabled there are two possible modes of matching: match any (any marker that matches one or more selected filters will be displayed) or match all (any marker that matches all selected filters will be displayed). New configurations can be added using the markerFieldConfiguration element or manually by the user. Below is the configuration section of the Problems view showing user configurations and configurations added via the extension point).

Groups

Markers can be grouped in two ways. The first way is to group marker types into groups using the markerTypeCategory element. This is shown when the user selects Group By > Type. Below is an example that shows grouping by 'Java Problem Type':

When marker types are grouped together they will be shown in those groups in the configurations dialog.

You can also create your own grouping using the markerGrouping element, which will show up as an extra entry in the Group By menu of the markers view it is defined in. Top level markerGroupings are shown in the Problems View (this is for compatibility with previous Eclipse versions). The IDE provides a grouping based on severity (the default). The Java development tools provide a Java Type grouping.

markerGroupings are resolved by checking attributes on the IMarker. markerAttributeMappings are used to specify these mappings. Below is the Group By menu with the extra markerGroupings that have been added. The Problems view is currently grouped by severity.

13.12.4. 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.

13.12.4.1. 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 in the help view.

13.12.4.2. 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.CorrectionMarkerResolutionGenerator">

</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.

The Problems view supports resolving multiple problems at once. The user can add other matching problems to the list of problems to fix, and then apply all fixes in one go. To support multiple resolutions, your implementation of IMarkerResolution must be a subclass of WorkbenchMarkerResolution.

13.12.5. 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.

13.12.6. 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 > Workspace 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();

13.13. Status handling

Status handling is a facility that allows to introduce a custom way of showing problems in the Eclipse based applications to users. The facility can be configured at both the application and the product level.

Status handlers

The handlers are responsible for presenting problems by logging or showing appropriate feedback to the user (generally dialogs).

All status handlers extend org.eclipse.ui.statushandlers.AbstractStatusHandler which requires each handler to implement handle(StatusAdapter status, int style). This method handles statuses based on a handling style. The style indicates how status handler should handle a status. See Acceptable styles.

There are two ways for contributing handlers to the Workbench:

using the org.eclipse.ui.statusHandlers extension point, see User assistance and status handling

using the workbench advisor and its method WorkbenchAdvisor#getWorkbenchErrorHandler()

NOTE! if there exists a handler associated with the product, it is used instead of the one defined in the advisor.

A status handler has an id and a set of parameters. The handler can use them during the handling process. If the handler is added as an extension, both the id and parameter set are set during initialization of the handler with the use of elements and attributes of the statusHandler element.

Logging with the use of the default logging mechanism

In order to log extra information with the use of the default logging mechanism, it is necessary to perform some additional steps. The status manager provides you with an API that allows you to hook into the mechanism.

StatusManager#addLoggedStatus(IStatus status)

And below is the example of addLoggedStatus(IStatus status) proper usage.

public void handle(final StatusAdapter statusAdapter, int style) {

...

if ((style & StatusManager.LOG) == StatusManager.LOG) {

StatusManager.getManager().addLoggedStatus(statusAdapter.getStatus());

WorkbenchPlugin.getDefault().getLog().log(statusAdapter.getStatus());

}

}

The default status handler

The platform supplies its own status handler implementation org.eclipse.ui.statushandlers.WorkbenchErrorHandler. It respects all acceptable styles. It uses the default logging mechanism and a dialog based on the JFace org.eclipse.jface.dialogs.ErrorDialog.

Registering the ErrorSupportProvider with your status handler

There is a simple way to contribute a support area in the JFace ErrorDialog. It is the ErrorSupportProvider which can be set in the JFace policy. Because the default workbench handler uses a subclass of the ErrorDialog for its dialog, setting the provider on the JFace policy will affect this dialog as well.

When a handler like the one below is created

public class CustomStatusHandler extends WorkbenchErrorHandler {

public CompanyStatusHandler() {

ErrorSupportProvider provider = createSupportProvider();

Policy.setErrorSupportProvider(provider);

}

private ErrorSupportProvider createSupportProvider() {

...

}

}

and is contributed to the Workbench in one of two available ways, all dialogs based on the ErrorDialog including the one shown by the default handler, will have an extra support area.

Status manager

The status manager is the entry point for all statuses that are to be displayed in the user interface. Handlers are not intended to be used directly. They should be referenced via the StatusManager which selects the appropriate handler to apply. The following methods are the API entry points to the StatusManager

StatusManager#handle(IStatus)

StatusManager#handle(IStatus, int)

StatusManager#handle(StatusAdapter)

StatusManager#handle(StatusAdapter, int)

The StatusManager singleton is accessed using

StatusManager.getManager()

The int parameters are used for supplying styles for handling. See Acceptable styles.

NOTE! the style is only a suggestion and may not be honored by the current handler. For instance a handler may choose not to show the user anything when the SHOW flag is sent. See Status handlers for more details.

If a handler is associated with a product, it is used instead of the one that was defined in advisor. However because product handler is lazy initialized, an error can occur during the first attempt to access it. If any creation error occurs in the product handler, the workbench handler will process this error.

Acceptable styles

Below is a list of StatusManager styles which can be combined with logical OR:

NONE - a style indicating that the status should not be acted on. This is used by objects such as log listeners that do not want to report a status twice

LOG - a style indicating that the status should be logged only

SHOW - a style indicating that handlers should show a problem to a user without blocking the calling method while awaiting user response. This is generally done using a non modal dialog

BLOCK - a style indicating that the handling should block the calling method until the user has responded. This is generally done using a modal window such as a dialog

StatusAdapter

The StatusAdapter wraps an instance of IStatus subclass and can hold additional information either by using properties or by adding a new adapter. It is used during status handling process.

Each handler should specify what properties or types it accepts, being aware of the fact that developers can pass status adapters with such extra information.

13.13.1. Defining a status handler

There are two ways for contributing handlers to the Workbench:

using the org.eclipse.ui.statusHandlers extension point, see User assistance and status handling

using a custom workbench advisor and overriding its getWorkbenchErrorHandler() method

Contributing a status handler using the custom advisor

First, a custom workbench advisor for your application has to be created. In this new custom advisor override getWorkbenchErrorHandler(). This will now return the default error handler.

public class CustomWorkbenchAdvisor extends WorkbenchAdvisor {

public AbstractStatusHandler getWorkbenchErrorHandler() {

...

return customStatusHandler;

}

}

In the custom application use your new advisor when creating the Workbench.

public class CustomApplication implements IApplication{

public Object start(IApplicationContext appContext) throws Exception {

...

Display display = createDisplay();

PlatformUI.createAndRunWorkbench(display, new CustomWorkbenchAdvisor());

...

}

protected Display createDisplay() {

return PlatformUI.createDisplay();

}

}

For details how to run a custom application see the org.eclipse.core.runtime.applications extension point documentation.

When the status handling facility is accessed, the facility will get instance of the handler that is provided by the custom advisor. This handler will be used for handling of status or error occurred in the application.

13.14. Services

The workbench defines a number of services that can be retrieved from the org.eclipse.ui.services.IServiceLocator. Services provide a way for components to retrieve information about the workbench without always having to go to the global singleton: PlatformUI.getWorkbench(). For example:

IHandlerService handlerService = (IHandlerService) getSite()

.getService(IHandlerService.class);

Some services provide 2 optional abilities, which are usually specified in the service interface javadoc:

Services that support listeners can clean up the listeners automatically when the component's org.eclipse.ui.services.IServiceLocator (used to get the service) is disposed.

Services can provide more localized services depending on which component they are requested from.

See Associating a handler programmically with a command... for an example of the activation localization provided by the org.eclipse.ui.handlers.IHandlerService.

Services Provided by the Workbench

The services provided to by the workbench now includes some of the most common services that an org.eclipse.ui.IWorkbenchPart would require. Support for service related optional abilities are specified in the service javadoc. The availability of the service is at and below the level specified.

Service

Description

Availability

IBindingService

Provides services related to the binding architecture (e.g., keyboard shortcuts) within the workbench.

Globally

ICommandService

Provides services related to the command architecture within the workbench.

Globally

ICommandImageService

Provides a look-up facility for images associated with commands.

Globally

IContextService

Provides services related to contexts in the Eclipse workbench, like context activation and definitions.

Globally

IContributionService

The IContributionService is a service provided at the workbench level that provides mechanisms that clients may use to work with user interface contributions. Currently, this is limited to providing sorters for particular contribution types but this may be expanded on in the future. An instance of this service is present for the entire lifetime of the workbench.

Globally

IEvaluationService

Evaluate a core expression against the workbench application context and report updates using a Boolean property. Also provides the main source for the workbench application context.

Globally

IFocusService

Tracks focusGained and focusLost events for a Control registered with this service, and provides the control and its registered ID as variables to the application evaluation context for evaluation by the various services.

Globally

IHandlerService

Provides services related to activating and deactivating handlers within the workbench.

Globally

IMenuService

Provides services related to the menu architecture within the workbench. It can be used to populate MenuManagers and ToolBarManagers by components.

Globally

IPageService

A page service tracks the page and perspective lifecycle events within a workbench window.

Workbench Window

IPartService

A part service tracks the creation and activation of parts within a workbench window.

Workbench Window

IProgressService

The progress service is the primary interface to the workbench progress support.

Globally

IWorkbenchSiteProgressService

The part progress service is an IProgressService that adds API for jobs that change the state in a IWorkbenchPartSite while they are being run.

Part Site

ISaveablesLifecycleListener

Parts that implement org.eclipse.ui.ISaveablesSource should notify their ISaveablesLifecycleListener, available as a service from their site, about org.eclipse.ui.Saveable objects that have been added to or removed from the part. Implementations of ISaveablesSource that are not parts should retrieve this service from the org.eclipse.ui.IWorkbench object.

Globally

ISelectionService

A selection service tracks the selection within an a workbench window.

Workbench Window

Contributing a Service

Plug-in developers providing a framework in eclipse may also want to provide a service. This can be done using the org.eclipse.ui.services extension point. Define a service factory based on org.eclipse.ui.services.AbstractServiceFactory and then specify what services the factory can return. A factory can create more than one type of service, but multiple factories cannot contribute to the same service type.

<serviceFactory

factoryClass="org.eclipse.ui.examples.contributions.model.PersonServiceFactory">

<service

serviceClass="org.eclipse.ui.examples.contributions.model.IPersonService"/>

</serviceFactory>

The create method of the org.eclipse.ui.services.AbstractServiceFactory passes in 3 parameters:

serviceInterface - the interface class that needs to be created

parentLocator - an org.eclipse.ui.services.IServiceLocator that can be used to get the parent service for serviceInterface

locator - the org.eclipse.ui.services.IServiceLocator that requested this service be created. It can be used to retrieve dependent services.

Workbench services are organized so there is a global service providing the needed functionality, and a chain of children services. The children services provide the listener cleanup and any data needed to localize the service request.

A service factory should return the global service when the parentLocator returns null when asked for the serviceInterface. The global service may be the implementation, or a proxy to a plugin provided manager or OSGi service.

Object parentService = parentLocator.getService(IPersonService.class);

if (parentService == null) {

return new PersonService(locator);

}

return new PersonServiceSlave(locator, (IPersonService) parentService);

// or ...

Object parentService = parentLocator.getService(IPersonService.class);

if (parentService == null) {

return Activator.getDefault().getPersonManager();

}

return new PersonServiceSlave(locator, (IPersonService) parentService);

//

// or as a front to an OSGi service.

Object parentService = parentLocator.getService(IPersonService.class);

if (parentService == null) {

PersonServiceProxy proxy = new PersonServiceProxy(locator,

Activator.getDefault().getBundle().getBundleContext(),

0);

proxy.findService();

return proxy;

}

PersonServiceProxy proxy = new PersonServiceProxy(locator,

Activator.getDefault().getBundle().getBundleContext(),

((PersonServiceProxy)parentService).getLevel()+1);

proxy.findService();

return proxy;

In the OSGi service snippet above, instead of having each child IPersonService know its parent, each IPersonService talks directly to the OSGi service and uses the parent to calculate its "level" which allows it to localize any functionality it provides.

14. 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.

14.1. 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.

Three generic editor inputs are provided in the platform. IFileEditorInput represents an input that is a file in the file system, and IURIEditorInput represents an input based on a URI. 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.

14.2. 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.

As an an alternative implementation of a text editor, you can also decide to rely on the Extensible and Generic Editor, which is a faster and simpler approach tp implement textual language support, but with less control and some limitations compared to defining a full new editor.

14.3. 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.getDefault().getJavaPartitionScanner(), JavaPartitionScanner.JAVA\_PARTITION\_TYPES);

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().getJavaPartitionScanner(), JavaPartitionScanner.JAVA\_PARTITION\_TYPES);

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.

14.4. 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.

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 AbstractDecoratedTextEditor.

protected ISourceViewer createSourceViewer(Composite parent, IVerticalRuler ruler, int styles) {

...

ISourceViewer sourceViewer= new SourceViewer(parent, ruler, fOverviewRuler, isOverviewRulerVisible(), styles);

fSourceViewerDecorationSupport= new SourceViewerDecorationSupport(sourceViewer, fOverviewRuler, fAnnotationAccess, sharedColors);

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 AbstractDecoratedTextEditor for the decoration support.

protected void configureSourceViewerDecorationSupport() {

Iterator e= fAnnotationPreferences.getAnnotationPreferences().iterator();

while (e.hasNext())

fSourceViewerDecorationSupport.setAnnotationPreference((AnnotationPreference) e.next());

fSourceViewerDecorationSupport.setAnnotationPainterPreferenceKeys(DefaultMarkerAnnotationAccess.UNKNOWN, UNKNOWN\_INDICATION\_COLOR, UNKNOWN\_INDICATION, UNKNOWN\_INDICATION\_IN\_OVERVIEW\_RULER, 0);

fSourceViewerDecorationSupport.setCursorLinePainterPreferenceKeys(CURRENT\_LINE, CURRENT\_LINE\_COLOR);

fSourceViewerDecorationSupport.setMarginPainterPreferenceKeys(PRINT\_MARGIN, PRINT\_MARGIN\_COLOR, PRINT\_MARGIN\_COLUMN);

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.

14.5. 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() : new DefaultAutoIndentStrategy());

}

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.

14.6. 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(), hoverRegion.getLength());

} 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;

}

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.

14.7. 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.getJavaCodeScanner());

reconciler.setDamager(dr, IDocument.DEFAULT\_CONTENT\_TYPE);

reconciler.setRepairer(dr, IDocument.DEFAULT\_CONTENT\_TYPE);

dr= new DefaultDamagerRepairer(new SingleTokenScanner(new TextAttribute(provider.getColor(JavaColorProvider.JAVADOC\_DEFAULT))));

reconciler.setDamager(dr, JavaPartitionScanner.JAVA\_DOC);

reconciler.setRepairer(dr, JavaPartitionScanner.JAVA\_DOC);

dr= new DefaultDamagerRepairer(new SingleTokenScanner(new TextAttribute(provider.getColor(JavaColorProvider.MULTI\_LINE\_COMMENT))));

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.

14.8. 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\_CONTENT\_TYPE);

assistant.setContentAssistProcessor(new JavaDocCompletionProcessor(), JavaPartitionScanner.JAVA\_DOC);

...

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.format(JavaEditorMessages.getString("CompletionProcessor.Proposal.ContextInfo.pattern"), new Object[] { fgProposals[i] })); //$NON-NLS-1$

result[i]= new CompletionProposal(fgProposals[i], documentOffset, 0, fgProposals[i].length(), null, fgProposals[i], info, MessageFormat.format(JavaEditorMessages.getString("CompletionProcessor.Proposal.hoverinfo.pattern"), new Object[] { fgProposals[i]})); //$NON-NLS-1$

}

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.ContextInfo.display.pattern"), new Object[] { new Integer(i), new Integer(documentOffset) }),

MessageFormat.format(JavaEditorMessages.getString("CompletionProcessor.ContextInfo.value.pattern"), new Object[] { new Integer(i), new Integer(documentOffset - 5), new Integer(documentOffset + 5)}));

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().getColor(new RGB(150, 150, 0)));

return assistant;

}

Show the content assist action in the menu

If the content assistant is correctly supplied as mentioned above then the key binding (Ctrl+Space per default) will work out of the box. To make the action appear in the main menu the editor action bar contributor has to be extended:

...

private RetargetTextEditorAction fContentAssist;

public MyEditorActionContributor() {

fContentAssist= new RetargetTextEditorAction();

String commandId= ITextEditorActionDefinitionIds.CONTENT\_ASSIST\_PROPOSALS;

fContentAssist.setActionDefinitionId(commandId);

}

public void contributeToMenu(IMenuManager menu) {

IMenuManager editMenu= menu.findMenuUsingPath(M\_EDIT);

editMenu.appendToGroup(MB\_ADDITIONS, fContentAssist);

}

public void setActiveEditor(IEditorPart part) {

IAction editorAction= getAction(part, ITextEditorActionConstants.CONTENT\_ASSIST);

fContentAssist.setAction(editorAction);

}

...

14.9. 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();

fContentAssistProposal= new RetargetTextEditorAction(JavaEditorMessages.getResourceBundle(), "ContentAssistProposal."); //$NON-NLS-1$

...

fContentAssistTip= new RetargetTextEditorAction(JavaEditorMessages.getResourceBundle(), "ContentAssistTip."); //$NON-NLS-1$

...

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(), "ContentAssistProposal.", this, ISourceViewer.CONTENTASSIST\_PROPOSALS); //$NON-NLS-1$

a.setActionDefinitionId(ITextEditorActionDefinitionIds.CONTENT\_ASSIST\_PROPOSALS);

setAction("ContentAssistProposal", a);

a= new TextOperationAction(JavaEditorMessages.getResourceBundle(), "ContentAssistTip.", this, ISourceViewer.CONTENTASSIST\_CONTEXT\_INFORMATION); //$NON-NLS-1$

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\_RIGHT);

addAction(menu, ITextEditorActionConstants.GROUP\_EDIT, ITextEditorActionConstants.SHIFT\_LEFT);

}

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.

14.10. 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.)

14.11. 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) {

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();

}

}

}

15. 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.

15.1. 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.

15.2. 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.

16. 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.

16.1. Merging multiple files

In some cases, the ability to merge files without the assist of a user is desirable. The extension point org.eclipse.team.core.storageMergers allows you to contribute a class that merges three different files (or storage units) into a single output stream. Storage mergers can be associated with file extensions or bound to a particular content type. The Platform defines a storage merger for merging three files of plain text:

<extension

point="org.eclipse.team.core.storageMergers">

<storageMerger

extensions="txt"

class="org.eclipse.team.internal.ui.mapping.TextStorageMerger"

id="org.eclipse.team.ui.textStorageMerger">

</storageMerger>

<contentTypeBinding

contentTypeId="org.eclipse.core.runtime.text"

streamMergerId="org.eclipse.team.ui.textStorageMerger">

</contentTypeBinding>

</extension>

The storage merger itself is described in the storageMerger 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 storage merger should be used.

You may also use the contentTypeBinding element to associate a content type with a storage merger.

Storage mergers must implement IStorageMerger. This simple interface merges the contents from three different storage units into a single output stream. The not-so-simple implementation depends upon your plug-in and its content types.

16.2. 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.javaSource"

contentMergeViewerId="org.eclipse.jdt.internal.ui.compare.JavaContentViewerCreator">

</contentTypeBinding>

</extension>

<extension

point="org.eclipse.compare.contentMergeViewers">

<viewer

extensions="properties,properties2"

class="org.eclipse.jdt.internal.ui.compare.PropertiesFileMergeViewerCreator"

id="org.eclipse.jdt.internal.ui.compare.PropertiesFileMergeViewerCreator">

</viewer>

<contentTypeBinding

contentTypeId="org.eclipse.jdt.core.javaProperties"

contentMergeViewerId="org.eclipse.jdt.internal.ui.compare.PropertiesFileMergeViewerCreator">

</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 adapt to an IDocumentRange instead of an IDocument.

The TextMergeViewer also supports the use of a shared document (i.e. document's that are shared by multiple open editors) when the input adapts to an ISharedDocumentAdapter. Subclasses of TextMergeViewer that provide syntax highlighting must implement both the getDocumentPartitioner() and getDocumentPartitioning() methods to support shared documents.

Range differencing

RangeDifferencer finds the longest sequences of matching and non-matching comparable entities in text content. 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.

16.3. 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.

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, DocumentRangeNodes 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 DiffNodes 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:

Map the input data into a tree of objects implementing both the IStructureComparator and IStreamContentAccessor interfaces

Perform the compare operation by means of the Differencer

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 IStructureComparators 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.

For text based inputs, clients should subclass the StructureCreator class. This will enable the use of a shared document between multiple editors open on the same file. Subclasses of StructureCreator that provide syntax highlighting must implement both the getDocumentPartitioner() and getDocumentPartitioning90 methods to support shared documents.

16.4. 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.

Comparing a single file in an editor

The CompareEditorInput supports the comparison of an arbitrary file/folder structure which can be displayed in an editor, dialog or view. However, the specific case of comparing a single file in an editor warrants special handling because, in that case, the desired behavior should be close to that of editing the file. So, for this purpose a subclass of CompareEditorInput has been provided. The SaveableCompareEditorInput class provides this capability and has the following features:

provides a static method createFileElement for creating a type element that represents a file.

supports the use of shared documents (i.e. file buffers).

supports the workbench Saveables API.

auto-closes when changes are resolved.

initializes asynchronously.

Working with patches

The ApplyPatchOperation provides the ability to launch the Apply Patch wizard programmatically. The pages shown by the wizard are determined using the inputs to the operation. The class also contains the static method parsePatch which can be used by clients to deal with patches in a custom manner.

17. 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 workflows 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 FileModificationValidator 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.

Support for Logical Model Integration [new in 3.2]

The logical model integration support defines API that allows logical models to participate in team operations. In model based views, model providers can use this support in order to have team decorations and operations appear on their model elements. In resource based views (e.g. Resource Navigator), model providers can use this support to validate that operations performed on resources will not corrupt the model and to ensure that all the resources that constitute a model element are included in team operations. For team merge operations, the API allows model providers to use model semantics during merge operations and to participate in the preview of merge operation. The description of the support for logical model integration is divided into a Repository Roadmap for Logical Model Integration and a Model Roadmap for Logical Model Integration.

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.

17.1. 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 with 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\*/, null);

} 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 TeamException {

// 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:

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>

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 functionality 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, FileModificationValidator 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.

getFileModificationValidator2 - answers an FileModificationValidator 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. CVS uses this hook when using the watch/edit mode, but by default CVS uses an optimistic versioning scheme that does not require 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

17.2. 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:

Those which call out to an external program that then operates directly against the file system.

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.ccvs.ui.CVSProjectSetSerializer"/>

</extension>

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

17.3. Synchronization Support

Eclipse includes 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 for a resource

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.

17.3.1. 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.ui.examples.filesystem plug-in. You can check the project out from the Git 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 3.3.2 version tag (most likely R3\_3\_2) or a date tag of June 10, 2007.

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 DateFormat.getDateTimeInstance().format(new Date(state.getModificationTime()));

}

public byte[] asBytes() {

return null;

}

public IFileState getFileState() {

return state;

}

}

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 its 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, IResourceVariantComparator comparator) {

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="icons/full/wizards/synced.gif"

class="org.eclipse.team.examples.localhistory.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) { //$NON-NLS-1$

protected SynchronizeModelOperation getSubscriberOperation(ISynchronizePageConfiguration configuration, IDiffElement[] elements) {

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.

private 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.examples.localhistory.LocalHistorySynchronizeWizard"

icon="icons/full/wizards/synced.gif"

description="Synchronize resources with their previous contents in the local history"

name="Synchronize with Latest From Local History"

id="ExampleSynchronizeSupport.wizard1"/>

</extension>

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

LocalHistoryParticipant participant = new LocalHistoryParticipant();

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.

17.3.2. 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.ui.examples.filesystem plug-in. You should check the project out from the Git 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.ui.examples.filesystem plug-in. The code in the following description is kept to a minimum since it is available from the Eclipse Git 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 Git 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.

17.4. Team Support for Logical Model Integration

There are two viewpoints of interest when describing the Team support for logical model integration:

Repository Provider: A Repository Provider is the connection between the local workspace and a remote repository. Details of the logical model support from the standpoint of a Repository Provider can be found in the Repository Roadmap for Logical Model Integration.

Model Provider: A Model Provider is the tooling that allows the user to work with the model elements that are stored in the resources in the local workspace. Details of how Model Providers can leverage this support is found in Model Roadmap for Logical Model Integration.

The following points summarize the features covered by the Team logical model support.

Maintaining Workspace Consistency: Operations performed directly on resources may have undesirable side effects on model elements that are persisted in, or are otherwise associated with, those resources. Clients can use the ResourceChangeValidator to validate that changes to resources will not have undesirable side effects on models while models can implement the ModelProvider#validateChange method to validate a resource change.

Team Operations and Decorations: It has always been possible to have team operations and decorations appear on model elements that have a one-to-one relationship by adapting the model element to the corresponding IResource. It is now possible to have operation and decorations appear on model elements that have more complex relationships to resources by adapting a model element to a ResourceMapping.

Semantic Merges of Model Elements: Model Providers can participate in headless merges by associating an IStorageMerger with a particular file type, if there is a one-to-one correspondence between model elements and resources. For more complex relationships, Model Providers can adapt their ModelProvider to an IResourceMappingMerger to have access to the full content of the merge operation.

Model participation in team viewers: The team views now make use of the Common Navigator framework. By extending a Common Navigator extension point and a Team extension point, and supplying a content provider and a label provider, a Model Provider can appear in the team views. With a few more additional steps, it is also possible to provide Merge Preview support for a model.

Remote Discovery: Model Providers can participate in Remote Discovery through the use of the Team ProjectSetCapability class to obtain an URI from project set entries. This URI can then be used with the Eclipse File System API to access remote contents.

Model History: Model Providers can access individual File History through the FileHistory API, and present a model history as they wish in a custom History Page which get displayed in the History View.

17.4.1. Repository Roadmap for Logical Model Integration

To provide full support for logical models, a repository provider can perform the following steps:

Contribute the appropriate repository operations to elements that adapt to ResourceMapping.

Ensure that operations performed on resource mappings include all the appropriate model elements and resources by using an ISynchronizationScope and supporting API.

Allow model providers to participate in headless merging through the IMergeContext interface and supporting API.

Allow model providers to participate in merge previews by using the teamContentProviders for the models involved in the merge. A ModelSynchronizeParticipant class is provided to help manage the relationship between the model content, a merge context and the Compare framework.

Provide access to the history of workspace files through the IFileHistoryProvider API

Provide access to remote configurations using the Eclipse File System API in the org.eclipse.core.filesystem plug-in and link this to workspace projects through the ProjectSetCapability

Support logical model element decoration by providing a workspace Subscriber for use with the SynchronizationStateTester API.

Allow models to group related changes by implementing the IChangeGroupingRequestor API.

The following sections describe each of these points in more detail. The org.eclipse.ui.examples.filesystem plug-in contain an example that illustrate several of these points. You can check the project out from the Git 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 3.3 version tag (most likely R3\_3) or a date tag of June 30, 2007.

Contributing Actions to Resource Mappings

The Basic Resource Mapping API

The resource mapping API consists of the following classes:

ResourceMapping: The class to which logical model elements adapt to indicate that the model corresponds to a set of resources in the workspace. The methods of interest are:

Object getModelObject(): The model object from which the mapping was derived (or adapted).

ResourceTraversal[] getTraversals(ResourceMappingContext, IProgressMonitor): The resource traversal that cover the resources that constitute the model object.

ResourceTraversal: A ResourceTraversal contains a set of resources and a depth flag that indicates the depth to which the resources in the traversal are associated with the originating model object. Resource traversals are provided to a client by a resource mapping in order to describe the contents of a model in such a way that the client (e.g. a repository provider) can perform its operations in as efficient a means as possible. Methods of interest are:

getResources()

getDepth()

ResourceMappingContext: a context that is provided to the resource mapping by the client when obtaining traversals. This context allows the logical model to determine what the remote state of the model is so that the proper resources can be covered by the resource traversals returned by the resource mapping. The use of the ResourceMappingContext and RemoteResourceMappingContext is a bit more complicated and is described later.

There are two types of plugins that should be interested in resource mappings. Those who provide a model that consists of, or is persisted in, resources in the workspace and those that want to perform operations on resources. The former case will be covered in the model roadmap and the later case is covered in the next section.

Resource Mappings and Object Contributions

Plug-ins that contribute extensions to adaptable extension points will have to make two changes to support the new ResourceMapping APIs:

Update any objectContributions of the popupMenus extension point in their plugin.xml file to target ResourceMapping instead of IResource (for those for which this is appropriate).

Update their actions to work on ResourceMapping instead of IResource and respect the depth constraints provided in the traversals.

Plug-ins that add object contributions to IResource can now add them to ResourceMapping instead, if the action can apply to multiple resources. Here is an XML snippet that contributes a menu action to objects that adapt to resource mappings:

<extension

point="org.eclipse.ui.popupMenus">

<objectContribution

objectClass="org.eclipse.core.resources.mapping.ResourceMapping"

adaptable="true"

id="org.eclipse.team.ccvs.ui.ResourceMapperContributions">

<enablement>

<adapt type="org.eclipse.core.resources.mapping.ResourceMapping">

<test

property="org.eclipse.core.resources.projectPersistentProperty"

args="org.eclipse.team.core.repository,org.eclipse.team.cvs.core.cvsnature" />

</adapt>

</enablement>

<action

label="%UpdateAction.label"

definitionId="org.eclipse.team.cvs.ui.update"

class="org.eclipse.team.internal.ccvs.ui.actions.UpdateAction"

tooltip="%UpdateAction.tooltip"

menubarPath="team.main/group2"

id="org.eclipse.team.cvs.ui.update">

</action>

...

</objectContribution>

</extension>

Contributions to ResourceMapping will automatically apply to objects that adapt to IResource. This transitive association is handled by the Workbench. Filtering of the contributions to resource mappings can be done using enablement expressions. An expression for filtering by project persistent property has been added to allow repository providers to have their menus appear on projects that are mapped to their repositories.

Actions that have been contributed to the ResourceMapping class will be given a selection that contains one or more ResourceMappings. It is the actions responsibility to translate the resource mapping into a set of resources to be operated on. This can be done by calling getTraversals to get the traversals of the mapping. Traversals are used to allow the clients of the traversal to optimize their operations based on the depth of the resources being traversed. A client may traverse the resource manually or may use the resource and the depth as input into an operation that the action delegates to do the work. As an example, if the user performs a CVS update on a java package and the java package resource mapping maps to a folder of depth one, CVS would issue an appropriate command ("cvs update -l" for those who are curious) which would perform a shallow update on the folder the package represents.

Although it is possible to obtain a set of traversals directly from the selected resource mappings, there are model relationships (or repository relationships) that may require the inclusion of additional resources or model elements in an operation. The next section describes how to ensure that all required resources are included in an operation

Operation Scope

For team operations, the selected mappings need to be translated into the set of mappings to be operated on. This process involves consulting all model providers to ensure that they get included in operations on resources that match their enablement rules. The term we use to describe the complete set of resource mappings to be operated on is the operation scope. The following API has been provided for this:

ISynchronizationScope: Interface that defines the API for accessing the scope of the operation. It provides access to all the resource mappings being operated on and the traversals for those mappings as they were calculated during the scope building process.

ISynchronizationScopeManager: Interface that defines the API for creating and managing a scope.

SynchronizationScopeManager: Extendable class that provides a default implementation of the ISynchronizationScopeManager API.

ModelOperation: Extendable operation class that generates a scope using a provided scope manager and prompts if additional resources or mappings have been included in the operation due to model provider relationships.

The initialize(IProgressMonitor) method of the SynchronizationScopeManager class handles the entire process of converting an input set of resource mappings into the complete set of mappings that need to be operated on as well as the complete set of traversals that cover these mappings. A repository provider can tailor the process by:

Providing a RemoteResourceMappingContext for use when obtaining resource traversals from resource mappings.

Overriding SynchronizationScopeManager to tailor the scope management process as required.

The next two sections describe these points in more detail.

Remote Resource Mapping Context

In order to guarantee that all necessary resources get included in a team operation, the model provider may need the ability to glimpse at the state of one or more resources in the repository. For some models, this may not be required. For instance, a java package is a container visited to a depth of one, regardless of the remote state of the model. Given this, a repository provider can easily determine that outgoing deletions should be included when committing or that incoming additions should be included when updating. However, the resources that constitute some logical models may change over time. For instance, the resources that constitute a model element may depend of the contents of a manifest file (or some other similar mechanism). In order for the resource mapping to return the proper traversal, it must access the remote contents of the manifest file (if it differs from the local contents) in order to see if there are additional resources that need to be included. These additional resources may not exist in the workspace but the repository provider would know how to make sure they did when the selected action was performed.

In order to support these more complex models, a RemoteResourceMappingContext can be passed to the ResourceMapping#getTraversals method. When a context is provided, the mapping can use it to ensure that all the necessary resources are included in the traversal. If a context is not provided, the mapping can assume that only the local state is of interest.

The remote resource mapping context provides three basic queries:

What type of comparison is being performed: two-way or three-way?

For two-way, does the local contents differ from the remote contents?

For three-way, are there local changes and are there remote changes?

What are the contents of the remote (and base for three-way) counterparts of the file?

What are the remote members of a folder?

The answer to the first question above depends on the type of operation that is being performed. Typically, updates and merges are three-way while comparisons and replace operations (at least for CVS) are two-way.

The Eclipse Team API includes a Subscriber class that defines an API for providing the synchronization state between the local workspace and a remote server. A SubscriberResourceMappingContext is provided that uses a Subscriber to access the necessary remote state. Clients that have a Subscriber do not need to do any additional work to get a resource mapping context.

Subclassing SynchronizationScopeManager

The SynchronizationScopeManager class can be subclassed to tailor the scope generation and management process. The two main reasons for subclassing the scope manager are:

The repository provider needs to include additional resources due to some repository level relationship (e.g. change set). This can be accomplished by overriding the adjustInputTraversals(ResourceTraversal[]) method.

The synchronization has a longer lifecycle (e.g. Synchronize view vs. dialog) and needs the potential to react to scope changes. The ISynchronizationScopeParticipant interface defines the API that model providers can use to participate in the scope management process. The SubscriberScopeManager class is a Subscriber based subclass of SynchronizationScopeManager that involves participants in the scope management process. An example of why this type of process is needed is working sets. If a working set is one of the resource mappings in a scope, the set of traversals covered by the scope would increase if resources were added to the working set.

Model-based Merging

The main repository operation type that requires model participation is merging. In many cases, models only need to participate at the file level. For this, the IStorageMerger API was introduced to allow model providers to contribute mergers that should be used to merge files of a particular extension or content type. However, in some cases, models may need additional context to participate properly in a merge. For this purpose, we introduced the IResourceMappingMerger and IMergeContext APIs.

Merge operations are still triggered by actions associated with a repository provider. However, once a merge type operation is requested by the user, the repository provider needs to involve the model providers in the merge process to ensure that the merge does not corrupt the model in some way.

There are two main pieces of repository provider API related to the model-based merging support.

API to describe the synchronization state of the resources involved in the merge.

API to allow model providers to merge model elements.

The following sections describe these two pieces.

API for Synchronization State Description

An important aspect of model-based merging is the API used to communicate the synchronization state of the resources involved to the model provider. The following interfaces are used to describe the synchronization state:

ISynchronzationContext: provides access to the synchronization state of all resources within the scope of the operation. The following API pieces are used to describe the synchronization state of resources.

IDiff: A generic path-based diff that consist of a diff kind (NO\_CHANGE, ADD, REMOVE, CHANGE).

ITwoWayDiff: A diff that provides more details about a change between two states of an element (CONTENT, MOVE\_TO, MOVE\_FROM, COPY\_FROM, REPLACE).

IThreeWayDiff: A diff that consists of either a local or remote two-way diff or both and provides a direction description for the change (INCOMING, OUTGOING, CONFLICTING) based on the presence of either or both the local and remote change.

IResourceDiff: A two-way diff that applies to resources and provides access to the before state and after state of files through the IFileRevision API.

IDiffTree: A generic path-based data structure for accessing a tree of diffs.

IResourceDiffTree: A resource-based data structure that contains the synchronization state for all resources in the operation scope that are out-of-sync.

Abstract classes are provided for all these interfaces with the convention that the class names match the interface names with the "I" prefix removed. The only class that repository providers must override is the ResourceDiff class so that appropriate before and after file revisions can be provided.

API for Model Merging

The IMergeContext interface extends synchronization context with additional methods that support merging. Callback methods exist for:

Merging one or more diffs.

Rejecting the remote changes in one or more diffs

Indicating that one or more conflicts have been merged by the model

Performing a group of operations while holding an appropriate scheduling rule.

An abstract MergeContext class is provided that contains default implementations for much of the merging behavior and also uses the IStorageMerger to perform three-way merges. A SubscriberMergeContext class is also provided which handles the population and maintenance of the synchronization state description associated with the merge context.

An operation class, ModelMergeOperation is provided which uses the IResourceMappingMerger API to perform a model-based merge operation. Subclasses need to override the initializeContext(IProgressMonitor) method to return a merge context. The operation uses this context to attempt a headless model-based merge. If conflicts exist, the preview of the merge is left to the subclass. As we'll see in the next section, there is a ModelParticipantMergeOperation that provides preview capabilities using a ModelSynchronizeParticipant.

Model Content in Team Viewers

Support for the display of logical models in a team operation is provided using the Common Navigator framework which was introduced in Eclipse 3.2. Logical models can associate a content extension with a model provider using the org.eclipse.team.ui.teamContentProviders extension point. Team providers access these content providers through the ITeamContentProviderManager.

There are several places where a team provider may wish to display logical models:

When the scope of an operation has been expanded due to model provider relationships. The ModelOperation class provides a prompt that uses the registered team content providers to inform the user of a scope expansion.

When a merge preview is required. The ModelSynchronizeParticipant class makes use of the team content providers and a merge context to display a merge preview. Team providers can extend this class in several ways to tailor the preview.

The ModelSynchronizeParticipant provides integration into the Synchronize view or any container that can display ISynchronizePages. The participant makes use of both the pre-existing synchronization participant capabilities and the Common Navigator capabilities to allow for team providers and models to tailor the toolbar, context menu and other aspects of the merge preview. The ModelSynchronizeParticipant provides the following:

A synchronize page that defines the link between a scope manager, merge context and team content providers.

Toolbar actions for displaying direction modes (Incoming, Outgoing, Both and Conflicting modes).

Toolbar action for cycling through models involved (i.e. Show All, Show Resources, Show Java, etc).

Refresh and scheduled refresh actions.

Action to open elements in a compare editor.

Change Navigation actions.

Standard Merge actions (Merge, Overwrite and Mark as Merged) for which models can register handlers and team providers can provide appropriate labels (e.g. Update for CVS).

Here's a checklist of steps for tailoring a model synchronize participant for a particular Team provider:

Define the participant using the org.eclipse.team.ui.synchronizeParticipants extension point. This will require team providers to create a subclass of ModelSynchronizeParticipant

Define a viewer for the participant using the org.eclipse.ui.navigator.viewer extension point. The viewer definition is used to specify what menu groups appear in the context menu.

Override the initializeConfiguration method of the participant in order to set the ISynchronizePageConfiguration.P\_VIEWER\_ID to the id of the viewer specified in the previous step.

Override the createMergeActionGroup to provide a custom subclass of MergeActionGroup in order to tailor the appearance of the merge related actions.

If the participant is persistable, override the createScopeManager and restoreContext methods to recreate the scope manager and merge context for the participant when it is restored on startup.

Implement a subclass of ModelParticipantMergeOperation for handling the transition from a model-based merge to a preview in a dialog or Synchronize view.

The following XML snipets illustrate how the CVS participant class is registered and how it's viewer is defined.

<extension point="org.eclipse.team.ui.synchronizeParticipants">

<participant

name="CVS"

icon="$nl$/icons/full/eview16/cvs\_persp.gif"

class="org.eclipse.team.internal.ccvs.ui.mappings.WorkspaceModelParticipant"

id="org.eclipse.team.cvs.ui.workspace-participant">

</participant>

</extension>

<extension point="org.eclipse.ui.navigator.viewer">

<viewer viewerId="org.eclipse.team.cvs.ui.workspaceSynchronization">

<popupMenu

allowsPlatformContributions="false"

id="org.eclipse.team.cvs.ui.workspaceSynchronizationMenu">

<insertionPoint name="file"/>

<insertionPoint name="edit" separator="true"/>

<insertionPoint name="synchronize"/>

<insertionPoint name="navigate" separator="true"/>

<insertionPoint name="update" separator="true"/>

<insertionPoint name="commit" separator="false"/>

<insertionPoint name="overrideActions" separator="true"/>

<insertionPoint name="otherActions1" separator="true"/>

<insertionPoint name="otherActions2" separator="true"/>

<insertionPoint name="sort" separator="true"/>

<insertionPoint name="additions" separator="true"/>

<insertionPoint name="properties" separator="true"/>

</popupMenu>

</viewer>

</extension>

File History

A file history API has been added to allow models to access the history of files. The file history API consists of the following interfaces:

IFileHistoryProvider: Obtained from a RepositoryProvider, the file history provider allows clients to obtain the file history of a particular file.

IFileHistory: A data structure that allows access to the history of a file

IFileRevision: The description of a single revision or state of a file at a particular point in time in its history.

Along with this API, a generic file History view has been added. This will allow Team providers to display their file/resource history in a shared view and also allows models to display model element history for elements that do not map directly to files. The History view is a page-based view which obtains a page for the selected element in the following way:

A history page for workspace resources is obtained by adapting the RepositoryProvider associated with the project that contains the resource to an IHistoryPageSource.

A history page for other objects is obtained by adapting the object itself to an IHistoryPageSource.

Project Set Capability

Methods have been added to ProjectSetCapability to support the translation between a reference string used to identify a mapping between a project and remote content and URIs that identify a file-system scheme registered with the org.eclipse.core.filesystem.filesystems extension point. Team providers can optionally provide support for this in order to allow logical models to performing remote browsing and project loading.

Decorating Model Elements

Team providers can decorate model elements by converting their lightweight decorators to work for resource mappings in the same way object contributions are converted to work for resource mappings. However, there is one aspect of logical model element decoration that is problematic. If a model element does not have a one-to-one mapping to a resource, the model element may not receive a label update when the underlying resources change.

To address this issue, the ITeamStateProvider was introduced in order to give model providers access to state changes that may affect team decorations. In addition, model views can use a SynchronizationStateTester to determine when the labels of logical model elements need to be updated. This API relies on the ITeamStateProvider interface to determine when the team state of resource has changed and can be passed to a team decorator as part of an IDecorationContext.

Grouping Related Changes

Some logical models need to ensure that a set of changed files get committed or checked-in to the repository at the same time. To facilitate this, repository providers can adapt their RepositoryProviderType to an instance of IChangeGroupingRequestor. This API allows models to request that a set of files get committed or checked-in as a single unit.

17.4.2. Model Roadmap for Logical Model Integration

Here is a list of what model providers can do to take advantage of the Team logical model support:

Adapt model elements in model views to ResourceMapping in order to allow resource based operations to appear on your model elements.

Register a ModelProvider to ensure that your model is consulted when operations are performed on the resources related to your model.

Use the ResourceChangeValidator when performing operations on resources in order to ensure that any potential side effects on the model elements related to those resources are made known to the user.

Implement an IResourceMappingMerger in order to participate in headless merges that involve the resources related to your model.

Register a teamContentProvider in order to participate in Team viewers such as merge previews.

Provide an IHistoryPageSource to show logical model history in the Team history view.

Use the Eclipse File System API to access the remote state of model projects.

Use the SynchronizationStateTester API to ensure proper decoration of model elements that do not have a one-to-one mapping to resources.

Use the Saveable API to manage the save life cycle of model elements.

Use the ElementLocalHistoryPageSource API to show local history for sub-file model elements (e.g. Java methods)

Use the ChangeTracker API to group related changes together.

The following sections describe each of these points in more detail. The org.eclipse.ui.examples.filesystem plug-in contain an example that illustrate several of these points. You can check the project out from the Git 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 3.3 version tag (most likely R3\_3) or a date tag of June 30, 2007.

Resource Mappings

The Basic Resource Mapping API

The resource mapping API is purposely simple with logical model manipulations omitted. A client can't use this interface to display logical models or gain any interesting additional knowledge about it. It's purpose is simply to map one or more model elements to workspace resources.

The API consists of the following classes:

ResouceMapping: ResourceMapping is the Class to which logical model elements adapt to indicate that the model corresponds to a set of resources in the workspace. The methods of interest are:

Object getModelObject(): The model object from which the mapping was derived (or adapted).

ResourceTraversal[] getTraversals(ResourceMappingContext, IProgressMonitor): The resource traversal that cover the resources that constitute the model object.

ResourceTraversal: A ResourceTraversal contains a set of resources and a depth flag that indicates the depth to which the resources in the traversal are associated with the originating model object. Resource traversals are provided to a client by a resource mapping in order to describe the contents of a model in such a way that the client (e.g. a repository provider) can perform its operations in as efficient a means as possible. Methods of interest are:

getResources()

getDepth()

ResourceMappingContext: A context that is provided to the resource mapping by the client when obtaining traversals. This context allows the logical model to determine what the remote state of the model is so that the proper resources can be covered by the resource traversals returned by the resource mapping. The use of the ResourceMappingContext and RemoteResourceMappingContext is a bit more complicated and is described in the Resource Mapping Context section.

There are two types of plugins that should be interested in resource mappings. Those who provide a model that consists of, or is persisted in, resources in the workspace and those that want to perform operations on resources. The former is covered in the next section while the later is covered in the Repository Roadmap for Logical Model Integration.

Adapting a Model to a ResourceMapping

Plug-ins that adapted their model objects to IResource in order to get resource specific actions shown in the context menu can now adapt to ResourceMapping if a richer description of how the object adapts to resources is beneficial. However, they are not required to do so if there is no benefit. For instance a Java compilation unit (i.e. \*.java file shown in a JDT view) that now currently adapts to IFile need not adapt to ResourceMapping since nothing is gained. However, a Java package should adapt to ResourceMapping in order to indicate that the package consists of only the files in the corresponding folder and not the subfolders.

The preferred way to adapt model elements to a resource mapping is to use an adapter factory. The following is the XML markup for contributing an adapter factory in a plug-in manifest.

<extension

point="org.eclipse.core.runtime.adapters">

<factory

class="org.eclipse.example.library.logical.AdapterFactory"

adaptableType="org.eclipse.example.library.Book">

<adapter type="org.eclipse.core.resources.mapping.ResourceMapping"/>

</factory>

<factory

class="org.eclipse.example.library.logical.AdapterFactory"

adaptableType="org.eclipse.example.library.Library">

<adapter type="org.eclipse.core.resources.mapping.ResourceMapping"/>

</factory>

...

</extension>

The adapter factory implementation would look something like this:

public class AdapterFactory implements IAdapterFactory {

public Object getAdapter(Object adaptableObject, Class adapterType) {

if((adaptableObject instanceof EObject) && adapterType == ResourceMapping.class) {

return new EObjectResourceMapping((EObject)adaptableObject);

}

return null;

}

public Class[] getAdapterList() {

return new Class[] {ResourceMapping.class};

}

}

Model objects can implement the IAdaptable interface. When they do so, they must ensure that the Platform adapter manager is consulted. This can be done by either subclassing PlatformObject or by using the following line of code:

Platform.getAdapterManager().getAdapter(Object, Class)

The above is the preferable approach. However, the model object can implement the IAdaptable interface and provide a getAdapter(Class) implementation that creates returns an instance of ResourceMapping explicitly when asked for one. This is a more straightforward approach but the least desirable as the model must have explicit knowledge of the adaptation to resources.

In some cases, the provider of a logical model may not want their model to adapt to IResource in every context or may want the object to adapt differently for object contributions than for other contexts. The workbench UI provides a special intermediate adapter API, IContributorResourceAdapter, for this purpose. When objects are being adapted to IResource in the context of object contributions, the workbench first tries to adapt the resource to IContributorResourceAdapter before trying to adapt to IResource directly. A new sub-interface of this interface, IContributorResourceAdapter2, has been added which provides the same capability for ResourceMapping. The only difference is that the model provider should register a factory for IContributorResourceAdapter since the Workbench does an instanceof check to see if the contributed adapter is also an instance of IContributorResourceAdapter2.

The implementation of the ResourceMapping subclass for a Java package would look something like this.

public class JavaPackageResourceMapping extends ResourceMapping {

IPackageFragment package;

...

public getModelObject() {

return package;

}

public ResourceTraversals[] getTraversals(

ResourceMappingContext context,

IProgressMonitor monitor) {

return new ResourceTraversal[] {

new ResourceTraversal(

new IResource[] { package.getCorrespondingResource() },

IResource.DEPTH\_ONE, IResource.NONE)

}

}

}

This is a fairly straightforward mapping so the implementation is not complex. The complexity of the resource mapping implementation will, of course, vary from model to model.

Resource Mapping Context

One of the advantages of a Resource Mapping API is that it allows plug-ins to implement any operations they desire in terms of resource mappings (e.g. Git pull, Git commit, Git tag, dirty decoration, etc.). However, the API that has been introduced so far deals only with the local state of the model. When working with a model that may be shared between developers, you end up in a situation where the remote state of the model (i.e. the state of the model that another user has checked-in to the repository) may differ from the state in the workspace. If you performed a Git pull, you would want the local state of the model to match the remote state even if it meant that additional files needed to be included or some files needed to be removed.

This is not an issue for some logical models. For instance, a java package is a container visited to a depth of one, regardless of the remote state of the model. Given this, a repository provider can easily determine that outgoing deletions should be included when committing or that incoming additions should be included when updating. However, the resources that constitute some logical models may change over time. For instance, the resources that constitute a model element may depend of the contents of a manifest file (or some other similar mechanism). In order for the resource mapping to return the proper traversal, it must access the remote contents of the manifest file (if it differs from the local contents) in order to see if there are additional resources that need to be included. These additional resources may not exist in the workspace but the repository provider would know how to make sure they did when the selected action was performed.

In order to support these more complex models, a RemoteResourceMappingContext can be passed to the ResourceMapping#getTraversals method. When a context is provided, the mapping can use it to ensure that all the necessary resources are included in the traversal. If a context is not provided, the mapping can assume that only the local state is of interest.

When does a ResourceMapping need to worry about the RemoteResourceMappingContext?

A ResourceMapping need only worry about a context supplied to the getTraversals method in cases were the resources that make up a model change over time and the relationship between the model and resources cannot be described by a simple traversal that is guaranteed to encompass those resources (and only those resources) that constitute the model. For example, although the resources of a Java package may change over time, the package can be described as a folder of depth of one so a resource mapping for java packages would not need to make use of the resource mapping context.

As a more complicated example, consider an HTML file that contains several images. Let's make the assumption that any images references from an HTML file are part of the model of that file. When updating the local contents of the HTML file from a repository, the user would expect that any new images would be included. The getTraversals method for a ResourceMapping for the HTML file model would look something like this:

public class HTMLResourceMapping extends ResourceMapping {

private HTMLFile htmlFile;

public ResourceTraversal[] getTraversals(ResourceMappingContext context,

IProgressMonitor monitor)

IResource[] resources = htmlFile.getResources();

if (context instanceof RemoteResourceMappingContext) {

// Look for any additional resources on the server

RemoteResourceMappingContext remoteContext = (RemoteResourceMappingContext)context;

IFile file = htmlFile.getFile();

if (remoteContext.hasRemoteChange(file, monitor)) {

IStorage storage = remoteContext.fetchRemoteContents(file, monitor);

IResource[] additionalResources = getReferences(storage.getContents());

resources = combine(resources, additionalResources);

}

if (remoteContext.isThreeWay() && remoteContext.hasLocalChange(file, monitor)) {

IStorage storage = remoteContext.fetchBaseContents(file, monitor);

IResource[] additionalResources = getReferences(storage.getContents());

resources = combine(resources, additionalResources);

}

}

return new ResourceTraversal[] {

new ResourceTraversal(resources, IResource.DEPTH\_ZERO, IResource.NONE)};

}

}

Notice that there are two sets of resources included in the model: those derived from the local contents of the HTML file in the workspace and those obtained from the contents of the remote file and base file. In either of these two sets, there may be resources that do not exist in the workspace. For instance, the local HTML file may contain a relative link to an image that does not exist in the workspace. This resource should be included so that it will be fetched if it exists remotely. As for the remote file, it may contain a new copy that references additional images that should be fetched when the new remote contents are downloaded.

Model Providers

Model providers are a means to group related resource mappings together. Here is a link to the ModelProvider class. This class serves three main purposes:

From a model provider, clients can then obtain additional API pieces for performing operations on a set of resource mappings using the adaptable mechanism. For example, the IResourceMappingMerger for the model is obtained by adapting the model provider.

Given a set of file-system resources, clients can query whether a model provider has model elements persisted in those resources and, if it does, obtain the set of resource mappings that describe the relationship.

For operations on a set of resources, the resource change validator will query the model providers in order to determine if there are any potential side effects of an operation that users should be made aware of. This is covered in a separate section on change validation.

The following is an example of a modelProvider extension definition.

<extension

id="modelProvider"

name="Library Example"

point="org.eclipse.core.resources.modelProviders">

<modelProvider

class="org.eclipse.team.examples.library.adapt.LibraryModelProvider"

name="Library Example"/>

<extends-model id="org.eclipse.core.resources.modelProvider"/>

<enablement>

<test property="org.eclipse.core.resources.projectNature"

value="org.eclipse.team.examples.library.view.nature" />

</enablement>

</extension>

The LibraryModelProvider is a subclass of ModelProvider. The enablement rule is used to match resources that the Library model persists its model in. In the above example, the model provider will match any resource in a project that has the library nature.

Once the model provider is defined, the ResourceMapping#getModelProviderId() method should be overridden to return the id of the model provider.

public String getModelProviderId() {

return "org.eclipse.team.examples.library.adapt.modelProvider";

}

To get the proper inverse mapping of resources to resource mapping for those resources that match your provider's enablement rule, you should also override one or both of the getMapping methods. The method that you need to override depends on whether your model has elements that contain multiple resources or not. If your model elements map to a single resource, you can override the method that accepts a singleIResource argument. Otherwise, you will need to override the method that accepts an array of resources. Here's an example using the single resource case.

The following example method wraps a library model file in an appropriate resource mapping. It also wraps folders that contain files that are of interest to the model provider.

public class LibraryModelProvider extends ModelProvider {

public ResourceMapping[] getMappings(IResource resource,

ResourceMappingContext context, IProgressMonitor monitor) {

if (isModelFile(resource)) {

// Return a resource mapping on the file

return new LibraryResourceMapping(resource);

} if (containsModelFiles(resource)) {

// Create a deep resource mapping on the container

return new LibraryContainerResourceMapping(resource);

}

// The resource is not of interest to this model provider

return null;

}

}

Clients can then access the model provider to determine whether the model providers cares about the resources that are about to be operated on. The next section describes API that will be provided to team operations that use the model provider API to determine the complete set of resource mappings to be operated on when a team operation is performed on a set of selected resources or model elements.

Resource Change Validation

Operations performed on resources should be validated first to ensure that the user is aware of any potential side effects. Here are the steps required to validate a resource change.

Build up a description of the change using the IResourceChangeDescriptionFactory. The factory produces an IResourceDelta that mirrors what the resulting resource delta will look like once the operation is performed.

Validate the change using the ResourceChangeValidator. The validator consults all the model providers that have registered an interest in the affected resources. The result is one or more status that contain the id of the originating model and a description of the potential side effect of the operation on the model.

Make the user aware of any potential side effects from models that are unknown to the originator of the operation. For instance, if a Java refactoring has received a side effect from the Java model, it can be ignored since the refactoring understands the semantics of the Java model. However, if a side effect from the Library model is returned, it should be made known to the user since Java has no knowledge of the Library model.

Model-based Merging

When a Team provider is attempting a headless merge, it will do the following:

Obtain the resource mappings from the selected elements

Determine the model providers involved using the ResourceMapping#getModelProvider() method.

Expand the scope of the operation to include all necessary resource mappings.

Build up a description of the synchronization state between the local and remote. This description is provided to the models through the IMergeContext API.

Adapt the model providers to IResourceMappingMerger.

Invoke the validateMerge method on each merger, passing in the synchronization description, to ensure there is not some condition that should prevent an attempted merge.

Delegate the merge to the model mergers to perform the merge.

Model providers can delegate the merge of individual files back to the Team provider if they only want to control the order of the merge or may perform the merge themselves and signal to the team provider when they are done.

Model Content in Team Operation Views

The display of model elements in the context of a Team operation is made possible by the Common Navigator framework.

Provide an extension of the org.eclipse.ui.navigator.navigatorContent extension.

Provide an extension of the org.eclipse.team.ui.teamContentProviders extension point to associate the content provider with a model provider.

Implement a content provider and label provider for the navigatorContent extension. It is best to subclass SynchronizationContentProvider and SynchronizationLabelProvider for this. They wrap existing content/label providers and handle interactions with the team provider such as sync state decoration and filtering based on sync state.

The above steps will allow models to appear in dialogs used by team operations. There are additional steps required to integrate into a merge preview.

Adapt the model provider to ISynchronizationCompareAdapter. This is done to allow the model to open model specific compare editors on model elements.

Implement an acti on provider for the navigatorContent extension as a subclass of SynchronizationActionProvider. This will allow the model to register merge handlers that will be invoked when users perform merge operations on the model provider's elements in the merge preview.

History View

The following improvements have been made in the area of file history and model element history:

The History view has been generalized to display the history of any object that adapts to an IHistoryPageSource. Thus, model providers can render the history for their logical model elements.

An IFileHistoryProvider API has been added for obtaining and querying the remote history of workspace files. Logical models can obtain and interpret this history.

An IHistoryCompareAdapter is used to hook up the selection in an IHistoryPage to compare editors so that history pages can be reused in the context of compare operations.

Remote Browsing

The following has been provided to support remote browsing:

An Eclipse File System API has been introduced in the org.eclipse.core.filesystem plug-in.

API has been added to the ProjectSetCapability to obtain the URI from project set entries. These URIs can be used to obtain a FileSystem for the purposes of browsing remote contents.

Models that require an IResource base project can use the file system to create a team-private project.

Methods have been added to the ProjectSetCapability for loading a URI as a project.

Decorating Model Elements with Team State

Team providers can decorate model elements by converting their lightweight decorators to work for resource mappings in the same way object contributions are converted to work for resource mappings. However, there is one aspect of logical model element decoration that is problematic. If a model element does not have a one-to-one mapping to a resource, the model element may not receive a label update when the underlying resources change.

To address this issue, the ITeamStateProvider was introduced in order to give model providers access to state changes that may affect team decorations. In addition, model views can use a SynchronizationStateTester to determine when the labels of logical model elements need to be updated. This API relies on the ITeamStateProvider interface to determine when the team state of resource has changed and can be passed to a team decorator as part of an IDecorationContext.

Using the Saveable API

Logical model providers may use the Saveable API to manage the Save lifecycle of their model elements. If they do so, they should adapt their Saveable instances to a ResourceMapping to ensure that repository providers (and other tools) can determine which resources would be affected if the saveable was saved. For instance, this is important in the case where a repository provider wants to ensure that there are no open dirty editors on a set of resources before a repository operation is performed.

Showing the history of sub-file elements

The ElementLocalHistoryPageSource class can be used by models to show the local history for a sub-file element such as a Java method. History can be shown in the History view by calling the TeamUI#showHistory method with the model element and history page source as arguments. Models that call this also need to provide a subclass of StructureCreator as described in the Advanced compare techniques section. The history can also be shown in a dialog using the HistoryPageCompareEditorInput class. Subclasses can control whether the dialog is to be used for comparison purposes or as a dialog that supports replacing the current element with an element from the history.

Grouping related changes

Model providers that wish to ensure that changes to sets of resources are committed or checked-in to a repository together can subclass the ChangeTracker class. This class is used to track local changes and issue requests to the repository provider to group related changes together.

17.5. 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 functionality 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 menus 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 editing 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 operations that invoke the move/delete hook or fileModificatonValidator.

Use the resource rule factory to ensure that the proper rules are obtained for resource 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.

17.5.1. 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

id="team.main"

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.

Commands can be contributed in a similar way:

<extension point="org.eclipse.core.expressions.definitions">

<definition id="org.eclipse.ui.example.ccvs.ui.IFileContributions">

<iterate ifEmpty="false">

<adapt type="org.eclipse.core.resources.IFile">

<test property="org.eclipse.core.resources.projectPersistentProperty"

value="org.eclipse.team.core.repository=org.eclipse.team.cvs.core.cvsnature"/>

</adapt>

</iterate>

</definition>

</extension>

<extension point="org.eclipse.ui.menus">

<menuContribution locationURI="popup:team.main?after=group3">

<command commandId="org.eclipse.team.ccvs.ui.ignore"

id="org.eclipse.ui.example.ccvs.ui.ignore"

style="push">

<visibleWhen checkEnabled="false">

<or>

<with variable="activeMenuSelection">

<reference definitionId="org.eclipse.ui.example.ccvs.ui.IFileContributions"/>

</with>

<with variable="activeMenuEditorInput">

<reference definitionId="org.eclipse.ui.example.ccvs.ui.IFileContributions"/>

</with>

</or>

</visibleWhen>

</command>

</menuContribution>

</extension>

17.5.2. 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>

17.5.3. 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>

...

17.5.4. 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.ccvs.ui.CVSProjectSetSerializer"/>

</extension>

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

17.5.5. 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:

Those which call out to an external program that then operates directly against the file system.

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.canHandleLinkedResourceURI() 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(IResource.CHECK\_ANCESTORS) to determine if a resource is a link or is the descendant of a link.

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

17.5.6. 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.

17.5.7. 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.

17.5.8. Adding History support

There are two parts to adding History support to your repository:

Providing implementations for the core history interfaces; see the Providing History section.

Creating the UI elements for the History Page which gets displayed in the History View; see the Showing History section.

Providing History

The core history interfaces allow users to access the history of an item by querying the IFileHistoryProvider returned by the RepositoryProvider. To do this, you need to:

Provide an implementation for IFileHistory by extending FileHistory.

Provide an implementation for IFileRevision by extending FileRevision.

Provide an implementation for IFileHistoryProvider by extending FileHistoryProvider.

Provide an implementation for getFileHistoryProvider to allow users to fetch the IFileHistoryProvider from your RepositoryProvider.

Showing History

To display history for items in your repository, you can provide one or more HistoryPages which will be shown in the History view. To do this, you need to:

Provide an implementation for IHistoryPage by extending HistoryPage.

Provide an implementation for IHistoryPageSource by extending HistoryPageSource. When a resource is selected to be shown in the History view it will try to adapt the FileHistoryProvider it gets from your repository provider to a HistoryPageSource. If it doesn't succeed, it will then try to adapt the repository provider itself. For non-resource objects, the History view will try to adapt the object to a HistoryPageSource. The FileSystem History view example registers an adapter factory that adapts a FileHistoryProvider to an IHistoryPageSource.

Create a Show History Action, provide an object contribution for the Action. See org.eclipse.ui.popupMenus for more info.

Showing History for Sub-file Elements

To display the history of a model element that is contained in a file, you can do the following:

Implement a StructureCreator for the file.

Subclass ElementLocalHistoryPageSource and adapt your sub-file element(s) to an instance of the subclass.

To show the history in the History View, pass your history page source and element to the showHistoryFor method of TeamUI.

To show the history in a dialog, provide History actions (show, compare, replace) that make use of a HistoryPageCompareEditorInput to show the history.

FileSystem History example

The FileSystem example illustrates how you can provide history and a history page for repository items. Select a resource in a FileSystem managed project and choose Team > Show History. The FileSystem History page will be displayed in the History view with all of the local revisions as well as the FileSystem revision displayed. You can right click on any revision and select Open from the context menu to open the revision.

FileRevisionEditorInput

To support the Open functionality, the FileSystem history example had to create its own FileSystemRevisionEditorInput (see org.eclipse.team.examples.filesystem.ui.FileSystemRevisionEditorInput). This class adapts to IFileRevision and to IFileState, which can be used to determine whether the editor contains remote contents (IFileRevision) or local history contents (IFileState).

Likewise, the existing FileRevisionEditorInput's that come with the SDK and are used by the CVS History page and Local History page all adapt to IFileRevision which can be used to determine if the revision contained in the editor is that of a remote revision.

18. 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.

18.1. 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.

18.1.1. 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.JavaLocalApplicationLaunchConfigurationDelegate"

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.

18.1.1.1. 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

name="%javaSourceLocatorName"

class="org.eclipse.jdt.launching.sourcelookup.JavaSourceLocator"

id="org.eclipse.jdt.launching.javaSourceLocator">

</sourceLocator>

</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 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>

...

18.1.1.2. 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.

18.1.1.3. 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.

18.1.1.4. 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

18.1.2. 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>

18.1.3. 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\_application"

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.

18.1.4. 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.localJavaApplication">

</launchConfigurationTypeImage>

...

18.1.5. 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 Workbench Command Expressions for an explanation of the XML syntax for enabling conditions. The complete syntax is described in the documentation for org.eclipse.debug.ui.launchShortcuts.

18.1.6. Launch groups

The Launch Group launch configuration type allows you to launch multiple other launch configurations sequentially, with configurable actions after launching each group member:

None: Continue launching the next member right away

Wait until terminated: Continue launching the next member only after this member has terminated

Delay: Delay launching the next member for a given amount of seconds.

Wait for console output (regexp): Allows to delay further processing of launch group elements until a certain output (matching a regular expression) appears on the console of the given group element

New launch groups can be created via the Run > Run Configurations... or Run > Debug Configurations... dialogs.

The Adopt launch if already running option allows to control behaviour of the group when the launch configuration referenced by the launch group element is already running (no matter who launched it). If checked, the launch configuration will not be launched again if it is running already. The existing launch will be adopted by the group (i.e. terminating the group will also terminate this launch). Any configured post launch action will still be executed!

18.2. 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.

18.2.1. 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.

18.2.2. 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 functionality 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.

18.2.3. 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.

18.2.4. 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.DetailsViewerConfiguration">

</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.

18.2.5. 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 functionality 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

19. 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.

19.1. 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.

19.2. 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().

Examples:

<eclipse.incrementalBuild/>

<eclipse.incrementalBuild project="MyProject" kind="incremental"/>

project the name of the project to be built

kind the kind of build to perform. Possible values include:

"incremental" to perform an incremental build

"full" to discard existing builder state and rebuild everything from scratch

"clean" to discard existing builder state but not rebuild

builder the name of the builder to run; If this value is not specified all builders are run

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"/>

19.3. 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\_PROGRESS\_MONITOR);

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.

19.4. Developing Ant tasks and types within Eclipse

Ant tasks and types must be loaded by an Ant classloader instead of a plug-in classloader. This can cause problems when developing and testing new tasks and types to be run in the same VM as Eclipse. To avoid these issues, it is necessary to store the tasks and types in a location that is not visible to any plug-in classloader.

The following guidelines should be followed when developing and debugging new tasks and types to be run within the same VM as Eclipse.

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 using PDE Launch, additional configuration is required:

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 Plug-in Development > Runtime Classpath by removing the Ant output directories from the plug-ins classpath.

The JAR file containing the Ant tasks and types and specified in the org.eclipse.ant.core.extraClasspathEntries extension, must be built manually, or using an external tool builder.

The org.eclipse.ant.core plugin can be used as an example of how to set up a plugin that contributes tasks within Eclipse. For development and testing of this plugin within Eclipse, an external tool builder explictly generates the tasks JAR when triggered on specific resource changes.

Also see Contributing tasks and types.

19.5. Expanding the Ant classpath

Plug-ins can contribute extra JAR files to the Ant classpath. The plug-in contributing the JARs is added to the Ant classpath, and as a result, classes inside the contributed JAR files will have access to all classes available to the plug-in. However, the reverse is not true. Since the contributed JAR files must be separate JARs, and not plug-in libraries, the plug-in classes do not have access to the classes provided by the contributed JARs. The org.eclipse.ant.core.extraClasspathEntries extension point provides an example of how to specify the extra JARs in the plugin.xml file.

20. User assistance support

What is user assistance?

User assistance is a set of components that together introduce the user to the product, guide the user through tasks, and help the user find more information about the product. There are four basic components:

Welcome

Help

Cheat Sheets

Status Handling

Welcome

Welcome (also called 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.

Help

The help system provides 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.

Help also includes context help support, where a user can summon help for a particular element in the UI (e.g. by pressing F1 on Windows). In addition, help provides the ability to perform a single search that finds information from any number of sources (federated information search).

Cheat Sheets

Cheat sheets guide users through tasks. The task is broken down into steps and presented to the user one step at a time, and the user checks off the steps as he/she completes them. Cheat sheets come in two forms: simple (one task, several steps), and composite (many sub-tasks, each having many steps). Which one you use depends on the size and complexity of the task you want to guide the user through.

Status Handling

Status handling is a facility that allows to introduce a custom way of showing problems in the product to users. A part of the facility is an extension point which allows to plug new status handler specific for the product. This handler associated to the product is intended to present the problem in the useful for users way.

20.1. Welcome

Welcome (also referred to as Intro) support is a set of extension points that control the initial user experience of a product or rich client application. These extension points allow plug-ins to define specialized pages that introduce a platform product to new users. The Intro pages are shown the first time a product is started guiding users into discovering a product's functionality. An Intro 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 extensions define a class which extends the abstract class org.eclipse.ui.part.IntroPart which is responsible for painting the contents of the screen. An IntroPart shares some similarities with views and editors, which are also Workbench parts. 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.

It is possible to write an intro part from scratch but there are advantages to using the customizable intro part described below.

The Customizable Intro Part

The intro part used by the Eclipse Platform and SDK is based on the class org.eclipse.ui.intro.config.CustomizableIntroPart. An intro of this kind will be based on a configuration which can be extended by individual plugins. The configuration uses an xml file contributed using the extension point org.eclipse.ui.intro.config to define the intro content. Individual plugins can extend the configuration using the org.eclipse.ui.intro.configExtension. extension point. The customizable intro part has several benefits:

The content is separated from the presentation.

Individual plugins can extend the intro content using the org.eclipse.ui.intro.configExtension. extension point.

The presentation uses HTML allowing css to be used to set the background, layout the elements and style the text. for

On systems where a browser is unavailable the intro content will be presented using SWT.

Universal Intro

Universal Intro is an intro configuration based on the customizable intro part and is used by the Eclipse Platform and SDK. The Universal Intro configuration provides a standardized set of pages and anchors which can be extended by other plugins. Use of Universal Intro is encouraged because it allows a plugin to contribute intro content to any product that uses Universal Intro. Without Universal Intro it would be necessary to write a config extension for each intro configuration that a plugin might possibly contribute to.

Related Topics

Defining an Intro Part

Using the Customizable Intro Part

Universal Intro

20.1.1. 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.

20.1.1.1. 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:

use the org.eclipse.ui.intro extension point to register an intro part implementation and bind this intro part to your product.

implement the org.eclipse.ui.intro.IIntroPart interface and use this class as the class attribute in the above extension point contribution.

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\_GRADIENT));

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.

20.1.2. Using the CustomizableIntroPart

The platform's CustomizableIntroPart is an intro part which reads intro content from XML files and presents the content using HTML. Cascading Style Sheets (css) can be used to control most aspects of the presentation including background image, font style and layout. The following extensions are required to use CustomizableIntroPart.

org.eclipse.intro to defines an intro id and specifies that it will use the CustomizableIntroPart

org.eclipse.intro to define the product binding which specifies which intro a product will use

org.eclipse.ui.intro.config to defines the intro content for a particular intro id

In addition org.eclipse.ui.intro.configExtension extensions can be used from any plugin to extend the intro content. 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.

Related Topics

Defining intro content

Defining a minimal intro configuration

Defining an intro config

20.1.2.1. 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.

20.1.2.2. 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 Eclipse SDK. The snippet below shows an example of this:

<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</title>

<group id="links-background">

<group id="page-links">

<link label="Overview" url="http://org.eclipse.ui.intro/showPage?id=overview" id="overview" style-id="left">

<text>Find out what Eclipse is all about</text>

</link>

<link label="Tutorials" url="http://org.eclipse.ui.intro/showPage?id=tutorials" id="tutorials" style-id="left">

<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" style-id="right">

<text>Explore Eclipse development through code samples</text>

</link>

<link label="Whats New" url="http://org.eclipse.ui.intro/showPage?id=news" id="news" style-id="right">

<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.

20.1.2.3. Defining a minimal intro configuration

The first stage in creating a new intro configuration is to add the necessary extension points and create a minimal amount of content. This stage is often the hardest. Below are the steps required to create a minimal intro configuration and have it show up in a product.

Create a plugin project and add extension points

Create an new plugin project "org.eclipse.intro.minimal" which has a dependency on org.eclipse.ui.intro. In the plugin manifest editor add the extension org.eclipse.ui.intro. Edit plugin.xml until it looks like this:

<?xml version="1.0" encoding="UTF-8"?>

<?eclipse version="3.2"?>

<plugin>

<extension

point="org.eclipse.ui.intro">

<intro

class="org.eclipse.ui.intro.config.CustomizableIntroPart"

icon="$nl$/icons/image\_obj.gif"

id="org.eclipse.intro.minimal"

label="Minimal Intro"/>

</extension>

<extension

point="org.eclipse.ui.intro.config">

<config

content="$nl$/introContent.xml"

id="org.eclipse.intro.minimal.config"

introId="org.eclipse.intro.minimal">

<presentation

home-page-id="root" standby-page-id="standby">

<implementation

kind="html">

</implementation>

</presentation>

</config>

</extension>

</plugin>

Create the intro config file

Create a file introcontent.xml in the plugin project org.eclipse.intro.minimal with these contents.

<?xml version="1.0" encoding="utf-8" ?>

<introContent>

<!-- Root page -->

<page id="root" style="html/style.css" style-id="page">

<title style-id="intro-header">Minimal Intro</title>

<link url="http://org.eclipse.ui.intro/switchToLaunchBar" label="Go to theWorkbench" id="workbench" />

<link label="Link to tutorials" url="http://org.eclipse.ui.intro/showPage?id=tutorials" id="tutorials" />

</page>

<!-- Standby page -->

<page id="standby" style="html/style.css" style-id="page">

<title style-id="intro-header">Standby Page</title>

</page>

<!-- Tutorials page -->

<page id="tutorials" style= "html/style.css" style-id="page">

<title style-id="intro-header">Tutorials</title>

<text>This page under construction</text>

</page>

</introContent>

Define a product binding

In this step we need to determine the product which is being used and create an intro product binding and modify its plugin.xml file. If you are using the Eclipse SDK this will be in the plugin project org.eclipse.sdk which can be found in the plugins directory of your Eclipse installation. The following directions assume you are using the Eclipse SDK, with slight modifications these will work equally well for an RCP or other Eclipse based product. Open plugin.xml and locate the following section:

<extension

point="org.eclipse.ui.intro">

<introProductBinding

introId="org.eclipse.ui.intro.universal"

productId="org.eclipse.sdk.ide">

</introProductBinding>

</extension>

Change the introId to "org.eclipse.intro.minimal" so that the extension code looks like this.

<extension

point="org.eclipse.ui.intro">

<introProductBinding

introId="org.eclipse.intro.minimal"

productId="org.eclipse.sdk.ide">

</introProductBinding>

</extension>

Test and Customize

Launch a new Eclipse application from your workbench. Your intro screen may already be showing (depending on the state when Eclipse last exited). If it is not showing select Help/Minimal Intro from the workbench menu. By editing the file html/style.css you can modify the appearance of your intro. You can now extend your intro either by editing the intro.xml file or by adding config extensions.

20.1.2.4. Using XHTML as intro content

Depending on the usage scenario of the intro framework, XHTML files can be contributed as intro content. The 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 are 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.

20.1.2.5. 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 simple example, and to contribute static content we will use the following as content:

<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.

20.1.2.6. Extending an intro config

An intro configuration can be extended in several ways:

Themes can be contributed to define the overall appearance.

The content of an existing intro config can be extended.

A custom standby content part, such as a Cheat Sheet, can be contributed to provide content for the standby area of the Intro part.

Custom IntroURL actions can be defined.

20.1.2.6.1. 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 tool platform - an open extensible IDE for anything and nothing in particular. It provides a feature-rich development environment that allows the developer to efficiently create tools that integrate seamlessly into the Eclipse Platform.</text>

<group id="overview-links">

<link label="Workbench basics" url="http://org.eclipse.ui.intro/showHelpTopic?id=/org.eclipse.platform.doc.user/concepts/concepts-2.htm" id="basics">

<text>Learn about basic Eclipse workbench concepts</text>

</link>

<link label="Team support" url="http://org.eclipse.ui.intro/showHelpTopic?id=/org.eclipse.platform.doc.user/concepts/concepts-26.htm" id="team">

<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/overview.css"

path="overview/page-content/overview-links/jdtAnchor">

<link label="Java development" url="http://org.eclipse.ui.intro/showHelpTopic?id=/org.eclipse.jdt.doc.user/gettingStarted/qs-BasicTutorial.htm" id="java">

<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.

20.1.2.6.2. Defining intro themes

A theme defines the overall appearance of the intro. It is simply a way of grouping all the presentation files (styles and images) in one place that can be pointed at or switched as one.

Definition

To define a theme, extend your intro configuration using the org.eclipse.ui.intro.configExtension and use the theme element, as shown in the example below:

<extension

point="org.eclipse.ui.intro.configExtension">

<theme

default="true"

id="org.eclipse.ui.intro.universal.circles"

name="%theme.name.circles"

path="$nl$/themes/circles"

previewImage="themes/circles/preview.png">

<property

name="launchbarBackground"

value="#a1c2cb"/>

<property

name="launchbarOverviewIcon"

value="$theme$graphics/launchbar/overview16.png"/>

<property

name="launchbarFirststepsIcon"

value="$theme$graphics/launchbar/firststeps16.png"/>

<property

name="launchbarTutorialsIcon"

value="$theme$graphics/launchbar/tutorials16.png"/>

<property

name="launchbarSamplesIcon"

value="$theme$graphics/launchbar/samples16.png"/>

<property

name="launchbarWhatsnewIcon"

value="$theme$graphics/launchbar/whatsnew16.png"/>

<property

name="launchbarMigrateIcon"

value="$theme$graphics/launchbar/migrate16.png"/>

<property

name="launchbarWebresourcesIcon"

value="$theme$graphics/launchbar/webresources16.png"/>

</theme>

</extension>

Consult the schema documentation for details on all the available attributes relating to themes.

Each theme has a unique identifier, translatable name, preview image, and a path to the root theme folder. The intro plug-in does not provide any UI for theme manipulation. The only way to select a theme is via the preference org.eclipse.ui.intro/INTRO\_THEME in plugin\_customization.ini.

Enabling themes

Theme support by itself does not make intro implementations theme-enabled. Concrete intro implementations can choose to expose themes in a more substantial way (and in fact Universal Welcome implementation does exactly that with the General > Welcome preference page.

Theme-enabled intro implementation must make all the references to style and presentation resources using the $theme$ substitution variable. Absolute paths for images, pages, styles, etc. will be computed by resolving the substitution variable using the path of the currently active theme.

20.1.2.6.3. 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&input=org.eclipse.pde.helloworld. 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.

20.1.2.6.4. 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&amp;class=org.eclipse.pde.ui.internal.samples.ShowSampleAction&amp;id=org.eclipse.sdk.samples.swt.examples

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&amp;class=org.eclipse.pde.ui.internal.samples.ShowSampleAction">

</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&amp;class=org.eclipse.pde.ui.internal.samples.ShowSampleAction&amp;id=org.eclipse.sdk.samples.swt.examples

20.1.2.6.5. Intro Content File XML Format

Identifier:

Intro Content File XML Format..

Description:

This document describes the intro content file structure as a series of DTD fragments.

Configuration Markup:

<!ELEMENT introContent ((page | group)+ | (extensionContent | replacementContent))>

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, extensions to anchor points defined in other configurations, or replacements of existing elements.

<!ELEMENT page (group | link | text | head | img | include | html | title | anchor | contentProvider)+>

<!ATTLIST page

id CDATA #REQUIRED

url CDATA #IMPLIED

style CDATA #IMPLIED

alt-style CDATA #IMPLIED

filteredFrom (swt|html)

content CDATA #IMPLIED

style-id CDATA #IMPLIED

shared-style (true | false)

bgImage 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 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.

id - A unique name that can be used to identify this page.

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.

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.

bgImage - an optional URL of the image to use as a background for this group

<!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)

computed (true | false) "false"

bgImage CDATA #IMPLIED

expandable (true | false) "false"

expanded (true | false) "false">

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.

computed - if true, the children of this group will be provided by the intro configurer at run time. It is advised not to define any children statically in this case.

bgImage - an optional URL of the image to use as a background for this group

expandable - specifies whether or not the group can be expanded and collapsed. Defaults to false if not specified. If true, the expanded attribute will be consulted to determine whether or not the group should be expanded by default or not.

expanded - specifies whether the expandable group should be expanded by default or not. Defaults to false (collapsed) if not specified. This only applies to expandable groups (where expandable was set to true). Has no effect otherwise.

<!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 srcipts or extra styles sheets. If the content contains substitution segments of the form $plugin:plugin\_id$ then they will be replaced with the absolute path to the plugin with id plugin\_id. 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. If the content contains substitution segments of the form $plugin:plugin\_id$ then they will be replaced with the absolute path to the plugin with id plugin\_id.

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

url CDATA #REQUIRED

id CDATA #IMPLIED

label CDATA #IMPLIED

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.

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

execute - executes the specified command. See the serialize() method on org.eclipse.core.command.ParameterizedCommand for details of the command serialization format. Since 3.2.

command - a serialized ParameterizedCommand

standby (optional) = ("true" | "false") "false" - indicate whether to set the intro into standby mode after executing the command

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 receive 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&amp;decode=true

will process the message parameter as "This is a message".

id - A unique id that can be used to identify this link

label - The text name of this link

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>

<!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 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.

<!ELEMENT img EMPTY>

<!ATTLIST img

src CDATA #REQUIRED

id CDATA #IMPLIED

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).

src - the file to load the image from

id - unique identifier of this image

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. If the content contains substitution segments of the form $plugin:plugin\_id$ then they will be replaced with the absolute path to the plugin with id plugin\_id.

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. If the content contains substitution segments of the form $plugin:plugin\_id$ then they will be replaced with the absolute path to the plugin with id plugin\_id.

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

path CDATA #REQUIRED

configId CDATA #IMPLIED

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.

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.

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.

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

path CDATA #REQUIRED

style CDATA #IMPLIED

alt-style CDATA #IMPLIED

content CDATA #IMPLIED

id CDATA #IMPLIED

name CDATA #IMPLIED>

The content to be added to the target anchor. Only one extensionContent or replacementContent 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 could not be found) then the pages and/or groups in the extension need to be ingnored.

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.

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.

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

id - (since 3.2) a unique identifier of this extension required when used in conjunction with intro configurer.

name - (since 3.2) a translatable name of this extension required when used in conjunction with intro configurer

<!ELEMENT replacementContent (text | group | link | html | include)+>

<!ATTLIST replacementContent

path CDATA #REQUIRED

style CDATA #IMPLIED

alt-style CDATA #IMPLIED

content CDATA #IMPLIED>

(since 3.3) The content to replace the target element. Only one extensionContent or replacementContent is allowed in a given configExtension because if this extension could not be resolved (if the config could not be found, or the target element could not be found) then the pages and/or groups in the extension need to be ignored.

path - the path that uniquely represents the path to the element to be replaced. (e.g. page/group1/group2/elementId) within the target configuration to be extended. To replace content contributed from an extensionContent, you may use a path of the form pageId/@extension\_id/path\_in\_extension and it will be resolved to the specified extension.

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.

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

class CDATA #REQUIRED

pluginId CDATA #IMPLIED>

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.

class - A class that implements the IContentProvider interface

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.

<!ELEMENT hr EMPTY>

<!ATTLIST hr

id CDATA #IMPLIED

style-id CDATA #IMPLIED

filteredFrom (swt|html) >

a horizontal rule.

id - unique identifier of this hr

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.

Copyright (c) 2004, 2006 IBM Corporation and others.

This program and the accompanying materials are made available under the terms of the Eclipse Public License 2.0 which accompanies this distribution, and is available at https://www.eclipse.org/legal/epl-2.0/ SPDX-License-Identifier: EPL-2.0

20.1.2.6.6. SWT Properties for Intro

The properties below can be used to affect the appearance of intro on systems which do not support the browser widget and use the SWT based Intro presentation. They have no effect on the HTML presentation, which is the default presentation for most systems.

! Alternative style file for SWT presentation

! Theme property should be used to indicate that the file is part

! of the intro theme (new in 3.2). When in theme, relative images are resolved

! relative to the property file location. Otherwise (false),

! images are resolved relative to the contributing bundle. This

! is the backward-compatible behavior.

theme = true/false

! General properties that can only be specified in a shared style

! file because they apply to the whole presentation.

! Background color of all presentation

! ------------------------------------

bg = #rrggbb hex

! Color for presentation title

! -----------------------------

title.fg = #rrggbb hex

! Backgournd Image for presentation title

! ----------------------------------------

title.image = <file name>

title.image.repeat = true/false

! Flag to display the Home Page using custom layout. Default is true.

! ----------------------------------------

home-page-custom-layout = true/false

! Home Page settings

! ------------------

! The home page layout supports all customization that is supported by a Page,

! except what is specified under Subtitle, Description and Fonts.

! In addition, the following customizations applies only to home page:

! Color for home page link description (default is UI forms TITLE color, which is

! a system color)

! -------------------------------------

<homePageId>.hover-text.fg = #rrggbb hex

! Default icons for links in the navigation bar which is added to each page

! -------------------------------------------------------------------------

<homePageId>.small-link-icon = <relative file name>

<homePageId>.small-hover-icon = <relative file name>

! Icon for a given link in the Home Page, and its corresponding link in

! the navigation bar

! ----------------------------------------------------------------------

<homePageId>.<linkId>.link-icon = <relative file name>

<homePageId>.<linkId>.hover-icon = <relative file name>

<homePageId>.<linkId>.small-link-icon = <relative file name>

<homePageId>.<linkId>.small-hover-icon = <relative file name>

! Page settings

! -------------

! Note: in any of the page settings, if the <pageId> is omitted and the

! property starts with a ".", then this property applies to this whole page,

! and to any page that inherits the properties of this page. So in effect,

! this property becomes like a shared property.

!

! A short form of the property that uses element id can be used:

! <pageId>.<elementId>.property = value

! Use this for elements that do not have fixed path (when late target

! path resolution is used).

! Flag to display link description in a given page. Default is true.

<pageId>.show-link-description = true/false

.show-link-description = true/false

! Flag to display Root page navigation links in a given page. Default is true.

<pageId>.show-home-page-navigation = true/false

.show-home-page-navigation = true/false

! Layout:

! -------

! Number of columns in this page or group.

<pageId>.layout.ncolumns = <integer>

<pageId>.<path\_to\_group>.layout.ncolumns = <integer>

! Equal width of columns

<pageId>.<path\_to\_group>.layout.equalWidth = true/false

! vertical/horizantal spacing of all objects in a given page or group.

<pageId>.<path\_to\_group>.layout.vspacing = <integer>

<pageId>.<path\_to\_group>.layout.hspacing = <integer>

! Number of columns/rows an element spans in its group or page.

<pageId>.<path\_to\_element>.layout.colspan = <integer>

<pageId>.<path\_to\_element>.layout.rowspan = <integer>

! Icons:

! ------

! default icon/hover icon used for all links in the page.

<pageId>.link-icon = <relative file name>

<pageId>.hover-icon = <relative file name>

! icon/hover icon for specific link/Image link.

<pageId>.<path\_to\_link>.link-icon = <relative file name>

<pageId>.<path\_to\_link>.hover-icon = <relative file name>

! Sub Title:

! ----------

! The path of the child Text element that will be used as subtitle of this page.

! (Can only be a child Text element)

<pageId>.subtitle-id = <path to subtitle child>

! The style id of the direct child Text element that will be used as subtitle of any page.

! It would be used if a specific id has not been specified at the page.

! (Can only be a child Text element)

subtitle-style-id = <style id of child subtitle Text element>

! Description:

! ------------

! The path of the child Text element that will be used as description of this page.

! (Can only be a child Text element)

<pageId>.description-id = <path to child description Text element>

! The path of the child Text element that will be used as description of this group.

! (Can only be a child Text element)

<pageId>.<path\_to\_group>.description-id = <path to child description Text element>

! The style id of the direct child Text element that will be used as description of any

! page or any group. It would be used if a specific id has not been specified at the page

! or group level. (Can only be a child Text element)

description-style-id = <style id of child description Text element>

! Page elements Font:

! -------------------

! Set the color for a specific Text or Group label. If applied to a group the label of

! the group gets the color and not the description text. To color description text,

! set its color explicitly.

! note: it is not recommended to set the color of a link because it changes the

! theme of the link colors.

<pageId>.<path\_to\_element>.font.fg = #rrggbb hex

! make a Text element bold. Note that if the Text string itself has any formatting

! tags like <b> and <li> then this bold property is ignored.

<pageId>.<path\_to\_text>.font.bold = true/false

! The style id that will make any Text element bold. It would be used if that Text element

! does not have a bold property explicitly set.

bold-style-id = <style id of child subtitle Text element>

! Color for separator elements:

<pageId>.separator.fg = #rrggbb hex

20.1.3. Universal intro

What is the universal intro?

The universal intro is complete, ready out-of-the-box implementation of intro with a pre-defined page structure that all products can rely on (as API). It is a complete implementation, including presentation, but without content, and can be used by any product.

Motivation

Prior to 3.2, the intro framework provided for defining welcome pages with unique contribution points or 'anchors'. Other components could provide extensions that add content into these anchors. Although this mechanism worked well for closed solutions, it is increasingly hard to maintain for large products. Due to the 'bottom-up' nature of contributions from clients into the welcome pages, it is hard to control the final result and resolve conflicting contributions.

Take, for example, the provider of a reusable eclipse feature. He/she has created a useful feature consisting of a number of plug-ins. The feature can take part in several products, each one having its own Welcome implementation. There are several problems that the feature provider will face when creating Welcome contribution:

Since it must contribute into a number of different products, the feature must know about the page identifiers for each of the products upfront.

It is entirely possible that one of the products will have a Welcome implementation that is completely different from what the feature provider expects; the contribution will not show up in these products.

Even if the products share a common Welcome implementation, the importance of the feature (and hence its relative position on the page) is not going to be the same. The feature can be one of the most important components in the product, or may be ranked as 'See Also'. Nevertheless, the feature provider is asked to fully spell out the location on the target page in advance.

It is clear that the Welcome contribution of a reusable feature is be much easier to develop with the following conditions in place:

There is a common Welcome implementation that can be counted on (as a form of API)

Delayed target page resolution (so that the decision of the final position of the contribution is left to the product assembler)

Enter universal intro

The universal intro is based on experience with intro implementations and the need to have an implementation in place that contributors can count on. The implementation is built using the existing intro framework, and for this reason, all existing intro implementation will continue to work and clients will be able to switch to the new model according to their schedules. Of course, as long as they don't switch, they will not reap the benefits of the new implementation such as intelligent content merging, new visual solution etc.

Benefits

Some of the benefits of using the universal intro include:

Products can simply to point to the universal intro without the need to implement an intro from scratch

Each product should be able to configure the intro to a certain degree (title, branding image, presentation theme)

A number of root pages that cover most of the products' needs are be provided. The list of pages to show is also configurable

The presentation is sufficiently product-agnostic to better suit the increasingly componentized world. It works well with both monolithic products and 'best of breed' eclipse products assembled with components created by different companies

End-users can configure the universal intro in a user-friendly way, via the preference page

20.1.3.1. Contributing to universal intro

In order to hook into to the universal intro, developers should have a product-based eclipse application. This requires the use of the org.eclipse.runtime.products extension point (the assumption is that the product has already been defined):

<extension

id="foo"

point="org.eclipse.core.runtime.products">

<product

application="org.eclipse.ui.ide.workbench"

description="Product Foo to use for testing the universal intro"

name="Product Foo">

</product>

</extension>

For this example, assume that the extension is defined in the plugin.xml file in the com.example.intro plug-in. A long-standing feature of the eclipse intro support is the ability to hook products and intros using the product-intro binding:

<extension

point="org.eclipse.ui.intro">

<introProductBinding

introId="org.eclipse.ui.intro.universal"

productId="com.example.intro.foo"/>

</extension>

The extension above binds the universal intro implementation (org.eclipse.ui.intro.universal) and our product id.

Universal intro customization is split between product branding properties and preferences. Product branding properties are set by the product and cannot be modified. They include product title, branding image and branding image text:

<product

application="org.eclipse.ui.ide.workbench"

description="Product Foo to use for testing the universal intro"

name="Product Foo">

<property

name="introTitle"

value="Welcome to Product Bar"/>

<property

name="introBrandingImage"

value="product:eclipse.png"/>

<property

name="introBrandingImageText"

value="XYZ Company"/>

</product>

Product properties whose values represent a file name relative to the product bundle must be qualified with the 'product:' prefix. The following properties are supported:

introTitle - the value of the property will be used at the top of the root page (assuming that the current presentation theme elected to show the root page title)

introBrandingImage - the value of the property represents the file name of the image to be used as the root page brand mark. The image should ideally be PNG with alpha blending but other formats are also valid. Image file names relative to the product bundle must have a 'product:' prefix.

introBrandingImageText - an alternative text to be used for the branding image.

introDescription-<pageId> - an option description text that appears below the title on each of the sections, where pageId is one of the values listed further below (e.g. introDescription-overview, introDescription-samples etc.). This value should be translated in plugin.properties file.

As of version 4.6, the universal intro supports replacing file-based properties such as introBrandingImage with a theme-specific alternative. For example, a product may contribute a introBrandingImage as "product:gizmo.png", but this image does not match the the com.acme.theme theme. A more suitable image can be placed in the product bundle in com.acme.theme/gizmo.png. When a replacement image is not found, the resolver will fall back to the general image gizmo.png image found at the root of the product bundle.

Second half of universal intro variables are accessible as preferences. The split is due to the fact that these variables can be configured by users and are exposed in the new Welcome preference page. The initial values for these preferences should be placed in the 'plugin\_customization.ini' file that is referenced from the product extension. These preferences are:

org.eclipse.ui.intro.universal/INTRO\_ROOT\_PAGES - a comma-separated list of root page identifiers that should be visible in the home page. Valid values are:

overview

firststeps

tutorials

samples

whatsnew

migrate

webresources

org.eclipse.ui.intro.universal/INTRO\_DATA - a file name pointing at the XML file with the page layout settings (see more about this below). The file name relative to the product bundle must have 'product:' prefix.

org.eclipse.ui.intro/INTRO\_THEME - a unique identifier of the presentation theme to be used for this product. Note that this preference comes from 'org.eclipse.ui.intro plug-in' because active theme selection is performed at the framework level.

org.eclipse.ui.intro/INTRO\_START\_PAGE - the id of the page which will be shown when Eclipse starts the first time.

org.eclipse.ui.intro/INTRO\_HOME\_PAGE - the id of the page which will be shown when the home button is pressed.

org.eclipse.ui.intro/INTRO\_STANDBY\_PAGE - the id of the page which will be shown when welcome is displayed in a non-maximized form.

When using universal intro valid values for the three page related properties are are root, overview, firststeps, tutorials, samples, whatsnew, migrate or webresources or the name of a page contributed using a config extension.

20.1.3.2. Adding the preference page

The combination of product properties and default preference values can fully configure the Universal Welcome if no further customization is desired. For products that want to allow users to customize Welcome, a preference page is available. The following code should be added to the product's plugin.xml:

<extension

point="org.eclipse.ui.preferencePages">

<page

category="org.eclipse.ui.preferencePages.Workbench"

class="org.eclipse.ui.intro.universal.ExtensionFactory:welcomeCustomization"

id="com.example.intro.introCustomization"

name="%introCustomizationPreference.name">

<keywordReference id="org.eclipse.ui.ide.appearance"/>

</page>

</extension>

This code will add the Welcome customization preference page. The page allows users to select the presentation theme. The original choice is provided by the theme preference in the plugin\_customization.ini file. In addition, users can choose from the list of available root pages. Checking the root page causes the related tab to appear at the top of the dialog.

When saved, this preference page will prefix the variables with the product id so that it does not interfere with the settings made for other products in the same workbench. Alternatively, selecting the checkbox above will not prefix the variables, making the stored settings visible to all the products.

20.1.3.3. Extending the universal intro

When launching the universal Welcome configured as shown above, you will notice that all of the second level page that branch off the root page have an empty content area. This is because shared Welcome does not have content of its own. These areas are populated using intro config extensions.

The traditional way of contributing config extensions is to specify a full path of the anchor element in the target page. This method is still supported but it has proven to be somewhat fragile. As mentioned before, specifying an exact place where the extension will end up in the target is premature for extension authors. They may not know all the products in which their extension will end up, and therefore don't know where they should appear according to each product's focus. In the universal Welcome, this decision is left to product authors. Extension authors now have the option of specifying an incomplete target path, allowing the product author to fill in the blanks.

Config extensions that want to use late target path resolution need to fulfill three conditions:

Config extension needs to specify a unique identifier using the 'id' attribute. Extensions without the id will be ignored. This is important because id will be used to refer to the extension by the product author.

Config extension needs to specify a user-friendly name using the 'name' attribute. This is important if the customization preference page will be used because extensions will show up in the UI.

The target path should have the following form: "page\_id/@" where 'page\_id' is the identifier of the target page. When '/@' is detected in the target path, an attempt will be made to dynamically resolve the path into the expected full form.

The ability to resolve extension target paths is a new intro support feature in Eclipse 3.2 added to make universal Welcome more powerful. The feature itself is generic in that it can be used in other Welcome implementations, not just the universal one. In Eclipse 3.2, CustomizableIntroPart can accept an optional intro customizer class that can affect its behavior in several ways. One of the roles of the customizer is to perform late target path resolution. Of course, as far as product authors are concerned, this is all just implementation detail because in universal Welcome implementation, late target path resolution is performed using the data file mentioned earlier. This file is stored using the org.eclipse.ui.intro/INTRO\_DATA preference.

The 'INTRO\_DATA' file uses XML format and allows product authors to control the content of the main Welcome pages. Welcome contribution authors are required to specify only the page Id in their target paths. The rest is defined in this file. The file contains a sequence of 'page' elements, each containing a number of 'group' elements. Group elements specify page-relative path and have contributions for children. Contribution elements are used to specify two aspects:

<extensions>

<page id="overview">

<group path="page-content/top-left">

<extension id="foo1" importance="high"/>

<extension id="bar" importance="high"/>

<extension id="foo2" importance="medium"/>

</group>

<group path="page-content/top-right">

<extension id="foo3" importance="low"/>

<extension id="foo4" importance="low"/>

<extension id="foo5" importance="callout"/>

</group>

<hidden>

<extension id="foo6"/>

<extension id="foo7"/>

</hidden>

</page>

<page id="whatsnew">

...

</page>

</extensions>

In the example above, contributions 'foo1', 'bar' and foo2' will be in the left, and 'foo3', 'foo4' and 'foo5' in the right column on the page. The relative order of contributions is also extracted from this file. In addition, contributions are classified based on the importance attribute that can have four valid values: high, medium, low and callout, with low as the default. Each value has matching presentation that makes it stand out on the page. First three are simply different levels of importance according to the product author (note that the same contribution may receive a different importance classification in two separate products). The last one (callout) is used to single out contributions that are of a completely different nature (for example, a contribution that offers links to videos or animation).

Contributions that are not listed are appended after the listed contributions and assigned a low importance value. This is important for contributions added after the product has shipped - they still need to show up.

Contributions not relevant to the project can be hidden by explicitly listing them as children of the 'hidden' element.

Contributing into extensions

An extension contributed into one of the root pages can itself contain anchors, providing for others to add content. This causes a problem because the final resolved path of the content in the extension is not known in advance. For this reason, target paths for content in extensions that use late path resolution must itself be resolved:

<?xml version="1.0" encoding="utf-8" ?>

<introContent>

<extensionContent id="extra" name="Extra" alt-style="css/swt.properties" style="css/overview.css" path="overview/@">

<group id="extra-group" style-id="content-group">

<link label="Extra Overview link" url="http://org.eclipse.ui.intro/showPage?id=extraOverview"

id="extra-overview">

<text>Showing the third-level extra page for overview</text>

</link>

<anchor id="additions"/>

</group>

</extensionContent>

</introContent>

In order to contribute into the anchor 'additions' in the extension above, we should use the following path: "overview/@extra/extra-group/additions". The segment '@extra' will be replaced with the resolved path of the extension with the id 'extra' in the 'overview' page. For example, if the extension is placed in the upper left segment of the page, the resolved path will be: "overview/page-content/upper-left/extra-group/additions".

20.1.3.4. Configuring product defaults

In addition to providing intro customization to the end users for the first time, Welcome preference page will allow product authors to create the introData.xml file to be used for the first startup. The procedure is simple:

Launch the product for the first time

Open the Welcome preference page (a quick way to do this is from the local tool bar of the Welcome view where a new tool bar button opens the Preferences into the right page)

For each page, manipulate the available extensions by moving them between the four page quadrants (upper-left, upper-right, lower-left, lower-right) and the Available list, or moving extensions up or down within the same list (relative order). Use drag and drop to get the desired results.

If needed, add the separator between the extension within the quadrant.

All extensions have the default importance setting of 'Low'. To change it, click on the extension image to get the drop-down list to open and pick a different setting from the list (e.g. High, Medium or Callout).

Press OK to see the preferences applied to Welcome.

If satisfied, open the preference page again and press 'Save As...' button.

Store the file as 'introData.xml' in the product plug-in.

Set org.eclipse.ui.intro.universal/INTRO\_DATA preference to the value 'product:introData.xml'.

Start the product on a fresh workspace to test if the data from the file has been picked up.

List 'Available Extensions' holds all the extensions that are currently hidden. They can be moved to one of the four quadrants of each page.

20.1.3.5. Managing links

Most of the extensions coming into the main Welcome pages offer links to open further pages contributed by the extensions themselves. It is recommended to create these pages in such a way as to continue with the shared Welcome visual design. The main requirement is to show the appropriate navigator at the top of the page.

It is fairly easy to do that using the 'include' element. The idea is to include the navigation content at the beginning of contributed page. Unlike the current Welcome implementation, shared Welcome main pages have navigation section that is different for each page. For this reason, pages that branch off these pages should include the navigation from the parent page. For example:

<introContent>

<extensionContent id="extra" name="Extra" alt-style="css/swt.properties" style="css/overview.css" path="overview/@">

<link

label="Extra Overview link"

url="http://org.eclipse.ui.intro/showPage?id=extraOverview"

id="extra-overview">

<text>Showing the third-level extra page for overview</text>

</link>

</extensionContent>

<page id="extraOverview" style="$theme$/html/overview.css" style-id="page">

<title style-id="intro-header">$introTitle$</title>

<group id="extra-group1" filteredFrom="swt"/>

<include path="overview/navigation-links"/>

<group id="page-content">

<group id="content-header" label="EXTRA OVERVIEW" filteredFrom="swt">

</group>

<text style-id="page-title" id="page-title" filteredFrom="html">EXTRA OVERVIEW</text>

<text style-id="page-description" id="page-description">Extra overview page description.</text>

<!-- Add content here -->

</group>

</page>

</introContent>

The extension above contributed into the overview page contributes a link that shows another page whose definition is part of the extension. Note how we define the title element using substitution variable (the variable will be resolved based on the product property). We also include the group 'navigation-links' that belongs to the 'overview' page at the beginning of the page content definition. We also add the 'extra-group1' div because it is can be used by some themes for adding additional page decoration (indeed, it is used in 'Purple Mesh' theme for that exact purpose).

20.2. 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.

20.2.1. Configuration/setup

This section contains topics related to installing, configuring, and customizing the help system to work in different environments and products.

Help modes

The help system can run in three modes: workbench (normal), information center, and standalone. Workbench mode is used for serving help integrated with the product, usually via a Help menu. This mode also offers context help and the help view, which are not available in the two other modes. Standalone mode has the same goal as workbench mode, but is for products that are not eclipse-based (the help will be less integrated). Information center mode is used to serve help content to the masses over the Web. Consult the links above for details.

Rich Client Platform (RCP) support

The Help system is an optional Rich Client Platform (RCP) component. That is, it is not part of the minimal RCP, but can be added to it to provide help in your RCP application.

Product customization

Help provides preferences that products can use to customize the help system to their needs. These include ways to customize branding, appearance, turn on/off functionality, etc.

Pre-indexing

For products that offer a large number of help documents, the initial indexing phase that occurs when searching for the first time can be lengthy. To avoid this delay, you can pre-index your help contents and ship the pre-built index along with your documentation. This is a trade-off between performance (avoiding the indexing phase) and space (to store the index).

20.2.1.1. Rich Client Platform (RCP) help

The Help system is an optional Rich Client Platform (RCP) component. That is, it is not part of the minimal RCP, but can be added to it. The following plug-ins and all of their dependencies, including optional dependencies must be added to the RCP base in order to run the Help system:

org.eclipse.help.ui

org.eclipse.help.webapp

Note: Help contents will only be made available to the user if there is content available. If there are no topics in the table of contents, the help will not be available.

Setting the Help Server Port and Hostname

The help server port and host name can be specified using virtual machine arguments as follows:

-Dserver\_port=<port number> specifies the port number that the help server will use.

-Dserver\_host=<host name> specifies the host name which will be used when the help system creates URLs which reference help pages.

20.2.1.2. Standalone help

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 workbench-integrated context help, the help view, 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 the information center 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.

Download the eclipse Platform Runtime Binary driver from eclipse.org.

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

Make sure d:\myApp\eclipse\plugins\org.eclipse.help.base\_[version].jar is on your classpath, where [version] is the version of the plugin you're using (e.g. org.eclipse.help.base\_3.2.0.jar). The class you use to start, launch, and shut down the help system is org.eclipse.help.standalone.Help.

Create an array of String objects 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" };

In your application, create an instance of the Help class by passing in the options. This object should be held onto until the end of your application.

Help helpSystem = new Help(options);

To start the help system:

helpSystem.start();

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");

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.

At the end of your application, to shutdown the help system:

helpSystem.shutdown();

How to call the help from the command line

The org.eclipse.help.standalone.Help class has a main method you can use to launch stand-alone help from the command line. The command line arguments syntax is:

-command start | shutdown | ((displayHelp | displayHelpWindow) [href]) [-eclipsehome eclipseInstallPath] [-data instanceArea] [-host helpServerHost] [-port helpServerPort] [-dir rtl] [platform options] [-vmargs JavaVMarguments]

The command start starts a headless help server. shutdown shuts down the server. The command displayHelp starts the server and causes the help UI to display in an Eclipse shell. displayHelpWindow is similar to displayHelp, the difference being that the help server will shut down when the shell is closed.

A simple way to display help is to invoke

java -classpath d:\myApp\eclipse\plugins\org.eclipse.help.base\_[version].jar org.eclipse.help.standalone.Help -command displayHelp

from within d:\myApp\eclipse directory, where version is the plug-in's version. To display a specific TOC file or topic use

java -classpath d:\myApp\eclipse\plugins\org.eclipse.help.base\_[version].jar org.eclipse.help.standalone.Help -command displayHelp /com.mycompany.mytool.doc/tasks/task1.htm

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\_[version].jar org.eclipse.help.standalone.Help -command start

[Optional] Installing a minimal stand-alone help system

The stand-alone help does not require the entire eclipse Platform package. It is possible to run the stand-alone help using only those plugins from the feature org.eclipse.help. To do this perform the following steps.

Download an eclipse SDK build and upzip it into two different locations, <location1> and <location1>

Remove the eclipse plugins directory from location1

Start Eclipse in location2 and import the org.eclipse.help and org.eclipse.rcp features into your workspace using File/Import/Plug-in Development/Features

Export the help and rcp features and their plugins to a newly created directory location3 using File/Export/Plug-in Development/Deployable Features.

Replace the plugins and features directories under the location1/eclipse directory with those from location3.

copy org.eclipse.equinox.launcher.\* from location2/eclipse/plugins to location1/eclipse/plugins.

From location1 start standalone help from the command line as described in the previous section.

Some documentation plug-ins may have dependencies on other plug-ins, usually by specifying required plug-ins in their manifest. The dependent plug-ins need to be installed as well.

See Product customization for more information on customizing the help system.

20.2.1.3. Information Center

You can allow your users to access the help system over the Internet or an intranet by installing an information center 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 information center help system can be used for both client and web applications, either of which can have their help accessed remotely. All features of help system except context help, active help, command support, and the help view are supported.

The information center help system allows passing number of options that can be used to customize various aspects of the information center. The following options are supported:

-eclipsehome eclipseInstallPath - specifies the Eclipse installation directory. This directory is the parent of the plugins directory and Eclipse executable. The option must be provided when the current directory from which the information center is launched is not the same as the Eclipse installation directory.

-data instanceArea - specifies a path that Eclipse can use to write instance data. The value can be the absolute path of a directory or a path relative to Eclipse installation directory. The option must be provided when Eclipse is installed in a read only location, or has been customized to override the osgi.instance.area or osgi.instance.area.default properties.

-host helpServerHost - specifies the host name of the interface that the help server will use. This overrides the host name specified in the application server plugin preferences.

-port helpServerPort - specifies the port number that the help server will use. This overrides the port number specified in the application server plugin preferences.

-locales localeList - specifies a list of locales that the information center will recognize and provide customized content for. If the option is not specified, the information center will build navigation and index documents for each preferred locale of the browsers accessing the information center. When the option is present, locales from browser requests will be matched with locales in the list. If no browser preferred locale exists in the list, but its language part does, it will be used instead. 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) match 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 the information center to operate in 10 locales. All other locales will receive content from the en locale.

-dir ltr or -dir rtl - forces left-to-right or right-to-left rendering direction of the help UI in the browser for all languages. By default, the direction is determined by the browser locale.

-noexec - indicates that the Eclipse executable should not be used. You need to use this option when running on a platform for which the Eclipse executable is not available.

Additionally, most options accepted by the Eclipse executable are allowed. They are especially useful during debugging and for applying customization to Eclipse. For example, passing options -vmargs -Xmx256M increases memory available to the information center and will allow serving of 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.

Download the Eclipse Platform Runtime Binary driver from eclipse.org.

Install (unzip) the driver in a directory, say 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 information center from command line

The org.eclipse.help.standalone.Infocenter class has a main() method that you can use to launch the information center from the command line. The command line argument syntax is:

-command start | shutdown | [-eclipsehome eclipseInstallPath] [-data instanceArea] [-host helpServerHost] [-locales localeList] [-port helpServerPort] [-dir rtl] [-noexec] [platform options] [-vmargs JavaVMarguments]

To start an information center on port 8081, issue a start command by running

java -classpath d:\myApp\eclipse\plugins\org.eclipse.help.base\_[version].jar org.eclipse.help.standalone.Infocenter -command start -eclipsehome d:\myApp\eclipse -port 8081

To shut down the information center issue a shutdown command by running

java -classpath d:\myApp\eclipse\plugins\org.eclipse.help.base\_[version].jar org.eclipse.help.standalone.Infocenter -command shutdown -eclipsehome d:\myApp\eclipse

Using the information center

Start the information center using the instructions above. Point a web browser to the "/help/index.jsp" starting point for the Web application running on the port specified when starting the information center (e.g. 8081). For example, from the machine on which the information center is installed, this would be http://localhost:8081/help/index.jsp.

How to start and stop an information center from Java

When including an information center as part of another application, it may be more convenient to start it and stop it using Java API calls instead of system commands. If this is the case, follow the steps:

Make sure d:\myApp\eclipse\plugins\org.eclipse.help.base\_[version].jar is on your application's classpath. The class you use to start and shut down the information center is org.eclipse.help.standalone.Infocenter .

Create an array of String objects containing options that you want to pass to the information center. Typically, the eclipsehome and port options are needed.

String[] options = new String[] { "-eclipsehome", "d:\\myApp\\eclipse" , "-port", "8081" };

In your application, create an instance of the Help class by passing in the options.

Infocenter infocenter = new Infocenter(options);

To start the help system:

infocenter.start();

To shut down the information center:

infocenter.shutdown();

Making information center available on the web

Eclipse contains a complete information center and does not require any other server software to run. However, in an unsecure environment like the Internet, it is not recommended to allow direct access by the clients, but instead 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, you can configure a proxy module of the Apache HTTP Server to redirect requests made to http://mycompany.com/myproduct/infocenter to http://internalserver:8081/help that runs an information center. Adding the lines

LoadModule proxy\_module modules/ApacheModuleProxy.dll

ProxyPass /myproduct/infocenter http://internalserver:8081/help

ProxyPassReverse /myproduct/infocenter http://internalserver:8081/help

to the conf/httpd.conf file of Apache server running the mycompany web site accomplishes this.

Some versions of the Apache HTTP server may contain an AddDefaultCharset directive enabled in configuration file. Remove the directive or replace it with

AddDefaultCharset Off

to have browsers display documents using correct character set.

Running multiple instances of an information center

Multiple instances of an information center can be run on a machine from one installation. Each started instance must use its own port and must be provided with a workspace, hence the -port and -data options must be specified. The instances can serve documentation from different sets of plug-ins by providing a valid platform configuration with the -configuration option.

If -configuration is not used and the configuration directory is shared among multiple information center instances with overlapping sets of locales, you must be ensure that all search indexes are created by one information center instance before another instance is started. Indexes are saved in the configuration directory, and write access is not synchronized across information center instances.

Filtering

Filtering support is turned off when running in information center mode, causing all content, including filtered content, to be visible. If you intent to host your documentation in both workbench and information center modes, you should use filters in a way that makes sense even if filtering is turned off.

Some documentation plug-ins may have dependencies on other plug-ins, usually by specifying the required plug-ins in their bundle manifest. The dependent plug-ins need to be installed on the information center as well.

See the Product customization topic for more information on customizing the help system.

20.2.1.4. Help system customization

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.

org.eclipse.help plug-in:

Preference key

Usage

Default

HELP\_DATA

Specifies the path of an XML file containing help data, such as the order in which the table of contents should be displayed, and which books and index contributions should be hidden from the user. The path may either be of the form PLUGINS\_ROOT/ pluginId/path or "path". If the value does not start with PLUGINS\_ROOT the path is resolved relative to the product plugin. For example, "helpData.xml" or "PLUGINS\_ROOT/org.eclipse.platform/helpData.xml". Consult the schema documentation for details.

filterInfocenter

Specifies whether enablement filters should apply to an information center. Enablement filters are always applied to help in workbench and standalone mode.

false

baseTOCS (deprecated; use HELP\_DATA instead)

Toc ordering. Ordered list of help TOC's (books) or TOC categories as they would appear on the bookshelf. All the other TOCS and categories will follow these. Non-present TOCs or categories on this list will be ignored. For TOCs, use the location of the TOC as /pluginId/path/to/toc.xml. For categories, use the category id.

ignoredTOCS (deprecated; use HELP\_DATA instead)

Disabling TOCs. List of help TOCs (books) or TOC categories that will be ignored by the help system. The disabled TOCs/categories will not appear in the list of books, cannot be linked. Topics defined by disabled TOCs will not be available from search. Non-present TOCs or categories in this list will be ignored. For TOCs, use the location of the TOC as /pluginId/path/to/toc.xml. For categories, use the category id.

ignoredIndexes (deprecated; use HELP\_DATA instead)

Disabling keyword indexes. List of help keyword index contributions that will be ignored by the help system. Keywords from the disabled indexes will not appear in the index view. For index XML files, use the path to the file in the form /pluginId/path/to/index.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

footer

Location of the footer page to display in the bottom frame

Example: footer=/org.eclipse.help.webapp/advanced/banner.html

footer\_height

Height of the footer frame

Example: footer\_height=60

titleResource

The name of a project resource of the form <plugin\_name>/<resource\_name> which contains the title of the help page.

Example: titleResource=org.eclipse.help.webapp/help\_webapp\_plugin\_name

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\_home.html

page\_not\_found

The page to show in the content area when a topic file cannot be opened. If this variable is not specified the browser will show its default 404 error page.

/org.eclipse.help.base/doc/page\_not\_found.html

showBreadcrumbs

Set to true or false to control the visibility of breadcrumbs.

true

bookmarksView

Set to true or false to control the visibility of the bookmarks view.

Note: this option has no effect in the information center.

true

indexView

Set to true or false to control the visibility of the index view.

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

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. May be the empty string.

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

advanced.syncDefault

The default boolean value for whether the toc should automatically be synchronized with the contents.

true

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 information center 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 information center.

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 on the browser available on a given system

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" %1 - on Windows,

"konqueror %1" - on Linux

"mozilla %1" - on other platforms

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 information center setup, where active help is disabled.

Accepted values:

true - default active help actions enabled

false - active help framework disabled

true

restrictTopicParameter

Since 3.4. When true prevents topic parameters with an http or other protocol from causing an external URL to open in the content frame of the information center. This improves security and the recommended setting is true.

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

help\_view\_open\_mode

Controls where links in the help view are opened.

Accepted values:

in\_place - open documents in the help view

in\_editor - open document in an editor

in\_browser - open document in a browser

in\_place

showSearchDescription

If true descriptions are shown with search results.

true

showSearchCategories

If true search results are organized by book.

false

remoteHelpOn

Controls whether or not remote help is enabled.

Accepted values:

true - remote help enabled

false - remote help disabled

false

The 6 remote help preferences below can each have multiple comma separated values if there is more than one information center contributing remote content.

remoteHelpName

Specifies a name for this remote host which will appear in the table on the preference page

remoteHelpHost

Specifies the host name to access for remote help content. This must be specified as a host name and cannot be a URL (i.e. do not include "http://").

remoteHelpPath

Specifies the context root of the information center application running on the specified host.

remoteHelpProtocol

Specifies the protocol of the information center application running on the specified host. If no protocol is specified, http will be used. Supported protocols are http and https.

remoteHelpPort

Specifies the port to use to access remote help content.

remoteHelpICEnabled

Specifies that the information center will contribute remote content

The five css preferences allow for the control of page appearance by inserting css files into every page served by the help server. Each can each contain zero or more comma separated paths of the form /plugin/path. These paths should contain only ASCII characters.

If a path contains ${os} then that will be replaced with the name of the OS, any other parameters of the form ${parameter} are reserved for future use.

topic\_css

A list of css file(s) to include in every non navigation page served by help system.

nav\_css

A list of css file(s) to include in every navigation page served by help system.

/PRODUCT\_PLUGIN/book.css

narrow\_css

A list of css file(s) to include in every page displayed in the help view or help tray .

/PRODUCT\_PLUGIN/narrow\_book.css,/PRODUCT\_PLUGIN/${os}\_narrow\_book.css

remote\_css

A list of css file(s) to include in every page served by help system which originates from a remote information center.

disabled\_css

A list of css file(s) to include in every page served by help system from a capability that is not enabled.

/PRODUCT\_PLUGIN/disabled\_book.css

These two parameters allow for the restriction of the number of print operation.

maxConnections

Maximum number of connections for concurrent print(print selected topic and all subtopics).

Assign a value no greater than Integer.MAX\_VALUE.

Example: maxConnections=20

10

maxTopics

Maximum number of topics allowed for print in one request.

Assign a value no greater than Integer.MAX\_VALUE.

Example: maxTopics=200

500

org.eclipse.help.appserver plug-in:

Deprecated: This plug-in is no longer used by help. See table for alternative usage.

Preference key

Usage

Default

port (deprecated)

The port number on which the sever listens for http requests. If port is not given, an arbitrary port is picked by the system.

Start Eclipse with "-vmargs -Dserver\_port=<port>" instead

host (deprecated)

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 information center), can set this preference to "127.0.0.1" to ensure help server is not exposed to other users on the network.

Start Eclipse with "-vmargs -Dserver\_host=<host>" instead

org.eclipse.tomcat plug-in:

Deprecated: This plug-in is no longer used by help. No alternative commands available.

Preference key

Usage

Default

acceptCount (deprecated)

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 (deprecated)

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 (deprecated)

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

20.2.1.4.1. Help Data

Identifier:

org.eclipse.help.HELP\_DATA

Since:

3.3

Description:

The help data XML file is used by products to control the order of books in the help table of contents, as well whether or not books or keyword index sets should be displayed at all. The file must be referenced in the product's plugin\_customization.ini file using the org.eclipse.help/HELP\_DATA property.

Configuration Markup:

<!ELEMENT extensions (tocOrder? , hidden?)>

The extension data for Help.

<!ELEMENT tocOrder (toc | category)\*>

Specifies the order in which top-level table of contents entries (also called "books") or categories of books should appear in Help. If one of the items listed is not available, it is ignored. If there are items available that are not listed and not hidden, they will be displayed after the ones listed here.

<!ELEMENT toc EMPTY>

<!ATTLIST toc

id CDATA #REQUIRED>

A reference to a top-level table of contents (TOC) entry, also called a "book".

id - The unique identifier for this book. For XML file TOC contributions, this is a path to the file in the form "/<plugin\_id>/<path>/<file>" (e.g., "/org.eclipse.platform.doc.user/toc.xml"). In general, this is the ID of the TocContribution supplied by its originating AbstractTocProvider.

<!ELEMENT category EMPTY>

<!ATTLIST category

id CDATA #REQUIRED>

A reference to a category of top-level table of contents (TOC) entries (books). Categories are implicitly created when a table of contents contribution declares itself to be of that category, for example, by specifying a category attribute for the toc element in the org.eclipse.help.toc extension point.

id - The unique id of the category.

<!ELEMENT hidden (toc | category | index)\*>

Contains a set of help items that should be hidden from the user.

<!ELEMENT index EMPTY>

<!ATTLIST index

id CDATA #REQUIRED>

A reference to a contribution of help index keywords.

id - The unique identifier for this contribution of keywords. For XML file index contributions, this is a path to the file in the form "/<plugin\_id>/<path>/<file>" (e.g., "/org.eclipse.platform.doc.user/index.xml"). In general, this is the ID of the IndexContribution supplied by its originating AbstractIndexProvider.

Examples:

The following example shows how to arrange the following books in the order shown:

Book #1: "Introduction to XYZ" (category: "user.intro") in /com.xyz.doc.user/introToc.xml

Book #2: "Using XYZ" (category: "user.content") in /com.xyz.doc.user/usingToc.xml

Book #3: "Troubleshooting" (category: "user.reference") in /com.xyz.doc.user/refToc.xml

As well as hide the following books/categories and related keyword indexes:

Book #4: "Platform ABC" (category: none) in /org.abc.doc.isv/toc.xml

Book #5: "DEF Toolkit" (category: "isv.reference") in /com.def.doc.isv/toc.xml

Book #6: "GHI Support" (category: "isv.reference") in /com.ghi.doc.isv/toc.xml

The markup would be the following:

<extensions>

<tocOrder>

<toc id="/com.xyz.doc.user/introToc.xml"/>

<category id="user.content"/>

<toc id="/com.xyz.doc.user/refToc.xml"/>

</tocOrder>

<hidden>

<toc id="/org.abc.doc.isv/toc.xml"/>

<category id="isv.reference"/>

<index id="/org.abc.doc.isv/index.xml"/>

<index id="/com.def.doc.isv/index.xml"/>

</hidden>

</extensions>

Supplied Implementation:

This API is supported by any help implementation that is based on org.eclipse.help, including the default help implementation provided by Eclipse. Copyright (c) 2006, 2011 IBM Corporation and others.

All rights reserved. This program and the accompanying materials are made available under the terms of the Eclipse Public License v1.0 which accompanies this distribution, and is available at http://www.eclipse.org/legal/epl-v10.html

20.2.1.4.2. Generated navigation topics

If a topic in the table of contents does not specify an href attribute, or a toc element does not specify a topic element, Help will automatically generate a topic containing the topic's title, a heading to denote the start of the list of subtopics, and an unordered list of subtopic links (direct subtopics only).

Products may customize the appearance of these generated topics using CSS. Help will look for a book.css file at the root of the currently running product's plug-in, and will use this stylesheet to style the generated document. You may rely on the elements with the following classes to be present in the generated document, and use these to style it:

NavTitle: The class of the heading containing the topic's title.

NavListTitle: The class of the heading before the list of topic links.

NavList: The class of the list containing the links to the direct subtopics.

For example, to hide the list bullets, you would use the following CSS rule:

UL.NavList {

list-style-type: none;

}

20.2.1.4.3. The Help Menu

The Eclipse Platform and SDK contain menu items for Help Contents, Search and Dynamic Help. A menu item to show the keyword index can be added to the help menu in an Eclipse based product by adding the following extension point to the plugin.xml file of the product plugin. The same approach will work equally well for a Rich Client application.

<extension point="org.eclipse.ui.menus">

<menuContribution locationURI="menu:help?before=group.assist">

<command commandId="org.eclipse.help.ui.indexcommand"

mnemonic="I"

style="push">

</command>

</menuContribution>

</extension>

20.2.1.4.4. Information center customization

All of the customizations which can apply to the Eclipse help system can also be applied to an Eclipse information center including product specific branding, style sheets, banner and ordering of books. The easiest way to do this is to create a plug-in which implements the org.eclipse.core.runtime.products extension point and place all the customization in that plugin.

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.

Creating a product plug-in

A product plugin can be created using the menu item File/New/Project and selecting Plug-in Project. Uncheck the check box titled "Create Java Project", give the project a name, hit "next" and "finish" to exit the wizard. In the extensions tab of the plug-in manifest editor add the extension "org.eclipse.core.runtime.products" and give it a name. Create a product element beneath that and give it a name. Beneath the product element add a property element and set "preferenceCustomization" to "plugin\_customization.ini". plugin.xml will now contain entries similar to the following.

<extension

id="custom"

point="org.eclipse.core.runtime.products">

<product

application="org.eclipse.ui.ide.workbench"

description="This is my custom product"

name="Custom Product">

<property

name="preferenceCustomization"

value="plugin\_customization.ini">

</property>

</product>

</extension>

Installing the product plug-in

Export the product plug-in as a deployable plug-in. Copy the jar file for the exported plug-in to the "dropins" folder in the eclipse installation which will be used to start the information center. Start the information center with an additional argument to specify the product. For example if the product plugin is called help.product and its product id is "custom" add the argument "-product help.product.custom". Open the information center, the title will now show the customized product name.

Additional customization

Modify plugin\_customization.ini in the product plugin to add more customizations, see Help system customization, re-export and replace the plug-in in the dropins folder. To modify the banner a line like this could be added to plugin\_custiomization.ini.

org.eclipse.help.base/banner=/org.eclipse.help.webapp/advanced/banner.html

20.2.1.4.5. Using about.html to debug information centers

When developing an information center sometimes not all of the books show as expected or customizations do not have the desired effect. The about.html page can be used to get information about installed plug-ins and help system preferences. If the home page of the help system is

http://<hostname>:<port>/help/index.jsp

the about page has a url of

http://<hostname>:<port>/help/about.html

Without any parameters the about page shows a list of all plug-ins installed on the system. Parameters can be added to show other kinds of information as follows:

about.html?show=preferences shows the values of the help system preferences.

about.html?show=agent shows the user agent information from the Web Browser used to read the page.

20.2.1.4.6. Using AbstractHelpScope to filter an information center

An information center can be used to serve information for more than one product. Using the AbstractHelpScope class it is possible to define documentation subsets called help scopes which allow for users to see only the information which applies to one or more specific products.

A help scope consists of an id and a class which subclasses AbstractHelpScope and is defined in the extension point org.eclipse.ua.help.base.scope. The AbstractHelpScope class contains methods for determining whether a table of contents, topic or index entry should be included in the scope. Once a help scope with an id of myId has been declared it can be applied by opening an infocenter with a url of this form:

http://<host>:<port>/help/index.jsp?scope=myId

This will present an infocenter which shows only those topics which fall within the scope myId.

Scopes can be joined together to retrieve the intersection or union of multiple scopes. A basic query language can be used to perform these multi-scope operations.

Examples:

Filter help info to include anything in A,B, or C:

scope=A|B|C

scope=(A|B|C)

Filter help info to include anything in A,B, and C:

scope=A^B^C

scope=(A^B^C)

scope=A&scope=B&scope=C

Filter help to include A or B, and C:

scope=(A|B)^C

scope=A|B&scope=C

Note: (A|B^C) will not be recognized by the parser, and will only use the last operation, so it will look like (A^B^C)

Filter help to include A and B, or, A and C:

scope=(A^B)|(A^C)

The URL character for 'and' (intersection) is '^', and 'or' (union) is '|'.

20.2.1.5. Pre-indexing documentation

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, for example

<extension

point="org.eclipse.help.toc">

<index

path="index">

</index>

</extension>

Add the help.buildHelpIndex ANT task to the build.xml file in the plugin project by adding the lines below. A build.xml file can be created by right clinking on MANIFEST.MF in the package explorer and selecting the menu item PDE Tools/Create Ant Build File. This example build the index for the default locale and also for "nl/fr". This should be modified to match the locales which you are using.

<target name="build.index" description="Builds search index for the plug-in: org.eclipse.platform.doc.user." if="eclipse.running">

<help.buildHelpIndex manifest="plugin.xml" destination="."/>

<help.buildHelpIndex manifest="plugin.xml" destination="nl/fr"/>

</target>

create an index by building the target "build.index". The ant task must be run in the same JRE as the workspace. To do this right click on build.xml, select the menu item Run As/Ant build... the ant dialog will appear. On the targets tab check only "build.index" and on the JRE tab select the radio button "Run in the same JRE as the workspace". When you hit the "Run" button the indexes will be built.

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=outputDirectory -DindexLocale=locale

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.my.plugin -DindexLocale=en

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

20.2.2. Help content

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" />

20.2.2.1. 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.

Dynamic content

Dynamic content is available for the table of contents in the form of filters and extensions. For example, you may want a topic to show up in the table of contents only when running on a specific operating system.

Includes are not supported here because they are not needed; use links instead.

20.2.2.2. 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="PLUGINS\_ROOT/another\_plugin\_id/ref/ref1.html"/>

Here PLUGINS\_ROOT will be resolved at runtime and replaced with the root directory for the plugins. You can specify your own plug-in id for another\_plugin\_id. 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="PLUGINS\_ROOT/org.eclipse.platform.doc.isv/guide/help.html"/>

Prior to 3.2, references to documents in other plug-ins were made by using '..' to go up to the plug-in level, then referencing the plug-in Id followed by the HREF to the topic inside the plug-in. The recommended way to do this now is to use PLUGINS\_ROOT instead of '..'. Using this variable avoids these up/down trips in references, and can be used for all the resource URLs in the help documents (images, links, CSS files, java script files etc.)

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="PLUGINS\_ROOT/PRODUCT\_PLUGIN/book.css"

refers to a style sheet residing in the plug-in for the currently running product.

20.2.2.3. 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:

20.2.2.4. Adding criteria to help content

This page will explain how to activate criteria. To enable criteria you need define which criteria are associated with each topic and also set preferences which tell the help system to show criteria. The association between criteria and topics can be either in a toc file or using a criteria provider class.

What are criteria?

Some topics of the plug-in might only be of interest to certain users, for example a topic may be specific to the linux platform. The document writer can define a criterion "platform" with values such as "linux", "windows" etc applied to different topics. The user can then filter the documentation to show only documents applicable to a specific platform.

How to define criteria in a table of contents file

When defining a table of contents file criteria information can be appended according to these steps:

Add criteria tag as children to the topic.

Use id to define 'name' and 'value' attribute of the criteria element

For example:

<toc href="../topic/org.eclipse.platform.doc.isv/guide/tochref">

<criteria name="platform" value="AIX"/>

<criteria name="platform" value="Windows"/>

<topic label="label1" href="../topic/org.eclipse.platform.doc.isv/guide/href1">

<criteria name="platform" value="Windows"/>

</topic>

<topic label="label2" href="../topic/org.eclipse.platform.doc.isv/guide/href2">

<criteria name="platform" value="AIX"/>

</topic>

</toc>

Some rules to follow when adding criteria:

The criteria name is case insensitive but the criteria value is case sensitive.

It is important to ensure that parent topics contain all of the criteria of any of their children, the parent and child will not show in the table of contents if filtered using criteria which only the child satisfies.

Any criterion with an empty value is invalid and will be ignored.

Multiple comma separated values can be specified in a single <criteria> element, for example <criteria name="Platform" value="AIX,Linux"/>. The name attribute takes a single value.

How to define criteria using a criteria provider

As an alternative to defining criteria in a table of contents file the extension point org.eclipse.criteriaProvider can be used to define a class which will associate criteria with topics. Criteria defined using this extension point will be merged with those defined in the table of contents and if multiple criteria providers are defined each will contribute to the criteria of an ITopic or IToc.

Criteria localization

The names and values of criteria can be localized. Follow these steps to add localization information for criteria:

In plugin.xml, use the extension point org.eclipse.help.criteriaDefinition to provide the criteria definition file. (if no translation file is provided, then the id of criteria name and value will be displayed). For example:

<extension point="org.eclipse.help.criteriaDefinition">

<criteriaDefinition file="criteria.xml"/>

</extension>

In the definition file, define the translation of criteria information. Node 'criterion' represents the criterion name. Node 'criterion-value' represents the criterion value. All 'id' attributes represent the id of criterion name or value. All 'name' attributes represent the display name of criteria name or value in one specific locale. For example:

<criteriaDefinition>

<criterion id="platform" name="Platform">

<criterion-value id="AIX" name="AIX Server"/>

<criterion-value id="Windows" name="Windows Server"/>

</criterion>

</criteriaDefinition>

How to set the preferences to enable criteria

Define the necessary preferences in your plugin customization file.

enableCriteria. This preference must be true to enable filtering by criteria.

example: org.eclipse.help/enableCriteria = true

supportedCriteria. A comma separated list of the criteria which will be supported in the help system.

example: org.eclipse.help/supportedCriteria = platform,version

which means that 'platform' and 'version' are the supported criteria in help system.

See Also:

The criteria definition extension point.

The criteria provider extension point.

20.2.2.5. Building nested documentation structures

As plug-ins contribute functionality 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 concepts">

<topic>

...

</topic>

</toc>

20.2.2.6. Contributing XHTML help documents

Why use XHTML?

The help system provides the ability to produce dynamic help content by annotating your XHTML markup with special tags to filter, include, and extend documents. These features are not available when using HTML.

How to contribute XHTML

XHTML help documents are contributed in exactly the same way as HTML documents by referencing them from a table of contents (TOC) file. Since Eclipse 3.4 it is no longer necessary to bind the "org.eclipse.help.base.xhtml" search participant to your doc plugin.

XHTML include format

If you wish to use includes in your XHTML, the format of the path attribute is as follow: (explained below)

<plugin\_id>/<path\_to\_xhtml\_file>/<filename\_xhtml>/<element\_id>

Where the fields are:

plugin\_id: The id of the plug-in containing the content to include (e.g. org.eclipse.help)

path\_to\_xhtml\_file: The plug-in relative path to the file (e.g. /my\_folder/my\_sub\_folder/)

filename\_xhtml: The name of the XHTML file, including extension (e.g. my\_file.xhtml)

element\_id: The unique identifier for the element you wish to include. This is set by adding an id attribute to that element (e.g. my.element.id)

For example, if you wish to include the paragraph (<p> element) with the id my\_copyright from the file /copyrights/copyright.xhtml in plugin my.product.plugin, you would specify the following:

my.product.plugin/copyrights/copyright.xhtml/my\_copyright

20.2.2.7. Remote Help

What is remote help?

The help system provides the ability to retrieve content, not only from plug-ins installed in the local instance of Eclipse, but also from one or more remote servers running help in information center mode. This allows products to ship without help content and provide it from a server to reduce their download and install size. It also allows user the ability to include help content from plug-ins not installed in their environment.

How does remote help work?

Content from the remote help servers is merged seamlessly with the local content. There will be no difference perceptible to the user other than potential network delays. This is done by adding a second implementation of AbstractTocProvider called RemoteTocProvider that supplies the content from the remote help server alongside the local TocFileProvider. For performance reasons, where both are available, local content will be supplied rather than remote content.

How do I setup a remote help server?

All you have to do to setup a remote help server is run the help system (version 3.3 or later) in information center mode.

How do I access a remote help server?

A remote help server can be specified in two ways as follows:

Product customization: There are 6 options that can be included in the plugin\_customization.ini file to setup remote help defaults. They are remoteHelpOn, remoteHelpHost, remoteHelpPath, remoteHelpPort, remoteHelpName, remoteHelpICEnabled..

Help Content preferences: There is a preference page that allows users to configure remote help server settings manually.

20.2.2.8. 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 the cheatsheet "Check out a CVS project". We will take a look at how to create and reference your own actions.

Open a cheatsheet.

20.2.2.8.1. 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.");

}

}

});

}

}

20.2.2.8.2. Invoking the action from HTML

To include active help links in your documentation, you must first declare the use of the supporting JavaScript code. The live help JavaScript is located in the org.eclipse.help plug-in. You refer to it using the help system's cross plug-in referencing technique. This script reference should be placed in the HEAD section of your HTML:

<script language="JavaScript" src="PLUGINS\_ROOT/org.eclipse.help/livehelp.js"> </script>

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.

20.2.2.8.3. 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 "liveHelp.js" (make a copy of this from the plugins/org.eclipse.help plugin and change your script statement to refer to your local copy) 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;

}

});

}

20.2.2.9. Embedding commands in help

It is possible to create links to workbench commands in help content. When the user clicks the link, the command will be executed. This feature is similar to Active Help in that it uses JavaScript to bridge between the documentation HTML and the Eclipse Java runtime. Also like Active Help, the command framework is extensible; so you can contribute new commands with interesting behaviors. However, unlike Active Help, there are a large number of useful commands already defined in the workbench and more are being added all the time. This should be appealing to documentation authors because linking to an existing command does not require writing any Java code.

Command links can be used to:

Open a specific preference page, such as General > Appearance.

Select a new perspective or switch to a pre-determined perspective like the Java Browsing. perspective

Select a view to open or open a pre-determined view like the Bookmarks. view

Open wizards like the New Plug-in Project wizard and the Export Preferences. wizard

Open a Cheat Sheet.

Open a message dialog.

Open the About Eclipse SDK dialog.

And much more!

To get an idea of the range of commands available, look at the General > Keys preference page. This page is used to bind key sequences to commands so it shows the list of available commands. Note that not every command will do something useful from help content. Many commands are designed to work within a somewhat narrow context (like a particular view, or expecting a certain type of selection).

If you want to define your own commands, refer to the documentation for the org.eclipse.core.commands API package and the org.eclipse.ui.commands and org.eclipse.ui.handlers extension-points.

20.2.2.9.1. Authoring a command link

To include command links in your documentation, you must first declare the use of the supporting JavaScript code. The live help JavaScript is located in the org.eclipse.help plug-in. You refer to it using the help system's cross plug-in referencing technique. This script reference should be placed in the HEAD section of your HTML:

<script type="text/javascript" src="PLUGINS\_ROOT/org.eclipse.help/livehelp.js">

</script>

Now in the body of your documentation you may invoke the executeCommand function. Here is an example:

<a href='javascript:executeCommand("org.eclipse.ui.help.aboutAction")'>

Open the About dialog</a>

The parameter for the executeCommand function is a serialized ParameterizedCommand. See the ParameterizedCommand.serialize() method for full details on this format.

The example above shows the bare minimum required to embed a command in an HTML link. The Eclipse documentation supplements this with two extra pieces of information. First a class attribute is specified to allow for tuning the look of the link via CSS. Second, an image tag is included before the link text. The image serves to distinguish command links from ordinary links to other HTML pages. Supplementing our initial example with these two extra features will look like this:

<a class="command-link" href='javascript:executeCommand("org.eclipse.ui.help.aboutAction")'>

<img src="../topic/org.eclipse.platform.doc.isv/guide/PLUGINS\_ROOT/org.eclipse.help/command\_link.svg">

Open the About dialog</a>

In the examples above, the About dialog command does not require any parameters, so the serialization is merely its command id: org.eclipse.ui.help.aboutAction. Below is another example showing a command with a parameter. Note the command id is followed by the parameter id and value in parentheses:

<a href='javascript:executeCommand(

"org.eclipse.ui.window.preferences(preferencePageId=org.eclipse.ui.preferencePages.Views)")'>

Show a preference page</a>

Another example demonstrates that multiple parameters are possible. They are comma separated and the order of the parameters is not important.

<a href='javascript:executeCommand(

"org.eclipse.ui.dialogs.openMessageDialog(imageType=3,buttonLabel2=Maybe,title=Opinion Poll,message=Do you like command links?,buttonLabel0=Yes,defaultIndex=0,buttonLabel1=No)")'>

Open a message dialog</a>

20.2.2.10. Processing Help Content

The Eclipse help system performs preprocessing on html and xhtml pages to add breadcrumbs and JavaScript. The processing is performed by output filter which operate on the output stream. Eclipse has an extension point to allow user defined filters to be added. This can be used for example to inject javascript into every help page. See the extension point org.eclipse.help.webapp.contentFilter for more information.

20.2.2.11. Adding Child Links to Help Pages

The Eclipse Help System recognizes two special comments in the source code which can be used to cause child links to appear in help topics. This is useful in the case of a help topic which is the parent of an anchor.

If the sequence

<!--INSERT\_CHILD\_LINKS-->

appears in the source code it will be replaced with html containing hyperlinks to all child topics.

If the sequence

<!--INSERT\_CHILD\_LINK\_STYLE-->

appears in the source code it will be replaced by either "has\_child\_topics" or "no\_child\_topics" depending on whether or not the topic has children. This can be used to assign a style to a section of html so that it will only be shown if the topic has children.

20.2.2.12. Help Placeholders

A help placeholder is used when documentation is not installed at the same time as the corresponding software bundles. A placeholder contains a reference to a help page which describes how to install help bundles.

As an example suppose that for reasons of size the documentation bundle org.eclipse.myfeature.doc was a separate download from org.eclipse.myfeature. Users who installed only org.eclipse.myfeature without the documentation would not see the help content for myfeature and a help search would not search org.eclipse.myfeature.doc. If the developers of myfeature add a toc placeholder the user will be made aware that documentation is available and be able to get instructions for installing it.

A help placeholder is defined using the placeholder element of the extension point org.eclipse.help.toc. The easiest way to manage placeholders is to create bundles which contain one or more help placeholder extensions and no code. The code bundles will have a dependency to these help placeholder bundles.

As an example here is how to setup a placeholder for the ( nonexistent ) help bundle org.eclipse.myfeature.doc A bundle org.eclipse.myfeature.doc.placeholder is created which contains the extension point below and no code.

<extension

point="org.eclipse.help.toc"\*gt;

<placeholder

placeholderPage="http://www.eclipse.org/installingmyfeaturedocs.html"

plugin="org.eclipse.myfeature.doc"\*gt;

</placeholder\*gt;

</extension\*gt;

A dependency is added from the Java bundle org.eclipse.myfeature to org.eclipse.myfeature.doc.placeholder. If the help system is launched with org.eclipse.myfeature installed and org.eclipse.myfeature.doc not installed the home page will be replaced with a page informing of the missing documentation bundle. In this example clicking on the link "org.eclipse.myfeature.doc" will open the page http://www.eclipse.org/installingmyfeaturedocs.html.

20.2.3. 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.

20.2.3.1. 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.

Note: The default implementation of help will display the help dialog tray only if the dialog is either large enough to accomodate it, or is resizable. Otherwise, an infopop will be shown.

20.2.3.2. Describing and packaging context-sensitive help content

Context-sensitive help is described by associating the context id declared in the UI code with a description and list of links to related topics or commands in the online help. These associations are made inside an XML file located within the plug-in that contains the topics in question. 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, as well as command links to perform any operation for the user, e.g. open a cheat sheet.

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>

<command serialization="org.eclipse.ui.cheatsheets.openCheatSheet(cheatSheetId=org.eclipse.panic.button.cheatsheet)quot; 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 qualified.

A plug-in containing context files contributes them using the org.eclipse.help.contexts extension point.

<extension point="org.eclipse.help.contexts">

<contexts file="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 file="myContextHelp.xml" plugin="com.example.helpExample" />

</extension>

When a context id is declared in an extension point the Eclipse help system will create a fully qualified context id of the form <plug-in name>.<context id> and use this when matching against the context ids used in the Java source. <plug-in name> is the value of the "plugin" attribute, or if not specified the name of the plug-in in which the org.eclipse.help.contexts extension is declared.

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 plugin attribute must be used in the extensions if multiple plugins will contribute to the same context. 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 content

Dynamic content is available for the context help in the form of filters on context help topic links. For example, you may want a topic link to show up in the context help only when running on a specific operating system.

Adding Context Help to your Java Code

See declaring a context Id for more information.

20.2.3.3. Dynamic 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.

Note: In addition to using context providers (or alternatively), you can use XML annotations to filter topics in context help.

20.2.3.4. 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.

20.2.4. Help 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.

20.2.4.1. Plugging in search engines

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=20&amp;m=all">

</param>

</engine>

20.2.5. Deploying the information center as a Web Archive

Using Eclipse 3.4 or later it is possible to configure the help plugins to be deployed as a web archive (war file) which will act as a fully

functioning information center. The instructions below assume a Tomcat server has been installed, but with minor modifications

these steps should work for any full featured server.

In your Eclipse installation locate the plugin org.eclipse.help.webapp.<version>.jar and copy it to a temporary directory,

unzip the copy into that folder.

In the webapp plugin locate the web-archive directory and underneath that there will be two directories titled "help" and

"org.eclipse.help.infocenter-feature".

Import the org.eclipse.help.infocenter-feature using File->Import->Existing Project.

Export org.eclipse.help.infocenter-feature as a deployable feature and set the destination to be web-archive/help/WEB-INF in the area where org.eclipse.help.webapp.<version>.jar was unzipped.

Add some documentation plugins to the webapps/help/WEB-INF/plugins directory.

Download org.eclipse.equinox.http.servletbridge\_<version>.jar and org.eclipse.equinox.servletbridge\_<version>.jar from the equinox download site. Select a version of Equinox that matches the version of Eclipse you are running to take you to the downloads page.

Add the file org.eclipse.equinox.servletbridge\_<version>.jar. to the help/WEB-INF/lib directory. You may need to create this directory.

Add the file org.eclipse.equinox.http.servletbridge\_<version>.jar to the help/WEB-INF/plugins directory

At this stage you can create a war file from the help directory or you can copy the directory and its contents to the webapps folder of your Tomcat installation.

For Tomcat only. In conf/server.xml add URIEncoding="UTF-8" to the connector element, for example

<Connector port="8080" URIEncoding="UTF-8" etc.>

If this step is not performed search will fail if the search term contains non ASCII characters.

Start Tomcat and see the help system start up.

Notes: If you look in the config.ini in the help.war file under directory help/WEB\_INF/configuration you will notice the

line eclipse.product=org.eclipse.productname. If your product has help system customizations in a product plugin you can

activate these by changing this line to point to your product plugin.

20.3. Cheat sheets

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 steps 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 step.

Composite cheat sheets can guide the user through larger problems by breaking that problem into tasks. Each task is represented by a simple cheat sheet.

Related links

Working with cheat sheets

Working with composite cheat sheets

Simple cheat sheets

Composite cheat sheets

Authoring guidelines

Cheat Sheet Content File Format

20.3.1. Simple cheat sheets

Cheat sheets 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="../topic/org.eclipse.platform.doc.isv/guide//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 java project, and a java class that will print "hello world" in the console when run.

Let's get started!

</description>

</intro>

<item

href="../topic/org.eclipse.platform.doc.isv/guide//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 paramN 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="../topic/org.eclipse.platform.doc.isv/guide//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 world!"); and save your changes. Press the "click when done" button below when finished.

</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.

Related links

Working with cheat sheets

Creating composite cheat sheets

Authoring guidelines

org.eclipse.ui.cheatsheets.cheatSheetContent extension point

Cheat sheet content file specification

20.3.2. Cheat Sheet Content File XML Format

Identifier:

org.eclipse.ui.cheatsheets.cheat\_sheet\_schema

Since:

3.2

Description:

This document describes the cheat sheet content file structure as a series of DTD fragments (machine readable XML schema).

A cheat sheet consists of a series of items (steps) which must be completed in order. Items can be divided into subitems and can launch commands or actions which will perform some of the steps for the user.

Configuration Markup:

<!ELEMENT cheatsheet (intro , item+)>

<!ATTLIST cheatsheet

title CDATA #REQUIRED

>

The root element of a cheatsheet.

title - The title of this cheat sheet. The title will be displayed at the head of the cheat sheet view when the cheat sheet is opened.

<!ELEMENT intro (description)>

<!ATTLIST intro

contextId CDATA #IMPLIED

href CDATA #IMPLIED

>

The <intro> element is used to define the introductory text to be displayed when the cheat sheet is opened.

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).

<!ELEMENT description (#PCDATA)>

The <description> element holds the description of a cheat sheet or of a cheat sheet item. The description consists of text interspersed with form text markup. The cheat sheet automatically formats and lays out the text to make it show up reasonably in the UI. Within the text, balanced <b>...</b> tags cause the enclosed text to be rendered in a bold font, and the <br/> 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 "&lt;", "&gt;", "&amp;", "&apos;", and "&quot;" 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 <br/> tags is ignored, as is whitespace immediately before the </description> tag.

<!ELEMENT item (description , (action | command | perform-when | (subitem | conditional-subitem | repeated-subitem)\*) , onCompletion?)>

<!ATTLIST item

title CDATA #REQUIRED

dialog (true | false) "false"

skip (true | false) "false"

contextId CDATA #IMPLIED

href CDATA #IMPLIED

>

Each <item> element describes one top-level step in a cheat sheet. An <item> may contain <subitem> elements.

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 or command. 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>, <command> or <perform-when>)element is associated with a button that the user can press to perform the step's action or command. If no action or command is present, the step is one that the user must carry out manually and then overtly indicate that they have successfully completed the step.

Steps may be 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.

title - The title of this step.

dialog - if "true" means this step involves opening a modal dialog. This is a hint to the system that it should allow the user to continue using the cheat sheet while in the modal dialog. This attribute will only affect dialogs launched from a command or action.

skip - if "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).

<!ELEMENT subitem (description? , (action | command | 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.

Sub-items may have an optional action or command. The presence of an <action>, <command> or <perform-when> element is associated with a button that the user can press to perform the sub-step's action or command. If no action or command 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 completed 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.)

Since version 3.4 a description has been allowed in a subitem in place of the label attribute, which allows for formatting tags to be used.

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. A subitem may have either a label or a description element but not both. A description element may contain the formating tags such as <b>, </b> or <br/>.

skip - if "true" this 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.

<!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.

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>

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).

<!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.

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>

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).

<!ELEMENT action EMPTY>

<!ATTLIST action

class CDATA #REQUIRED

pluginId CDATA #REQUIRED

paramN CDATA #IMPLIED

confirm (true | false) "false"

when CDATA #IMPLIED

required (true | false) "true"

translate CDATA #IMPLIED

>

Each <action> element describes an action in a cheat sheet.

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.

paramN - For action classes that also implement org.eclipse.ui.cheatsheets.ICheatSheetAction, the string values of attributes param1, param2 up to param9 are passed to the action when it is invoked. You can pass up to 9 parameters to a cheat sheet action , 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 - If "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.

required - if "true" this item or subitem can only be completed by performing this action (it may still be skipped if skip="true"). If "false" two buttons will be created, one to perform the task and one to mark it as complete, either will complete this step or substep.

translate - A comma separated list of parameters which are translatable. Any parameters not in the list are considered non-translatable. While this attribute is optional it is strongly recommended that it be provided for any cheat sheat which may end up being translated. If this attribute is not specified it means that there is no translation hint.

Examples:

translate = "param2, param3" means translate param2 and param3 only.

translate = "" means do not translate any parameters for this action.

<!ELEMENT command EMPTY>

<!ATTLIST command

serialization CDATA #REQUIRED

returns CDATA #IMPLIED

confirm (true | false) "false"

when CDATA #IMPLIED

required (true | false) "true"

translate CDATA #IMPLIED

>

Each <command> element describes an command in a cheat sheet.

Below is an example of an item with a command which opens a dialog box and stores the result in the cheat sheet variable "result".

<item title="View Selection">

<description>Select a view which will be opened in the next step.</description>

<command returns = "result"

serialization="org.eclipse.ui.dialogs.openMessageDialog(title=Select View,buttonLabel0=Package Explorer,message=Select a view ,buttonLabel1=Search View)"/>

<onCompletion> Selected the ${result}. </onCompletion>

</item>

serialization - A serialized ParameterizedCommand, which is a string containing the command name and parameters. See the ParameterizedCommand.serialize() method for full details on this format.

returns - An optional attribute which specifies the name of a cheat sheet variable which will be used to store the return value of the command. This allows a command to set a cheat sheet variable which is used in a later <perform-when>, <conditional-subitem> or <repeated-subitem>.

confirm - if "true" indicates that this step (or sub-step) requires the user to manually confirm that the command has been completed.

when - Indicates this command 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 <command> element is not a child of a <perform-when> element.

required - if "true" this item or subitem can only be completed by performing this command (it may still be skipped if skip="true"). If "false" two buttons will be created, one to perform the task and one to mark it as complete, either will complete this step or substep.

translate - A comma separated list of parameters which are translatable. Any parameters not in the list are considered non-translatable. While this attribute is optional it is strongly recommended that it be provided for any cheat sheat which may end up being translated. If this attribute is not specified it means that there is no translation hint.

Examples:

translate = "param2, param3" means translate param2 and param3 only.

translate = "" means do not translate any parameters for this command.

<!ELEMENT onCompletion (#PCDATA)>

Contains text which will be displayed when an item is completed. This is particularly useful in the final step of the cheat sheet to acknowledge completion of the entire task. The description consists of text interspersed with form text markup following the same rules as for a <description> element. <onCompletion> elements may also contain references to cheat sheet variables of the form "${var}", which will be expanded using the actual value of the cheat sheet variable var at the time this step was completed.

<!ELEMENT perform-when (action | command)+>

<!ATTLIST perform-when

condition CDATA #REQUIRED

>

Each <perform-when> element describes an action in a cheat sheet.

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" />

<command when="c" serialization="org.eclipse.search.ui.views.SearchView"/>

</perform-when>

</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>

condition - Arbitrary string value used to select which child <action> or <command> 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).

Examples:

The following is an example of a simple cheat sheet content file which demonstrates the use of commands, perform-when and conditional subitems.

<?xml version="1.0" encoding="UTF-8"?>

<cheatsheet title="Sample Cheat Sheet">

<intro>

<description>A cheat sheet which demonstrates the use of perform-when and conditional subitems</description>

</intro>

<item title="View Selection">

<description>Select a view which will be opened in the following steps.</description>

<command returns = "result"

serialization="org.eclipse.ui.dialogs.openMessageDialog(title=Select View,buttonLabel0=Package Explorer,message=Select a view ,buttonLabel1=Search View)"/>

<onCompletion> Selected the ${result}. </onCompletion>

</item>

<item title="Close Views">

<description>Close the search view and package explorer if open</description>

</item>

<item title="Open the view from a perform when item" skip = "true">

<description>Uses perform when to open the view seleted previously.</description>

<perform-when condition = "${result}">

<command when = "Package Explorer"

serialization="org.eclipse.ui.views.showView(org.eclipse.ui.views.showView.viewId=org.eclipse.jdt.ui.PackageExplorer)"/>

<command when = "Search View"

serialization="org.eclipse.ui.views.showView(org.eclipse.ui.views.showView.viewId=org.eclipse.search.ui.views.SearchView)"/>

</perform-when>

</item>

<item title="Close Views">

<description>Close the search view and package explorer if open</description>

</item>

<item title="Open the view from a perform when subitem">

<description>Uses perform when to open the view seleted previously.</description>

<subitem label="Perform when subitem" skip = "true">

<perform-when condition = "${result}">

<command when = "Package Explorer"

serialization="org.eclipse.jdt.ui.PackageExplorer"/>

<command when = "Search View"

serialization="org.eclipse.search.ui.views.SearchView"/>

</perform-when>

</subitem>

</item>

<item title="Close Views">

<description>Close the search view and package explorer if open</description>

</item>

<item title="Open the view from a conditional subitem">

<description>Uses perform when to open the view seleted previously.</description>

<conditional-subitem condition="${result}">

<subitem when="Package Explorer" label="Open package explorer.">

<command serialization = "org.eclipse.jdt.ui.PackageExplorer"/>

</subitem>

<subitem when="Search View" label="Open Search View">

<command serialization = "org.eclipse.search.ui.views.SearchView"/>

</subitem>

</conditional-subitem>

</item>

</cheatsheet>

Copyright (c) 2004, 2007 IBM Corporation and others.

This program and the accompanying materials are made available under the terms of the Eclipse Public License 2.0 which accompanies this distribution, and is available at https://www.eclipse.org/legal/epl-2.0/ SPDX-License-Identifier: EPL-2.0

20.3.3. Composite cheat sheets

Composite cheat sheets provide guidance through complex problems by breaking the problem into a set of smaller tasks. Composite cheat sheets are registered using the the org.eclipse.ui.cheatsheets.cheatSheetContent extension point.

Content file format

The composite cheat sheet content itself is defined in a separate file which conforms to the composite cheat sheet content file specification. The content file is an XML file and consists of a set of tasks organized in groups into a tree structure.

<compositeCheatsheet> is the root element of a composite cheat sheet. It will have a single root task which may be a <task> or <taskGroup>. <taskGroup> elements may have one or more children each of which can be a <task> or <taskGroup>. A <task> does not have child tasks.

Tasks and task groups may contain <intro> elements which contain the text to be displayed before the task has been started and <onCompletion> elements which contain the text to be displayed once the task is completed. Both the <intro> and <onCompletion> elements may contain form text markup, in the example below the tags <b> and </b> are used to make text bold.

Tasks may also contain <param> elements. A cheat sheet task may have any of the following parameters: "id" is the id of a registered cheatsheet, "path" is the relative path or URL of the cheat sheet content file and "skipIntro" is a boolean parameter which if true causes the cheat sheet to start at the first step rather than at the introduction. Either "id" or "path" but not both must be specified.

A <dependency> node from task "B" to task "A" represents a requirement that task A is completed before task B can be started.

Cheat sheet task parameters

If a task has kind = "cheatsheet" a cheat sheet will be opened when that task is started, There are three possible parameters to a cheat sheet task.

Parameter name

Description

id

The id of a cheat sheet which has been registered using the extension point org.eclipse.ui.cheatsheets.cheatSheetContent. This identifies the cheatsheet which will be associated with this task. Either the id or the path parameter (but not both) should be specified.

path

The URL of a cheat sheet content file. This may be an absolute URL, or relative to the content file for the composite cheat sheet. If both id and path are specified the path will be used to locate the content file and the id parameter will be ignored.

showIntro

A boolean parameter with default value of true. If "false" the cheat sheet when started will initially show the first step rather than the introduction..

Example of a composite cheat sheet

The file below is an example of how to create a composite cheat sheet from existing cheat sheets. It shows how to create task groups and make tasks skippable.

<?xml version="1.0" encoding="UTF-8"?>

<compositeCheatsheet name="Composite cheat sheet example">

<taskGroup name= "Composite cheat sheet example">

<intro> This is an example of a <b>composite cheat sheet</b> built from existing cheat sheets.

<br/><br/>You can select a task to work on either by following the hyperlinks or by

selecting a task in the tree.

</intro>

<onCompletion>Congratulations you have completed all the tasks.</onCompletion>

<task kind="cheatsheet" name= "Branching and merging using CVS" skip="true">

<param name = "id" value = "org.eclipse.platform.cvs\_1" />

<intro>This cheat sheet is intended for CVS users. If you are not using CVS or do

not intend to branch and merge you may skip this task.

</intro>

<onCompletion>Congratulations you now know how to branch and merge.</onCompletion>

</task>

<taskGroup name= "Create Java Projects" kind = "sequence">

<intro> First you will learn how to create a simple java project, then you will create

an java project which uses SWT.

<br/><br/>This task group is a sequence which means that

if you click on the subtask "Standalone SWT Application" it will not let that task be started

until "Create a java project" has been completed.

</intro>

<onCompletion>Congratulations you have built both Java applications.</onCompletion>

<task kind="cheatsheet" name= "Create a java project" id = "createJavaProject">

<param name="id" value = "org.eclipse.jdt.helloworld"/>

<param name="showIntro" value = "false"/>

<intro>This cheat sheet walks through the process of creating a simple hello world application.

The cheat sheet can launch wizards to create a new project and a new class.

</intro>

<onCompletion>Congratulations you have succeeded in creating a hello world application</onCompletion>

</task>

<task kind="cheatsheet" name= "Standalone SWT Application">

<intro>Eclipse plugins which contribute to the user interface use The Standard Widget Toolkit (SWT).

This task guide can be used to learn more about SWT.

</intro>

<param name = "id" value = "org.eclipse.jdt.helloworld.swt" />

<onCompletion>Congratulations you have succeeded in creating an SWT application.</onCompletion>

</task>

</taskGroup>

</taskGroup>

</compositeCheatsheet>

Composite Cheat Sheet Extensibility - Provisional in Eclipse 3.2

Composite cheat sheets are extensible, however in Eclipse 3.2 this extensibility is provisional and the classes could change before they become API. Composite cheat sheet support can be extended by using the extension point org.eclipse.ui.cheatsheets.cheatSheetContent which has two new elements taskEditor and taskExplorer which allow for contribution of task editors and task explorers.

Contributing a task editor defines a new kind of task which displays in the task detail section. To contribute a task editor implement a concrete subclass of TaskEditor, then add a taskEditor element to plugin.xml.

The representation of the task explorer is also configurable with an extension point, a tree explorer is included with the Eclipse platform. By default the explorer for a composite cheat sheet when first opened is a tree, an attribute on the <compositeCheatSheet> element will allow that default to be overridden. If more than one explorer is registered the view menu will contain a menu item to switch between explorers. To contribute a task explorer first implement a concrete subclass of TaskExplorer, then add a taskExplorer element to plugin.xml.

Related links

Working with cheat sheets

Working with composite cheat sheets

Creating cheat sheets

Authoring guidelines

Composite cheat sheet content file specification

org.eclipse.ui.cheatsheets.cheatSheetContent extension point

20.3.4. Composite Cheat Sheets

Identifier:

org.eclipse.ui.cheatsheets.composite\_schema

Since:

3.2

Description:

The schema definition for a composite cheat sheet content file. A composite cheat sheet consists of a set of tasks organized into task groups. Each task can be a simple cheat sheet or a user contributed task kind. (machine readable XML schema.

Configuration Markup:

<!ELEMENT compositeCheatsheet (taskGroup | task)>

<!ATTLIST compositeCheatsheet

name CDATA #REQUIRED>

The root element of a composite cheatsheet

name - The name of the composite cheat sheet which will be displayed in large font when the composite cheat sheet is opened.

<!ELEMENT taskGroup ((task | taskGroup)+ , intro? , onCompletion? , dependsOn\*)>

<!ATTLIST taskGroup

kind (set|sequence|choice) "set"

name CDATA #REQUIRED

id CDATA #IMPLIED

skip (true | false) "false">

A task group represents a collection of related tasks. If the kind is "choice" only one of the child tasks need to be completed. If the kind is "set" or "sequence" all children must be completed. Note that the child elements <task>, <taskGroup>, <intro>, <onCompletion> and <dependsOn> may occur in any order. The order of the <task> and <taskGroup> elements determines the order in which they will be displayed and in the case of a sequence the order in which they must be performed.

kind - The kind of this task group which can be "set", "sequence" or "choice". If the kind is set or sequence this task group is complete when all of its child tasks/task groups have been completed. In addition the subtasks of a sequence must be completed in order. A choice is complete when any of its subtasks has been completed.

name - The name of this task group which will be displayed in the task explorer.

id - An id for this task group which is required if this task group is referenced by a dependsOn element.

skip - If true this group of tasks may be skipped.

<!ELEMENT task (intro? , onCompletion? , param\* , dependsOn\*)>

<!ATTLIST task

kind CDATA "set"

name CDATA #REQUIRED

id CDATA #IMPLIED

skip (true | false) "false">

A leaf task within a composite cheat sheet. A task does not have children, but it does have a task editor which shows in the lower/right hand pane of the cheat sheet view. The task kind determines which task editor will be opened, a task kind of "cheatsheet" represents a simple cheatsheet. Other task kinds may be contributed. Note that the child elements <intro>, <onCompletion>, <param> and <dependsOn> may occur in any order.

kind - The task kind. A task kind of "cheatsheet" represents a simple cheatsheet, other task kinds can be contributed using the extension point org.eclipse.ui.cheatsheets.cheatSheetContent.

name - The name of this task which will be displayed in the task explorer.

id - An id for this task group which is required if this task group is referenced by a dependsOn element.

skip - If true this task may be skipped.

<!ELEMENT param EMPTY>

<!ATTLIST param

name CDATA #REQUIRED

value CDATA #REQUIRED>

A parameter to a task within a composite cheatsheet. Each parameter has a name and value, both of which are strings. A task may have any number of parameters, two parameters for a single task may not share the same name.

name - The name of this parameter. A task of kind="cheatsheet" can have the following parameters:

id: The id of a cheat sheet which has been registered using the extension point org.eclipse.ui.cheatsheets.cheatSheetContent. This identifies the cheatsheet which will be associated with this task. Either the id or the path parameter (but not both) should be specified.

path: The URL of a cheat sheet content file. This may be an absolute URL, or relative to the content file for the composite cheat sheet. If both id and path are specified the path will be used to locate the content file and the id parameter will be ignored.

showIntro: A boolean parameter with default value of true. If "false" the cheat sheet when started will initially show the first step rather than the introduction.

value - The value of this parameter.

<!ELEMENT intro (#PCDATA)>

Contains the text which will be displayed before this task has been started. May contain form text markup.

<!ELEMENT onCompletion (#PCDATA)>

Contains the text which will be displayed in the completion panel for this task. May contain form text markup.

<!ELEMENT dependsOn EMPTY>

<!ATTLIST dependsOn

task CDATA #REQUIRED>

Creates a dependency between two tasks or subtasks.

task - The id of the task or task group which must be completed before this task can be started.

Copyright (c) 2006, 2007 IBM Corporation and others.

This program and the accompanying materials are made available under the terms of the Eclipse Public License 2.0 which accompanies this distribution, and is available at https://www.eclipse.org/legal/epl-2.0/ SPDX-License-Identifier: EPL-2.0

20.3.5. Cheat sheet authoring guidelines

Cheat sheets and composite cheat sheets enable a user to achieve a goal by completing a sequence of steps. These guidelines discuss when to use cheat sheets and how to best write a cheat sheet.

When to create cheat sheets

Cheat sheets are well suited to tasks which consist of steps which lead towards a tangible goal, such as building a simple application. The goal must be well defined so that the user can see success when all the steps in cheat sheet have been completed. Tutorials are often good candidates for cheat sheets, in a tutorial the goal is to learn how to perform a specific task. Cheat sheets will usually contain up to 10 steps and can be completed in a half an hour or less. For larger tasks consider using a composite cheat sheet.

When to create composite cheat sheets

Composite cheat sheets are used to for providing guidance through a task which is too large to describe in a single cheat sheet or which has multiple goals. A composite cheat sheet can be used to describe a sequence of tasks each of which builds on its predecessor.

When not to use cheat sheets

Cheat sheets work best when problem can be solved by a sequence of simple steps. Cheat sheets are not a substitute for the help system which allows for creation of HTML pages with rich graphics and random access of information using search and hyperlinks. Cheat sheets are not intended for tasks which require a large amount of text to be input by the user.

Guidelines for creating a cheat sheet

Choose a short and meaningful title.

Utilize links to the help system in cheat sheet tasks whenever possible to provide background or more detailed information

Consider rewriting as a composite cheat sheet if there are more than 10 steps.

Utilize commands where possible to perform simple tedious tasks such as opening perspectives, opening resources, executing wizards. Be sure to also provide a description of how to achieve the same effect without using the command.

Ensure commands can be skipped if the user decides to do the steps manually

Specify dialog="true" for steps which open dialogs. This will cause the cheat sheet to open in a "tray" to the right of most dialogs.

Organize the steps in such a way that the user sees visible signs of success frequently. A cheat sheet that makes a number of different changes to java source and then launches an application could be rewritten so that the application was launched after each source change.

Make sure the user understands why they are performing each step.

Rather than describing multiple ways to perform a step pick the simplest way requiring the least amount of description.

Use break (<br/>) tags to improve readability.

Subitems can be used to break steps down into smaller steps.

Not all cheat sheets will require subitems.

Start subitem label text using action word such as "Input", "Expand", "Select", etc

For subitem labels, clearly identify interface text using quotes to allow users to differentiate instruction text from interface text they need to locate and interact with.

Each subitem should be responsible for one atomic task such as inputting one value.

Always add an <onCompletion> element to the last step of a cheat sheet.

Test cheat sheets with different sizes for the cheat sheet view.

Test cheat sheets with the error log view open so you can see if any warnings are written to the log.

Guidelines for creating a composite cheat sheet

A composite cheat sheet should contain tasks that are related by a common theme.

Each task within a composite cheat sheet should have a well defined goal.

Use <onCompletion> elements to recognize success.

Task names should be short and concise.

Complex problems can be solved by using tasks that build upon their predecessors.

Use skip="true" if a task is optional.

For consistency, make sure the simple cheat sheet introduction and name match the introduction and name specified in the composite cheat sheet..

For task descriptions, clearly identify interface text using bold formatting.

Sequences should be used only when it is important to enforce order in which tasks can be started.

Related links

Working with cheat sheets

Working with composite cheat sheets

Simple cheat sheets

Composite cheat sheets

20.4. Status handling

Status handling is a facility that allow to introduce a custom way of showing problems in the product to users. A part of the facility is an extension point which allows to plug new status handler specific for the product. This handler associated to the product is intended to present the problem in the useful for users way.

The status handling is contributed using org.eclipse.ui.statusHandlers extension point. This contribution can be part of product plug-ins or can be provided separately in its own status handling plug-in. Separating the product handler into a separate plug-in is beneficial in those situations where the product and status handling are provided by different groups, e.g. CompanyA provides the ProductA and other companies provide status handlers showing problems in ProductA differently.

If there is no handler defined for the product, the facility will use a handler defined for application in the workbench advisor. When the advisor shipped with Eclipse is used the WorkbenchErrorHandler will be used for handling of the product problems.

20.4.1. Defining a product status handler

The AbstractStatusHandler class and the org.eclipse.ui.statusHandlers extension point make up the generic mechanism that can be used to customize problem handling for the given product. The main purpose of this extension is to define status handlers (which extends AbstractStatusHandler) and to specify the binding between a product and a status handler. For example, the following contribution defines a hypothetical status handler to be used by the workbench:

<extension point="org.eclipse.ui.statusHandlers">

<statusHandler

class="org.eclipse.ui.statushandlers.SampleStatusHandler"

id="sampleStatusHandler"/>

<statusHandlerProductBinding

handlerId="sampleStatusHandler"

productId="productId">

</introProductBinding>

</extension>

This contribution first defines the status handler and assigns it the id "sampleStatusHandler". It then binds this handler to a product whose id is "productId". When the status handling facility is accessed first time, the class specified in the class attribute will be instantiated by the workbench and used for handling of status or error occurred in the product.

The platform supplies its own AbstractStatusHandler implementation called WorkbenchErrorHandler.

20.4.2. Contributing a sample handler

We will now contribute a very basic status handler just to illustrate the steps needed to contribute a status handler implementation to the Workbench. To do this:

use the org.eclipse.ui.statusHandlers extension point to register a status handler implementation and bind it to your product

extend the org.eclipse.ui.statushandlers.AbstractStatusHandler class and use as the class attribute in the above extension point contribution

run your Eclipse based product with the correct product id

Here is the org.eclipse.ui.statusHandlers extension point markup:

<extension point="org.eclipse.ui.statusHandlers">

<statusHandler

class="org.eclipse.ui.statushandlers.SampleStatusHandler"

id="sampleStatusHandler"/>

<statusHandlerProductBinding

handlerId="sampleStatusHandler"

productId="productId">

</statusHandlerProductBinding>

</extension>

A statusHandler is registered with the Workbench. The statusHandlerProductBinding contribution tells the Workbench that the status handler we just defined is bound to our product with the id "productId". The status handling facility uses lazy initialization, so the handler will be instantiated during first access to the facility in the product.

The class defined by the class attribute extends org.eclipse.ui.statushandlers.AbstractStatusHandler. The following is sample code that simply writes to the console the message set in the status adapter.

public void handle(final StatusAdapter statusAdapter, int style) {

System.out.println(statusAdapter.getStatus().getMessage());

}

When you run the product with the id defined in the statusHandlerProductBinding you will start to use your product handler. For example, if you are self hosting, create a new runtime-workbench launch configuration, choose the "Run a product" option, and select productId from the dropdown.

20.5. Dynamic content

What is dynamic content?

Several user assistance components allow you to provide dynamic content. This means the content can change depending on the user's environment or context. For example, you may want a section of your help document or welcome page to only appear if the user has a certain functionality enabled.

There are two basic ways to create dynamic content. One is to add tags and attributes (annotations) to your XML markup instructing the system to filter out sections or documents, include sections from other documents, or extend documents by contributing into anchors or replacing elements.

The other method is to write your own content producer by plugging in some Java code that will write the content from scratch or process existing static documents on-the-fly. Let's examine both approaches.

XML Annotations

The simplest way to create dynamic content is to use the XML markup annotations. You simply add some special attributes and tags to your XML markup and the system will process the instructions. You can use this markup to:

Filter out documents or parts of documents

Include sections from other documents, and

Extend documents using anchors, contributions, and replacements.

To learn more about each aspect of the markup, follow the links above or find the topics in the table of contents.

This is demonstrated in the XHTML fragment shown below:

<p>

This is a static paragraph.

</p>

<!-- This tag will be replaced with the referenced tag -->

<include path="my.plugin.id/path/my\_other\_document.xhtml/element\_id"/>

<ul>

<li>

This is a static list item.

</li>

<li>

This only shows when running on Windows and plugin com.myplugin is not installed.

<enablement>

<systemTest property="osgi.os" value="win32"/>

<not>

<with variable="platform">

<test property="org.eclipse.core.runtime.isBundleInstalled" args="com.myplugin"/>

</with>

</not>

</enablement>

</li>

</ul>

<!-- An element with an id that can be replaced by another element -->

<p id="my.paragraph">

This paragraph can be replaced because it has an id.

</p>

<!-- A place where others can add content -->

<anchor id="my.anchor.id"/>

Content producers

A content producer is a mechanism for plugging-in Java code to produce the document content on-the-fly. Content producers are more powerful than XML annotations, but are more complex to use.

Content producers are supported in the following areas of user assistance:

Welcome XML/XHTML

Help topics (any format)

Help table of contents

Help keyword index

Context-sensitive help

20.5.1. Filters

If some sections of a document (e.g. a list item in a XHTML help topic) should only be displayed to the user if certain conditions are met, you can specify an enablement expression declaring the criteria that must be met in order to display the element.

Filters are specified by adding an <enablement> element as a child of the element that should conditionally be filtered. The syntax used is core expressions, which is the same syntax used to filter menu contributions, etc from the UI.

System tests

Expressions check criteria by performing tests. One type of test is a system test, which tests a system property against an expected value. Some common system properties to test by are listed below:

Property

Meaning

Possible values

osgi.os

operating system

win32, win32, linux, macosx, aix, solaris, hpux, qnx

osgi.ws

windowing system

win32, motif, gtk, photon, carbon

osgi.arch

processor architecture

x86, x86\_64, ia64, ia64\_32, ppc, PA\_RISC, sparc

Here are a few examples of system tests:

<p>

This paragraph should only be displayed on Windows.

<enablement>

<systemTest property="osgi.os" value="win32"/>

</enablement>

</p>

<p>

This paragraph should \*not\* be displayed on GTK.

<enablement>

<not>

<systemTest property="osgi.ws" value="gtk"/>

</not>

</enablement>

</p>

<p>

This paragraph should only be displayed on PowerPC Macs.

<enablement>

<systemTest property="osgi.os" value="macosx"/>

<systemTest property="osgi.arch" value="ppc"/>

</enablement>

</p>

Note: When several sub-expressions are listed with no boolean operator, they are by default ANDed together. See the complete expressions syntax specification for more details.

Property tests

In addition to system tests, you can test any property of an available object as long as there is a property tester available for it.

Tests always perform their test on a property of some object, and each object has different properties you can test. User assistance provides two variables that are used to select the object to test on:

platform - The underlying platform you're running on

workbench - The UI workbench

You can perform a test on one of these by using the <with> element, as shown below:

<enablement>

<with variable="platform">

<test property="x" value="y"/>

</with>

</enablement>

Each property has a namespace, which is a prefix such as org.eclipse.ui that is used to minimize the chances of duplicate properties being defined by two components. The table below shows some common properties you can test by:

Property

Object (variable)

Meaning

org.eclipse.core.runtime.product

platform

tests if the expected value matches the currently active product's unique id

org.eclipse.core.runtime.isBundleInstalled

platform

tests if the bundle with the symbolic name (e.g. org.eclipse.help) specified as a single argument is installed on the platform

org.eclipse.ui.isActivityEnabled

workbench

tests if the activity with the id specified as a single argument exists and is currently enabled in the workbench

org.eclipse.ui.isCategoryEnabled

workbench

tests if the category of activities with the id specified as a single argument exists and is currently enabled in the workbench. A category is enabled if all its activities are enabled

Here are a few examples using these properties:

<p>

This paragraph should only be displayed when running the Eclipse SDK.

<enablement>

<with variable="platform">

<test property="org.eclipse.core.runtime.product" value="org.eclipse.sdk.ide"/>

</with>

</enablement>

</p>

<p>

This paragraph should only be displayed when either the com.myactivity.a or

com.myactivity.b activity (or both) is enabled in the workbench.

<enablement>

<with variable="workbench">

<or>

<test property="org.eclipse.ui.isActivityEnabled" args="com.myactivity.a"/>

<test property="org.eclipse.ui.isActivityEnabled" args="com.myactivity.b"/>

<or>

</with>

</enablement>

</p>

<p>

This paragraph should only be displayed when the com.mycategory category

is enabled and the com.mybundle bundle is not installed.

<enablement>

<with variable="workbench">

<test property="org.eclipse.ui.isCategoryEnabled" args="com.mycategory"/>

</with>

<with variable="platform">

<not>

<test property="org.eclipse.core.runtime.isBundleInstalled" args="com.mybundle"/>

</not>

</with>

</enablement>

</p>

Note: Make sure you use the value and args attributes for the appropriate tests. In general, string properties like product will use value where boolean tests (isSomethingTrue) will use args. See the complete expressions syntax specification for more details on how to write expressions.

Defining your own test

The core expressions framework allows you define your own test that can test on any arbitrary property of any object that is accessible through the variables defined for the context you're in. You do this by defining a property tester.

Note: The variables platform and workbench resolve to the org.eclipse.core.runtime.Platform Class object (we use the class object for "static" tests) and the singleton org.eclipse.ui.IWorkbench instance, respectively. Your property tester must declare one of these two classes for its type.

Using Filters in Table of Contents (TOC) Files

Filters can be used in Table of Contents files to filter out content that does not apply to the current installation. The example below is a topic which is only included in the TOC if plugin x.y.z is running.

<topic href="html/subtopic.html" label="This topic is shown only if plugin x,y.z is running">

<enablement>

<with variable="platform">

<test property="org.eclipse.core.runtime.isBundleInstalled" args="x.y.z"/>

</with>

</enablement>

</topic>

Filter attributes/elements (deprecated)

Prior to the 3.3, filters were specified using filter attributes or elements. The use of these filters is now deprecated, and you should use expressions (described above) instead.

The table below contains a complete list of all the filter properties and their possible values for use with filter elements and attributes.

(deprecated)

Property

Meaning

Possible Values

os

operating system

win32, win32, linux, macosx, aix, solaris, hpux, qnx

ws

windowing system

win32, motif, gtk, photon, carbon

arch

processor architecture

x86, x86\_64, ia64, ia64\_32, ppc, PA\_RISC, sparc

product

eclipse product identifier

Any product identifier (e.g., for SDK, org.eclipse.sdk.ide)

plugin

plug-in presence

Any plug-in identifier (e.g. org.eclipse.help)

category

category of activities

Any activity category identifier (e.g. for Team category, org.eclipse.categories.teamCategory)

activity

activity (capability)

Any activity identifier (e.g. for CVS Support activity, org.eclipse.team.cvs)

If the name does not match any pre-defined property, the help system will use the JVM's system property of that name. For example, you can pass in any user-defined property at launch, such as -Dlocation=paris,france and filter by that property.

There are two ways to specify filters on an element; using attributes, or elements.

Filter Attribute (deprecated)

The first form is to add a filter attribute to the element. The general form is:

<element filter="[name][operator][value]">

Some text.

</element>

The name is the name of the property by which to filter, for example, os for operating system. The operator is either = to denote a match (exact match, case sensitive), or != to denote does not match. The value is what the property should (or shouldn't) match. For example, for os, one of the possible values is win32 (Windows). A complete list of filter properties and their values is available in a table below.

The example below shows how to display a paragraph of text in an XHTML document when running on Linux only.

<p filter="os=linux">

This message will only appear when viewed on Linux.

</p>

In this second example, the link will only appear when plugin com.my.plugin is not installed:

<a href="../topic/org.eclipse.platform.doc.isv/guide/..." filter="plugin!=com.my.plugin">

Click here to download plugin com.my.plugin.

</a>

Filter Element (deprecated)

The second form is to use a filter element as a child of the element you wish to filter. This form is slightly longer than the attribute form, but it is more powerful because you can specify any number of filters on an element. The general form is:

<element attribute="value">

<filter name="[name]" value="[modifier][value]"/>

</element>

The name and value here are the same as with the attribute. However, since they are separated, we need another way to specify whether or not it should match. By default, if you do not provide a modifier, match is assumed. If it should not match, set the modifier to "! "

Here is the first example shown above in the second form:

<p>

<filter name="os" value="linux"/>

This message will only appear when viewed on Linux.

</p>

And the second example:

<a href="../topic/org.eclipse.platform.doc.isv/guide/...">

<filter name="plugin" value="!com.my.plugin"/>

Click here to download plugin com.my.plugin.

</a>

Information Center

Filtering support is turned off by default when running help in information center mode, causing all content, including filtered content, to be visible. If you intend to host your documentation in both workbench and information center modes, you should use filters in a way that makes sense even if filtering is turned off. If you wish to turn on filtering in an information center set the customization preference filterInfocenter to true, see help system customization.

Where can I use filters?

Filtering can be used in any XML-based user assistance document, such as help XHTML topics, help table of contents, welcome pages, cheat sheets, etc. You cannot use filtering in HTML documents.

In any case, you must not place filters on any element where removing that element would result in invalid XML. For example, you should not place a filter on the html element in XHTML, because without that element it is no longer valid XHTML.

20.5.2. Includes

Includes are used include parts of other documents into your document. The markup specifies the document and element to include, and before the document is shown to the user, the include element will be replaced with the element referred to.

For example, you may wish to show a copyright at the end of each help document you provide. You could copy and paste it in each document, but if the copyright changes (e.g. the copyright year), you have to modify every one of your documents.

Instead, you can place the copyright in a separate XHTML file called copyright.xhtml and assign it a unique identifier, like this:

<p id="copyright">

Copyright 2006 My Company. All rights reserved.

</p>

Then for each document, simply include the copyright paragraph:

<p>

This is my document. It should have a copyright at the end.

</p>

<include path="my.plugin.id/path/copyright.xhtml/copyright"/>

Before the document is sent to the browser to be displayed, the include element will be replaced by the copyright paragraph.

Where can I use includes?

See below for a list of the document types in which includes can be used. The format of the path element of the include depends on the type of document you're including from. As you can see in the example above, the format for XHTML help documents is "<plugin\_id>/<path\_to\_file>/<element\_id>". Refer to the links below for the format for your document type.

Welcome/Intro XML/XHTML format

Help XHTML format

20.5.3. Content extensions

A content extension is either a contribution of content into a known anchor (similar to an extension into an extension point), or the replacement of an existing element in a document.

Contributing into anchors

Anchor elements (e.g. <anchor id="my\_anchor"/>) can be specified in any user assistance XML document, and represent places at which other components are allowed to extend this document. Specifying an anchor is as simple as adding the element in your markup.

To contribute content into an anchor, you must write the content to be added in a separate XML file and hook it into the platform using one of two extension points. For help or cheat sheet content, you must use the org.eclipse.help.contentExtension extension point. For welcome (intro) extensions, use org.eclipse.ui.intro.configExtension. The table of contents markup also allows the special constructs link and link\_to which perform a similar function to includes and contributions, but perform the additional operation of merging extra documents to index for searching.

For example, let's say component A's documentation provides a listing of file formats it supports. If component B extends A's support to several more formats, you can place an anchor in A's list, and extend the list from B. For example:

<p>The following list shows the supported formats:</p>

<ul>

<li>Portable Network Graphics (.png)</li>

<li>Joint Photographic Experts Group (.jpeg)</li>

<li>Graphical Interchange Format (.gif)</li>

<!-- Extra formats go here -->

<anchor id="image\_format\_list"/>

</ul>

Then component B can make a contribution to the anchor to add more formats to the list by specifying the following extension:

<extension point="org.eclipse.help.contentExtension">

<contentExtension

file="path/to/extension.xml"/>

</extension>

Where the file path/to/extension.xml might contain:

<contentExtension>

<contribution

content="docs/mydoc.xhtml#mycontent"

path="/component\_a\_plugin/docs/doc\_with\_anchor.xhtml#anchorId">

</contribution>

</contentExtension>

The element with the id attribute mycontent in document docs/mydoc.xhtml would contain the content you wish to contribute into the anchor. The result will be that when the user views the document, the extra content will appear at the anchor as though it was explicitly added into the original document.

Replacing content

Any XML element in a user assistance document that has an id attribute that is unique to that document can be replaced with any other element.

As with contributions into anchors, to replace an element, you must write the new element to replace with in a separate XML file and hook it into the platform using one of two extension points. For help or cheat sheet content, you must use the org.eclipse.help.contentExtension extension point. For welcome (intro) extensions, use org.eclipse.ui.intro.configExtension.

The only difference with replacements as opposed to contributions into anchors is the name of the XML element used in the markup; use the replacement element instead of contribution (or replacementContent in the case of welcome). For example:

<contentExtension>

<replacement

content="docs/mydoc.xml#myelement"

path="/plugin.id/path/doc.xml#elementId">

</replacement>

</contentExtension>

(Note the markup is different for welcome, as specified in the org.eclipse.ui.intro.configExtension extension point)

Path format

The format of the path attribute is /pluginId/path/file.xml#elementId, except welcome where it is pageId/path/to/elementId where the pageId is the welcome page id, and all other path segments are ids of container elements like groups, leading to the target element.

21. Provisioning platform (p2)

Installation and updates are managed in Eclipse using a provisioning platform called p2. Fundamentally, p2 is a technology for provisioning and managing Eclipse- and Equinox-based applications. You can use p2 to install or manage any aspect of your application, from the physical plug-ins and native code, to the configuration of the installed software - file permissions, command line arguments, etc. Installation with p2 does not consist simply in adding or removing files in the file system, but more generally the sequence of events that must occur to lay down and configure a system that is ready to run. Core aspects of p2 include:

Automatic resolution of dependencies between software components. With p2, you state the root set of items you want installed or uninstalled, and p2 computes the complete set of required system changes automatically.

Transport of software components and configuration data from remote repositories to the system being installed. p2 includes sophisticated algorithms for performing multi-threaded transfers, including support for mirrors and automatic re-balancing of transfers based on network state (availability, throughput).

An extensible mechanism for "teaching" p2 how to install and configure various kinds of software. By default p2 knows how to install and configure Eclipse plug-ins, and basic native integration such as adding/removing files, setting permissions, and creating symbolic links.

A graphical user interface integrated into the Eclipse platform, to allow end users to examine and manage the application.

A suite of command line tools and Ant tasks, to allow developers and release engineers to build and configure p2-enabled applications.

The following documentation sections describe how to use p2 to build and deliver your own Eclipse-based applications and extensions.

Metadata management

Installation management

Repository management

21.1. API overview

This section provides an overview of the p2 API and introduces some of the key concepts that should be understood to work with the p2 API. The API can be generally thought of as three layers of API. The top most layer requires understanding of very few concepts, and each successive layer provides more flexibility, but is a bit more complex.

The User Interface API

The Operations API

The Core API

This section provides a description of each of these categories and introduces some of the key concepts.

The User Interface API

The UI provides wizards for installing, updating, and uninstalling software in a system. It also provides dialog pages for describing the installation and manipulating the repositories that are used to access software. Most of these building blocks are private ("black box") implementations, not intended to be extended by clients. A small package, org.eclipse.equinox.p2.ui, defines the API, which provides hooks for customizing the behaviour of the UI components, and class definitions that can be used in a plug-in's UI contributions.

Applications that simply want to provide their users a way to update or install additional software into the system need only include the relevant UI bundles. No deep knowledge of the p2 API is required. Customizing the p2 UI describes this level of working with the p2 UI. The The Policy class defines aspects of the UI that can be customized.

Operations API

The operations API provides high-level API for installing, updating, and uninstalling software in a headless system. The operations API is defined with "progressive disclosure" in mind. That is, the simplest and most common operations require little knowledge of underlying concepts.

Clients work with the operation in two passes. The operation must first be resolved, which means that the requirements and dependencies are calculated to determine if the desired operation is compatible with what is already installed in the system. If the operation can be resolved, then the actual provisioning work (downloading of artifacts, updating the system) can be performed. The two-pass nature of an operation allows the resolution status to be reported to a client to determine whether the operation should proceed. For example, the client can determine if the resolution is acceptable, or examine the detailed plan of what needs to be installed in order to continue with the operation. The resolution and provisioning phases can be performed synchronously or in the background as a job.

A simple example can help demonstrate the simplicity of operations. A common operation in many applications is to simply update the application to get the latest version of everything. This is possible with very little code. The following snippet shows the sequence for updating everything in the running application.

// create an operation. We already have a provisioning agent (to be explained later)

UpdateOperation op = new UpdateOperation(new ProvisioningSession(agent));

// resolve the operation to see if we can update

IStatus result = op.resolveModal(new NullProgressMonitor());

if (result.isOK()) {

// get a job that will perform the actual work and schedule it

op.getProvisioningJob(monitor).schedule();

}

If there is a problem resolving an operation, or the client code wants to do something slightly different, then a little more code might be needed. For example, an operation can be configured to update an application that is not the running application, or to consult a specific list of software repositories for the update.

Inside the operation, the underlying p2 IPlanner is performing the resolution phase, and the underlying p2 IEngine is performing the actual install. However, these subsystems do not need to be understood by a client working with operations.

For more information about operations, consult the javadoc for ProfileChangeOperation. Code snippets for working with operations can be found in the javadoc for InstallOperation, UpdateOperation, and UninstallOperation.

The Core API

The core API contains all the subsystems on which the Operation and UI APIs are built. Some of these constructs feed into the other API layers. For example, we saw above that an agent was needed in order to build an update operation. Now we'll take a look at these core concepts.

The Provisioning Agent

All access to the p2 API happens through the agent, IProvisioningAgent. The agent is the starting point of everything. One way to think about the IProvisioningAgent is that it is an "executable" representation of the p2 area (e.g. the p2 folder at the root of an eclipse installation). Among other things, the agent can be used to acquire p2 services for managing repositories, creating provisioning plans and perform installation requests.

The provisioning agent is acquired using the IProvisioningAgentProvider. This is generally done in one place in the client code, with the rest of the code simply accessing the agent through some variable or helper code. The following snippet shows how to acquire the provisioning agent.

ServiceReference sr = Activator.getContext().getServiceReference(IProvisioningAgentProvider.SERVICE\_NAME);

IProvisioningAgentProvider agentProvider = null;

if (sr == null)

return;

agentProvider = (IProvisioningAgentProvider) Activator.getContext().getService(sr);

IProvisioningAgent agent = agentProvider.createAgent(new URI("file:/Applications/eclipse36/p2"));

Accessing all p2 services through an agent allows for multiple instances of p2 to be running in isolation in the same VM. Note that the client creating the agent is responsible for destroying it.

Metadata

The Installable Unit (IU) is a chunk of metadata describing something that is installable. An IU is used throughout the API to describe something that is being installed, updated, or removed. An IU describes something that can be installed, including the name, description, license, copyright information, installation processing steps, and the requirements that must be satisfied. The IInstallableUnit javadoc describes all of the things one can do with an IU. You will notice that you can't change the properties of an IU. That is because an IU is immutable. Once created, it should never change.

An IU can be obtained either by querying a source of metadata (e.g. a repository) or by creating one programmatically.

Queries and Queriables

Every source of metadata is usually queryable (see org.eclipse.equinox.p2.query.IQueryable). To discover an IU, you can execute a query (see IQuery) against a metadata source. The result of a query is a collection of all of the IUs that meet the criteria of the query.

Queries can be created in multiple ways. The simplest way is to create a query using the QueryUtil. The following snippet creates a query that searches for all IUs that have the ID "org.eclipse.jdt":

QueryUtil.createIUQuery("org.eclipse.jdt");

Depending on the specificity of the query, there may be one or many IUs that satisfy the query. For example, you could query for a specific version of a specific IU, or you could use a wildcard to query for IUs that match a particular pattern. QueryUtil has API for retrieving the most commonly used queries. Additional queries that are OSGi specific can be found in the package org.eclipse.equinox.p2.eclipse.touchpoint. Finally, should you need to write more complex queries, p2 comes with a query language called p2 QL

Repositories and Repository Managers

There are two main types of repositories, Metadata Repositories and Artifact Repositories. Metadata repositories hold metadata (Installable Units), while artifact repositories hold "artifacts" (the actual downloadable bytes that make up an install). Repositories can be remote or local. They can be edited and queried. The javadoc for IMetadataRepository and IArtifactRepository describe the repository API in more detail.

Repositories are managed (created, loaded, removed, cached, etc...) using a Repository Manager. The repository manager can be acquired using the provisioning agent.

The following snippet shows how to acquire the metadata repository manager using the agent, and subsequently load the Helios repository.

IMetadataRepositoryManager manager = (IMetadataRepositoryManager) agent.getService(IMetadataRepositoryManager.SERVICE\_NAME);

IMetadataRepository repository = manager.loadRepository(new URI("http://download.eclipse.org/releases/helios"), new NullProgressMonitor());

Profiles and Profile Registries

A p2 profile tracks the set of software that composes the executable application. For example, your Eclipse Install has a profile that contains all the IUs that that comprise Eclipse. When you attempt to install new IUs, p2 modifies your current profile. If the new IUs conflict with your existing profile (or dependencies cannot be resolved), then p2 will report an error and the installation will not proceed. See the IProfile javadoc for a complete list of the Profile APIs.

The profile is managed by a profile registry. IProfileRegistry manages all the profiles for a given p2 agent. It can be acquired through the agent.

Putting it all together

The following snippet demonstrates everything that must be done to trigger the installation of an IU into the running application. We use the operations API to perform the install, so we don't have to work with the profile, planner, or engine subsystems. However, we do need to know enough about the core API to obtain an agent, and get some IUs from a repository. The operation manages the rest of the detail.

//get the agent

ServiceReference sr = Activator.getContext().getServiceReference(IProvisioningAgentProvider.SERVICE\_NAME);

IProvisioningAgentProvider agentProvider = null;

if (sr == null)

return;

agentProvider = (IProvisioningAgentProvider) Activator.getContext().getService(sr);

IProvisioningAgent agent = agentProvider.createAgent(new URI("file:/Applications/eclipse36/p2"));

//get the repository managers and define our repositories

IMetadataRepositoryManager manager = (IMetadataRepositoryManager) agent.getService(IMetadataRepositoryManager.SERVICE\_NAME);

IArtifactRepositoryManager artifactManager = (IArtifactRepositoryManager) agent.getService(IArtifactRepositoryManager.SERVICE\_NAME);

manager.addRepository(new URI("file:/Users/Pascal/tmp/demo/"));

artifactManager.addRepository(new URI("file:/Users/Pascal/tmp/demo/"));

//Load and query the metadata

IMetadataRepository metadataRepo = manager.loadRepository(new URI("file:/Users/Pascal/tmp/demo/"), new NullProgressMonitor());

Collection toInstall = metadataRepo.query(QueryUtil.createIUQuery("org.eclipse.equinox.p2.demo.feature.group"), new NullProgressMonitor()).toUnmodifiableSet();

//Creating an operation

InstallOperation installOperation = new InstallOperation(new ProvisioningSession(agent), toInstall);

if (installOperation.resolveModal(new NullProgressMonitor()).isOK()) {

Job job = installOperation.getProvisioningJob(new NullProgressMonitor());

job.addJobChangeListener(new JobChangeAdapter() {

public void done(IJobChangeEvent event) {agent.stop()}});

job.schedule();

}

21.2. Metadata management

As described in the overview, p2 provides facilities for software dependency management, and for performing all of the necessary steps to get an application physically installed and configured into an end user's system. The information that describes the dependencies between application components, and the steps required to properly configure a running system is called the p2 metadata.

In many cases this metadata can be computed directly from the information in your plug-in and feature manifests, and an extra step is simply required to publish this data into a format suitable for consumption by p2. In other cases, a plug-in or application developer may need to author or customize the p2 metadata for their software. The following sections describe in more detail how to author and publish p2 metadata for your application or plug-in.

Publishing metadata

Customizing metadata

Ant Tasks for publishing metadata

21.2.1. Publishing p2 metadata

The p2 metadata contains useful information about the elements that can be installed. These elements are called Installable Units (IUs). Among other things, the IUs describe dependencies, properties and configuration information.

There are three different ways p2 repositories can be created. 1) By using the export wizard, 2) using PDE Build, and 3) using the publisher. The Plug-in Development Environment Guide explains how the feature export wizard, product export wizard and PDE Build can be used to generate metadata. The remainder of this document explains how the publisher can be used.

The publisher is the means by which deployable entities get added to repositories. For example, the publisher can be used to create an IU from an OSGi Bundle or Eclipse Feature. The publisher consists of an extensible set of publishing actions, applications and Ant tasks that allow users to generate p2 repositories from a number of different sources.

In order to be backwards-compatible, p2 is able to install things from old-style update sites and extension locations by generating metadata for these things on-the-fly, but this of course is not the optimal situation. Ideally, plug-in developers should create p2 metadata when they produce their bundles, features, and products.

This document describes how to publish p2 metadata for your software using the publisher.

The publisher can be used in two ways:

Command line applications

Ant Tasks

Command Line Applications

p2 ships with four command line publisher applications that plug-in developers can run. The applications are contained in org.eclipse.equinox.p2.publisher and org.eclipse.equinox.p2.updatesite. These bundles are part of the Eclipse SDK. The four applications are:

UpdateSite Publisher

Features And Bundles Publisher

Product Publisher

Category Publisher

UpdateSite Publisher

The updatesite publisher application (org.eclipse.equinox.p2.publisher.UpdateSitePublisher) is a command line application that is capable of generating metadata (p2 repositories) from an update site containing a site.xml, bundles and features. The updatesite publisher can be invoked using the generic Eclipse launcher format as follows:

eclipse -application org.eclipse.equinox.p2.publisher.UpdateSitePublisher <publisherArgs>

Here are the supported command-line options:

-metadataRepository <URI>

the URI to the metadata repository where the installable units should be published

-artifactRepository <URI>

the URI to the artifact repository where the artifacts should be published

-source <path>

the location of the update site

-compress

a flag indicating that the repository should be compressed

-append

flag indicating that repositories will be appended to as opposed to over-written

-publishArtifacts

flag indicating that the artifacts should be published (copied) to the repository. When this flag is not set, the actual bytes underlying the artifact will not be copied, but the repository index will be created. When this option is not specified, it is recommended to set the artifactRepository to be in the same location as the source (-source).

Here is an example of how the updatesite publisher can be used to create a compressed p2 repository from an update site:

eclipse -application org.eclipse.equinox.p2.publisher.UpdateSitePublisher

-metadataRepository file:/<some location>/repository

-artifactRepository file:/<some location>/repository

-source /<location with a site.xml>

-configs gtk.linux.x86

-compress

-publishArtifacts

Once you run the publisher, you will see the generated artifact repository index file in artifacts.xml and the generated metadata repository index file in the content.xml (or artifacts.jar and content.jar if you used the -compress option). There is nothing forcing you to have the metadata and artifact repositories co-located, but you can do this if you wish to.

Features And Bundles Publisher

The features and bundles publisher application (org.eclipse.equinox.p2.publisher.FeaturesAndBundlesPublisher) is a command line application that is capable of generating metadata (p2 repositories) from pre-built Eclipse bundles and features. The features and bundle publisher can be invoked using the generic Eclipse launcher format as follows:

eclipse -application org.eclipse.equinox.p2.publisher.FeaturesAndBundlesPublisher <publisherArgs>

Here are command-line options:

-metadataRepository <URI>

the URI to the metadata repository where the installable units should be published

-artifactRepository <URI>

the URI to the artifact repository where the artifacts should be published

-source <path>

the location of the update site

-bundles <path>

the location of the bundles

-features <path>

the location of the features

-compress

a flag indicating that the repository should be compressed

-append

flag indicating that repositories will be appended to as opposed to over-written

-publishArtifacts

flag indicating that the artifacts should be published (copied) to the repository. When this flag is not set, the actual bytes underlying the artifact will not be copied, but the repository index will be created. When this option is not specified, it is recommended to set the artifactRepository to be in the same location as the source (-source).

The features and bundles publisher application can be invoked using the -source option by pointing at a directory that contains two sub-directories (features and plug-ins). The publisher can also be invoked by pointing at the bundles and features separately and using the -bundles and -features options.

Here is an example of how the features and bundles publisher can be used to create a compressed p2 repository from a collection of bundles and features:

eclipse -application org.eclipse.equinox.p2.publisher.FeaturesAndBundlesPublisher

-metadataRepository file:/<some location>/repository

-artifactRepository file:/<some location>/repository

-source /<location with a plugin and feature directory>

-configs gtk.linux.x86

-compress

-publishArtifacts

Product Publisher

The product publisher application (org.eclipse.equinox.p2.publisher.ProductPublisher) is a command line application that is capable of generating metadata (p2 repositories) from a .product file. The product publisher does not generate metadata for the bundles and features that make up the product. The product publisher can be invoked using the generic Eclipse launcher format as follows:

eclipse -application org.eclipse.equinox.p2.publisher.ProductPublisher <publisherArgs>

Here are the supported command-line options:

-metadataRepository <URI>

the URI to the metadata repository where the installable units should be published

-artifactRepository <URI>

the URI to the artifact repository where the artifacts should be published

-productFile <path>

the location of the product file

-executables <path>

the location of the executables feature

-flavor <String>

the flavor used for the configuration units (if unsure, using tooling)

-compress

a flag indicating that the repository should be compressed

-append

flag indicating that repositories will be appended to as opposed to over-written

-configs <spec>

a set of environment properties for the os, ws and arch parameters of the system which the product supports and can be provisioned on; can specify more than one platform in a comma-separated list of such triplets; if the product can support any environment, "-configs ANY" or "-configs ANY.ANY.ANY" should be used.

Here is an example of how the product publisher can be used to create a p2 repository from a .product file:

eclipse -application org.eclipse.equinox.p2.publisher.ProductPublisher

-metadataRepository file:/<some location>/repository

-artifactRepository file:/<some location>/repository

-productFile /<location>/<filename>.product

-append

-executables /<deltapack parent>/delta/eclipse/features/org.eclipse.equinox.executable\_3.3.200.v20090426-1530-7M-Fm-FI3UouOdcoUJz-7oc

-flavor tooling

-configs gtk.linux.x86

Category Publisher

The category publisher application (org.eclipse.equinox.p2.publisher.CategoryPublisher) is a command line application that is capable of categorizing a set of Installable Units in a given repository. The categorization is driven from a category file. The category publisher can be invoked using the generic Eclipse launcher format as follows:

eclipse -application org.eclipse.equinox.p2.publisher.CategoryPublisher <publisherArgs>

Here is an example of how the product publisher can be used to categorize a p2 repository from a category.xml file:

eclipse -application -application org.eclipse.equinox.p2.publisher.CategoryPublisher

-metadataRepository file:/<some location>/repository

-categoryDefinition file:/<some location>/category.xml

-compress

Categories are always appended to a repository.

Updated Documentation

Up-to-date information on the p2 publisher can be found on the Eclipse p2 wiki.

21.2.2. Ant tasks for publishing p2 metadata

p2 ships with two publisher ant tasks. The ant tasks are contained in org.eclipse.equinox.p2.publisher. This bundles is part of the Eclipse SDK.

Default Attributes

The p2.publish.\* ant tasks outlined below all support the following attributes:

metadataRepository

A URL specifying the metadata repository to publish to.

artifactRepository

A URL specifying the artifact repository to publish to.

repository

Sets both metadataRepository and artifactRepository.

metadataRepositoryName

When creating a new metadata repository, sets the name.

artifactRepositoryName

When creating a new artifact repository, sets the name.

repositoryName

Sets both metadataRepositoryName and artifactRepositoryName.

append

Whether to append to the repository. (Default is "true")

compress

When creating a new repository, whether or not to compress the metadata. (Default is "false")

publishArtifacts

Whether or not to publish the artifacts. (Default is "true")

reusePackedFiles

Whether or not to include discovered Pack200 files in the repository. (Default is "false")

<contextRepository>

Nested elements specifying context repositories, supports the following attributes:

location

A URL specifying the location of the repository.

artifact

"true" or "false": whether or not there is an artifact repository at this location.

metadata

"true" or "false": whether or not there is a metadata repository at this location.

If a given context repository contains metadata for one of the features or bundles that are being published, then that metadata will be re-used instead of generating new metadata.

p2.publish.featuresAndBundles

This task will publish metadata for pre-existing binary features and plug-ins.

This task supports the following attributes and elements:

source

A folder containing plugins and features subfolders to publish.

<features>

A nested fileset element specifying the locations of binary features to publish.

<bundles>

A nested fileset element specifying the locations of binary plug-ins to publish.

p2.publish.product

Publish a .product file. This task assumes everything included in the product already exists in the repository. (That is, all features and bundles have been previously published.)

This task supports the following attributes:

flavor

Set the flavor for the p2 metadata, default is "tooling". Products should consider using a unique flavor if they have special requirements for bundle start levels.

productFile

The location of the .product file describing the product.

executables

The location of the executables feature. This is the feature that is used for branding and publishing the executables.

<config>

Nested elements specifying configurations supported by this product. Config elements specify ws, os and arch:

<config ws="gtk" os="linux" arch="x86"/>

Use <config ws="ANY" os="ANY" arch="ANY"/> if the product can support any platform.

<advice>

Nested elements specifying specifying additional advice to use when creating the product. Currently the accepted kinds of advice are "featureVersions" and "pluginVersions".

<advice kind="featureVersions" file="finalFeaturesVersions.properties" />

<advice kind="pluginVersions" file="finalPluginsVersions.properties" />

PDE/Build will generate these version properties files when the builder sets the property generateVersionsLists".

In addition to the publisher tasks, metadata can be generated as part of PDE Build.

21.2.3. Customizing p2 metadata

Disclaimer: Authoring p2 metadata is something we expect to support with better tooling in a future release. The support for customizing metadata with a p2.inf file is provisional although we expect to maintain a basic level of compatibility for common tasks.

On occasion the metadata that is automatically generated by p2 for bundles, features and products does not provide everything required to successfully provision an installable unit. For those cases p2 supports the use of an advice file (a p2.inf file) that can be used to augment the metadata for an installable unit. In this release the p2.inf file will allow an author to customize capabilities, properties, and instructions. In addition support is provided for defining additional installable units that are related to the container IU.

The touchpoint advice file is a java properties file and can be placed:

In bundles (META-INF/p2.inf): The instructions are added to the installable unit for the bundle.

In features (a p2.inf file co-located with the feature.xml): The instructions are added to the installable unit for the feature group.

In products (a p2.inf file co-located with the .product file): The instructions are added to the root installable unit for that product.

Version substitution is a common practice and two special version parameters are supported:

$version$ - returns the string form of the containing IU's version

$qualifier$ - returns just the string form of the qualifier of the containing IU's version

One common use for advice files is to allow more fine grained control of the generation of categories used by the UI to group installable features

Capability Advice

Note: Capability advice will "replace" an existing capability of the same type on the IU if the name and namespace match.

There are three different type of capability advice:

provides - these are capabilities that an IU will offer to satisfy the needs of other IUs.

requires - these are the capabilities that an IU requires from other IUs in order to resolve correctly.

metaRequirements - these are capabilities that the IU puts on the profile that must already be installed before this IU can be installed.

provides.{#}.namespace = {namespace}

provides.{#}.name = {name}

provides.{#}.version = {version} (optional / default: 1.0.0)

requires.{#}.namespace = {namespace}

requires.{#}.name = {name}

requires.{#}.range = {range} (optional / default: 0.0.0)]

requires.{#}.matchExp = {p2QL expression} (note that in this case the namespace, name and range attributes are not used)

requires.{#}.greedy = {true|false} (optional / default: true)

requires.{#}.optional = {true|false} (optional / default: false)

requires.{#}.multiple = {true|false} (optional / default: false)

requires.{#}.min = {integer}

requires.{#}.max = {integer} (when min and max are specified the optional flag is ignored)

metaRequirements.{#}.namespace = {namespace}

metaRequirements.{#}.name = {name}

metaRequirements.{#}.range = {range} (optional / default: 0.0.0)

metaRequirements.{#}.matchExp = {p2QL expression} (note that in this case the namespace, name and range attributes are not used)

metaRequirements.{#}.greedy = {true|false} (optional / default: true)

metaRequirements.{#}.optional = {true|false} (optional / default: false)

metaRequirements.{#}.multiple = {true|false} (optional / default: false)

metaRequirements.{#}.min = {integer}

metaRequirements.{#}.max = {integer} (when min and max are specified the optional flag is ignored)

Where {#} is an index for the property, {namespace}, and {name} are the associated named strings, {version} and {range} are version and version range strings respectively.

For example:

provides.0.namespace = testNamespace1

provides.0.name = testName1

provides.0.version = 1.2.3.$qualifier$

provides.1.namespace = testNamespace2

provides.1.name = testName2

provides.1.version = $version$

requires.0.namespace = testNamespace1

requires.0.name = testName1

requires.0.range = [1.2.3.$qualifier$, 2)

requires.0.greedy = true

requires.0.optional = true

requires.0.multiple = true

requires.1.namespace = testNamespace2

requires.1.name = testName2

requires.1.range = [$version$, $version$]

requires.1.greedy = false

metaRequirements.0.namespace = testNamespace1

metaRequirements.0.name = testName1

metaRequirements.0.range = [1.2.3, 2)

metaRequirements.0.greedy = true

metaRequirements.0.optional = true

metaRequirements.0.multiple = true

metaRequirements.1.namespace = testNamespace2

metaRequirements.1.name = testName2

metaRequirements.1.range = $version$

metaRequirements.1.greedy = false

Property Advice

properties.{#}.name = {propertyName}

properties.{#}.value = {propertyValue}

Where {#} is an index for the property, {propertyName}, and {propertyValue} hold the name and value strings for the property.

For example:

properties.0.name = testName1

properties.0.value = testValue1

properties.1.name = testName2

properties.1.value = testValue2

Update descriptor advice

The update descriptor advice allows to override the default update descriptor generated by p2. Typically this is useful if an IU has been renamed and automatic update detection is still desired.

update.id = {id of IU}

update.range = {range of the IU being updated}

update.matchExp = {a match expression identifying the IU being updated}. (When this is specified the values of id and range are ignored)

update.severity = {0|1}

Touchpoint Instruction Advice

instructions.{phase} = {raw actions}

instructions.{phase}.import = {qualified action name} [,{qualified action name}]\* (optional)

Where {phase} is a p2 installation phase (collect, configure, install, uninstall, unconfigure, etc). Note:

The {raw actions} will be "appended" to the end of any instructions already being generated.

The qualified action names for the IU's touchpoint type are implicitly imported. All other actions need to be imported.

For example:

instructions.install = \

ln(targetDir:@artifact,linkTarget:foo/lib.1.so,linkName:lib.so);\

chmod(targetDir:@artifact,targetFile:lib/lib.so,permissions:755);

instructions.install.import= \

org.eclipse.equinox.p2.touchpoint.natives.ln,\

org.eclipse.equinox.p2.touchpoint.natives.chmod

Additional information on touchpoint instructions and action syntax can be found here.

Additional Installable Unit Advice

In addition to customizing attributes of the containing IU one can also author additional installable units that work with the container IU. Typically this mechanism is used to author an IU fragment that customizes the containing IU or one of its dependencies.

iu.{#}.id = {identifier}

iu.{#}.version = {version} (optional)

Where {#} is an index for the installable unit, so multiple installable units can be declared. A full range of IU customizations are supported including:

id

version

singleton

copyright

licenses

filter

touchpoint

update

artifacts

properties

provides

requires

metaRequirements

hostRequirements

instructions

To illustrate all the various settings for these customizations here's a more complete example of: (unit.0) a minimal IU and (unit.1) a full featured IU:

units.0.id = testid0

units.0.version = 1.2.3

units.1.id = testid1

units.1.version = 1.2.4

units.1.singleton = true

units.1.copyright = testCopyright

units.1.copyright.location = http://localhost/test

units.1.filter = test=testFilter

units.1.touchpoint.id = testTouchpointId

units.1.touchpoint.version = 1.2.5

units.1.update.id = testid1

units.1.update.range = (1,2)

units.1.update.severity = 1

units.1.update.description = some description

units.1.artifacts.0.id = testArtifact1

units.1.artifacts.0.version = 1.2.6

units.1.artifacts.0.classifier = testClassifier1

units.1.artifacts.1.id = testArtifact2

units.1.artifacts.1.version = 1.2.7

units.1.artifacts.1.classifier = testClassifier2

units.1.licenses.0 = testLicense

units.1.licenses.0.location = http://localhost/license

units.1.properties.0.name = testName1

units.1.properties.0.value = testValue1

units.1.properties.1.name = testName2

units.1.properties.1.value = testValue2

units.1.requires.0.namespace = testNamespace1

units.1.requires.0.name = testName1

units.1.requires.0.range = [1.2.3.$qualifier$, 2)

units.1.requires.0.greedy = true

units.1.requires.0.optional = true

units.1.requires.0.multiple = true

units.1.requires.1.namespace = testNamespace2

units.1.requires.1.name = testName2

units.1.requires.1.range = $version$

units.1.requires.1.greedy = false

units.1.requires.1.optional = false

units.1.metaRequirements.0.namespace = testNamespace1

units.1.metaRequirements.0.name = testName1

units.1.metaRequirements.0.range = [1.2.3.$qualifier$, 2)

units.1.metaRequirements.0.greedy = true

units.1.metaRequirements.0.optional = true

units.1.metaRequirements.0.multiple = true

units.1.metaRequirements.1.namespace = testNamespace2

units.1.metaRequirements.1.name = testName2

units.1.metaRequirements.1.range = $version$

units.1.metaRequirements.1.greedy = false

units.1.metaRequirements.1.optional = false

units.1.provides.0.namespace = testNamespace1

units.1.provides.0.name = testName1

units.1.provides.0.version = 1.2.3.$qualifier$

units.1.provides.1.namespace = testNamespace2

units.1.provides.1.name = testName2

units.1.provides.1.version = $version$

units.1.instructions.configure = addProgramArg(programArg:-startup); addProgramArg(programArg:@artifact);

units.1.instructions.unconfigure = removeProgramArg(programArg:-startup); removeProgramArg(programArg:@artifact);)

units.1.instructions.unconfigure.import = some.removeProgramArg

units.1.hostRequirements.0.namespace = testNamespace1

units.1.hostRequirements.0.name = testName1

units.1.hostRequirements.0.range = [1.2.3.$qualifier$, 2)

units.1.hostRequirements.0.greedy = true

units.1.hostRequirements.0.optional = true

units.1.hostRequirements.0.multiple = true

Category Generation Using p2.inf

The p2 UI allows for hierarchical organization of installable units based on the concept of "categories" where the children of categories are what's installable. On occasion we might want to take finer grained control of the contents of a category and what it contains. For example we might want to support further categorization of a feature's contents to allow individual plug-ins to be installed instead of the more typical features.

To support this we can tag a feature as a category as follows:

properties.1.name=org.eclipse.equinox.p2.type.category

properties.1.value=true

Another possibility is to use "additional IU advice" to create a specialized category IU like this:

units.1.id=my.product.category

units.1.version=1.0.0

units.1.provides.1.namespace=org.eclipse.equinox.p2.iu

units.1.provides.1.name=my.product.category

units.1.provides.1.version=1.0.0

units.1.properties.1.name=org.eclipse.equinox.p2.type.category

units.1.properties.1.value=true

units.1.properties.2.name=org.eclipse.equinox.p2.name

units.1.properties.2.value=My Category Name

requires.1.namespace=org.eclipse.equinox.p2.iu

requires.1.name=my.product

requires.1.range=[1.0.0,1.0.0]

requires.1.greedy=true

21.2.4. Provisioning Actions and Touchpoints

Introduction

A p2 Installable Unit (IU) is installed using the facilities provided by a touchpoint. The IU metadata consists of a reference to the touchpoint (Touchpoint Type) which also defines the version of the touchpoint (i.e an expectancy that it supports a certain set of operations), and describes instructions to execute in various p2 engine phases. The instructions are named after the phases - the phases "install", "uninstall", "configure", "unconfigure" are of interest when authoring, but there are also some internal phases such as "collect" and "checktrust" executed by the engine. Each instruction (e.g. "install") describes a sequence of actions to execute on the referenced touchpoint. Examples of actions are: create and remove directories, change permissions, install and remove bundles. Currently, two touchpoints (native, and eclipse) have been implemented. The native touchpoint has approximately 5 different actions, and the eclipse touchpoint has approximately 20. Most of these actions take parameters.

The instructions are grouped and described in a Touchpoint Data Element. The touchpoint data element uses a Map where the key is the name of a p2 engine phase (such as "install"), and the value is a string representation of a sequence of actions. Using multiple touchpoint data elements is useful as it allows separation between sets of actions for install/uninstall/configure/unconfigure which makes it easier to maintain the meta data.

Actions

Fully Qualified Action Names and Importing Actions

Each action has a fully qualified name; for example "org.eclipse.equinox.p2.touchpoint.eclipse.installBundle" is the FQN for the Eclipse touchpoints "installBundle" action. One can use the short name of an action if the action is imported or if the IUs "touchpoint type" matches the action.

Most bundle IUs use the Eclipse touchpoint type so you may be able to use:

installBundle(bundle:${artifact})

However, if you get an error such as the following, use the fully qualified name.

java.lang.IllegalArgumentException: No action found for: installBundle.

A valid call of this action using FQN might be something like:

org.eclipse.equinox.p2.touchpoint.eclipse.installBundle(bundle:${artifact})

Native Touchpoint Actions

Action Fully Qualified Name:

org.eclipse.equinox.p2.touchpoint.natives.action\_name

action

parameters

description

collect

-

collects all associated artifacts for an IU and places them in a local touchpoint addressable cache. This action may be called explicitly during the collect phase but is typically called as the "default" action called for IUs with the Native Touchpoint Type as part of the main Collect and Sizing phases.

cleanupzip

source, target

removes unzipped files and directories that where unzipped from source into target - i.e. an "undo operation" of an unzip instruction.

unzip

source, target

unzips the source into the target directory. The source can be the special value @artifact, which denotes the download cache location for the first artifact key in the installable unit.

mkdir

path

Creates the directory specified by the parameter path.

rmdir

path

Removes the directory specified by the parameter path. Action has no effect if the referenced directory contains files. Use the remove action for a forced recursive remove.

ln

targetDir, linkTarget, linkName, force

Performs the system action ln -s with the parameters linkTarget being the source-file, targetDir is the directory where the symbolic link will be created, and linkName is the name of the resulting link in the targetDir. The force parameter is a boolean in string form (i.e. "true"/"false") and indicates if an existing link with the same name should be removed before the new link is created.

This action is not available on platforms that do not have this command(i.e. Windows)

chmod

targetDir, targetFile, permissions, options

Changes permission on a file using the system chmod command. The targetDir parameter is either a path, or the special @artifact which is a reference to the directory where the first artifact included in the installable unit is located. The parameter targetFile is the name of a file, and permissions is written like for the chmod system command. The options parameter allows passing additional options like "-R" for recursive operation. If multiple parameters are needed separate them with a space (like on the command line).

This is not available on platforms that do not have this command (i.e. Windows)

remove

path

Removes a file, or a directory (and all files under this directory) as referenced by the parameter path.

(Since 3.5)

copy

source, target, overwrite

Copies a file or a directory (and all of its content) denoted by source path to the target path.

The boolean flag overwrite should be set to true if the copy action should overwrite existing files. If overwrite is false the operation will fail with an IO error in the files already exists.

(Since 3.5)

cleanupcopy

source, target

Cleans up what was installed earlier with a copy from source to target. I.e. this is an "undo" of a copy operation.

(Since 3.5)

checkAndPromptNativePackage

distro, package, [comparator, version]

Prompt the user for the package identified if it is not already installed. distro identifies a distribution for which the package is applicable. package is the package identifier as known by the OS. version an optional string capturing the expected version of the package. comparator an optional string from eq, gt, ge, lt, le to express more specific dependencies on the version to install. Know that at this point only Ubuntu is supported, and we welcome support for other OS.

Example:

org.eclipse.equinox.p2.touchpoint.natives.checkAndPromptNativePackage(distro:debian,package:sudo);

or

org.eclipse.equinox.p2.touchpoint.natives.checkAndPromptNativePackage(distro:debian,package:handbrake,comparator:ge,version:0.9);

(Since 4.5)

checkAndPromptNativePackageWindowsRegistry

package, key, [attributeName, attributeValue, downloadLink, version]

Prompt the user for the Windows library identified in the Windows registry if it is not already installed. package is a descriptive library identifier. version an descriptive version string. key is the full registry key. attributeName and attributeValue are an optional name and value of a registry attribute to check. downloadLink is a an optional URL to show in the UI from where the library can be obtained.

Example:

org.eclipse.equinox.p2.touchpoint.natives.checkAndPromptNativePackageWindowsRegistry(\

package:MS VC Runtime,\

version:2010,\

key:HKLM\\SOFTWARE\\Wow6432Node\\Microsoft\\VisualStudio\\10.0\\VC\\VCRedist\\x64,\

attributeName:Installed,\

attributeValue:1,\

downloadLink:http${#58}//www.microsoft.com/download/details.aspx?id=14632);

(Since 4.5)

Eclipse Touchpoint Actions

Action Fully Qualified Name:

org.eclipse.equinox.p2.touchpoint.eclipse.action\_name

instruction

parameters

description

collect

-

collects all associated artifacts for an IU and places them in a local touchpoint addressable cache. This action may be called explicitly during the collect phase but is typically called as the "default" action called for IUs with the Eclipse Touchpoint Type as part of the main Collect and Sizing phases.

installBundle

bundle

Installs a bundle artifact specified by the parameter bundle

uninstallBundle

bundle

Uninstalls a bundle artifact with a bundle-id specified by the paramter bundle

addSourceBundle

bundle

Installs a source bundle artifact with the bundle-id specified by the parameter bundle

removeSourceBundle

bundle

Removes/uninstalls the source bundle artifact with the bundle-id specified by the parameter bundle

installFeature

feature, featureId, version

Installs the feature referenced by the parameter feature (matched against artifacts in the iu). The feature is installed with the id specified by the parameter featureId, or if this parameter has the value default, with the id specified in the artifact referenced by feature. The features is installed with the version specified in version, or with the version specified in the artifact referenced by the feature parameter if the version parameter has the value default

uninstallFeature

feature, featureId, version

Uninstalls a feature. Parameters have the same meaning as for the command installFeature

setLauncherName

name

Sets the name of the launcher to name. The launcher name is used to configure launcher name specific ini files.

addProgramArg

programArg

Adds the string specified in the parameter programArg as an argument to the program. If the parameter is the special value @artifact, the location of the artifact referenced by the first artifact key in the IU is used as the parameter value.

removeProgramArg

programArg

Removes the program argument specified in the string programArg - if the parameter is the special value @artifact, the location of the artifact referenced by the first artifact key in the IU is used as the parameter value.

setStartLevel

startLevel

Sets the start level to the integer value specified in the parameter startValue

markStarted

started

Marks the bundle referenced by the first artifact key in the installable unit as started or not started, as controlled by the boolean parameter started. A parameter value of "true" will mark the bundle as started and "false" will mark the bundle as not started).

setFrameworkDependentProperty

propName, propValue

Sets the framework dependent property named propName to the value specified in propValue. Framework dependent properties are properties specific to the Equinox implementation of the OSGi framework.

setFrameworkIndependentProperty

propName, propValue

Sets the framework independent property named propName to the value specified in propValue. Framework independent properties do not specifically target Eclipse and are generally applicable to other OSGi frameworks.

addProgramProperty

propName, propValue

Adds the given value specified in propValue to the program property named propName. The program property value is treated as a comma-separated list and the given value is added to that list. Program properties are used by the executable program to among other things locate the jars needed to start Eclipse.

removeProgramProperty

propName, propValue

Remove the given value specified in propValue from the program property named propName. The program property value is treated as a comma-separated list and the given value is removed from that list. If there are no more elements in the list, then the property itself is removed. Program properties are used by the executable program to among other things locate the jars needed to start Eclipse.

setProgramProperty

propName, propValue

Sets the program property named propName to the value specified in propValue. Program properties are used by the executable program to among other things locate the jars needed to start Eclipse.

addJvmArg

jvmArg

Adds the string specified in the parameter jvmArg to the arguments passed to the JVM.

removeJvmArg

jvmArg

Removes the string specified in the parameter jvmArg from the arguments passed to the JVM.

setJvm

jvm

Sets the JVM to be the path specified in the parameter jvm. If a value of null is specified then the current JVM path value is removed.

checkTrust

-

collects the set of bundle files on which the signature trust check should be performed. The checkTrust action is not meant to be user callable and is done as part of the CheckTrust phase.

addRepository

location, type, enabled

Adds the repository at location of type type to the list of known repositories. The repository will then be available when installing or updating the profile in the future. The enabled parameter takes a boolean value ("true" or "false") indicating whether the add repository should be enabled. The value of the location parameter must be a well-formed URI. The type parameter value must be the value of one of the IRepository.TYPE\_\* constants, Specifically, type "0" indicates a metadata repository, and type "1" indicates an artifact repository.

Example:

instructions.configure=org.eclipse.equinox.p2.touchpoint.eclipse.addRepository(location:http${#58}//download.eclipse.org/birt/update-site/2.5/,type:0,name:BIRT 2.5,enabled:true); \

org.eclipse.equinox.p2.touchpoint.eclipse.addRepository(location:http${#58}//download.eclipse.org/birt/update-site/2.5/,type:1,name:BIRT 2.5,enabled:true);

removeRepository

location, type

Removes the repository at location of type type from the list of known

repositories. The value of the location parameter must be a well-formed URI. The type parameter value must be the value of one of the IRepository.TYPE\_\* constants, Specifically, type "0" indicates a metadata repository, and type "1" indicates an artifact repository.

Action Format

The Touchpoint Data Element has a Map that describes the actions to execute in the various p2 engine phases (e.g. "install", "uninstall", "configure", "unconfigure", "collect" and "checktrust"). The key of the Map entry is the name of a phase (i.e. when the actions should be executed), and the value is a statement-sequence:

statement-sequence :

| statement ';'

| statement-sequence statement

;

Where a statement is of the format:

statement :

| actionName '(' parameters ')'

;

parameters :

| // empty

| parameter

| parameters ',' parameter

;

parameter :

| paramName ':' paramValue

;

actionName, paramName, paramValue :

| String

;

In the p2 engine, the Phase will lookup the "actionName" using it's own phase specific actions (e.g. "collect") and also those made available by the associated touchpoint (e.g. "mkdir" in the native touchpoint, and "installBundle" in the Eclipse touchpoint) .

As an example - an "install" instruction for a bundle might consist of the following statement:

installBundle(bundle:${artifact});

installBundle is the action name

bundle is the parameter name

${artifact} is the parameter value. The value ${artifact} signifies the use of a pre-defined variable named "artifact".

Built-in Action Variables

What follows is a catalog of the variables made available by the phases and touchpoints. Many of these are mostly useful to the implementor of new actions and touchpoint types.

Variables Available in all phases

variable

description

#nnnn

the unicode value of a character.

Note: This is especially important for the six characters that require escaping.

$ = ${#36}

, = ${#44}

: = ${#58}

; = ${#59}

{ = ${#123}

} = ${#125}

profile

the profile being modified.

phaseId

the name of the phase e.g. collect, install, etc.

operand

the actions operand (e.g. IU pair)

Variables Available in all installable unit phases

e.g. collect, unconfigure, uninstall, install, configure, ...

variable

description

installFolder

the root folder for this profile.

touchpoint

the touchpoint associated with the IUs in the operand if applicable

Collect Phase Variables

variable

description

artifactRequests

A list that a touchpoints "collect" action will use to add mirroring requests to.

Unconfigure Phase Variables

variable

description

iu

The IU being unconfigured. This is set from the first IU of the operand pair.

artifact

The artifact id of the first artifact listed in the IU.

Uninstall Phase Variables

variable

description

iu

The IU being uninstalled. This is set from the first IU of the operand pair.

artifact

The artifact id of the first artifact listed in the IU.

Install Phase Variables

variable

description

iu

The IU being installed. This is set from the second IU of the operand pair.

artifact

The artifact id of the first artifact listed in the IU.

Configure Phase Variables

variable

description

iu

The IU being configured. This is set from the second IU of the operand pair.

artifact

The artifact id of the first artifact listed in the IU.

Eclipse Touchpoint Variables

variable

description

manipulator

an instance of the Manipulator class used to alter the configuration of an Eclipse install.

Native Touchpoint Variables

variable

description

backup

the BackupStore used to save transaction state during native file operations.

Additional Notes on Backup for File Operation Actions

The Native Touchpoint stores a temporary backup of files that are deleted or overwritten during the installation process. If an installation succeeds the backup is simply removed. If however the installation fails, the files in backup are restored to their original location. If the restore works as expected, the backup copy is also removed. Two "disaster" cases remain:

It was not possible to restore files (probably because of faulty hardware, running out of disk, or manual tampering with files during the installation).

The system crashed during the install

The backup is placed in a directory under the directory referenced by the system property "java.io.tmpdir". The backup directory has a name with the prefix ".p2bu" which is followed by a unique key per running backup instance. Under the ".p2bu..." directory files are stored in a hierarchy that reflects their original location.

In both of the disaster cases, the backup store under java.io.tempdir will contain copies of all files that were not automatically restored. If a restore is wanted, this can be performed manually by copying the content back to their original position (which is evident from the structure under ".p2bu"). This can be made with tools like zip (simply zip everything relative to .p2bu, and then unzip it from the real file system root).

In case something goes wrong during restore of a backup, the event is logged with details of what needs to be manually restored. This is best effort support so in the case of a system crash file loss is possible.

21.3. Installing software using the p2 director application

In addition to the "Software Updates" dialog, you can also perform provisioning operations from a command line or script. This is achieved using a tool called the director application. The director application is a command line tool for installing additional software or uninstalling software from an Eclipse-based product. This application is capable of provisioning a complete installation from scratch or simply extending your application. Depending on your needs, this application can be executed both inside and outside of the target product being provisioned.

Terminology

Director application: the application performing p2 operations such as install or uninstall. This application is provided by the org.eclipse.equinox.p2.director.app bundle.

Provisioning operation: an operation installing, uninstalling features.

Target product: the installation targeted by the provisioning operation.

Builder: an eclipse based application containing the director application bundle and its dependents.

Running inside the target application

In this mode, the provisioning operation happens from within the targeted product that you are provisioning. It is equivalent to starting up the targeted product and using the p2 UI to perform the equivalent operation.

This means that the target application has to be in a runnable state and has to contain the director application bundle. Also, since the target product will have run, cache files will have been created in the configuration folder (e.g. configuration/org.eclipse.osgi).

The following example shows the command line used to install CDT into the SDK.

<targetProductFolder>/eclipsec.exe

-application org.eclipse.equinox.p2.director

-repository http://download.eclipse.org/releases/helios/

-installIU org.eclipse.cdt.feature.group/<version>

Provisioning without running the target application

In this case the provisioning operation happens "outside" of the targeted product. The "targeted product" is \*not\* started. This allows one to both modify an existing installation and create a complete installation from scratch given proper metadata.

This also has the advantage that since the targeted product does not need to be started, the provisioning operation can be performed on any platform for any other platform (e.g. on my linux machine, one can add plug-ins to a windows-based target application).

Installing / uninstalling IUs into a target product

To install or uninstall something into an existing target product a few extra arguments than for the "inside" mode need to be set. These mostly consist in providing the provisioning operation the ID of the profile it needs to operate on, and where it is located on disk.

For example, if from a directory called "d:\builder" containing the builder, you want to install CDT into an existing SDK located into "d:\eclipse", you would use the following command line.

d:\builder\eclipsec.exe

-application org.eclipse.equinox.p2.director

-repository http://download.eclipse.org/releases/helios/

-installIU org.eclipse.cdt.feature.group

-tag AddCDT

-destination d:/eclipse/

-profile SDKProfile

Note that there is no need to describe the os/ws/arch of the platform being targeted because all this information is already available in the profile of the application in which we are provisioning.

Installing a complete product

The creation of a complete product using the director application only needs a few extra arguments compared to the previous example. Most of these consist of values used to initialize the profile in which the application will be provisioned.

The following example demonstrates how to create a linux/gtk installation of the Eclipse SDK provisioned into a folder called "d:\eclipse" using a director application located in "d:\builder".

d:\builder\eclipsec.exe

-application org.eclipse.equinox.p2.director

-repository http://download.eclipse.org/eclipse/updates/3.6

-installIU org.eclipse.sdk.ide

-tag InitialState

-destination d:/eclipse/

-profile SDKProfile

-profileProperties org.eclipse.update.install.features=true

-bundlepool d:/eclipse/

-p2.os linux

-p2.ws gtk

-p2.arch x86

-roaming

The -p2.\* arguments describe the os/ws/arch that the provisioned product is targeting.

Installing a complete product for macOS

The creation of a complete product for the macOS Operating System requires that the destination folder end with ".app/" as in the following example.

d:\builder\eclipsec.exe

-nosplash

-application org.eclipse.equinox.p2.director

-repository http://download.eclipse.org/releases/mars

-installIU org.eclipse.sdk.ide

-tag InitialState

-destination d:/eclipse/MyApp.app/

-profile SDKProfile

-profileProperties org.eclipse.update.install.features=true

-p2.os macosx

-p2.ws cocoa

-p2.arch x86\_64

-roaming

For a macOS App product, you may also want to edit the Info.plist file, such as to change CFBundleName to the name of your product.

Arguments Description

-application org.eclipse.equinox.p2.director: the application ID.

-metadataRepository: a comma separated list of metadata repository URLs where the software to be installed can be found.

-artifactRepository: a comma separated list of artifact repository URLs where the software artifacts can be found.

-repository: a comma separated list of repository URL where each entry denotes colocated meta-data and artifact repositories.

-installIU: a comma separated list of IUs to install. Each entry in the list is in the form <id> [ '/' <version> ]. If you are looking to install a feature, the identifier of the feature has to be suffixed with ".feature.group".

-uninstallIU: a comma separated list of IUs to uninstall. Each entry in the list is in the form <id> [ '/' <version> ].

-revert: Revert the installation to a previous state. The previous state can either be a tag (see -tag / -listtags) or a previous profile state [ the number representing the previous state of the profile as found in p2/org.eclipse.equinox.p2.engine/<profileId>/ ].

-purgeHistory: Remove the history of the profile registry.

-destination: the path of a folder in which the targeted product is located. For "macosx", it must end with ".app/".

-list: lists all IU's found in the given repositories in a property like format. The optional arguments can be an comma separated list of entries where each entry is in the form <id> [ '/' <version> ], or a p2 QL query prefixed with Q:, or Q:group as shortcut for groups.

-listTags: list all the tags found for the given profile.

-listInstalledRoots: list all the roots IUs found in the given profile. Each entry is the list in the form id / version

-listFormat: formats the list of IUs according to the given string. Use ${<property>} to access IU properties, e.g. ${org.eclipse.equinox.p2.name} for the IU's name. ID and version of an IU are available through ${id} and ${version}.

-profile: the profile id containing the description of the targeted product. This ID is is defined by the eclipse.p2.profile property contained in the config.ini of the targeted product. For the Eclipse SDK the ID is "SDKProfile"

-profileProperties: a comma separated list of <key>=<value> pair. The property org.eclipse.update.install.features=true will cause the update manager features to be installed.

-bundlepool: the location of where the plug-ins and features will be stored. This value is only taken into account when a new profile is created. For an application where all the bundles are located into the plugins/ folder of the destination, set it to <destination>.

-p2.os: the OS to use when the profile is created. This will be used to filter which OS specific installable units need to be installed.

-p2.ws: the windowing system to use when the profile is created. This will be used to filter which WS specific installable units need to be installed.

-p2.arch: the architecture to use when the profile is created. This will be used to filter which architecture specific installable units need to be installed.

-roaming: indicates that the product resulting from the installation can be moved. This property only makes sense when the destination and bundle pool are in the same location. This value is only taken into account when the profile is created.

-shared: use a shared location for the install. The path defaults to ${user.home}/.p2.

-tag: a label. This allows to tag the profile state resulting from the operation being executed.

-verifyOnly: only verify that the actions can be performed. Don't actually install or remove anything.

While doing these operations, you can disable the automatic mirror selection mechanism by setting the VM argument eclipse.p2.mirrors to false.

21.4.1. Mirroring repositories with p2

p2 provides two applications that support copying (mirroring) the content of remote repositories to a local repository. There is also a p2.mirror ant task.

Mirroring an artifact repository

The artifact mirroring application supports duplicating a complete artifact repository into a target repository. To perform this operation you simply need an eclipse installation that contains the org.eclipse.equinox.p2.artifact.repository bundle. The following command will copy the complete contents of a source repository into the destination repository. If the destination repository does not already exist, the mirroring application will create a new repository with the same properties as the source.

<eclipseInstall>\eclipse.exe

-application org.eclipse.equinox.p2.artifact.repository.mirrorApplication

-source http://download.eclipse.org/releases/ganymede

-destination file:d:/artifactLocalRepository/

Arguments description

-application org.eclipse.equinox.p2.artifact.repository.mirrorApplication: the application ID.

-source <source location>: the artifact repository to mirror from.

-destination <destination location>: the artifact repository to mirror to.

Additional arguments

-writeMode clean: removes all artifacts from the destination repository before performing the mirroring process.

-verbose: enables error messages and error logging.

-ignoreErrors: ensures the mirroring application continues to run in the event of an error during the mirroring process.

-raw: instructs the mirroring application to copy the exact artifact descriptors from source into the destination instead of initializing new artifact descriptors with properties from the source descriptors.

-compare: instructs the mirroring application to perform a comparison when a duplicate artifact descriptor is found.

-comparator <comparator ID>: specifies which comparator the mirroring application should use to compare artifact descriptors. Uses the "MD5 Comparator" if no comparator is defined which compares the MD5 hash property of the artifact descriptors.

-compareAgainst <baseline location>: specifies the location of a known good repository. When specified the mirror application will compare all artifacts from the source to the baseline with precedence given to artifacts found in the baseline.

-destinationName <destination name>: specifies what the destination repository should be named. If no name is specified, the source repository's name will be used as the destination's name if no repository exists at the destination.

-references <boolean>: enables or disables the mirroring of references. Default value is true.

Mirroring a metadata repository

The metadata mirroring application supports copying a complete metadata repository into a target repository. To perform this operation you simply need an eclipse installation that contains the org.eclipse.equinox.p2.metadata.repository bundle. The following command will copy the complete source repository into the destination repository. If the destination repository does not already exist, the mirroring application will create a new repository with the same properties as the source.

<eclipseInstall>\eclipse.exe

-application org.eclipse.equinox.p2.metadata.repository.mirrorApplication

-source http://download.eclipse.org/releases/ganymede

-destination file:d:/metadataLocalRepository/

Arguments description

-application org.eclipse.equinox.p2.metadata.repository.mirrorApplication: the application ID.

-source <source location>: the metadata repository to mirror from.

-destination <destination location>: the metadata repository to mirror to.

Additional arguments

-writeMode clean: removes all installable units from the destination repository before performing the mirroring process.

-destinationName <destination name>: specifies what the destination repository should be named. If no name is specified, the source repository's name will be used as the destination's name if no repository exists at the destination.

21.4.2. Composite repositories

As repositories continually grow in size they become harder to manage. The goal of composite repositories is to make this task easier by allowing site maintainers to have a parent repository which refers to multiple child repositories. Users are then able to reference the parent repository and the content of all the child repositories will be transparently available to them.

How it works

With the built-in repository types defined by p2, when a user connects to a server, p2 checks to see if there are any files that are recognized. In particular for metadata repositories it searches for a content.xml file or a content.jar and in the case of artifact repositories it looks for an artifacts.xml or artifacts.jar file.

So in order to create a composite repository, all one needs to do is create a new "index" file for the metadata and/or artifact repository and p2 will recognize that a composite repository has been defined and it will load it and its children. Composite repositories use a different index file name to help p2 recognize them as composite repositories.

Sample composite metadata repository

File: compositeContent.xml

<?xml version='1.0' encoding='UTF-8'?>

<?compositeMetadataRepository version='1.0.0'?>

<repository name='"Eclipse Project Test Site"'

type='org.eclipse.equinox.internal.p2.metadata.repository.CompositeMetadataRepository'

version='1.0.0'>

<properties size='1'>

<property name='p2.timestamp' value='1243822502499'/>

</properties>

<children size='2'>

<child location='http://example.eclipse.org/childOne'/>

<child location='http://example.eclipse.org/childTwo'/>

</children>

</repository>

Sample composite artifact repository

File: compositeArtifacts.xml

<?xml version='1.0' encoding='UTF-8'?>

<?compositeArtifactRepository version='1.0.0'?>

<repository name='"Eclipse Project Test Site"'

type='org.eclipse.equinox.internal.p2.artifact.repository.CompositeArtifactRepository'

version='1.0.0'>

<properties size='1'>

<property name='p2.timestamp' value='1243822502440'/>

</properties>

<children size='2'>

<child location='http://example.eclipse.org/childOne'/>

<child location='http://example.eclipse.org/childTwo'/>

</children>

</repository>

Composite repositories as part of the build

In order to automate composite repository actions in release engineering builds, Ant tasks have been provided which can be called to create and modify composite repositories. The tasks are defined in the org.eclipse.equinox.p2.repository.tools bundle.

p2.composite.repository

Attributes

failOnExists - Whether it should fail if the repository already exists. (Default is false)

validate - A comparator-id. Child repositories claiming to contain the same artifact are compared using the given comparator. These are extensions to the org.eclipse.equinox.p2.artifact.repository.artifactComparators extension point. Comparators provided by p2 are:

org.eclipse.equinox.p2.repository.tools.jar.comparator: Compare jars. Class files are disassembled and compared for equivalence, properties and manifest files are compared as such, all other files are compared byte-for-byte.

org.eclipse.equinox.artifact.md5.comparator: Compare the MD5 sums as recorded in the artifact repositories.

Sub-elements

add - A nested element containing a list of repositories to add to the composite.

remove - A nested element containing a list repositories to remove from the composite.

Examples

Some example Ant tasks for composite repositories and further explanation can be found on the p2 Ant tasks page.

21.4.3. Content categorization

By default, the p2 Install Software Dialog groups all the installable units by category. If an installable unit is not categorized, it will not be displayed in the user interface.

There are currently two supported methods for categorizing content.

A category definition file

A p2.inf file

Category Definition File

A category definition file is an XML file that describes which categories a feature should be listed in. The category definition file can be edited using the Category Definition Editor. The category definition can be specified when exporting a feature, building using PDE Build, or injected into a repository using the category publisher.

The p2.inf File

The p2.inf file is a properties file that can be used to customize capabilites, properties, and instructions. Information about this provisional file is available in the section Customizing p2 metadata.

21.4.4. Ant tasks for managing repositories

Specifying source repositories

Several p2 tasks accept "source" repositories. Source repositories can be specified in several different ways:

A source attribute directly on the ant task. The value is a URL to a location that contains both a metadata and artifact repository. For example:

<p2.mirror source="http://download.eclipse.org/releases/helios" .... />

A <source> element nested in the ant task. This element is an ant fileset, with additional location and kind attributes.

location : A URL specifying the location of a repository. If this attribute is set, all fileset based attributes are ignored.

kind: Either "metadata" or "artifact". If not set, the repository is both metadata and artifacts.

If no location attribute is set, all directories matched by the fileset are added as repositories and all \*.zip files are treated as zipped repositories (accessed using a jar: URL). For example:

<source kind="metatata" dir="/build/input" includes="\*.zip" />

<source location="http://download.eclipse.org/releases/helios" />

A <source> element containing a list of repositories. Each nested element is again a fileset with an additional kind attribute. For example:

<source>

<repository location="http://download.eclipse.org/releases/helios" />

<repository kind="metadata" file="/build/repos/cdt-master-6.0.0-I200903161435.zip" />

<repository kind="artifact" dir="/build/repos" includes="\*repo" />

</source>

Specifying destination repositories

Several p2 tasks accept "destination" repositories. Destination repositories can be specified in several different ways:

A destination attribute directly on the ant task. The value is a URL to a location for both a metadata and artifact repository.

A <repository> element nested in the ant task. (<destination> may also be used.) This element supports the following attributes:

location

A URL to the location of the repository

kind

Either "metadata" or "artifact". Default is both if not specified.

name

A name to give the repository if the repository does not already exist.

append

Append to a repository that is already there. (Default is "true")

compressed

Whether to compress the content/artifact xml into a jar (Default is "true")

format

A URL to another repository from which to copy the repository name and other properties.

Specifying Installable Units

Some p2 tasks allow specifying installable units to work with. This is done with nested <iu> elements which support the following attributes:

id

The id of the IU to match.

version

The version of the IU to match. If not specified, the highest versioned IU is returned. (requires id to be set)

query

Matches all IUs satisfying the query. Currently only "property" queries are supported. The format of the query attribute is intended to be "xpath-like". Eg:

<iu query="property[@name='org.eclipse.equinox.p2.type.category']" />

<iu query="property[@name='my.property.name' @value='specialValue']" />

Ant Tasks

p2.mirror

This task mirrors artifacts and metadata from the given source repositories to the destination. This task will traverse the source repository and recursively include all IUs that match the requirements of IUs being mirrored (this is called slicing the repository).

This task supports the following attributes and elements:

source

Source repositories, as outlined above

destination

Destination repositories, as outlined above

<iu>

Nested IU elements, as outlined above. The installable units to mirror. If none are specified, all IUs contained in the source repositories are mirrored.

log

A file to use for logging the results.

ignoreErrors

Whether or not to ignore errors. (Default is false)

raw

Copy the exact artifact descriptors from source into the destination instead of initializing new artifact descriptors with properties from the source descriptors. (Default is true)

verbose

Turn on verbose logging.

validate

Validate that all all source descriptors are present in the destination after mirroring is complete. (Default is false)

references

Enables or disables the mirroring of references. (Default value is true).

<comparator>

A nested comparator element for comparing against a baseline repository. If IUs from the source repository already exist in the baseline, then the artifacts will be mirrored from the baseline instead of the source. This element supports the following attributes:

<baseline>

A nested element specifying the baseline repository. See destination repositories above.

<exclude>

A nested element specifying artifacts to exclude from the comparison. Contains nested <artifact> elements which support the following attributes:

id

The artifact id.

classifier

The artifact classifier

version

The version to exclude.

range;

A range of versions to exclude

<property>

Nested <property> elements specifying properties that the artifact descriptor must have.

Example: Exclude all pack.gz artifacts:

<exclude>

<artifact>

<property name="format" value="packed" />

</artifact>

</exclude>

Exclude the specified documentation bundles:

<exclude>

<artifact id="org.eclipse.jdt.doc.isv" classifier="osgi.bundle" />

<artifact id="org.eclipse.jdt.doc.usr" classifier="osgi.bundle" />

<artifact id="org.eclipse.pde.doc.isv" classifier="osgi.bundle" />

</exclude>

comparator

Id of a comparator to use if the baseline artifacts should be compared against the source artifacts. These are extensions to the org.eclipse.equinox.p2.artifact.repository.artifactComparators extension point. Comparators provided by p2 are:

org.eclipse.equinox.p2.repository.tools.jar.comparator: Compare jars. Class files are disassembled and compared for equivalence, properties and manifest files are compared as such, all other files are compared byte-for-byte.

org.eclipse.equinox.artifact.md5.comparator: Compare the MD5 sums as recorded in the artifact repositories.

comparatorLog

A log file for the results of the comparison.

<slicingOptions>

A nested element for specifying how to slice the repositories to bring in additional IUs. This element supports the following attributes:

followStrict

Set to true if only strict dependencies should be followed. A strict dependency is defined by a version range only including one version (e.g. [1.0.0.v2009, 1.0.0.v2009]). (Default is false)

includeOptional

Whether or not to follow optional requirements. (Default is true).

includeNonGreedy

Whether or not to follow non-greedy requirements. (Default is true).

includeFeatures

Whether or not to include features. (Default is true).

platformFilter

An "os,ws,arch" triplet to set as the platform against which IU LDAP filters will be matched. IUs not matching the filter will not be mirrored.

filter

Set additional filter properties. Format is a comma separated list of "key=value" pairs.

followOnlyFilteredRequirements

Invert the filters, include only the IUs that don't match.

latestVersionOnly

Set to "true" to filter the resulting set of IUs to only included the latest version of each Installable Unit. By default, all versions satisfying dependencies are included.

Examples:

Creating a delta pack repository based on the contents of the platform, rcp, jdt and equinox.executable features:

<p2.mirror source="file://${buildRepo}">

<destination kind="metadata" location="file://${deltaTemp}" name="RCP Delta Pack Repo" />

<destination kind="artifact" location="file://${deltaTemp}" name="RCP Delta Pack Repo" />

<iu id="org.eclipse.platform.feature.group" version="" />

<iu id="org.eclipse.rcp.feature.group" version="" />

<iu id="org.eclipse.jdt.feature.group" version="" />

<iu id="org.eclipse.equinox.executable" version="" />

<slicingOptions includeOptional="false" includeNonGreedy="false" followStrict="true" followOnlyFilteredRequirements="true" />

</p2.mirror>

Mirror the org.eclipse.equinox.p2.user.ui feature and other required bundles locally (perhaps in preparation for a build).

<p2.mirror source="http://download.eclipse.org/releases/helios" destination="${p2Repo}">

<slicingOptions followStrict="true" />

<iu id="org.eclipse.equinox.p2.user.ui.feature.group" />

<iu id="org.eclipse.ecf"/>

<iu id="org.eclipse.ecf.filetransfer"/>

<iu id="org.eclipse.ecf.identity"/>

<iu id="org.eclipse.ecf.provider.filetransfer"/>

<iu id="org.eclipse.ecf.provider.filetransfer.httpclient"/>

<iu id="org.eclipse.ecf.provider.filetransfer.httpclient.ssl" />

<iu id="org.eclipse.ecf.provider.filetransfer.ssl"/>

<iu id="org.eclipse.ecf.ssl" />

<iu id="org.apache.commons.codec"/>

<iu id="org.apache.commons.httpclient"/>

<iu id="org.apache.ant"/>

<iu id="org.eclipse.core.runtime.compatibility.registry"/>

</p2.mirror>

Mirror the results of a build into a public location, and compare against the existing update site to ensure that artifacts haven't changed if their versions haven't been updated.

<p2.mirror source="file:${buildRepo}" ignoreErrors="true" log="${buildlogs}/mirrorlog.txt">

<destination compressed="true" name="${p2.repo.name}" location="file:${updateRepo}" append="true" />

<comparator comparator="org.eclipse.equinox.p2.repository.tools.jar.comparator" comparatorLog="${buildlogs}/comparatorlog.txt">

<repository location="file:${updateSite}" />

</comparator>

</p2.mirror>

p2.repo2runnable

Transform IUs into their installed form and add them to the destination repository. This allows compiling against folder shaped bundles that contain nested jars. This task is the main mechanism by which headless PDE/Builds can reuse metadata.

This task supports the following attributes and elements:

source

Source repositories, as outlined above

desination

Destination repositories, as outlined above

failOnError

Default is true. Set to "false" to ignore errors and complete the operation.

<iu>

Nested IU elements, as outlined above. The installable units to transform. If none are specified, all IUs contained in the source repositories are transformed.

p2.process.artifacts

Process a local, file-based artifact repository. This task will (optionally) sign and pack artifacts, as well as update the MD5 sums in the repository to match the actual artifacts on disk (since signing/conditioning will change the MD5 sums). This task will use the jarProcessor, which in turn uses ant's SignJar task which requires the jarsigner command line tool.

This task supports the following attributes and elements:

repositoryPath

A URL to an artifact repository to process. This must be a local, file-based repository.

pack

Whether or not to create pack.gz artifacts. (Default is false)

normalize

Whether or not to do pack200(pack + unpack) conditioning on the artifacts. (Default is false, however sign + pack will imply a normalize.)

<sign>

A nested element to enable signing. This element supports the following attributes:

alias

The alias to sign with

keystore

The location of the keystore.

keypass

Password for the private key (if different).

storepass

Password for the key store.

unsign

Whether or not to strip any existing signatures before signing. (Default is false)

<feature>

Nested elements specifying features IUs to process (see above, the query attribute is not supported here). If no features or plug-ins are specified, the entire repository is processed.

<plugin>

Nested elements specifying plugins IUs to process (see above, the query attribute is not supported here). If no features or plug-ins are specified, the entire repository is processed.

p2.remove.iu

Remove Installable Units from the given metadata and artifact repositories. The provided repositories must be modifiable.

This task supports the following attributes and elements:

<repository>

Nested elements specifying metadata and artifact repositories to remove Installable Units from. See destination repositories above. A metadata repository is required, the artifact repository is optional.

<iu>

Nested IU elements, as outlined above. The installable units to remove. Matching IUs are removed from the metadata repository, and all associated artifacts are removed from the artifact repository (if provided).

p2.composite.repository

Create a composite repository

This task supports the following attributes and elements:

failOnExists

Whether we should fail if the repository already exists. (Default is false)

validate

A comparator-id. Child repositories claiming to contain the same artifact are compared using the given comparator. These are extensions to the org.eclipse.equinox.p2.artifact.repository.artifactComparators extension point. Comparators provided by p2 are:

org.eclipse.equinox.p2.repository.tools.jar.comparator: Compare jars. Class files are disassembled and compared for equivalence, properties and manifest files are compared as such, all other files are compared byte-for-byte.

org.eclipse.equinox.artifact.md5.comparator: Compare the MD5 sums as recorded in the artifact repositories.

<add>

A nested element containing a list of repositories to add to the composite. See source repositories above.

<remove>

A nested element containing a list repositories to remove from the composite. See source repositories above.

21.5. Customizing the p2 UI

There are several different levels of integration with the p2 UI, depending on what kind of support you want to surface to your users.

If your goal is to reuse the Eclipse SDK UI inside your own product, you can include the org.eclipse.equinox.p2.user.ui feature in your product. If you want only a subset of the functionality, such as providing update support but not installation support, you can use a subset of the bundles in that feature and use the Eclipse SDK UI contributions as a model for your own contributions.

If your goal is to customize the way information is presented, you can configure a p2 UI Policy that changes aspects of the UI.

If you want to force update of your application, you can use the p2 Operations API to perform a customized search for updates on startup.

21.5.1. Reusing parts of the UI

You may reuse the p2 UI in its entirety, or you may pick and choose which aspects of the UI you want to include in your application.

Reusing the Eclipse SDK UI in its entirety

If your goal is to simply use the same update UI used in the SDK inside your RCP app, very few modifications are required. You'll need to include the org.eclipse.equinox.p2.user.ui feature in your application. This will add the following UI bundles to your application (in addition to all of the p2 core and other required bundles):

org.eclipse.equinox.p2.ui

org.eclipse.equinox.p2.ui.sdk

org.eclipse.equinox.p2.ui.sdk.scheduler

The contributions made in the org.eclipse.equinox.p2.ui.sdk assume that the application has defined a Help menu and an additions group for adding menu contributions.

A sample RCP Mail application which shows this kind of reuse can be found here.

Reusing the Eclipse SDK UI without automatic updating

If you want to use the SDK UI, but do not wish to add automatic update support, the application is modified as above. However, the following bundles can be eliminated from the product:

org.eclipse.equinox.p2.ui.sdk.scheduler

org.eclipse.equinox.p2.updatechecker

A sample RCP Mail application which shows how to do this can also be found here.

Modifying the UI contributions

If you want to include p2 update functionality in the UI, but you don't want these items to appear in exactly the same way as they do in the SDK, you can provide your own bundle that makes p2 UI contributions in lieu of the SDK bundle. This allows you to simply rearrange the way users encounter the install/update functionality, or provide more precise control of what can be done by the user. For example, you may wish to expose a subset of functionality, such as permitting updating, but not allowing the user to uninstall, install, or revert configurations.

The general approach for this kind of reuse is as follows:

Decide which p2 UI bundles to include in your product

Always include org.eclipse.equinox.p2.ui

If you wish to use automatic update checking, include org.eclipse.equinox.p2.updatechecker

If you wish to reuse the SDK UI for automatic update checking (the same pref page, popup, etc.), then you can also include org.eclipse.equinox.p2.ui.sdk.scheduler

Your application must replace the functionality provided by org.eclipse.equinox.p2.ui.sdk according to what functionality is needed.

Use the org.eclipse.equinox.p2.ui.sdk bundle as a model when determining which contributions to make. For example, if you are exposing user preferences, and you are including the SDK's automatic update support (and pref page), then you probably want the automatic update preferences to appear underneath your application's update preferences. If so, then you'll want to use the same preference page ID as used by org.eclipse.equinox.p2.ui.sdk so that the automatic update preference page contribution falls underneath it.

The IHandler classes in org.eclipse.equinox.p2.ui.sdk invoke the update and install function. If you are simply rearranging the menus in your application, you can copy these handler classes and command contributions to your bundle and revise the menu contributions as you wish. Or you can invoke the UI in a completely different way.

The org.eclipse.ui.about.installationPages contributions made by the SDK UI provide access to update functionality. Consider replacing or modifying the installation page contributions if some of the actions are not relevant.

21.5.2. Configuring the UI Policy

Some aspects of what the p2 UI presents and how it behaves are configurable using the Policy class. You can configure an instance of this class (or subclass) in order to customize the UI for your application. This policy is registered as an OSGi service, and retrieved by the UI in order to make decisions about presentation.

Depending on the application requirements, the policy instance may be configured in code, configured using internal preferences, or even by exposing some of the decisions to a user in a preference page. The policy allows you to control things such as

whether repositories (sites) are visible to the user, and whether the user is permitted to manipulate (add, enable, disable, remove) the sites that are used for install and update

what software (installable units) is visible to the user when browsing software sites

what software (installable units) is shown as the roots of the 'Installed Software' page

whether restart is required after updating an application

Example: Installing from the Cloud

This kind of customization is best described by example. A common scenario in RCP applications is that the user should be able to update the application, or install additional software, from a controlled set of update sites. This is common in managed installations, where a systems administrator is maintaining an internal update site. From the user point of view, updates come from one "cloud" rather than individual software sites, and there is no visibility of software sites.

In this example, we want the UI to behave in the following ways:

we do not want to expose any site management function or site preferences

we do not want to expose the standard SDK update preferences, but instead use our own values

we want to contribute the standard installation pages (allowing access to uninstall, update, and revert)

we want to allow automatic updating using the standard preference page

we group the application preferences and update menu items in a Tools menu

A sample RCP Mail application which does this kind of customization can be found here. The most important aspect of this example is that we need to configure the p2 UI Policy according to our requirements and register it.

We will register our policy in the startup code of the existing RCP Mail bundle class.

public void start(BundleContext context) throws Exception {

super.start(context);

plugin = this;

// XXX register the p2 UI policy

registerP2Policy(context);

getPreferenceStore().addPropertyChangeListener(getPreferenceListener());

}

The registration method creates a default policy instance, updates it according to some preference values, and registers it.

private void registerP2Policy(BundleContext context) {

policy = new CloudPolicy();

policy.updateForPreferences();

policyRegistration = context.registerService(Policy.class.getName(), policy, null);

}

In this example, we wish to initialize a policy instance that prevents the user from manipulating the repositories. There are several ways to go about this. The most direct way is that the policy initializes its desired values on construction.

public class CloudPolicy extends Policy {

public CloudPolicy() {

// XXX User has no visibility for repos

setRepositoriesVisible(false);

}

}

A more general approach is for the policy to derive its values from preference settings. These preferences are not exposed to the end user, but are used to control how the policy initializes itself. This is the approach taken in the cloud example. The advantage of such an approach is that the policy can be configured by altering the preference values in the plugin\_customization.ini file for the application.

The policy code simply reads preference settings and adjusts the policy accordingly.

public void updateForPreferences() {

IPreferenceStore prefs = Activator.getDefault().getPreferenceStore();

setRepositoriesVisible(prefs

.getBoolean(PreferenceConstants.REPOSITORIES\_VISIBLE));

setRestartPolicy(prefs.getInt(PreferenceConstants.RESTART\_POLICY));

setShowLatestVersionsOnly(prefs

.getBoolean(PreferenceConstants.SHOW\_LATEST\_VERSION\_ONLY));

setGroupByCategory(prefs

.getBoolean(PreferenceConstants.AVAILABLE\_GROUP\_BY\_CATEGORY));

setShowDrilldownRequirements(prefs

.getBoolean(PreferenceConstants.SHOW\_DRILLDOWN\_REQUIREMENTS));

if (prefs.getBoolean(PreferenceConstants.AVAILABLE\_SHOW\_ALL\_BUNDLES))

setVisibleAvailableIUQuery(QueryUtil.ALL\_UNITS);

else

setVisibleAvailableIUQuery(QueryUtil.createIUGroupQuery());

if (prefs.getBoolean(PreferenceConstants.INSTALLED\_SHOW\_ALL\_BUNDLES))

setVisibleAvailableIUQuery(QueryUtil.ALL\_UNITS);

else

setVisibleAvailableIUQuery(new UserVisibleRootQuery());

}

Now, the plugin\_customization.ini file can be edited according to the desired policy. A sample file that explains all of the configurable aspects of the policy is included in the example. It looks something like this.

# we can configure the update UI by using application preferences to initialize the default UI policy

# should user be able to see and manipulate repositories in the install wizard

org.eclipse.equinox.p2.examples.rcp.cloud/repositoriesVisible=false

# force restart after a provisioning operation (see possible values in org.eclipse.equinox.p2.ui.Policy.restartPolicy())

org.eclipse.equinox.p2.examples.rcp.cloud/restartPolicy=1

# show only latest versions when browsing for updates

org.eclipse.equinox.p2.examples.rcp.cloud/showLatestVersionOnly=true

# software should be grouped by category by default

org.eclipse.equinox.p2.examples.rcp.cloud/groupByCategory=true

# show only groups (features) in the available list, not every bundle

org.eclipse.equinox.p2.examples.rcp.cloud/showAllBundlesAvailable=false

# show only the install roots in the installed software list

org.eclipse.equinox.p2.examples.rcp.cloud/showAllBundlesInstalled=false

# do not drilldown into requirements in the wizards, just show the high level things

org.eclipse.equinox.p2.examples.rcp.cloud/showDrilldownRequirements=false

# automatic update options are defined in org.eclipse.equinox.p2.sdk.scheduler.PreferenceConstants

# check for updates on startup

org.eclipse.equinox.p2.ui.sdk.scheduler/enabled=true

org.eclipse.equinox.p2.ui.sdk.scheduler/schedule=on-startup

# remind the user every 4 hours

org.eclipse.equinox.p2.ui.sdk.scheduler/remindOnSchedule=true

# see AutomaticUpdatesPopup, values can be "30 minutes", "Hour", "4 Hours"

org.eclipse.equinox.p2.ui.sdk.scheduler/remindElapsedTime=4 Hours

# download updates before notifying the user

org.eclipse.equinox.p2.ui.sdk.scheduler/download=true

The rest of the UI code in the cloud example is based on code from org.eclipse.equinox.p2.ui.sdk. We copy the command handlers for install and update, and choose not to use any of the preference pages. The plugin.xml contributions for the standard installation pages are also included. The page implementations are contained in the p2 UI bundle, so we only need to define the names of the pages and point to the existing implementations.

The end result is an application whose install and update UI looks familiar. The most significant change is that the Install New Software... wizard no longer provides any control over which sites are shown. Only the content from sites preconfigured in the product (using the p2.inf file in our example) are shown.

Example: Customizing the UI Queries

The Eclipse SDK and the examples shown so far use the default p2 UI Policy queries that control the visibility of software items. In practice, this means that only Eclipse features are shown when browsing the various update sites. However, p2 does not have any specific knowledge of Eclipse features, nor does it require that installation and update operations be based on features. Similarly, only items actually installed by the end user, (or defined as installed items at product-build time), are shown in the Installed Software page. This is done to simplify the view of the installation.

Your application can change the visibility of items in the UI by defining customized queries for obtaining the available or installed software. In this example, we'll replace the filtered queries normally used by the Eclipse SDK with queries that show everything available. (We'll also show how this can be achieved by altering the preference settings in the Cloud example. The approach taken in this example is shown in order to demonstrate how the queries can be replaced with application-defined queries. For example, the query could be modified to show only IU's with a certain property, or only those whose ids are associated with the application).

A complete example is available here. When installing new software, every available bundle is shown in the install wizard. (Note that you must uncheck the Group items by category checkbox to see this list.)

Likewise, the Installed Software page shows every bundle in the installation, not just the "root" of the product and the user-installed items.

The steps for building the contributions are similar to those for the Cloud example above. This time, the policy class is called AllIUsAreVisiblePolicy.

public class AllIUsAreVisiblePolicy extends Policy {

public AllIUsAreVisiblePolicy() {

// XXX Use the pref-based repository manipulator

setRepositoryPreferencePageId(PreferenceConstants.PREF\_PAGE\_SITES);

// XXX All available IU's should be shown, not just groups/features

setVisibleAvailableIUQuery(InstallableUnitQuery.ANY);

// XXX All installed IU's should be shown, not just the user-installed.

setVisibleInstalledIUQuery(InstallableUnitQuery.ANY);

}

}

The queries that retrieve the visible available software, and the visible installed software, are set so that all Installable Units are shown.

In this example, we use OSGi declarative services to register the policy. Rather than manually register the service when our example Activator starts, we instead declare the service in a policy\_component.xml file. Using declarative services is not necessary in this particular example, but could become necessary if we were to separate our p2 UI contributions into another bundle. In that case, it becomes possible for p2 UI components that use the policy (the preference page or installation pages) to be invoked before the bundle that configures the policy starts. Declarative services ensures that the policy is found and the bundle starts when the service is needed.

As mentioned previously, the Cloud example preferences may also be used to make everything visible in the UI. This can be done by editing the plugin\_customization.ini file in the example.

# show only groups (features) in the available list, not every bundle

org.eclipse.equinox.p2.examples.rcp.cloud/showAllBundlesAvailable=true

# show only the install roots in the installed software list

org.eclipse.equinox.p2.examples.rcp.cloud/showAllBundlesInstalled=true

Modifying the p2 UI Policy while reusing the p2 UI feature

Because the p2 UI Policy is defined as an OSGi service, products that ship with the org.eclipse.equinox.p2.user.ui feature unmodified can still provide an alternate implementation of the UI Policy. The org.eclipse.equinox.p2.ui.sdk bundle declares the service with a default ranking value (0). This means that the product must supply a policy implementation with a higher service ranking. When the policy service is found, the highest ranking policy will win. A sample file that declares a policy with a ranking of 1000 can be found here.

21.5.3. Forced update on startup

Sometimes the simplest UI is no UI. In a highly-managed product installation, it may be desirable to automatically update the application each time it is started, with no intervention from the user. In this case, the update is not truly "headless," since a progress indicator is shown while searching for updates. The user may cancel the update, but otherwise cannot intervene with the update. If no updates are found, the user is notified.

In this configuration, the p2 UI class libraries bundle (org.eclipse.equinox.p2.ui) is not needed at all. Only the p2 core code is used to achieve the update. An example RCP application that uses this approach can be found here.

In this kind of scenario, you need to ensure that all of the p2 services are available at startup time, when the update check is to be performed. The example bundle itself must be started. Less obvious is that the org.eclipse.equinox.ds bundle must be started so that all of the declared services will be found. This can be accomplished in a number of ways, depending on how you are running the example:

set the bundle start level in the Configuration tab of the product editor (the .product file for your build)

force a start of the bundle in the config.ini of the already built app

set the bundle start level in the launch configuration if you are simply launching the example in Eclipse

The update check is performed in the postWindowOpen() method of the example's ApplicationWorkbenchWindowAdvisor class. This method sets up the progress monitoring, invokes the update search, and handles any errors or notifications. It uses a preference to remember if it is restarting after an update, so that the update search is not repeated when the application is restarted after updating.

The update check method itself is rather simple, because it does not attempt to involve the user in making any choices about the updates. It uses the p2 Operations API (new in Eclipse 3.6) to search for updates and perform the update.

public class P2Util {

// XXX Check for updates to this application and return a status.

static IStatus checkForUpdates(IProvisioningAgent agent, IProgressMonitor monitor) throws OperationCanceledException {

ProvisioningSession session = new ProvisioningSession(agent);

// the default update operation looks for updates to the currently

// running profile, using the default profile root marker. To change

// which installable units are being updated, use the more detailed

// constructors.

UpdateOperation operation = new UpdateOperation(session);

SubMonitor sub = SubMonitor.convert(monitor,

"Checking for application updates...", 200);

IStatus status = operation.resolveModal(sub.newChild(100));

if (status.getCode() == UpdateOperation.STATUS\_NOTHING\_TO\_UPDATE) {

return status;

}

if (status.getSeverity() == IStatus.CANCEL)

throw new OperationCanceledException();

if (status.getSeverity() != IStatus.ERROR) {

// More complex status handling might include showing the user what updates

// are available if there are multiples, differentiating patches vs. updates, etc.

// In this example, we simply update as suggested by the operation.

ProvisioningJob job = operation.getProvisioningJob(null);

status = job.runModal(sub.newChild(100));

if (status.getSeverity() == IStatus.CANCEL)

throw new OperationCanceledException();

}

return status;

}

}

22. Packaging and delivering Eclipse based products

The Eclipse platform is designed so you can add plug-ins that provide functionality 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.

22.1. 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)

artifacts.xml (bundle pool contents)

eclipse.exe

eclipse.ini

configuration/

config.ini

dropins/

jre/

features/ (installed features if any)

com.example.acme.acmefeature\_1.0.0/

feature.xml

...

plugins/

org.eclipse.equinox.launcher\_1.0.0.v20070530.jar

org.eclipse.equinox.launcher.win32.win32.x86/

com.example.acme.acmefeature\_1.0.0/

plugin.xml

about.ini

about.html

about.mappings

about.properties

acme.png

splash.bmp

com.example.acme.acmewebsupport\_1.0.0/

...

links/

...

p2/

...

There are actually two ways of defining a product in Eclipse. The preferred mechanism is to use the products extension point. This extension point allows you to define your product and customize branding such as splash screens, window icons, and the like.

22.1.1. 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="%productBlurb">

<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.

Product customization properties are also used by other platform services such as to configure the Universal Welcome/Intro.

22.1.2. 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.

<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 now discouraged in favor of the org.eclipse.ui.intro extension point. See Welcome/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.util.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.

Dialog settings defaults

If the custom preferences file is used as described above, setting the org.eclipse.ui/default\_dialog\_settings\_rootUrl property in that file allows to specify a folder containing default dialog settings values for product plug-ins. The property value must be a valid URL.

The preference can be set using the following URL schemes.

org.eclipse.ui/default\_dialog\_settings\_rootUrl=http://mycompany/dialog\_settings

org.eclipse.ui/default\_dialog\_settings\_rootUrl=file:/etc/mycompany/dialog\_settings

org.eclipse.ui/default\_dialog\_settings\_rootUrl=platform:/plugin/my.company.bundle/dialog\_settings

The directory specified by this URL must contain folders with dialog\_settings.xml files for every plug-in to be customized, each directory with the symbolic name of concrete plug-in, such as such as dialog\_settings/org.eclipse.ui.ide/dialog\_settings.xml, dialog\_settings/org.eclipse.ui.navigator/dialog\_sttings.xml, dialog\_settings/my.company.bundle/dialog\_settings.xml... Those dialog\_settings.xml files for each bundle can be retrieved from a running instance of the Eclipse Platform, under the folder ${workspace}/.metadata/.plugins. You can simply copy-paste those existing files into the directory set as value for the org.eclipse.ui/default\_dialog\_settings\_rootUrl preference.

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.

The location, size, and color of the progress bar and progress message shown in the splash screen during startup can be configured using the properties startupProgressRect, startupMessageRect, and startupForegroundColor. See IProductConstants for more information about these properties. Note that by default, no progress will be reported at startup. To enable startup progress reporting, set the following preference to true, for example in the preference customization file: IWorkbenchPreferenceConstants.SHOW\_PROGRESS\_ON\_STARTUP

22.2. 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.

Features do not contain any code. They merely describe a set of plug-ins that provide the functionality 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="3.3.0"

provider-name="%providerName">

<description>

%description

</description>

<license url="%licenseURL">

%license

</license>

<url>

<update label="%updateSiteName" url="http://update.eclipse.org/updates/3.3"/>

<discovery label="%updateSiteName" url="http://update.eclipse.org/updates/3.3/">

</url>

<plugin

id="org.apache.ant"

download-size="0"

install-size="0"

version="1.6.1"/>

<plugin

id="com.jcraft.jsch"

download-size="0"

install-size="0"

version="0.1.31"

unpack="false"/>

...

</feature>

22.3. Plug-ins and fragments

Features are described in terms of the plug-ins that comprise them. A plug-in is used to group your code into a modular, extendable and sharable unit.

Plug-ins are modular as each plug-in contains some portion of code. The plug-in specifies other plug-ins (or java packages) it requires to be available to run and it also specifies the set of java packages it provides. An Eclipse based product will contain multiple plug-ins, which can be added, replaced or removed to alter the functionality of the program.

Eclipse plug-ins are based on OSGi bundles. OSGi is used to manage the plug-ins in a product. A plug-in must contain a manifest file with valid OSGi headers for plug-in name and version.

Plug-ins are extendable using extensions and extension points. A plug-in can provide one or more extension points so other plug-ins can add to the functionality of the plug-in. A plug-in may also provide extensions to connect to other plug-ins. To use extensions you must provide a plugin.xml file.

A fragment is used to replace or extend the functionality of an existing plug-in. A common use for fragments is to put environment (operating system, architecture, etc.) specific code into fragments. Depending on the environment the plug-in is installed in the base plug-in code along with the correct fragment can be installed. Fragments are also ideal for shipping features like language or maintenance packs that typically trail the initial products for a few months.

When a fragment is detected by the platform and its parent plug-in is found, the fragment's libraries, extensions and extension points are "merged" with those of the parent plug-in. While this merging mechanism is good from a runtime point of view, developers need to view fragments as separate entities while working on them. Fragments require a valid OSGi manifest. To use extensions and extension points they must define a fragment.xml file which has the same structure as a plugin.xml file.

Plug-ins and fragments can be packaged in Plug-in archive files

22.4. 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

22.5. 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

artifacts.xml

eclipse.exe

install.ini

.config/

platform.cfg

jre/

dropins/

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.bmp

welcome.xml

com.example.acme.acmewebsupport\_1.0.0/

...

links/

...

p2/

...

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 (this includes the contents of the p2 and configuration folder)

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

Platform configuration information is copied to acmeweb/eclipse/configuration

Platform update support information is copied to acmeweb/eclipse/p2

create a marker file, .eclipseproduct, in acmeweb/eclipse. The marker file is a java.util.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 installer 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.

22.6. 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 functionality of the acmeweb product.

betterwebs/

eclipse/ (directory for installed Eclipse files)

.eclipseextension (marker file)

features/ (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/

features/ (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.util.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.

22.7. Updating a product or extension

By following the prescribed procedures for packaging and installing products, we can take advantage of platform provisioning support, 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 software sites. The platform itself defines a concrete implementation of features and sites. This concrete implementation is what allows the platform to upgrade and install additional features.

The platform software site 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 platform update support.

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 software site, 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.

Software site layout

The software site 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, content.xml. The 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. Prefer using the directory path as this will allow for future changes to the software site format. 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>/

artifacts.jar (compressed p2 artifact repository - present in p2 optimized site)

content.jar (compressed p2 metadata repository - present in p2 optimized site)

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

...

Update servers

An Eclipse project software site is provided for updating the platform itself. In addition, the platform update UI allows users to maintain a list of software sites 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.

Software sites can be created and manipulated by creating an "update site" project in PDE, or by using the publishing and repository manipulation command line tools and Ant tasks.

22.8. Deploying eclipse based application with Java Web Start

Applications built on Eclipse can 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 based on Eclipse 3.1 or later;

All deployed plug-ins must be jar'ed;

All plug-ins must be signed since the application needs full permission from the client.

The following steps describe how to setup a Java Web Start site serving up a feature based RCP application. These steps are for applications based on eclipse 3.3. Instructions on how to achieve the same for eclipse 3.1 and 3.2 can respectively be found in the 3.1 and 3.2 SDKs.

Step 1, creating a wrapper feature

Create a feature including all the features that are listed in your product definition;

Ensure that the org.eclipse.equinox.launcher plug-in is in the feature or in one of the included feature;

Step 2, exporting the wrapper feature

Note. Before proceeding with this step make sure to have a key store available. Eclipse does not provide any facility to create key stores. You can use the keytool application that comes with the JDK. 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 wrapper feature and do File > Export > Plug-in Development > Deployable Features. In the wizard, select the wrapper 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 key store 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)

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

org.eclipse.equinox.launcher\_<version>.jar

...

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 at the root. 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="../topic/org.eclipse.platform.doc.isv/guide/mail.jnlp"> <!-- URL to the site containing the jnlp application. It should match the value used on export. Href, the name of this file -->

<information>

<!-- user readable name of the application -->

<title> Mail Application </title>

<!-- vendor name -->

<vendor>My company</vendor>

<!-- vendor homepage -->

<homepage href="../topic/org.eclipse.platform.doc.isv/guide/My company website" />

<!-- product description -->

<description>This is a mail client</description>

<icon kind="splash" href="../topic/org.eclipse.platform.doc.isv/guide/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.equinox.launcher.WebStartMain">

<argument>-nosplash</argument>

</application-desc>

<resources>

<!-- Reference to the launcher jar. The version segment must be updated to the version being used-->

<jar href="../topic/org.eclipse.platform.doc.isv/guide/plugins/org.eclipse.equinox.launcher\_<version>.jar"/>

<!-- Reference to all the plugins and features constituting the application -->

<!-- Here we are referring to the wrapper feature since it transitively refers to all the other plug-ins necessary -->

<extension

name="Wrapper feature"

href="../topic/org.eclipse.platform.doc.isv/guide/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>

Tip: once you have created this file, you can store it in the wrapper feature in a folder such that on every export you will get the complete structure. This folder needs to be referenced from the root property of the build.properties (e.g: root=<folderContainingMainJNLPFile>/).

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 wrapper feature in order to facilitate the creation of the main jnlp file and ease the deployment. This wrapper 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.

Miscellaneous

Java Web Start on linux

When an eclipse application is started with Web Start on Linux the default windowing system is motif. If you want to run GTK, you need to set the property osgi.ws to "gtk" in the main jnlp file. For example you can add:

<resources os="Linux">

<property name="osgi.ws" value="gtk"/>

</resources>

Known limitations

Eclipse Update and Java Web Start

Those two deployment technologies can work together 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 referred in the prerequisites of features that needs to be installed by Update;

Request to exit the application with a restart code are ignored;

On the Mac, you can only use Web Start with Java 1.5 or later.

22.9. Associate and Open Files

An Eclipse based product is a stand-alone program built with the Eclipse platform. In many cases such a program would like to be associated with particular file types or extensions. For example, the Eclipse IDE could be configured to open all Java files. This would allow a user to double click on a Java file and have it be opened in a running Eclipse IDE instance or start a new instance of the Eclipse IDE if it was not previously running. In order to support this coordination is needed between the Eclipse native launcher, SWT, the workbench and the RCP application.

New command line options have been added to support this scenario:

--launcher.openFile Specifies a file to be opened.

--launcher.defaultAction Specifies an action to take when the launcher is started witout any "-" arguments on the command line.

The openFile argument opens the specified file in an instance of Eclipse. If an instance is not already running then a new instance will be started.

eclipse --launcher.openFile myFile.txt

A second option is needed to configure the launcher to automatically perform the open file request without requiring the user to always specify --launcher.openFile. A new "default action" argument has been introduced to accomplish this. This option can go in the launcher.ini (eclipse.ini) file, the value must be "openFile":

...

-showsplash

org.eclipse.platform

--launcher.defaultAction

openFile

-vmargs

-Xms256m

-Xmx768m

This tells the launcher that if none of the command line arguments start with "-" then all command line arguments should be treated as if they followed the "--launcher.openFile" argument.

eclipse myFile.txt

This option was added because without registry changes, this is the kind of command line the launcher will receive on windows when you double click a file that is associated with eclipse, or you select files and choose "Open With" or "Send To" Eclipse.

Talking to SWT

The launcher talks to SWT through the use of a hidden window. The launcher and SWT both need to agree on the name of this window. This allows the launcher to find an already running eclipse and tell it to open the file. Any RCP application will need to ensure they get this right for things to work.

The launcher bases this on its "official name". The official name can be set with the -name argument. If -name is not set, then the official name is derived from the launcher executable, the extension is removed and the first letter is capitalized: rcp.exe becomes Rcp.

SWT bases this on the value set with the Display.setAppName() function. Normally, this is set by the Workbench when it creates the display and the value is the "appName" taken from the product extension point.

Listening to SWT.OpenDocument events

The launcher communicates with SWT to inform SWT about a request to open one or more files. SWT then can fire the SWT.OpenDocument event. For an RCP application to take advantage of this it must register a listener for the SWT.OpenDocument event. It should register this listener before calling PlatformUI.createAndRunWorkbench so that the listener is in place before the workbench starts running the event loop.

The event loop will start running while the splash screen is still up, so events may arrive before the workbench is ready to actually open an editor for the file. This means that the listener should save the file paths it gets from the OpenDocument events so they can be opened at some later time. WorkbenchAdvisor.eventLoopIdle can be a good place to check for saved open file events.

Here is an example RCP application that does this. First is the IApplication implementation:

public class Application implements IApplication {

public Object start(IApplicationContext context) {

OpenDocumentEventProcessor openDocProcessor =

new OpenDocumentEventProcessor();

Display display = PlatformUI.createDisplay();

display.addListener(SWT.OpenDocument, openDocProcessor);

try {

int returnCode = PlatformUI.createAndRunWorkbench(display, new

ApplicationWorkbenchAdvisor(openDocProcessor));

if (returnCode == PlatformUI.RETURN\_RESTART) {

return IApplication.EXIT\_RESTART;

}

return IApplication.EXIT\_OK;

} finally {

display.dispose();

}

}

...

}

Next is an example SWT Listener that listens to the SWT.OpenDocument event:

public class OpenDocumentEventProcessor implements Listener {

private ArrayList<String> filesToOpen = new ArrayList<String>(1);

public void handleEvent(Event event) {

if (event.text != null)

filesToOpen.add(event.text);

}

public void openFiles() {

if (filesToOpen.isEmpty())

return;

String[] filePaths = filesToOpen.toArray(

new String[filesToOpen.size()]);

filesToOpen.clear();

for (String path : filePaths) {

// open the file path

}

}

}

Finally we need a WorkbenchAdvisor that will open the files durying eventLoopIdle:

public class ApplicationWorkbenchAdvisor extends WorkbenchAdvisor {

private OpenDocumentEventProcessor openDocProcessor;

public ApplicationWorkbenchAdvisor(

OpenDocumentEventProcessor openDocProcessor) {

this.openDocProcessor = openDocProcessor;

}

...

public void eventLoopIdle(Display display) {

openDocProcessor.openFiles();

super.eventLoopIdle(display);

}

}

23. 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.

Eclipse RCP applications are defined to use dependency injection and a service orientated architecture. The Eclipse renderer framework allow to control the appearance of the default user interface and clients can use either the default renderer for SWT or custom renderers for other UI technologies. The Eclipse platform provides the E4Application as default entry point but clients are free to create their own application.

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 determining what plug-ins should be required when using various platform API.

24. Http Service and JSP support

Equinox provides support for building and running OSGi based web applications both inside RCP and application server environments. In an OSGi web application use of the OSGi Http Service replaces the use of web.xml deployment file. The Http Service provides a means to register servlets and resources in a manner more consistent with the dynamic environment in Eclipse where bundles can come and go.

The platform SDK comes packaged with org.eclipse.equinox.http.jetty, an implementation of the OSGi Http Service that uses Jetty as its underlying servlet container. This implementation allows customization and configuration of many of the server settings important for integration.

Java Server Pages play a central role in many java server environments, allowing the presentation of dynamic content. On-the-fly compilation of JSPs is provided by org.eclipse.equinox.jsp.jasper's JSPServlet.

Http Service Extension Points

In many cases it's easier to use the more declarative style of interaction provided by the extension registry. To support this usage style org.eclipse.equinox.http.registry provides extension points for httpcontexts, servlets, and resources that allow the equivalent registration of web resources in the Http Service.

A JSP Extension Factory class in org.eclipse.equinox.jsp.jasper.registry provides JSP support for use in conjunction with the servlets extension point.

JDT Plug-in Developer Guide

Contents

1. Programmer's Guide

1.1. JDT Core

1.1.1. Java model

1.1.2. Manipulating Java code

1.1.3. Setting the Java build path

1.1.4. Compiling Java code

1.1.5. Using the Java search engine

1.1.6. Indexes for the Java search

1.1.7. Performing code assist on Java code

1.1.8. JDT Core options

1.1.9. Formatting Java code

1.2. JDT UI

1.2.1. JavaUI utilities

1.2.1.1. Opening a Java editor

1.2.1.2. Creating Java specific prompter dialogs

1.2.2. Presenting Java elements in a JFace viewer

1.2.3. Java wizard pages

1.2.4. Customizing Java editors

1.2.5. Writing Jar files

1.2.6. Contributing a clean up and a save action

1.2.7. Contributing a quick fix and a quick assist

1.3. JDT Debug

1.3.1. Running Java code

1.4. JDT JUnit integration

1.4.1. Observing JUnit test runs

1.5. JDT Annotation Processing

1.5.1. Getting Started

1.5.2. Headless Build

2. Reference

2.1. API Reference

2.1.1. org.eclipse.ant.launching

2.1.2. org.eclipse.ant.ui.launching

2.1.3. org.eclipse.jdt.apt.core.build

2.1.4. org.eclipse.jdt.apt.core.env

2.1.5. org.eclipse.jdt.apt.core.util

2.1.6. com.sun.mirror.apt

2.1.7. com.sun.mirror.declaration

2.1.8. com.sun.mirror.type

2.1.9. com.sun.mirror.util

2.1.10. org.eclipse.jdt.annotation

2.1.11. org.eclipse.jdt.core

2.1.12. org.eclipse.jdt.core.compiler

2.1.13. org.eclipse.jdt.core.compiler.batch

2.1.14. org.eclipse.jdt.core.dom

2.1.15. org.eclipse.jdt.core.dom.rewrite

2.1.16. org.eclipse.jdt.core.eval

2.1.17. org.eclipse.jdt.core.formatter

2.1.18. org.eclipse.jdt.core.index

2.1.19. org.eclipse.jdt.core.jdom

2.1.20. org.eclipse.jdt.core.manipulation

2.1.21. org.eclipse.jdt.core.refactoring

2.1.22. org.eclipse.jdt.core.refactoring.descriptors

2.1.23. org.eclipse.jdt.core.refactoring.participants

2.1.24. org.eclipse.jdt.core.search

2.1.25. org.eclipse.jdt.core.util

2.1.26. org.eclipse.jdt.debug.core

2.1.27. org.eclipse.jdt.debug.eval

2.1.28. org.eclipse.jdt.debug.ui

2.1.29. org.eclipse.jdt.debug.ui.actions

2.1.30. org.eclipse.jdt.debug.ui.breakpoints

2.1.31. org.eclipse.jdt.debug.ui.launchConfigurations

2.1.32. org.eclipse.jdt.launching

2.1.33. org.eclipse.jdt.launching.environments

2.1.34. org.eclipse.jdt.launching.sourcelookup

2.1.35. org.eclipse.jdt.launching.sourcelookup.advanced

2.1.36. org.eclipse.jdt.launching.sourcelookup.containers

2.1.37. org.eclipse.jdt.ui

2.1.38. org.eclipse.jdt.ui.actions

2.1.39. org.eclipse.jdt.ui.cleanup

2.1.40. org.eclipse.jdt.ui.dialogs

2.1.41. org.eclipse.jdt.ui.jarpackager

2.1.42. org.eclipse.jdt.ui.refactoring

2.1.43. org.eclipse.jdt.ui.search

2.1.44. org.eclipse.jdt.ui.text

2.1.45. org.eclipse.jdt.ui.text.folding

2.1.46. org.eclipse.jdt.ui.text.java

2.1.47. org.eclipse.jdt.ui.text.java.correction

2.1.48. org.eclipse.jdt.ui.text.java.hover

2.1.49. org.eclipse.jdt.ui.wizards

2.1.50. org.eclipse.jdt.junit

2.1.51. org.eclipse.jdt.junit.launcher

2.1.52. org.eclipse.jdt.junit.model

2.1.53. org.eclipse.jdt.junit.wizards

2.2. Extension Points Reference

2.2.1. org.eclipse.jdt.apt.core.annotationProcessorFactory

2.2.2. org.eclipse.jdt.core.annotationProcessorManager

2.2.3. org.eclipse.jdt.core.classpathContainerInitializer

2.2.4. org.eclipse.jdt.core.classpathVariableInitializer

2.2.5. org.eclipse.jdt.core.compilationParticipant

2.2.6. org.eclipse.jdt.core.codeFormatter

2.2.7. org.eclipse.jdt.core.manipulation.changeMethodSignatureParticipants

2.2.8. org.eclipse.jdt.core.javaFormatter

2.2.9. org.eclipse.jdt.debug.breakpointListeners

2.2.10. org.eclipse.jdt.debug.javaLogicalStructures

2.2.11. org.eclipse.jdt.debug.ui.vmInstallPages

2.2.12. org.eclipse.jdt.debug.ui.vmInstallTypePage

2.2.13. org.eclipse.jdt.junit.junitLaunchConfigs

2.2.14. org.eclipse.jdt.junit.testRunListeners

2.2.15. org.eclipse.jdt.launching.classpathProviders

2.2.16. org.eclipse.jdt.launching.executionEnvironments

2.2.17. org.eclipse.jdt.launching.libraryLocationResolvers

2.2.18. org.eclipse.jdt.launching.runtimeClasspathEntries

2.2.19. org.eclipse.jdt.launching.runtimeClasspathEntryResolvers

2.2.20. org.eclipse.jdt.launching.vmConnectors

2.2.21. org.eclipse.jdt.launching.vmInstalls

2.2.22. org.eclipse.jdt.launching.vmInstallTypes

2.2.23. org.eclipse.jdt.ui.classpathAttributeConfiguration

2.2.24. org.eclipse.jdt.ui.classpathContainerPage

2.2.25. org.eclipse.jdt.ui.classpathFixProcessors

2.2.26. org.eclipse.jdt.ui.cleanUps

2.2.27. org.eclipse.jdt.ui.foldingStructureProviders

2.2.28. org.eclipse.jdt.ui.javaCompletionProposalComputer

2.2.29. org.eclipse.jdt.ui.javaCompletionProposalSorters

2.2.30. org.eclipse.jdt.ui.javadocCompletionProcessor

2.2.31. org.eclipse.jdt.ui.javadocExportWizardPage

2.2.32. org.eclipse.jdt.ui.javaEditorTextHovers

2.2.33. org.eclipse.jdt.ui.javaElementFilters

2.2.34. org.eclipse.jdt.ui.queryParticipants

2.2.35. org.eclipse.jdt.ui.quickAssistProcessors

2.2.36. org.eclipse.jdt.ui.quickFixProcessors

2.3. Other Reference Information

2.3.1. JDT Plug-ins Map

3. Plug-in Migration Guide

3.1. Migrating to Eclipse JDT 4.13 from 4.12

3.1.1. Introduction

3.1.2. FAQ

3.1.3. Incompatibilities

3.1.4. Adopting 4.13 Mechanisms and API

3.2. Migrating to Eclipse JDT 4.12 from 4.11

3.2.1. Introduction

3.2.2. FAQ

3.2.3. Incompatibilities

3.2.4. Adopting 4.12 Mechanisms and API

3.3. Migrating to Eclipse JDT 4.11 from 4.10

3.3.1. Introduction

3.3.2. FAQ

3.3.3. Incompatibilities

3.3.4. Adopting 4.11 Mechanisms and API

3.4. Migrating to Eclipse JDT 4.10 from 4.9

3.4.1. Introduction

3.4.2. FAQ

3.4.3. Incompatibilities

3.4.4. Adopting 4.10 Mechanisms and API

3.5. Migrating to Eclipse JDT 4.9 from 4.8

3.5.1. Introduction

3.5.2. FAQ

3.5.3. Incompatibilities

3.5.4. Adopting 4.9 Mechanisms and API

3.6. Migrating to Eclipse JDT 4.8 from 4.7

3.6.1. Introduction

3.6.2. FAQ

3.6.3. Incompatibilities

3.6.4. Adopting 4.8 Mechanisms and API

3.7. Migrating to Eclipse JDT 4.7 from 4.6

3.7.1. Introduction

3.7.2. FAQ

3.7.3. Incompatibilities

3.7.4. Adopting 4.7 Mechanisms and API

3.8. Migrating to Eclipse JDT 4.6 from 4.5

3.8.1. Introduction

3.8.2. FAQ

3.8.3. Incompatibilities

3.8.4. Adopting 4.6 Mechanisms and API

3.9. Migrating to Eclipse JDT 4.5 from 4.4

3.9.1. Introduction

3.9.2. FAQ

3.9.3. Incompatibilities

3.9.4. Adopting 4.5 Mechanisms and API

3.10. Migrating to Eclipse JDT 4.4 from 4.3

3.10.1. Introduction

3.10.2. FAQ

3.10.3. Incompatibilities

3.10.4. Adopting 4.4 Mechanisms and API

3.11. Migrating to Eclipse JDT 4.3 from 3.8

3.11.1. Introduction

3.11.2. FAQ

3.11.3. Incompatibilities

3.11.4. Adopting 3.9 Mechanisms and API

3.12. Migrating to Eclipse JDT 3.8 from 3.7

3.12.1. Introduction

3.12.2. FAQ

3.12.3. Incompatibilities

3.12.4. Adopting 3.8 Mechanisms and API

3.13. Migrating to Eclipse JDT 3.7 from 3.6

3.13.1. Introduction

3.13.2. FAQ

3.13.3. Incompatibilities

3.13.4. Adopting 3.7 Mechanisms and API

3.14. Migrating to Eclipse JDT 3.6 from 3.5

3.14.1. Introduction

3.14.2. FAQ

3.14.3. Incompatibilities

3.14.4. Adopting 3.6 Mechanisms and API

3.15. Migrating to Eclipse JDT 3.5 from 3.4

3.15.1. Introduction

3.15.2. FAQ

3.15.3. Incompatibilities

3.15.4. Adopting 3.5 Mechanisms and API

3.16. Migrating to Eclipse JDT 3.4 from 3.3

3.16.1. Introduction

3.16.2. FAQ

3.16.3. Incompatibilities

3.16.4. Adopting 3.4 Mechanisms and API

3.17. Migrating to Eclipse JDT 3.3 from 3.2

3.17.1. Introduction

3.17.2. Incompatibilities

3.17.3. Adopting 3.3 mechanisms and API

3.18. Migrating to Eclipse JDT 3.2 from 3.1

3.18.1. Introduction

3.18.2. FAQ

3.18.3. Incompatibilities

3.18.4. Adopting 3.2 Mechanisms and API

4. Questions Index

5. Legal

1. JDT Programmer's Guide

The Eclipse platform is delivered with a full featured Java integrated development environment (IDE). Java development tooling (JDT) allows users to write, compile, test, debug, and edit programs written in the Java programming language.

The JDT makes use of many of the platform extension points and frameworks described in the Platform Plug-in Developer Guide. It's easiest to think of the JDT as a set of plug-ins that add Java specific behavior to the generic platform resource model and contribute Java specific views, editors, and actions to the workbench.

This guide discusses the extension points and API provided by the JDT. We assume that you already understand the concepts of plug-ins, extension points, workspace resources, and the workbench UI.

Given that the JDT supplies a full featured Java IDE, why would you need to use the JDT API? If you are building a plug-in that interacts with Java programs or resources as part of its function, you may need to do one or more of the following things:

Programmatically manipulate Java resources, such as creating projects, generating Java source code, performing builds, or detecting problems in code.

Programmatically launch a Java program from the platform

Provide a new type of VM launcher to support a new family of Java runtimes

Add new functions and extensions to the Java IDE itself

The JDT is structured into three major components:

JDT Core - the headless infrastructure for compiling and manipulating Java code.

JDT UI - the user interface extensions that provide the IDE.

JDT Debug - program launching and debug support specific to the Java programming language.

We'll examine each component's structure and the API it provides.

1.1. JDT Core

JDT Core (org.eclipse.jdt.core) is the plug-in that defines the core Java elements and API. You should always list this plug-in as a prerequisite when you are developing Java specific features.

JDT Core packages give you access to the Java model objects and headless Java IDE infrastructure. The JDT Core packages include:

org.eclipse.jdt.core - defines the classes that describe the Java model.

org.eclipse.jdt.core.compiler - defines an API for the compiler infrastructure.

org.eclipse.jdt.core.dom - supports Abstract Syntax Trees (AST) that can be used for examining the structure of a compilation unit down to the statement level.

org.eclipse.jdt.core.dom.rewrite - supports rewriting of Abstract Syntax Trees (AST) that can be used for manipulating the structure of a compilation unit down to the statement level.

org.eclipse.jdt.core.eval - supports the evaluation of code snippets in a scrapbook or inside the debugger.

org.eclipse.jdt.core.formatter - supports the formatting of compilation units, types, statements, expressions, etc.

org.eclipse.jdt.core.index - supports the generation of indexes that can be used by the JDT search engine.

org.eclipse.jdt.core.jdom - supports a Java Document Object Model (DOM) that can be used for walking the structure of a Java compilation unit. All types within this package are deprecated.

org.eclipse.jdt.core.search - supports searching the workspace's Java model for Java elements that match a particular description.

org.eclipse.jdt.core.util - provides utility classes for manipulating .class files and Java model elements.

org.eclipse.jdt.core.jdom is deprecated since version 3.0. The manipulation of the structure of a compilation unit should be done using org.eclipse.jdt.core.dom instead.

1.1.1. Java model

The Java model is the set of classes that model the objects associated with creating, editing, and building a Java program. The Java model classes are defined in org.eclipse.jdt.core. These classes implement Java specific behavior for resources and further decompose Java resources into model elements.

Java elements

The package org.eclipse.jdt.core defines the classes that model the elements that compose a Java program. The JDT uses an in-memory object model to represent the structure of a Java program. This structure is derived from the project's class path. The model is hierarchical. Elements of a program can be decomposed into child elements.

Manipulating Java elements is similar to manipulating resource objects. When you work with a Java element, you are actually working with a handle to some underlying model object. You must use the exists() protocol to determine whether the element is actually present in the workspace.

The following table summarizes the different kinds of Java elements.

Element Description

IJavaModel Represents the root Java element, corresponding to the workspace. The parent of all projects with the Java nature. It also gives you access to the projects without the java nature.

IJavaProject Represents a Java project in the workspace. (Child of IJavaModel)

IModuleDescription Represents a Java module descriptor. (Child of IJavaProject)

IPackageFragmentRoot Represents a set of package fragments, and maps the fragments to an underlying resource which is either a folder, JAR, or ZIP file. (Child of IJavaProject)

IPackageFragment Represents the portion of the workspace that corresponds to an entire package, or a portion of the package. (Child of IPackageFragmentRoot )

ICompilationUnit Represents a Java source (.java) file. (Child of IPackageFragment )

IPackageDeclaration Represents a package declaration in a compilation unit. (Child of ICompilationUnit )

IImportContainer Represents the collection of package import declarations in a compilation unit. (Child of ICompilationUnit )

IImportDeclaration Represents a single package import declaration. (Child of IImportContainer )

IType Represents either a source type inside a compilation unit, or a binary type inside a class file.

IField Represents a field inside a type. (Child of IType )

IMethod Represents a method or constructor inside a type. (Child of IType )

IInitializer Represents a static or instance initializer inside a type. (Child of IType )

IClassFile Represents a compiled (binary) type. (Child of IPackageFragment )

IModularClassFile Represents the class file of a module description ("module-info.class"). (Child of IPackageFragment )

ITypeParameter Represents a type parameter. (Not a child of any Java element, it is obtained using IType.getTypeParameter(String) or IMethod.getTypeParameter(String))

ILocalVariable Represents a local variable in a method or an initializer. (Not a child of any Java element, it is obtained using ICodeAssist.codeSelect(int, int))

IAnnotation Represents a Java 5 annotation. (Not a child of any Java element, it is obtained using IAnnotatable.getAnnotation(String) or IAnnotatable.getAnnotations())

IAnnotatable Represents a type, a field, a method, a local variable, or a package declaration that can be annotated with one or several IAnnotations.

All Java elements support the IJavaElement interface.

Some of the elements are shown in the Packages view. These elements implement the IOpenable interface, since they must be opened before they can be navigated. The figure below shows how these elements are represented in the Packages view.

Packages View showing elements implementing the IOpenable interface

The Java elements that implement IOpenable are created primarily from information found in the underlying resource files. The same elements are represented generically in the resource navigator view.

Resource Navigator showing elements implementing the IOpenable interface

Since 3.3, IOpenables that are the root of ITypes implement ITypeRoot . These are ICompilationUnit and IClassFile.

Other elements correspond to the items that make up a Java compilation unit. The figure below shows a Java compilation unit and a content outliner that displays the source elements in the compilation unit.

An editor and a content outliner illustrating the relation between corresponding source elements

These elements implement the ISourceReference interface, since they can provide corresponding source code. (As these elements are selected in the content outliner, their corresponding source code is shown in the Java editor).

Java elements and their resources

Many of the Java elements correspond to generic resources in the workspace. When you want to create Java elements from a generic resource the class JavaCore is the best starting point. The following code snippet shows how to get Java elements from their corresponding resources.

private void createJavaElementsFrom(IProject myProject, IFolder myFolder, IFile myFile) {

IJavaProject myJavaProject= JavaCore.create(myProject);

if (myJavaProject == null)

// the project is not configured for Java (has no Java nature)

return;

// get a package fragment or package fragment root

IJavaElement myPackageFragment= JavaCore.create(myFolder);

// get a .java (compilation unit), .class (class file), or

// .jar (package fragment root)

IJavaElement myJavaFile = JavaCore.create(myFile);

}

Once you have a Java element, you can use the JDT API to traverse and query the model. You may also query the non-Java resources contained in a Java element.

private void createJavaElementsFrom(IProject myProject, IFolder myFolder, IFile myFile) throws JavaModelException {

...

// get the non Java resources contained in my project.

Object[] nonJavaChildren = myJavaProject.getNonJavaResources();

...

}

Java projects

When you create a Java project from a simple project, JavaCore will check to see if the project is configured with the Java nature. The JDT plug-in uses a project nature to designate a project as having Java behavior. This nature (org.eclipse.jdt.core.JavaCore#NATURE\_ID ) is assigned to a project when the "New Java project" wizard creates a project. If the Java nature is not configured on a project, JavaCore will return null when asked to create the project.

JavaCore is also used to maintain the Java class path, including locations for finding source code and libraries, and locations for generating output binary (.class) files.

What are the unique characteristics of Java projects? They record their classpath in a ".classpath" file and add the Java incremental project builder to the project's build spec. Otherwise, they are just regular projects and can be configured with other natures (and other incremental builders) by plug-ins. Plug-ins that want to configure projects with Java behavior in addition to their own behavior typically use the NewJavaProjectWizardPage to assign the Java nature to the project in addition to their own custom natures or behavior.

IJavaModel can be considered the parent of all projects in the workspace that have the Java project nature (and therefore can be treated as an IJavaProject).

1.1.2. Manipulating Java code

Your plug-in can use the JDT API to create classes or interfaces, add methods to existing types, or alter the methods for types.

The simplest way to alter Java objects is to use the Java element API. More general techniques can be used to work with the raw source code for a Java element.

Code modification using Java elements

Generating a compilation unit

The easiest way to programmatically generate a compilation unit is to use IPackageFragment.createCompilationUnit. You specify the name and contents of the compilation unit. The compilation unit is created inside the package and the new ICompilationUnit is returned.

A compilation unit can be created generically by creating a file resource whose extension is ".java" in the appropriate folder that corresponds to the package directory. Using the generic resource API is a back door to the Java tooling, so the Java model is not updated until the generic resource change listeners are notified and the JDT listeners update the Java model with the new compilation unit.

Modifying a compilation unit

Most simple modifications of Java source can be done using the Java element API.

For example, you can query a type from a compilation unit. Once you have the IType, you can use protocols such as createField, createInitializer, createMethod, or createType to add source code members to the type. The source code and information about the location of the member is supplied in these methods.

The ISourceManipulation interface defines common source manipulations for Java elements. This includes methods for renaming, moving, copying, or deleting a type's member.

Working copies

Code can be modified by manipulating the compilation unit (and thus the underlying IFile is modified) or one can modify an in-memory copy of the compilation unit called a working copy.

A working copy is obtained from a compilation unit using the getWorkingCopy method. (Note that the compilation unit does not need to exist in the Java model in order for a working copy to be created.) Whoever creates such a working copy is responsible for discarding it when not needed any longer using the discardWorkingCopy method.

Working copies modify an in-memory buffer. The getWorkingCopy() method creates a default buffer, but clients can provide their own buffer implementation using the getWorkingCopy(WorkingCopyOwner, IProgressMonitor) method. Clients can manipulate the text of this buffer directly. If they do so, they must synchronize the working copy with the buffer from time to time using either the reconcile(int, boolean, WorkingCopyOwner, IProgressMonitor) method.

Finally a working copy can be saved to disk (replacing the original compilation unit) using the commitWorkingCopy method.

For example the following code snippet creates a working copy on a compilation unit using a custom working copy owner. The snippet modifies the buffer, reconciles the changes, commits the changes to disk and finally discards the working copy.

// Get original compilation unit

ICompilationUnit originalUnit = ...;

// Get working copy owner

WorkingCopyOwner owner = ...;

// Create working copy

ICompilationUnit workingCopy = originalUnit.getWorkingCopy(owner, null);

// Modify buffer and reconcile

IBuffer buffer = ((IOpenable)workingCopy).getBuffer();

buffer.append("class X {}");

workingCopy.reconcile(ICompilationUnit.NO\_AST, false, null, null);

// Commit changes

workingCopy.commitWorkingCopy(false, null);

// Destroy working copy

workingCopy.discardWorkingCopy();

The compilation unit's buffer can also be modified using the ICompilationUnit.applyTextEdit method.

// Get original compilation unit

ICompilationUnit originalUnit = ...;

// Get working copy owner

WorkingCopyOwner owner = ...;

// Create working copy

ICompilationUnit workingCopy = originalUnit.getWorkingCopy(owner, null);

// Get text edits

TextEdit edit = ...;

// Modify buffer and reconcile

workingCopy.applyTextEdit(edit, null);

workingCopy.reconcile(ICompilationUnit.NO\_AST, false, null, null);

// Commit changes

workingCopy.commitWorkingCopy(false, null);

// Destroy working copy

workingCopy.discardWorkingCopy();

Working copies can also be shared by several clients using a working copy owner. A working copy can be later retrieved using the findWorkingCopy method. A shared working copy is thus keyed on the original compilation unit and on a working copy owner.

The following shows how client 1 creates a shared working copy, client 2 retrieves this working copy, client 1 discards the working copy, and client 2 trying to retrieve the shared working copy notices it does not exist any longer:

// Client 1 & 2: Get original compilation unit

ICompilationUnit originalUnit = ...;

// Client 1 & 2: Get working copy owner

WorkingCopyOwner owner = ...;

// Client 1: Create shared working copy

ICompilationUnit workingCopyForClient1 = originalUnit.getWorkingCopy(owner, null);

// Client 2: Retrieve shared working copy

ICompilationUnit workingCopyForClient2 = originalUnit.findWorkingCopy(owner);

// This is the same working copy

assert workingCopyForClient1 == workingCopyForClient2;

// Client 1: Discard shared working copy

workingCopyForClient1.discardWorkingCopy();

// Client 2: Attempt to retrieve shared working copy and find out it's null

workingCopyForClient2 = originalUnit.findWorkingCopy(owner);

assert workingCopyForClient2 == null;

Code modification using the DOM/AST API

There are three ways to create a CompilationUnit. The first one is to use ASTParser. The second is to use ICompilationUnit#reconcile(...). The third is to start from scratch using the factory methods on AST (Abstract Syntax Tree).

Creating an AST from existing source code

An instance of ASTParser must be created with ASTParser.newParser(int).

The source code is given to the ASTParser with one of the following methods:

setSource(char[]): to create the AST from source code

setSource(IClassFile): to create the AST from a classfile

setSource(ICompilationUnit): to create the AST from a compilation unit

Then the AST is created by calling createAST(IProgressMonitor).

The result is an AST with correct source positions for each node. The resolution of bindings has to be requested before the creation of the tree with setResolveBindings(boolean). Resolving the bindings is a costly operation and should be done only when necessary. As soon as the tree has been modified, all positions and bindings are lost.

Note that some bindings recovery can also be done during this resolution with setBindingsRecovery(boolean). Using this recovery, some bindings - typically missing types - will no longer be null, hence improving the resilience of any clients using the AST tree.

Creating an AST by reconciling a working copy

If a working copy is not consistent (has been modified) then an AST can be created by calling the method reconcile(int, boolean, WorkingCopyOwner, IProgressMonitor). To request AST creation, call the reconcile(...) method with AST.JLS3 as first parameter.

Its bindings are computed only if the problem requestor is active, or if the problem detection is forced. Resolving the bindings is a costly operation and should be done only when necessary. As soon as the tree has been modified, all positions and bindings are lost.

Note that some bindings recovery can also be done during this resolution with by using the method reconcile(int, int, WorkingCopyOwner, IProgressMonitor) with the flag ENABLE\_BINDINGS\_RECOVERY set on the second parameter. Using this recovery, some bindings - typically missing types - will no longer be null, hence improving the resilience of any clients using the AST tree.

From scratch

It is possible to create a CompilationUnit from scratch using the factory methods on AST. These method names start with new.... The following is an example that creates a HelloWorld class.

The first snippet is the generated output:

package example;

import java.util.\*;

public class HelloWorld {

public static void main(String[] args) {

System.out.println("Hello" + " world");

}

}

The following snippet is the corresponding code that generates the output.

AST ast = AST.newAST(AST.JLS3);

CompilationUnit unit = ast.newCompilationUnit();

PackageDeclaration packageDeclaration = ast.newPackageDeclaration();

packageDeclaration.setName(ast.newSimpleName("example"));

unit.setPackage(packageDeclaration);

ImportDeclaration importDeclaration = ast.newImportDeclaration();

QualifiedName name =

ast.newQualifiedName(

ast.newSimpleName("java"),

ast.newSimpleName("util"));

importDeclaration.setName(name);

importDeclaration.setOnDemand(true);

unit.imports().add(importDeclaration);

TypeDeclaration type = ast.newTypeDeclaration();

type.setInterface(false);

type.modifiers().add(ast.newModifier(Modifier.ModifierKeyword.PUBLIC\_KEYWORD));

type.setName(ast.newSimpleName("HelloWorld"));

MethodDeclaration methodDeclaration = ast.newMethodDeclaration();

methodDeclaration.setConstructor(false);

List modifiers = methodDeclaration.modifiers();

modifiers.add(ast.newModifier(Modifier.ModifierKeyword.PUBLIC\_KEYWORD));

modifiers.add(ast.newModifier(Modifier.ModifierKeyword.STATIC\_KEYWORD));

methodDeclaration.setName(ast.newSimpleName("main"));

methodDeclaration.setReturnType2(ast.newPrimitiveType(PrimitiveType.VOID));

SingleVariableDeclaration variableDeclaration = ast.newSingleVariableDeclaration();

variableDeclaration.setType(ast.newArrayType(ast.newSimpleType(ast.newSimpleName("String"))));

variableDeclaration.setName(ast.newSimpleName("args"));

methodDeclaration.parameters().add(variableDeclaration);

org.eclipse.jdt.core.dom.Block block = ast.newBlock();

MethodInvocation methodInvocation = ast.newMethodInvocation();

name =

ast.newQualifiedName(

ast.newSimpleName("System"),

ast.newSimpleName("out"));

methodInvocation.setExpression(name);

methodInvocation.setName(ast.newSimpleName("println"));

InfixExpression infixExpression = ast.newInfixExpression();

infixExpression.setOperator(InfixExpression.Operator.PLUS);

StringLiteral literal = ast.newStringLiteral();

literal.setLiteralValue("Hello");

infixExpression.setLeftOperand(literal);

literal = ast.newStringLiteral();

literal.setLiteralValue(" world");

infixExpression.setRightOperand(literal);

methodInvocation.arguments().add(infixExpression);

ExpressionStatement expressionStatement = ast.newExpressionStatement(methodInvocation);

block.statements().add(expressionStatement);

methodDeclaration.setBody(block);

type.bodyDeclarations().add(methodDeclaration);

unit.types().add(type);

Retrieving extra positions

The DOM/AST node contains only a pair of positions (the starting position and the length of the node). This is not always sufficient. In order to retrieve intermediate positions, the IScanner API should be used. For example, we have an InstanceofExpression for which we want to know the positions of the instanceof operator. We could write the following method to achieve this:

private int[] getOperatorPosition(Expression expression, char[] source) {

if (expression instanceof InstanceofExpression) {

IScanner scanner = ToolFactory.createScanner(false, false, false, false);

scanner.setSource(source);

int start = expression.getStartPosition();

int end = start + expression.getLength();

scanner.resetTo(start, end);

int token;

try {

while ((token = scanner.getNextToken()) != ITerminalSymbols.TokenNameEOF) {

switch(token) {

case ITerminalSymbols.TokenNameinstanceof:

return new int[] {scanner.getCurrentTokenStartPosition(), scanner.getCurrentTokenEndPosition()};

}

}

} catch (InvalidInputException e) {

}

}

return null;

}

The IScanner is used to divide the input source into tokens. Each token has a specific value that is defined in the ITerminalSymbols interface. It is fairly simple to iterate and retrieve the right token. We also recommend that you use the scanner if you want to find the position of the super keyword in a SuperMethodInvocation.

Source code modifications

Some source code modifications are not provided via the Java element API. A more general way to edit source code (such as changing the source code for existing elements) is accomplished using the compilation unit's raw source code and the rewrite API of the DOM/AST.

To perform DOM/AST rewriting, there are two sets of API: the descriptive rewriting and the modifying rewriting.

The descriptive API does not modify the AST but use ASTRewrite API to generate the descriptions of modifications. The AST rewriter collects descriptions of modifications to nodes and translates these descriptions into text edits that can then be applied to the original source.

// creation of a Document

ICompilationUnit cu = ... ; // content is "public class X {\n}"

String source = cu.getSource();

Document document= new Document(source);

// creation of DOM/AST from a ICompilationUnit

ASTParser parser = ASTParser.newParser(AST.JLS3);

parser.setSource(cu);

CompilationUnit astRoot = (CompilationUnit) parser.createAST(null);

// creation of ASTRewrite

ASTRewrite rewrite = ASTRewrite.create(astRoot.getAST());

// description of the change

SimpleName oldName = ((TypeDeclaration)astRoot.types().get(0)).getName();

SimpleName newName = astRoot.getAST().newSimpleName("Y");

rewrite.replace(oldName, newName, null);

// computation of the text edits

TextEdit edits = rewrite.rewriteAST(document, cu.getJavaProject().getOptions(true));

// computation of the new source code

edits.apply(document);

String newSource = document.get();

// update of the compilation unit

cu.getBuffer().setContents(newSource);

The modifying API allows to modify directly the AST:

Request the recording of modifications (CompilationUnit.recordModifications()).

Perform the modifications on the AST Nodes.

And when the modifications are finished, generate text edits that can then be applied to the original source (CompilationUnit.rewrite(...)).

// creation of a Document

ICompilationUnit cu = ... ; // content is "public class X {\n}"

String source = cu.getSource();

Document document= new Document(source);

// creation of DOM/AST from a ICompilationUnit

ASTParser parser = ASTParser.newParser(AST.JLS3);

parser.setSource(cu);

CompilationUnit astRoot = (CompilationUnit) parser.createAST(null);

// start record of the modifications

astRoot.recordModifications();

// modify the AST

TypeDeclaration typeDeclaration = (TypeDeclaration)astRoot.types().get(0);

SimpleName newName = astRoot.getAST().newSimpleName("Y");

typeDeclaration.setName(newName);

// computation of the text edits

TextEdit edits = astRoot.rewrite(document, cu.getJavaProject().getOptions(true));

// computation of the new source code

edits.apply(document);

String newSource = document.get();

// update of the compilation unit

cu.getBuffer().setContents(newSource);

Responding to changes in Java elements

If your plug-in needs to know about changes to Java elements after the fact, you can register a Java IElementChangedListener with JavaCore.

JavaCore.addElementChangedListener(new MyJavaElementChangeReporter());

You can be more specific and specify the type of events you're interested in using addElementChangedListener(IElementChangedListener, int).

For example, if you're only interested in listening for events during a reconcile operation:

JavaCore.addElementChangedListener(new MyJavaElementChangeReporter(), ElementChangedEvent.POST\_RECONCILE);

There are two kinds of events that are supported by JavaCore:

POST\_CHANGE: Listeners of this event kind will get notified during the corresponding POST\_CHANGE resource change notification.

POST\_RECONCILE: Listeners of this event kind will get notified at the end of a reconcile operation on a working copy (see ICompilationUnit.reconcile(int, boolean, WorkingCopyOwner, IProgressMonitor)).

Java element change listeners are similar conceptually to resource change listeners (described in tracking resource changes). The following snippet implements a Java element change reporter that prints the element deltas to the system console.

public class MyJavaElementChangeReporter implements IElementChangedListener {

public void elementChanged(ElementChangedEvent event) {

IJavaElementDelta delta= event.getDelta();

if (delta != null) {

System.out.println("delta received: ");

System.out.print(delta);

}

}

}

The IJavaElementDelta includes the element that was changed and flags describing the kind of change that occurred. Most of the time the delta tree is rooted at the Java Model level. Clients must then navigate this delta using getAffectedChildren to find out what projects have changed.

The following example method traverses a delta and prints the elements that have been added, removed and changed:

void traverseAndPrint(IJavaElementDelta delta) {

switch (delta.getKind()) {

case IJavaElementDelta.ADDED:

System.out.println(delta.getElement() + " was added");

break;

case IJavaElementDelta.REMOVED:

System.out.println(delta.getElement() + " was removed");

break;

case IJavaElementDelta.CHANGED:

System.out.println(delta.getElement() + " was changed");

if ((delta.getFlags() & IJavaElementDelta.F\_CHILDREN) != 0) {

System.out.println("The change was in its children");

}

if ((delta.getFlags() & IJavaElementDelta.F\_CONTENT) != 0) {

System.out.println("The change was in its content");

}

/\* Others flags can also be checked \*/

break;

}

IJavaElementDelta[] children = delta.getAffectedChildren();

for (int i = 0; i < children.length; i++) {

traverseAndPrint(children[i]);

}

}

Since IAnnotations are not children of any Java element, annotation deltas are obtained using getAnnotationDeltas().

Several kinds of operations can trigger a Java element change notification. Here are some examples:

Creating a resource, e.g. IPackageFragment.createCompilationUnit (the delta indicates the addition of the compilation unit)

Modifying a resource, e.g. ICompilationUnit.createType (the delta indicates that the compilation unit has changed and that a type was added as a child of this compilation unit)

Modifying a project's classpath, e.g. IJavaProject.setRawClasspath (the delta indicates that package fragment roots have been added to the classpath, removed from the classpath, or reordered on the classpath)

Modifying a classpath variable value, e.g. JavaCore.setClasspathVariable (the delta also indicates that package fragment roots have been affected)

Reconciling a working copy with its buffer, e.g. ICompilationUnit.reconcile

Modifying an IFile that ends with ".java" and that is on the project's classpath, e.g. using IFile.setContents (the delta indicates that a compilation unit was changed, but no finer-grained information is provided as this was not done through a Java Model operation)

Similar to IResourceDelta the Java element deltas can be batched using an IWorkspaceRunnable. The deltas resulting from several Java Model operations that are run inside a IWorkspaceRunnable are merged and reported at once.

JavaCoreprovides a run method for batching Java element changes.

For example, the following code fragment will trigger 2 Java element change events:

// Get package

IPackageFragment pkg = ...;

// Create 2 compilation units

ICompilationUnit unitA = pkg.createCompilationUnit("A.java", "public class A {}", false, null);

ICompilationUnit unitB = pkg.createCompilationUnit("B.java", "public class B {}", false, null);

Whereas the following code fragment will trigger 1 Java element change event:

// Get package

final IPackageFragment pkg = ...;

// Create 2 compilation units

JavaCore.run(

new IWorkspaceRunnable() {

public void run(IProgressMonitor monitor) throws CoreException {

ICompilationUnit unitA = pkg.createCompilationUnit("A.java", "public class A {}", false, null);

ICompilationUnit unitB = pkg.createCompilationUnit("B.java", "public class B {}", false, null);

}

},

null);

1.1.3. Setting the Java build path

This section describes how to set the Java build path. The build path is the classpath that is used for building a Java project (IJavaProject).

A classpath is simply an array of classpath entries (IClasspathEntry) that describe the types that are available. The types can appear in source or binary form and the ordering of the entries on the path defines the lookup order for resolving types during a build.

The Java build path is reflected in the structure of a Java project element. You can query a project for its package fragment roots (IPackageFragmentRoot). Each classpath entry maps to one or more package fragment roots, each of which further contains a set of package fragments.

This discussion of the build path does not involve the Java runtime path, which can be defined separately from the build path. (See Running Java code for a discussion of the runtime classpath.)

Changing the build path

You can programmatically change a project's build path using setRawClasspath on the corresponding project's Java element. The following code sets the classpath for a project resource:

IProject project = ... // get some project resource

IJavaProject javaProject = JavaCore.create(project);

IClasspathEntry[] newClasspath = ...;

javaProject.setRawClasspath(newClasspath, someProgressMonitor);

(Note: The use of the term "raw" classpath is used to emphasize the fact that any variables used to describe entry locations have not been resolved.)

The Java build path is persisted into a file named '.classpath' in the project's file structure. The purpose of this file is to provide a way to share Java build path settings with others through some source code repository. In particular, this file should not be manually edited, since it may get corrupted.

Classpath entries

Classpath entries can be defined using factory methods defined on JavaCore. Classpath entries can reference any of the following:

a source folder - a folder containing source compilation units organized under their corresponding package directory structure. Source folders are used to better structure source files in large projects, and may only be referenced within the containing project. The corresponding factory method is newSourceEntry. Inside a given source folder, each compilation unit is expected to be nested in the appropriate folder structure according to its package statement. For example, compilation unit 'X.java' in package 'p1' must be located inside sub-folder 'p1' of a source folder. It is possible to use multiple source folders, as long as they don't overlap. A source folder may be assigned its own output location which determines where generated class files should be placed. If none is specified, then class files will be placed in the containing project's output location (see IJavaProject.setOutputLocation).

The following is an example classpath entry that denotes the source folder 'src' of project 'MyProject':

IClasspathEntry srcEntry = JavaCore.newSourceEntry(new Path("/MyProject/src"));

a binary library - either a class file folder (contained inside or outside the workspace) or a class file archive file (contained inside or outside the workspace). Archive libraries can have attached source archives, which are extracted when asking a class file element for its source (getSource). The factory method for libraries is newLibraryEntry.

The following is an example classpath entry that denotes the class file folder 'lib' of 'MyProject':

IClasspathEntry libEntry = JavaCore.newLibraryEntry(

new Path("/MyProject/lib"),

null, // no source

null, // no source

false); // not exported

The following classpath entry has a source attachment:

IClasspathEntry libEntry = JavaCore.newLibraryEntry(

new Path("d:/lib/foo.jar"), // library location

new Path("d:/lib/foo\_src.zip"), // source archive location

new Path("src"), // source archive root path

true); // exported

The source archive root path describes the location of the root within the source archive. If set to null, the root of the archive will be inferred dynamically.

a prerequisite project - another Java project. A prerequisite project always contributes its source folders to dependent projects. It can also optionally contribute any of its classpath entries which are tagged as exported (see factory methods supporting the extra boolean argument 'isExported'). This means that in addition to contributing its source to its dependents, a project will also export all classpath entries tagged as such. This allows prerequisite projects to better hide their own structure changes. For example, a given project may choose to switch from using a source folder to exporting a library. This can be done without requiring its dependent projects to change their classpath. The factory method for a project prerequisite is newProjectEntry.

The following classpath entry denotes a prerequisite project 'MyFramework'.

IClasspathEntry prjEntry = JavaCore.newProjectEntry(new Path("/MyFramework"), true); // exported

an indirect reference to a project or library, using some classpath variable - The location of projects or libraries can be dynamically resolved relative to a classpath variable, which is specified as the first segment of the entry path. The remainder of the entry path is then appended to the resolved variable path. The factory method for a classpath variable is newVariableEntry. Classpath variables are global to the workspace, and can be manipulated through JavaCore methods getClasspathVariable and setClasspathVariable.

It is possible to register an automatic classpath variable initializer which is invoked through the extension point org.eclipse.jdt.core.classpathVariableInitializer when the workspace is started.

The following classpath entry denotes a library whose location is kept in the variable 'HOME'. The source attachment is defined using the variables 'SRC\_HOME' and 'SRC\_ROOT' :

IClasspathEntry varEntry = JavaCore.newVariableEntry(

new Path("HOME/foo.jar"), // library location

new Path("SRC\_HOME/foo\_src.zip"), // source archive location

new Path("SRC\_ROOT"), // source archive root path

true); // exported

JavaCore.setClasspathVariable("HOME", new Path("d:/myInstall"), null); // no progress monitor

entry denoting a classpath container - an indirect reference to a structured set of project or libraries. Classpath containers are used to refer to a set of classpath entries that describe a complex library structure. Like classpath variables, classpath containers (IClasspathContainer) are dynamically resolved. Classpath containers may be used by different projects, causing their path entries to resolve to distinct values per project. They also provide meta information about the library that they represent (name, kind, description of library.) The factory method for a classpath variable is newContainerEntry. Classpath containers can be manipulated through JavaCore methods getClasspathContainer and setClasspathContainer.

It is possible to register an automatic classpath container initializer which is lazily invoked through the extension point org.eclipse.jdt.core.classpathContainerInitializer when the container needs to be bound.

The following classpath entry denotes a system class library container:

IClasspathEntry varEntry = JavaCore.newContainerEntry(

new Path("JDKLIB/default"), // container 'JDKLIB' + hint 'default'

false); // not exported

JavaCore.setClasspathContainer(

new Path("JDKLIB/default"),

new IJavaProject[]{ myProject }, // value for 'myProject'

new IClasspathContainer[] {

new IClasspathContainer() {

public IClasspathEntry[] getClasspathEntries() {

return new IClasspathEntry[]{

JavaCore.newLibraryEntry(new Path("d:/rt.jar"), null, null, false);

};

}

public String getDescription() { return "Basic JDK library container"; }

public int getKind() { return IClasspathContainer.K\_SYSTEM; }

public IPath getPath() { return new Path("JDKLIB/basic"); }

}

},

null);

Exclusion patterns

A classpath source entry may be assigned an exclusion pattern, which prevents certain resources in a source folder from being visible on the classpath. Using a pattern allows specified portions of the resource tree to be filtered out. Each exclusion pattern path is relative to the classpath entry and uses a pattern mechanism similar to Ant. Exclusion patterns can be used to specify nested source folders as long as the outer pattern excludes the inner pattern.

See getExclusionPatterns for more detail on exclusion patterns.

The Java project API isOnClasspath checks both inclusion and exclusion patterns before determining whether a particular resource is on the classpath.

Remarks:

Exclusion patterns have higher precedence than inclusion patterns; in other words, exclusion patterns can remove files from the ones that are to be included, not the other way around.

A nested source folder excluded from build path can be set as an output location. The following is an example classpath entry that denotes the source folder 'src' of project 'MyProject' with an excluded nested source folder used as an output location:

IPath sourceFolder = new Path("/MyProject/src");

IPath outputLocation = sourceFolder.append("bin");

IClasspathEntry srcEntry = JavaCore.newSourceEntry(

sourceFolder, // source folder location

new Path[] { outputLocation }, // excluded nested folder

outputLocation); // output location

Inclusion patterns

A classpath source entry may also be assigned an inclusion pattern, which explicitly defines resources to be visible on the classpath. When no inclusion patterns are specified, the source entry includes all relevant files in the resource tree rooted at this source entry's path. Specifying one or more inclusion patterns means that only the specified portions of the resource tree are to be included. Each path specified must be a relative path, and will be interpreted relative to this source entry's path. File patterns are case-sensitive. A file matched by one or more of these patterns is included in the corresponding package fragment root unless it is excluded by one or more of this entry's exclusion patterns.

See getExclusionPatterns for a discussion of the syntax and semantics of path patterns. The absence of any inclusion patterns is semantically equivalent to the explicit inclusion pattern \*\*.

The Java project API isOnClasspath checks both inclusion and exclusion patterns before determining whether a particular resource is on the classpath.

Examples:

The inclusion pattern src/\*\* by itself includes all files under a root folder named src.

The inclusion patterns src/\*\* and tests/\*\* includes all files under the root folders named src and tests.

The inclusion pattern src/\*\* together with the exclusion pattern src/\*\*/Foo.java includes all files under a root folder named src except for ones named Foo.java.

Classpath resolution

Since classpath variables and containers allow you to define dynamically bound classpath entries, the classpath API distinguishes between a raw and a resolved classpath. The raw classpath is the one originally set on the Java project using setRawClasspath, and can be further queried by asking the project for getRawClasspath. The resolved classpath can be queried using getResolvedClasspath. This operation triggers initialization of any variables and containers necessary to resolve the classpath. Many Java Model operations implicitly cause the Java build path to be resolved. For example, computing a project's package fragment roots requires the build path to be resolved.

1.1.4. Compiling Java code

See the Java development user guide for using the batch compiler and using the ant javac adapter.

The JDT plug-ins include an incremental and batch Java compiler for building Java .class files from source code. There is no direct API provided by the compiler. It is installed as a builder on Java projects. Compilation is triggered using standard platform build mechanisms.

The platform build mechanism is described in detail in Incremental project builders.

Compiling code

You can programmatically compile the Java source files in a project using the build API.

IProject myProject;

IProgressMonitor myProgressMonitor;

myProject.build(IncrementalProjectBuilder.INCREMENTAL\_BUILD, myProgressMonitor);

For a Java project, this invokes the Java incremental project builder (along with any other incremental project builders that have been added to the project's build spec). The generated .class files are written to the designated output folder. Additional resource files are also copied to the output folder.

In the case of a full batch build, all the .class files in the output folder may be 'scrubbed' to ensure that no stale files are found. This is controlled using a JDT Core Builder Option (CORE\_JAVA\_BUILD\_CLEAN\_OUTPUT\_FOLDER). The default for this option is to clean output folders. Unless this option is reset, you must ensure that you place all .class files for which you do not have corresponding source files in a separate class file folder on the classpath instead of the output folder.

The incremental and batch builders can be configured with other options that control which resources are copied to the output folder. The following sample shows how to set up a resource filter so that files ending with '.ignore' and folders named 'META-INF', are not copied to the output folder:

Hashtable options = JavaCore.getOptions();

options.put(JavaCore.CORE\_JAVA\_BUILD\_RESOURCE\_COPY\_FILTER, "\*.ignore,META-INF/");

JavaCore.setOptions(options);

Filenames are filtered out if they match one of the supplied patterns. Entire folders are filtered out if their name matches one of the supplied folder names which end in a path separator.

The incremental and batch builders can also be configured to only generate a single error when the .classpath file has errors. This option is set by default and eliminates numerous errors. See JDT Core Builder Options for a complete list of builder-related options and their defaults.

The compiler can also be configured using JavaCore options. For example, you can define the severity that should be used for different kinds of problems that are found during compilation. See JDT Core Compiler Options for a complete list of compiler-related options and their defaults.

When programmatically configuring options for the builder or compiler, you should specify the scope of the option. For example, setting up a resource filter may apply to a particular project only:

Hashtable options = myProject.getOptions(false); // get only the options set up in this project

options.put(JavaCore.CORE\_JAVA\_BUILD\_RESOURCE\_COPY\_FILTER, "\*.ignore,META-INF/");

myProject.setOptions(options);

Problem determination

JDT Core defines a specialized marker (marker type "org.eclipse.jdt.core.problem ") to denote compilation problems. To programmatically discover problems detected by the compiler, the standard platform marker protocol should be used. See Resource Markers for an overview of using markers.

The following snippet finds all Java problem markers in a compilation unit.

public IMarker[] findJavaProblemMarkers(ICompilationUnit cu)

throws CoreException {

IResource javaSourceFile = cu.getUnderlyingResource();

IMarker[] markers =

javaSourceFile.findMarkers(IJavaModelMarker.JAVA\_MODEL\_PROBLEM\_MARKER,

true, IResource.DEPTH\_INFINITE);

}

Java problem markers are maintained by the Java project builder and are removed automatically as problems are resolved and the Java source is recompiled.

The problem id value is set to one of the constants defined in IProblem . The problem's id is reliable, but the message is localized and therefore can be changed according to the default locale. The constants defined in IProblem are self-descriptive.

An implementation of IProblemRequestor should be defined to collect the problems discovered during a Java operation. Working copies can be reconciled with problem detection if a IProblemRequestor has been supplied for the working copy creation. To achieve this, you can use the reconcile method. Here is an example:

ICompilationUnit unit = ..; // get some compilation unit

// create requestor for accumulating discovered problems

IProblemRequestor problemRequestor = new IProblemRequestor() {

public void acceptProblem(IProblem problem) {

System.out.println(problem.getID() + ": " + problem.getMessage());

}

public void beginReporting() {}

public void endReporting() {}

public boolean isActive() { return true; } // will detect problems if active

};

// use working copy to hold source with error

ICompilationUnit workingCopy = unit.getWorkingCopy(new WorkingCopyOwner() {}, problemRequestor, null);

((IOpenable)workingCopy).getBuffer().setContents("public class X extends Zork {}");

// trigger reconciliation

workingCopy.reconcile(NO\_AST, true, null, null);

You can add an action on the reported problems in the acceptProblem(IProblem) method. In this example, the reported problem will be that Zork cannot be resolved or is not a valid superclass and its id is IProblem.SuperclassNotFound.

1.1.5. Using the Java search engine

Your plug-in can use the JDT API to search Java projects in the workspace for Java elements, such as method references, field declarations, implementors of an interface, etc.

The entry point for Java search is the SearchEngine class. You can search for particular patterns inside a Java element and scope the search to specific elements. Search patterns can be created using createPattern. A pattern is scoped using createJavaSearchScope. Once a pattern and scope are defined, the search method is used to collect the results.

Search results are reported to a SearchRequestor which you must extend in order to access the results.

Preparing for search

A search operation will use both a pattern for describing the nature of the search, and a scope for restraining the range of investigation.

Creating a Java search pattern

A search pattern defines how search results are found. You can either create a search pattern from a Java element (see createPatternPattern(IJavaElement element, int limitTo)) or from a string (see createPattern(String, int, int, int).) The last method supports wildcards (i.e. '\*') and can be used to widen the search results.

For example, creating a search pattern for searching for references to a given method is done as follows:

// Get the method

IMethod method = ...;

// Create search pattern

SearchPattern pattern = SearchPattern.createPattern(method, IJavaSearchConstants.REFERENCES);

Or creating a search pattern for searching for declarations of all types starting with "Obj":

// Create search pattern

SearchPattern pattern = SearchPattern.createPattern("Obj\*", IJavaSearchConstants.TYPE, IJavaSearchConstants.DECLARATIONS, SearchPattern.R\_PATTERN\_MATCH | SearchPattern.R\_CASE\_SENSITIVE);

The following search patterns are supported:

Package declarations

Type declarations

Field declarations

Method (and constructor) declarations

Package references

Type references

Interface implementors

Field references

Field write accesses

Module declarations

Module references

Field read accesses

Method (and constructor) references

Combinations of the above patterns using the OR pattern (see createOrPattern)

Note that these patterns are created using the following possible rules:

R\_EXACT\_MATCH

R\_PREFIX\_MATCH

R\_PATTERN\_MATCH

R\_REGEXP\_MATCH

R\_CAMELCASE\_MATCH

R\_CAMELCASE\_SAME\_PART\_COUNT\_MATCH

which may be also combined with one of the following flags:

R\_CASE\_SENSITIVE

R\_ERASURE\_MATCH

R\_EQUIVALENT\_MATCH

For example,

// a case insensitive prefix match is requested

SearchPattern pattern1 = SearchPattern.createPattern("Hash", IJavaSearchConstants.TYPE, IJavaSearchConstants.DECLARATIONS, SearchPattern.R\_PREFIX\_MATCH);

// a camel case match is requested

SearchPattern pattern2 = SearchPattern.createPattern("HM", IJavaSearchConstants.TYPE, IJavaSearchConstants.DECLARATIONS, SearchPattern.R\_CAMEL\_CASE\_MATCH);

// a camel case with a strict expected number of parts match is requested

SearchPattern pattern3 = SearchPattern.createPattern("HM", IJavaSearchConstants.TYPE, IJavaSearchConstants.DECLARATIONS, SearchPattern.R\_CAMELCASE\_SAME\_PART\_COUNT\_MATCH);

Note that using the patterns created above reduce the number of possible results among others:

pattern1: Hashtable, HashMap, HashMapEntry, etc.

pattern2: HashMap, HashMapEntry, etc.

pattern3: HashMapEntry, etc.

Using fine grain flags in a Java search pattern

Some references patterns can be refined by adding one or several fine grain flags to the limitTo parameter.

For example, only the type references used in a cast expression will match the pattern created as follows:

// Get the type

IType type = ...;

// Create search pattern

SearchPattern pattern = SearchPattern.createPattern(type, IJavaSearchConstants.REFERENCES | IJavaSearchConstants.CAST\_TYPE\_REFERENCE);

Note that the fine grain flags can be combined together but only for the same kind of search (e.g. only a combination of flags for type references will be meaningful if the searchFor parameter is set to TYPE).

Flags for type references:

FIELD\_DECLARATION\_TYPE\_REFERENCE: type references used as the type of a field declaration.

LOCAL\_VARIABLE\_DECLARATION\_TYPE\_REFERENCE: type references used as the type of a local variable declaration.

PARAMETER\_DECLARATION\_TYPE\_REFERENCE: type references used as the type of a method parameter.

SUPERTYPE\_TYPE\_REFERENCE: type references used as a super type or as a super interface.

THROWS\_CLAUSE\_TYPE\_REFERENCE: type references used in a throws clause.

CAST\_TYPE\_REFERENCE: type references used in a cast expression.

CATCH\_TYPE\_REFERENCE: type references used in a catch header.

CLASS\_INSTANCE\_CREATION\_TYPE\_REFERENCE: type references used in class instance creation.

RETURN\_TYPE\_REFERENCE: type references used as a method return type.

IMPORT\_DECLARATION\_TYPE\_REFERENCE: type references used in an import declaration.

ANNOTATION\_TYPE\_REFERENCE: type references used as an annotation.

TYPE\_ARGUMENT\_TYPE\_REFERENCE: type references used as a type argument in a parameterized type or a parameterized method.

TYPE\_VARIABLE\_BOUND\_TYPE\_REFERENCE: type references used as a type variable bound.

WILDCARD\_BOUND\_TYPE\_REFERENCE: type references used as a wildcard bound.

INSTANCEOF\_TYPE\_REFERENCE: type references used in an instance of condition.

Flags for field or method references:

SUPER\_REFERENCE: super field accesses or super method invocations (e.g. using the super qualifier).

QUALIFIED\_REFERENCE: qualified field accesses or qualified method invocations.

THIS\_REFERENCE: primary field accesses or primary method invocations (e.g. using the this qualifier).

IMPLICIT\_THIS\_REFERENCE: field accesses or method invocations without any qualification.

Creating a Java search scope

If you are interested in search results in a given project or even in a given package, or if you know that search results can only be found in the hierarchy of a given type, you can create the appropriate search scope using createJavaSearchScope(IJavaElement[]) or createHierarchyScope(IType).

For example, creating a search scope on a given package is done as follows:

// Get the package

IPackageFragment pkg = ...;

// Create search scope

IJavaSearchScope scope = SearchEngine.createJavaSearchScope(new IJavaElement[] {pkg});

Or creating a search scope on the hierarchy of a given type is:

// Get the type

IType type = ...;

// Create search scope

IJavaSearchScope scope = SearchEngine.createHierarchyScope(type);

Finally, you can create a search scope comprising the entire workspace using createWorkspaceScope:

// Create search scope

IJavaSearchScope scope = SearchEngine.createWorkspaceScope();

Searching

Once you have created a search pattern and a search scope, and you have extended SearchRequestor, you can start a search query as follows:

// Get the search pattern

SearchPattern pattern = ...;

// Get the search scope

IJavaSearchScope scope = ...;

// Get the search requestor

SearchRequestor requestor = ...;

// Search

SearchEngine searchEngine = new SearchEngine();

searchEngine.search(pattern, new SearchParticipant[] {SearchEngine.getDefaultSearchParticipant()}, scope, requestor, null);

A notification that the search starts is sent to your search requestor using the beginReporting method. Then, each search result is reported using the acceptSearchMatch method. Finally endReporting indicates that the search has ended.

Collecting search results

Search results are reported using the acceptSearchMatch(SearchMatch) method. Paragraphs below highlight some features of SearchMatch.

Resources and Java elements

A search result can correspond to a Java element (e.g. a type declaration) or it can be contained in a Java element (e.g. a reference to a type inside a method). The search engine always tries to find the innermost Java element that corresponds to or that contains the search result. For example, searching for references to a method could find such a reference in an initializer. The initializer that contains this method reference is the element of the search match.

The search engine also tries to find the resource that contains the Java element. If the Java element is contained in a compilation unit or a class file, the resource is the corresponding IFile. If the Java element is contained in a .jar file, the returned resource is that .jar file if it is in the workspace, null otherwise.

Source positions

Source positions getOffset and getLength are given relative to the compilation unit that contains the search result. If the search result is contained in a .jar file, the source positions are relative to the attached source. They are (-1, -1) if there is no source attached to the .jar file.

Accurate versus inaccurate search results

In most cases search results are accurate, meaning that the search engine was able to determine that the given match is what was asked for. However in some cases the search engine is unable to do so, in such cases the match is inaccurate. Some possible reasons why a match could be inaccurate are:

The classpath on the project that contains the result is not properly set. For example, it refers to a project that is not accessible, a jar on the classpath requires another jar that is not on the classpath, etc.

The user code would not compile. For example, it refers to a class that is not yet defined.

1.1.6. Indexes for the Java search

The JDT search creates indexes for all the sources and the dependent JARs. As a JAR gets added onto the classpath, it is indexed. This operation could take some time depending upon the size of the JAR. To improve the performance, one can pre-build the index and specify the index file for a JAR when it gets added onto the classpath. This is only supported for JARs.

Generating the index

An index can be pre-generated in the following ways:

An API JavaIndexer#generateIndexForJar.

A command line application JavaIndexer

An Ant task eclipse.buildJarIndex

Specifying the index

The index can be specified as a classpath attribute of the library that is getting added. The index file has to be in a valid URL form. If the index file is not found, JDT generates an index file and uses it for searches.

1.1.7. Performing code assist on Java code

The JDT API allows other plug-ins to perform code assist or code select on some Java elements. Elements that allow this manipulation should implement ICodeAssist.

There are two kinds of manipulation:

Code completion - compute the completion of a Java token.

Code selection - answer the Java element indicated by the selected text of a given offset and length.

In the Java model there are two elements that implement this interface: IClassFile and ICompilationUnit. Code completion and code selection only answer results for a class file if it has attached source.

Code completion

Performing a code completion

One way to programmatically perform code completion is to invoke ICodeAssist.codeComplete. You specify the offset in the compilation unit after which the code completion is desired. You must also supply an instance of CompletionRequestor to accept the possible completions.

The method in CompletionRequestor.accept(CompletionProposal) accepts all kinds of proposals for code completion. The methods of CompletionProposal give information that describes the proposed element (its name, declaring type, etc.), its proposed position for insertion in the compilation unit, and its relevance.

A completion requestor can accept many different kinds of completions. This kind is given by CompletionProposal.getKind.

Some of the possible completion kinds are (The complete list of possible completion kinds can be seen on CompletionProposal):

annotation attribute - ANNOTATION\_ATTRIBUTE\_REF

anonymous type - ANONYMOUS\_CLASS\_DECLARATION

type reference - TYPE\_REF

field reference- FIELD\_REF

keyword - KEYWORD

label reference - LABEL\_REF

local variable reference - LOCAL\_VARIABLE\_REF

method reference - METHOD\_REF

method declaration - METHOD\_DECLARATION

package import or reference - PACKAGE\_REF

variable name - VARIABLE\_DECLARATION

The completion requestor must also be able to accept compilation errors.

Completion relevance

Because there may be many different possible completions, the notion of relevance is used to compare the relevance of a suggested completion to other proposals. Relevance is represented by a positive integer. The value has no implicit meaning except to be used relative to the value for other proposals. The relevance of a code completion candidate can be affected by the expected type of the expression, as it relates to the types in the surrounding code, such as variable types, cast types, return types, etc. The presence of an expected prefix or suffix in a completion also affects its relevance.

Completion context

An instance of CompletionRequestor can also accept a completion context. This context is given by the method CompletionRequestor.acceptContext(CompletionContext) and does not depend on a specific completion proposal. The methods of CompletionContext give information that describe the general context like the offset of completion, the completed token, the completed token kind (name or string literal) and its position.

A CompletionContext can also give some information about elements (IJavaElement) which are related to the completion location. These elements are based on the content of the completed compilation unit's buffer and are not the result of the last reconcile operation.

Some of these methods are:

getEnclosingElement() - This method returns the innermost enclosing element which contains the completion location

getVisibleElements(String) - This method returns the elements which are visible from the completion location and which can be assigned to the given type

Code completion options

The JDT Core plug-in defines options that control the behavior of code completion. These options can be changed by other plug-ins.

Activate Visibility Sensitive Completion

When this option is active, code completion will not answer elements that are not visible in the current context. (For example, it will not answer private methods of a super class.)

Automatic Qualification of Implicit Members

When this option is active, completion automatically qualifies completion on implicit field references and message expressions.

Additional options allow you to specify prefixes and suffixes for the proposed completion names for fields, static fields, local variables, and method arguments.

See JDT Core Code Assist Options for more information about the code assist options and their defaults.

Code selection

Performing a code selection

Code selection is used to find the Java element represented by a range of text (typically the selected text) in a compilation unit. To programmatically perform code selection, you must invoke ICodeAssist.codeSelect. You must supply the starting index location of the selection and its length. The result is an array of Java elements. Most of the time there is only one element in the array, but if the selection is ambiguous then all the possible elements are returned.

In the following example, code select is invoked for a compilation unit.

// Get the compilation unit

ICompilationUnit unit = ...;

// Get the offset and length

int offset = ...;

int length = ...;

// perform selection

IJavaElement[] elements = unit.codeSelect(offset, length);

System.out.println("the selected element is " + element[0].getElementName());

Selection at cursor location

When the selection length is specified as 0, a selection will be computed by finding the complete token that encloses the specified offset. Consider the following example method:

public void fooMethod(Object) {

}

If you specify an offset after the first character of fooMethod, and you specify a length of 0, then the selection will be computed to include the entire token fooMethod. If instead, you specify a length of 5, the selection will considered as ooMet.

1.1.8. JDT Core options

JDT Core options control the behavior of core features such as the Java compiler, code formatter, code assist, and other core behaviors. The APIs for accessing the options are defined in JavaCore. Options can be accessed as a group as follows:

JavaCore.getDefaultOptions() - Answers the default value of the options.

JavaCore.getOptions() - Answers the current values of the options.

JavaCore.setOptions(Hashtable newOptions) - Replaces the options values by new values.

Options can also be accessed individually by a string name.

JavaCore.getOption(String optionName) - Answers the value of a specific option.

Options that can configure the severity of a problem can also be found.

JavaCore.getOptionForConfigurableSeverity(int problemID) - Answers the option to use to configure the severity of the problem identified by problemID.

Options are stored as a hash table of all known configurable options with their values. Helper constants have been defined on JavaCore for each option ID and its possible constant values.

The following code fragment restores the value of all core options to their defaults except for one (COMPILER\_PB\_DEPRECATION), which is set specifically.

// Get the default options

Hashtable options = JavaCore.getDefaultOptions();

// Change the value of an option

options.put(JavaCore.COMPILER\_PB\_DEPRECATION, JavaCore.ERROR);

// Set the new options

JavaCore.setOptions(options);

The following code fragment keeps the value of the current options and modifies only one (COMPILER\_PB\_DEPRECATION):

// Get the current options

Hashtable options = JavaCore.getOptions();

// Change the value of an option

options.put(JavaCore.COMPILER\_PB\_DEPRECATION, JavaCore.ERROR);

// Set the new options

JavaCore.setOptions(options);

Project specific options

The values of options can be overridden per project using protocol in IJavaProject.

The following code fragment retrieves the value of an option (COMPILER\_PB\_DEPRECATION) for a specific project in two different ways. The boolean parameter controls whether only the project-specific options should be returned in a query or whether the project's option values should be merged with the values in JavaCore.

// Get the project

IJavaProject project = ...;

// See if the value of an option has been set in this project

String value = project.getOption(JavaCore.COMPILER\_PB\_DEPRECATION, false);

if (value == null) {

// no specific option was set on the project

...

}

// Get the value of an option from this project. Use the value from

// JavaCore value if none is specified for the project

String value = project.getOption(JavaCore.COMPILER\_PB\_DEPRECATION, true);

JDT Core options descriptions

The following tables describe the available JDT Core options. The option id is shown in parentheses and the default value is shown in bold italics.

Options categories

Compiler options

Builder options

JavaCore options

Formatter options

CodeAssist options

Compiler options

Description Values

Annotation Based Null Analysis (COMPILER\_ANNOTATION\_NULL\_ANALYSIS)

This option controls whether the compiler will use null annotations for improved analysis of (potential) null references.

When enabled, the compiler will interpret the annotation types defined using COMPILER\_NONNULL\_ANNOTATION\_NAME and COMPILER\_NULLABLE\_ANNOTATION\_NAME as specifying whether or not a given type includes the value null.

The effect of these analyses is further controlled by the options COMPILER\_PB\_NULL\_SPECIFICATION\_VIOLATION, COMPILER\_PB\_NULL\_ANNOTATION\_INFERENCE\_CONFLICT, COMPILER\_PB\_NULL\_UNCHECKED\_CONVERSION, COMPILER\_PB\_REDUNDANT\_NULL\_ANNOTATION, COMPILER\_PB\_MISSING\_NONNULL\_BY\_DEFAULT\_ANNOTATION, COMPILER\_PB\_SYNTACTIC\_NULL\_ANALYSIS\_FOR\_FIELDS, COMPILER\_PB\_NONNULL\_PARAMETER\_ANNOTATION\_DROPPED, and COMPILER\_INHERIT\_NULL\_ANNOTATIONS.

ENABLED

DISABLED

Inline JSR Bytecode Instruction (COMPILER\_CODEGEN\_INLINE\_JSR\_BYTECODE)

When enabled in conjunction with a Java target platform lesser than or equal to "1.4", the compiler will no longer generate JSR instructions, but rather inline corresponding subroutine code sequences (mostly corresponding to try finally blocks). The generated code will thus get bigger, but will load faster on virtual machines since the verification process is then much simpler. This mode is adding support for the Java Specification Request 202 to pre-"1.5" Java target platforms.

For a Java target platform greater than or equal to "1.5", the inlining of the JSR bytecode instruction is mandatory and this option is ignored. ENABLED

DISABLED

Generating Method Parameters Attribute (COMPILER\_CODEGEN\_METHOD\_PARAMETERS\_ATTR)

When generated, this attribute will enable information about the formal parameters of a method (such as their names) to be accessed from reflection libraries, annotation processing, code weaving, and in the debugger, from platform target level 1.8 and later. GENERATE

DO\_NOT\_GENERATE

Setting Target Java Platform (COMPILER\_CODEGEN\_TARGET\_PLATFORM)

For binary compatibility reason, .class files are tagged with VM versions that are defined for each level of the reference specification. The target Java platform specifies the minimum runtime level required to execute the generated class files.

The compliance, source and target levels must satisfy a set of constraints summarized in a compatibility table below. VERSION\_1\_1

VERSION\_1\_2

VERSION\_1\_3

VERSION\_1\_4

VERSION\_1\_5

VERSION\_1\_6

VERSION\_1\_7

VERSION\_1\_8

VERSION\_CLDC\_1\_1

Preserving Unused Local Variables (COMPILER\_CODEGEN\_UNUSED\_LOCAL)

Unless requested to preserve unused local variables (i.e. never read), the compiler will optimize them out, potentially altering debugging. PRESERVE

OPTIMIZE\_OUT

Setting Compliance Level (COMPILER\_COMPLIANCE)

Select the compliance level for the compiler, which will then behave according to the said level of the reference specification.

The compliance, source and target levels must satisfy a set of constraints summarized in a compatibility table below. VERSION\_1\_3

VERSION\_1\_4

VERSION\_1\_5

VERSION\_1\_6

VERSION\_1\_7

VERSION\_1\_8

VERSION\_9

Javadoc Comment Support (COMPILER\_DOC\_COMMENT\_SUPPORT)

When this support is disabled, the compiler will ignore all javadoc problems options settings and will not report any javadoc problem. It will also not find any reference in javadoc comment and DOM AST Javadoc node will be only a flat text instead of having structured tag elements. ENABLED

DISABLED

Inheritance of Null Annotations (COMPILER\_INHERIT\_NULL\_ANNOTATIONS)

When enabled, the compiler will check for each method without any explicit null annotations (see COMPILER\_ANNOTATION\_NULL\_ANALYSIS): If it overrides a method which has null annotations, it will treat the current method as if it had the same annotations as the overridden method.

Annotation inheritance will use the effective nullness of the overridden method after transitively applying inheritance and after applying any default nullness (see COMPILER\_NONNULL\_BY\_DEFAULT\_ANNOTATION\_NAME) at the site of the overridden method.

If different implicit null annotations (from a nonnull default and/or overridden methods) are applicable to the same type in a method signature, this is flagged as an error and an explicit null annotation must be used to disambiguate.

ENABLED

DISABLED

Generating Line Number Debug Attribute (COMPILER\_LINE\_NUMBER\_ATTR)

When generated, this attribute will enable source code highlighting in the debugger (.class file is then bigger). GENERATE

DO\_NOT\_GENERATE

Generating Local Variable Debug Attribute (COMPILER\_LOCAL\_VARIABLE\_ATTR)

When generated, this attribute will enable local variable names to be displayed in the debugger, only in places where variables are definitely assigned (.class file is then bigger). GENERATE

DO\_NOT\_GENERATE

Name of Annotation Type for Non-Null Types (COMPILER\_NONNULL\_ANNOTATION\_NAME)

This option defines a fully qualified Java type name that the compiler may use to perform special null analysis.

If the annotation specified by this option is applied to a type in a method signature or variable declaration, this will be interpreted as a specification that null is not a legal value in that position. Currently supported positions are: method parameters, method return type and local variables.

For values declared with this annotation, the compiler will never trigger a null reference diagnostic (as controlled by COMPILER\_PB\_POTENTIAL\_NULL\_REFERENCE and COMPILER\_PB\_NULL\_REFERENCE), because the assumption is made that null will never occur at runtime in these positions.

The compiler may furthermore check adherence to the null specification as further controlled by COMPILER\_PB\_NULL\_SPECIFICATION\_VIOLATION, COMPILER\_PB\_NULL\_ANNOTATION\_INFERENCE\_CONFLICT and COMPILER\_PB\_NULL\_UNCHECKED\_CONVERSION.

This option only has an effect if the option COMPILER\_ANNOTATION\_NULL\_ANALYSIS is enabled.

The qualified name of a Java annotation type

Default is: org.eclipse.jdt.annotation.NonNull

Names of Secondary Annotation Types for Non-Null Types (COMPILER\_NONNULL\_ANNOTATION\_SECONDARY\_NAMES)

This option defines a comma-separated list of fully qualified Java type names that the compiler may use to perform special null analysis.

The annotation types identified by the names in this list are interpreted in the same way as the annotation identified by {@link #COMPILER\_NONNULL\_ANNOTATION\_NAME}. The intention is to support libraries using different sets of null annotations, in addition to those used by the current project. Secondary null annotations should not be used in the project's own source code.

JDT will never actively use any secondary annotation names from this list, i.e., inferred null annotations and content assist proposals mentioning null annotations are always rendered using the primary name from COMPILER\_NONNULL\_ANNOTATION\_NAME.

This option only has an effect if the option COMPILER\_ANNOTATION\_NULL\_ANALYSIS is enabled.

The qualified name of a Java annotation type

Default is: ""

Name of Annotation Type for Nullable Types (COMPILER\_NULLABLE\_ANNOTATION\_NAME)

This option defines a fully qualified Java type name that the compiler may use to perform special null analysis.

If the annotation specified by this option is applied to a type in a method signature or variable declaration, this will be interpreted as a specification that null is a legal value in that position. Currently supported positions are: method parameters, method return type and local variables.

If a value whose type is annotated with this annotation is dereferenced without checking for null, the compiler will trigger a diagnostic as further controlled by COMPILER\_PB\_POTENTIAL\_NULL\_REFERENCE.

The compiler may furthermore check adherence to the null specification as further controlled by COMPILER\_PB\_NULL\_SPECIFICATION\_VIOLATION, COMPILER\_PB\_NULL\_ANNOTATION\_INFERENCE\_CONFLICT and COMPILER\_PB\_NULL\_UNCHECKED\_CONVERSION.

This option only has an effect if the option COMPILER\_ANNOTATION\_NULL\_ANALYSIS is enabled.

The qualified name of a Java annotation type

Default is: org.eclipse.jdt.annotation.Nullable

Names of Secondary Annotation Types for Nullable Types (COMPILER\_NULLABLE\_ANNOTATION\_SECONDARY\_NAMES)

This option defines a comma-separated list of fully qualified Java type names that the compiler may use to perform special null analysis.

The annotation types identified by the names in this list are interpreted in the same way as the annotation identified by COMPILER\_NULLABLE\_ANNOTATION\_NAME. The intention is to support libraries using different sets of null annotations, in addition to those used by the current project. Secondary null annotations should not be used in the project's own source code.

JDT will never actively use any secondary annotation names from this list, i.e., inferred null annotations and content assist proposals mentioning null annotations are always rendered using the primary name from COMPILER\_NULLABLE\_ANNOTATION\_NAME.

This option only has an effect if the option COMPILER\_ANNOTATION\_NULL\_ANALYSIS is enabled.

The qualified name of a Java annotation type

Default is: ""

Name of Annotation Type to specify a nullness default for unannotated types. (COMPILER\_NONNULL\_BY\_DEFAULT\_ANNOTATION\_NAME)

This option defines a fully qualified Java type name that the compiler may use to perform special null analysis.

If the annotation is applied without an argument, all unannotated types in method signatures within the annotated element will be treated as if they were specified with the non-null annotation (see COMPILER\_NONNULL\_ANNOTATION\_NAME).

If the annotation is applied with the constant false as its argument, all corresponding defaults specified using this annotation at outer scopes will be canceled for the annotated element.

This option only has an effect if the option COMPILER\_ANNOTATION\_NULL\_ANALYSIS is enabled.

The qualified name of a Java annotatin type

Default is: org.eclipse.jdt.annotation.NonNullByDefault

Names of Secondary Annotation Types to specify a nullness default for unannotated types (COMPILER\_NONNULL\_BY\_DEFAULT\_ANNOTATION\_SECONDARY\_NAMES)

This option defines a comma-separated list of fully qualified Java type names that the compiler may use to perform special null analysis.

The annotation types identified by the names in this list are interpreted in the same way as the annotation identified by COMPILER\_NONNULL\_BY\_DEFAULT\_ANNOTATION\_NAME. The intention is to support libraries using different sets of null annotations, in addition to those used by the current project. Secondary null annotations should not be used in the project's own source code.

This option only has an effect if the option COMPILER\_ANNOTATION\_NULL\_ANALYSIS is enabled.

The qualified name of a Java annotatin type

Default is: ""

Reporting Use of Annotation Type as Super Interface (COMPILER\_PB\_ANNOTATION\_SUPER\_INTERFACE)

Plug-in Development Environment Guide

Contents

1. PDE Overview

2. Concepts

2.1. Extensions and Extension Points

2.2. Feature

2.3. Fragment

2.4. Plug-in

2.5. Product

2.6. Target Platform

2.7. Update Site

3. Tasks

3.1. PDE Build

3.1.1. Automated Building of RCP applications from a product configuration file

3.1.2. Automated Building of Features

3.1.3. Automated Building of Plug-ins

3.1.4. Generating Ant scripts from PDE

3.1.5. Generating Ant scripts from scripts

3.1.6. Using Shared Licenses

3.2. PDE Build Advanced Topics

3.2.1. Customizing a Headless Build

3.2.2. Feature and Plug-in Custom Build Steps

3.2.3. Fetching from Repositories

3.2.4. Controlling the output format of the build

3.2.5. Compilation Order and Parallelization

3.2.6. Version Qualifiers

3.2.7. Controlling compilation environment

3.2.8. Compiler options and settings

3.2.9. Source Plug-in and Feature generation

3.2.10. Individual Source Bundles

3.2.11. Repackaging Eclipse Components

3.2.12. Adding Files to the Root of a Build

3.2.13. Topic

3.3. PDE Build Integration with p2

3.3.1. Building p2 Repositories and products

3.3.2. Feature builds with p2

3.3.3. Product builds with p2

3.3.4. PDE/Build extensions to p2

3.3.5. Integration with the 3.4 metadata generator

3.3.6. Feature Metadata Layout

3.3.7. Reusing Metadata

3.3.8. Configuring Products

3.4. API Tools

3.4.1. Setting up API Tools

3.4.2. Setting up a baseline

3.4.3. Creating filters

3.4.4. Remove filters

3.4.5. Changing API Tools options

3.4.6. Comparing to an API baseline

3.5. PDE UI

3.5.1. Export and Install Into Running Host

3.5.2. Import Projects from a Repository

4. Reference

4.1. API Tools

4.1.1. API Tools Ant Tasks

4.1.1.1. Analysis Ant Task

4.1.1.2. Analysis Report Conversion Ant Task

4.1.1.3. API Freeze Ant Task

4.1.1.4. API Freeze Report Conversion Ant Task

4.1.1.5. API Deprecation Ant Task

4.1.1.6. API Deprecation Report Conversion Ant Task

4.1.1.7. API Use Ant Task

4.1.1.8. API Use Report Conversion Task (Producer)

4.1.1.9. API Use Report Conversion Task (Consumer)

4.1.1.10. API Use Migration Ant Task

4.1.1.11. API Use Migration Report Conversion Task

4.1.1.12. File Generation Ant Task

4.1.1.13. Compare Ant Task

4.1.1.14. API Use Scan Problems Ant Task

4.1.1.15. API Use Scan Problems Report Conversion Ant Task

4.1.2. API Tools Command Line Applications

4.1.2.1. API Analysis application

4.1.3. API Evolution

4.1.4. Defining API Restrictions

4.1.5. Javadoc @since Tag Management

4.2. Build Configuration

4.2.1. Builder Configuration properties

4.2.2. Feature and Plug-in Build configuration

4.3. Editors

4.3.1. Plug-in Manifest Editor

4.3.1.1. Plug-in Overview

4.3.1.2. Plug-in Dependencies

4.3.1.3. Plug-in Runtime

4.3.1.3.1. Access Rules

4.3.1.4. Plug-in Extensions

4.3.1.4.1. New Extension Wizard

4.3.1.5. Plug-in Extension Points

4.3.1.6. Plug-in Build

4.3.2. Feature Manifest Editor

4.3.2.1. Feature Overview

4.3.2.2. Feature Information

4.3.2.3. Included Plug-ins

4.3.2.4. Included Features

4.3.2.5. Feature Dependencies

4.3.2.6. Feature Installation

4.3.3. Site Manifest Editor

4.3.3.1. Site Map

4.3.3.2. Site Information

4.3.4. Extension Point Schema Editor

4.3.4.1. Schema Overview Page

4.3.4.2. Schema Definition Page

4.3.5. Product Configuration Editor

4.3.5.1. Product Overview

4.3.5.2. Product Content

4.3.5.3. Product Configuration

4.3.5.4. Product Launching

4.3.5.5. Product Splash

4.3.5.6. Product Branding

4.3.5.7. Product Licensing

4.3.6. Target Definition Editor

4.3.6.1. Target Definition

4.3.6.2. Target Content

4.3.6.3. Target Environment

4.3.7. Simple Cheat Sheet Editor

4.3.7.1. Simple Cheat Sheet Page

4.3.7.1.1. Content Section

4.3.7.1.2. Title Section

4.3.7.1.3. Introduction Section

4.3.7.1.4. Step Section

4.3.7.1.5. Sub-Step Section

4.3.7.1.6. Command Section

4.3.7.1.6.1. Command Composer Dialog

4.3.7.1.6.1.1. Commands Section

4.3.7.1.6.1.2. Command Details

4.3.7.1.7. Help Section

4.3.7.2. Register Cheat Sheet Wizard

4.3.8. Composite Cheat Sheet Editor

4.3.8.1. Composite Cheat Sheet Page

4.3.9. Table of Contents Editor

4.3.9.1. Table of Contents Page

4.3.10. Context Help Editor

4.3.11. Category Definition Editor

4.4. Launchers

4.4.1. Eclipse Application Launcher

4.4.1.1. Main Tab

4.4.1.2. Arguments Tab

4.4.1.3. Plug-ins Tab

4.4.1.4. Configuration Tab

4.4.1.5. Tracing Tab

4.4.1.6. Environment Tab

4.4.1.7. Common Tab

4.4.1.8. Prototype Tab

4.4.2. OSGi Framework Launcher

4.4.2.1. Bundles Tab

4.4.2.2. Arguments Tab

4.4.2.3. Settings Tab

4.4.2.4. Tracing Tab

4.4.2.5. Environment Tab

4.4.2.6. Common Tab

4.4.3. JUnit Plug-in Test Launcher

4.4.3.1. Test Tab

4.4.3.2. Main Tab

4.4.3.3. Arguments Tab

4.4.3.4. Plug-ins Tab

4.4.3.5. Configuration Tab

4.4.3.6. Tracing Tab

4.4.3.7. Environment Tab

4.4.3.8. Common Tab

4.5. Menus and Actions

4.5.1. API Tools Setup

4.5.2. Compare with API baseline

4.6. Preferences

4.6.1. Plug-in Development

4.6.2. API Baselines

4.6.3. API Errors and Warnings

4.6.4. Compilers

4.6.4.1. Plug-in Compiler Preferences

4.6.4.2. Extension Point Schema Compiler Preferences

4.6.4.3. Feature and Update Site Compiler Preferences

4.6.5. Editors

4.6.6. OSGi Frameworks

4.6.7. Target Platform

4.6.8. API Use Scans

4.7. Property Pages

4.7.1. API Errors and Warnings

4.7.2. API Problem Filters

4.7.3. Runtime Classpath

4.7.4. Plug-in Manifest Compiler

4.8. Quickfixes

4.8.1. Create API Problem Filter

4.8.2. Missing API Baseline

4.8.3. Since Tags

4.8.4. Version Numbering

4.8.5. Remove Unused API Problem Filters

4.9. Views

4.9.1. Plug-in Dependencies

4.9.2. Plug-in Image Browser

4.9.3. Plug-in Registry

4.9.4. Plug-ins

4.9.5. Target Platform State

4.10. Wizards and Dialogs

4.10.1. New Project Creation Wizards

4.10.1.1. Plug-in Project

4.10.1.1.1. Plug-in Project Structure

4.10.1.1.2. Plug-in Content

4.10.1.1.3. Templates

4.10.1.2. Plug-in from existing JAR archives

4.10.1.3. Fragment Project

4.10.1.3.1. Fragment Project Structure

4.10.1.3.2. Fragment Content

4.10.1.4. Feature Project

4.10.1.5. Feature Patch

4.10.1.6. Update Site Project

4.10.2. New File Creation Wizards

4.10.2.1. Category Definition

4.10.2.2. Extension Point Schema

4.10.2.3. Product Configuration

4.10.2.4. Target Definition

4.10.2.5. Cheat Sheet

4.10.2.6. Context Help

4.10.2.7. Table of Contents

4.10.3. Import Wizards

4.10.3.1. Plug-in Import

4.10.3.2. Feature Import

4.10.4. Export Wizards

4.10.4.1. Plug-in Export

4.10.4.2. Feature Export

4.10.4.3. Product Export

4.10.4.4. Target Export

4.10.5. Task Wizards

4.10.5.1. Converting Java Projects to Plug-in Projects

4.10.5.2. Externalizing Strings

4.10.5.3. Organizing Manifest Files

4.10.5.4. Updating Classpaths

4.10.6. API Tools

4.10.6.1. API Tools Setup Wizard

4.10.6.2. API Baseline Wizard

4.10.6.3. API Compare Wizard

4.10.6.4. API Use Reports

4.10.6.5. API Use Patterns Wizard

4.10.7. Target Wizards

4.10.7.1. New Target Wizard

4.10.7.2. Edit Target Wizard

4.10.7.2.1. Locations Tab

4.10.7.2.2. Content Tab

4.10.7.2.3. Environment Tab

4.10.7.2.4. Arguments Tab

4.10.7.2.4.1. Import Arguments Dialog

4.10.7.2.5. Implicit Dependencies Tab

4.10.7.2.5.1. Implicit Dependencies Selection Dialog

4.10.7.3. Move Target Wizard

4.11. API Reference

4.11.1. org.eclipse.pde.api.tools.annotations

4.11.2. org.eclipse.pde.build

4.11.3. org.eclipse.pde.core

4.11.4. org.eclipse.pde.core.build

4.11.5. org.eclipse.pde.core.plugin

4.11.6. org.eclipse.pde.core.project

4.11.7. org.eclipse.pde.core.target

4.11.8. org.eclipse.pde.launching

4.11.9. org.eclipse.pde.ui

4.11.10. org.eclipse.pde.ui.launcher

4.11.11. org.eclipse.pde.ui.target

4.11.12. org.eclipse.pde.ui.templates

4.12. Extension Points Reference

4.12.1. org.eclipse.ui.trace.traceComponents

4.12.2. org.eclipse.pde.build.fetchFactories

4.12.3. org.eclipse.pde.core.bundleClasspathResolvers

4.12.4. org.eclipse.pde.core.javadoc

4.12.5. org.eclipse.pde.core.pluginClasspathContributors

4.12.6. org.eclipse.pde.core.source

4.12.7. org.eclipse.pde.core.targetLocations

4.12.8. org.eclipse.pde.core.targets

4.12.9. org.eclipse.pde.ui.launchShortcuts

4.12.10. org.eclipse.pde.ui.newExtension

4.12.11. org.eclipse.pde.ui.osgiFrameworks

4.12.12. org.eclipse.pde.ui.pluginContent

4.12.13. org.eclipse.pde.ui.samples

4.12.14. org.eclipse.pde.ui.targetLocationProvisioners

4.12.15. org.eclipse.pde.ui.targetProvisioners

4.12.16. org.eclipse.pde.ui.templates

4.13. Other Reference Information

4.13.1. PDE Plug-ins Map

5. Tips and Tricks

6. What's new

6.1. Dialogs, Wizards and Views

6.2. API Tools

6.3. PDE Compiler

7. Legal

1. Plug-in Development Environment Overview

The Plug-in Development Environment (PDE) provides tools to create, develop, test, debug, build and deploy Eclipse plug-ins, fragments, features, update sites and RCP products.

PDE also provides comprehensive OSGi tooling, which makes it an ideal environment for component programming, not just Eclipse plug-in development.

PDE is broken into three main components:

UI - A rich set of models, tools and editors to develop plug-ins and OSGi bundles

API Tools - Tooling to assist API documentation and maintenance

Build - Ant based tools and scripts to automate build processes

Discover the latest features in the What's New section.

PDE UI

PDE UI provides a editors, wizards, launchers, views and other tools to create a full featured environment to develop and deploy Eclipse plug-ins, fragments, features, update sites, RCP products and OSGi bundles.

Some of the PDE tools include:

Form-Based Manifest Editors: Multi-page editors that centrally manage all manifest files of a plug-in or feature.

RCP Tools: Wizards and a form-based editor that allow you to define, brand, test and export products to multiple platforms.

New Project Creation Wizards: Create a new plug-in, fragment, feature, feature patch and update sites.

Import Wizards: Import plug-ins and features from the file system.

Export Wizards: Wizards that build, package and export plug-ins, fragments and products with a single click.

Launchers: Test and debug Eclipse applications and OSGi bundles.

Views: PDE provides views that help plug-in developers inspect different aspects of their development environment.

Miscellaneous Tools: Wizards to externalize and clean up manifest files.

Conversion Tools: Wizard to convert a plain Java project or plain JARs into a plug-in project.

Integration with JDT: Plug-in manifest files participate in Java search and refactoring.

To get started, try out the following cheat sheets:

Open plug-in cheat sheetCreating an Eclipse Plug-in

Open RCP cheat sheetCreating a Rich Client Application

API Tools

API Tools assists in the documentation and maintenance of APIs provided by plug-ins and OSGi bundles.

Some of the features include:

Compatibility Analysis : Identify binary compatibility issues relative to a previous version of a plug-in.

API Restriction Tags : Javadoc tags are provided to explicitly define restrictions associated with types and members.

Version Number Validation : Identify invalid plug-in version numbers relative to a previous version of a plug-in.

Javadoc @since Tag Validation : Identify missing and invalid @since tags on types and members.

API Leak Analysis : Identify API types and methods that leak non-API types.

Quick Fixes : Quick fixes are provided to adjust plug-in versions and @since tags appropriately.

To get started, see the tasks:

Setting up API Tools

Setting up a baseline

Changing options

Creating filters

Removing filters

PDE Build

PDE Build facilitates the automation of plug-in build processes. PDE Build produces Ant scripts based on development-time information provided by, for example, the plugin.xml and build.properties files. The generated Ant scripts, can fetch the relevant projects from a CVS repository, build jars, Javadoc, source zips, put everything together in a format ready to ship and send it out to a remote location (e.g., a local network or a downloads server).

To get started, see the following:

Automated building of RCP applications from product configurations

Automated building of features

Automated building of plug-ins

Generating Ant scripts from PDE

Generating Ant scripts from scripts

Building p2 repositories and products.

Builder Configuration Properties

Feature and Plug-in build properties

2.1. Extensions and Extension Points

A basic rule for building modular software systems is to avoid tight coupling between components. If components are tightly integrated, it becomes difficult to assemble the pieces into different configurations or to replace a component with a different implementation without causing a ripple of changes across the system.

Loose coupling in Eclipse is achieved partially through the mechanism of extensions and extension points. The simplest metaphor for describing extensions and extension points is electrical outlets. The outlet, or socket, is the extension point; the plug, or light bulb that connects to it, the extension. As with electric outlets, extension points come in a wide variety of shapes and sizes, and only the extensions that are designed for that particular extension point will fit.

When a plug-in wants to allow other plug-ins to extend or customize portions of its functionality, it will declare an extension point. The extension point declares a contract, typically a combination of XML markup and Java interfaces, that extensions must conform to. Plug-ins that want to connect to that extension point must implement that contract in their extension. The key attribute is that the plug-in being extended knows nothing about the plug-in that is connecting to it beyond the scope of that extension point contract. This allows plug-ins built by different individuals or companies to interact seamlessly, even without their knowing much about one another.

The Eclipse Platform has many applications of the extension and extension point concept. Some extensions are entirely declarative; that is, they contribute no code at all. For example, one extension point provides customized key bindings, and another defines custom file annotations, called markers; neither of these extension points requires any code on behalf of the extension.

Another category of extension points is for overriding the default behavior of a component. For example, the Java development tools include a code formatter but also supply an extension point for third-party code formatters to be plugged in. The resources plug-in has an extension point that allows certain plug-ins to replace the implementation of basic file operations, such as moving and deletion.

Yet another category of extension points is used to group related elements in the user interface. For example, extension points for providing views, editors, and wizards to the UI allow the base UI plug-in to group common features, such as putting all import wizards into a single dialog, and to define a consistent way of presenting UI contributions from a wide variety of other plug-ins.

Related references

Extensions (Plug-in Editor)

Extension Points (Plug-in Editor)

Related concepts

Plug-in

2.2. Feature

A Feature is used to package a group of plug-ins together into a single installable and updatable unit.

Features have a manifest that provides basic information about the feature and its content. Content may include plug-ins, fragments and any other files that are important for the feature. A feature can also include other features. The delivery format for a feature is a JAR, but each included plug-in will be provided as a separate JAR.

Once you have created your plug-ins and fragments you can create a new feature by creating a New Feature Project.

You can also create Feature Patches in PDE. A feature patch is developed in the same fashion as a normal feature, but is used to edit an existing feature rather than create a new one.

Related references

Feature Editor

New Feature Project

New Feature Patch Project

Related concepts

Plug-in

Fragment

Product

Update Site

2.3. Fragment

A fragment is used to replace or extend the functionality of an existing plug-in. A common use for fragments is to put environment (operating system, architecture, etc.) specific code into fragments. Depending on the environment the plug-in is installed in the base plug-in code along with the correct fragment can be installed. Fragments are also ideal for shipping features like language or maintenance packs that typically trail the initial products for a few months.

When a fragment is detected by the platform and its parent plug-in is found, the fragment's libraries, extensions and extension points are "merged" with those of the parent plug-in.

While this merging mechanism is good from a runtime point of view, developers need to view fragments as separate entities while working on them.

PDE provides full support for fragment development. Fragments can be viewed as "limited plug-ins". They have all of the capability of regular plug-ins but have no concept of life-cycle. Fragments have no top-level class with "startup" and "shutdown" methods.

To create a fragment use the New Fragment Project wizard. Editing fragments is very similar to editing plug-ins.

Related references

Plug-in Editor

New Fragment

Related concepts

Plug-in

Feature

2.4. Plug-in

A plug-in is used to group your code into a modular, extendable and sharable unit.

Plug-ins are modular as each plug-in contains some portion of code. The plug-in specifies other plug-ins (or java packages) it requires to be available to run and it also specifies the set of java packages it provides. An Eclipse based program or product will contain multiple plug-ins, which can be added, replaced or removed to alter the functionality of the program.

Plug-ins are extendable using extensions and extension points. A plug-in can provide one or more extension points so other plug-ins can add to the functionality of the plug-in. A plug-in may also provide extensions to connect to other plug-ins.

Plug-ins are sharable. A plug-in can be exported as a directory or as a jar which can be added to other applications. Plug-ins can be grouped into features which can be distributed and installed into applications.

Eclipse plug-ins are based on OSGi bundles. OSGi is used to manage the plug-ins in an Eclipse application. A plug-in must contain a manifest file with valid OSGi headers for plug-in name and version. Extensions and extension points functionality added by Eclipse in addition to OSGi. To use extensions you must provide a plugin.xml file. PDE provides a full featured project and editor for creating and editing these files.

Eclipse plug-ins can also be packaged as a Java modular project ( Java 9 or later).

Related references

Plug-in Editor

New Plug-in Project

Related concepts

Fragment

Feature

2.5. Product

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.

PDE provides a product configuration file and associated editor to make it easy to create products in the correct format.

Related references

Product Editor

New Product Configuration

Related concepts

Plug-in

Fragment

Feature

2.6. Target Platform

The Target Platform refers to the plug-ins which your workspace will be built and run against. It describes the platform that you are developing for. When developing with PDE, the target platform is used to:

Compile - Plug-ins in the workspace are built against the target platform so you do not have to have everything in your workspace

Launch - When using PDE's launchers you can choose the set of plug-ins you want to launch. By default the Eclipse Application Launcher will start with all plug-ins in the target, but will use workspace plug-ins instead when available

Calculate Dependencies - PDE assists you in finding the plug-ins your plug-ins was include/require to compile correctly by looking at what is available in the target platform

State - An OSGi state is created for the target platform so it can be used by other tooling. The state can also be viewed in detail using the Target Platform State View

Other Tools - Other PDE tools use the target platform to determine their options, such as the import wizards

Whereas the target platform refers to your currently active bundles, a Target Definition is a way of determining the plug-ins to add to the state. You can have multiple target definitions, but only one definition can be selected as the target platform.

The target platform and your target definitions are managed on the Target Platform Preference Page. This page lists all target definitions that PDE has access to and displays which definition is being used as your current target platform. Target definitions can be created and edited here using the Target Definition Content Wizard. To make for easier sharing among a development team, targets can also be created and edited in the workspace as xml files with the extension ".target". These files can be edited using the target definition editor and can be created using the New Target Definition Wizard.

By default PDE provides a default target platform which simply points to the plug-ins in your currently running Eclipse instance (often referred to as your host). For Eclipse plug-in developers this might be all you need. RCP developers must be more careful managing their target as adding dependencies increases the size and complexity of their application. When creating a new target definition for RCP, consider using the RCP template (available with or without source).

For up to the minute target troubleshooting help see the Target Definitions Wiki Page.

Related references

Target Platform Preference Page

Edit Target Definition Wizard

Target Editor

Eclipse Application Launcher

Eclipse Application Launcher Plug-ins Tab

2.7. Update Site

Update Sites are used to organize and export features so they can be installed into Eclipse applications.

To create an update site you must develop a site.xml file and build the site. PDE provides an editor and project for creating sites. A site will contain one or more features organized into categories.

When the site is built, the included features (along with all plug-ins part of those features) will be exported into an installable form. The exported plug-ins and features will be put into two folders "plug-ins" and "features". Two other files, "content.xml" and "artifacts.xml" will also be generated and contain metadata for the exported files that make installing easier. These files, along with "site.xml", collectively form an Eclipse update site. To make the update site available to others you must make all these files available in a shared directory or web site.

Related references

Site Editor

New Update Site Project

Related concepts

Feature

Plug-in

3.1.1. Building an RCP application from a product configuration file

PDE Build comes with infrastructure to build a complete RCP application from a product configuration file. Most of the setup necessary for such a build can be done through a few modifications to the template build.properties provided in PDE build. The following section focuses on setting up a simple product build assuming that all plug-ins and features (both to build and pre-built) referenced by the product file are already locally available on disk.

Basic setup

The first step in setting up a build is to create the directory in which the build will take place. This directory will be referred to as the build directory and will contain the plug-ins and features to build as well as all the generated artifacts. Next, copy your plug-ins and features respectively into "plugins" and "features" subdirectories. Your product configuration file is expected to be in one of these plugins/features.

The second step in setting up a build is to copy the template build.properties file from org.eclipse.pde.build/templates/headless-build to a new directory which will be the build configuration directory (also referred to as the configuration directory). Edit the copied build.properties file and set the following properties:

product: the location of your product configuration file in the form "/<plug-in or feature id>/path/to/.product"

baseLocation: the location of an eclipse install containing all the pre-built features and plug-ins that your product requires in features/ and plugins/ subdirectories. The org.eclipse.equinox.executable feature (it is available from the eclipse p2 repository) is mandatory as it includes the org.eclipse.platform.launchers feature which contains the launchers and root files necessary for a product.

buildDirectory: the directory the build will take place in. Set this to the full path of the build directory created previously.

configs: list the configurations for which you want your product to be built. You can uncomment the configuration(s) provided (be careful of the line continuations).

archivePrefix: the name of the directory of your product once installed on disk.

Overview of the directory structure

<buildDirectory>/

plugins/

pluginToBuildA

pluginToBuildB

myProduct.product

...

features/

featureToBuild

...

<buildConfiguration>/

build.properties

<baseLocation>

plugins/

org.eclipse.core.runtime

org.eclipse.ui

...

features/

org.eclipse.rcp

org.eclipse.platform.launchers

...

Running the build

To run the build you will use the org.eclipse.ant.core.antRunner application. When invoking eclipse with this application to perform a build you need to set two arguments on the command line:

-buildfile </path/to/productBuild.xml>: This is the path to the productBuild.xml provided by pde build. It is located in the org.eclipse.pde.build/scripts/productBuild directory. This is the build file that drives the whole product build process.

-Dbuilder=</path/to/configuration folder>: This is the path to the build configuration folder.

Run the antRunner application using the following command:

java -jar <eclipseInstall>/plugins/org.eclipse.equinox.launcher\_<version>.jar -application org.eclipse.ant.core.antRunner -buildfile <<eclipseInstall>/plugins/org.eclipse.pde.build\_<version>/scripts/productBuild/productBuild.xml> -Dbuilder=<path to the build configuration folder>

Advanced scenarios

Behind the scenes this setup will generate a feature containing the plug-ins or features specified in your product configuration file. This generated feature is then used to drive a normal feature based headless build. If you require more customization of the build, i.e. fetching from a repository, see the Advanced PDE Build topics for more information.

3.1.2. Building features

PDE Build comes with infrastructure to automate building features. Most of the setup necessary for such a build can be done through a few modifications to the template build.properties provided by PDE build. The following section focuses on setting up a simple build assuming that all plug-ins and features (both to-build and pre-built) referenced from the feature being built are already locally available on disk.

Basic Setup

The Build Configuration Directory

The first step in setting up a build is to create the build configuration directory. This is the directory in which the files that control and customize the headless build are kept.

The main file for controlling the build is the top level build configuration build.properties file. When setting up a new build, copy the template file from the org.eclipse.pde.build bundle in your eclipse install that you will be using to run your build. The template file is located under org.eclipse.pde.build/templates/headless-build/build.properties.

This directory is refered to by the builder properties which is normally set on the ant command line.

The buildDirectory

The buildDirectory property specifies the directory in which PDE/Build will generate scripts and perform the actual build. Plug-ins and features that are being built should placed in ${buildDirectory}/plugins and ${buildDirectory}/features sub-directories. If bundles and features are being fetch from a repository (See Fetching From Repositories), then they will be placed here as well.

This directory may be the same as the build configuration directory, or it may be a different directory. It may be desirable to use a separate directory, especially if the build configuration is kept in CVS and you don't want to pollute it with generated build artifacts.

Initial Properties

To set up a basic feature build, you need to edit the build configuration build.properties file and set the following properties:

topLevelElementId: Set to the id of the feature you wish to build.

baseLocation: the location of an eclipse install containing pre-built features and plug-ins in features/ and plugins/ subdirectories. These are the binaries against which you will be compiling. If you have no pre-requisite binaries and are building everything from scratch, this property can be left un-set.

buildDirectory: the directory the build will take place in.

configs: list the configurations for which you want your feature to be built. You can uncomment the configuration(s) provided (be careful of the line continuations). If the feature you are building is platform independent, this property does not need to be set or can be set to \*,\*,\*.

archivePrefix: the name of the directory in which your feature will be unzipped on disk.

Previous version of PDE/Build also required an allElements.xml file, this is no longer necessary in 3.4. The allElements.xml can still be used, but is only required if you wish to customize certain aspects of your build. See Customizing a Headless Build.

Running the build

To run the build you will use the org.elipse.ant.core.antRunner application. When invoking eclipse with this application to perform a build you need to set two arguments on the command line:

-buildfile=</path/to/build.xml>: This is the path to the build.xml provided by pde build. It is located in the org.eclipse.pde.build/scripts directory. This is the build script that drives the whole build process.

-Dbuilder=</path/to/configuration folder>: This is the path to the build configuration folder.

Run the antRunner application using the following command:

java -jar <eclipseInstall>/plugins/org.eclipse.equinox.launcher\_<version>.jar -application org.eclipse.ant.core.antRunner -buildfile <<eclipseInstall>/plugins/org.eclipse.pde.build\_<version>/scripts/build.xml> -Dbuilder=<path to the build configuration folder>

Once the build is complete, you can get the result in the build directory in the folder named I.TestBuild (this name can be configured by setting the buildLabel property).

Advanced scenarios

If you require more customization of the build, ie fetching from a repository, see the Advanced PDE Build topics for more information.

3.1.3. Building plug-ins / bundles

In order to build plug-ins, bundles or fragments, PDE build requires that you create a feature listing all the elements to be built. In this case the feature is only used to drive the build and will not be present in the final archive nor will it force you to use update sites.

A feature used to drive a plug-in build is a standard feature. What makes it different is that its build.properties file does not contain the property bin.includes. It is possible to automatically generate a feature containing the plug-ins you wish to build. This lets you build your plug-ins without requiring you to maintain a feature. See the eclipse.generateFeature task and the Customizing a Headless Build page.

Example:

build.properties: topLevelElementId = acme.build.feature

customTargets.xml :

<target name="preSetup">

<eclipse.generateFeature

featureId="acme.build.feature"

buildDirectory="${buildDirectory}"

baseLocation="${baseLocation}"

productFile="${product}"

verify="false"

pluginPath="${pluginPath}"

configInfo="${configs}"

pluginList="acme.plugin.A,acme.plugin.B"

featureList="${featureList}"

includeLaunchers="false"

/>

</target>

Once you have created the feature to drive the build, follow the instructions on building features.

3.1.4. Generating Ant scripts

Ant is a simple open-source scripting engine that is capable of running scripts written in XML format. Ant is ideal for executing tasks usually found in automated builds.

The variables set in the plug-in, fragment or feature build.properties will be used to generate scripts for Ant. PDE generates Ant scripts for creating individual plug-in and fragment build files and one overall script for building the feature JAR. This "main" script is also responsible for running individual script files in the right order (defined by the plug-in dependency chain). Each build file has the same name (build.xml) and is created as a sibling of the manifest files in the corresponding projects.

Since Ant scripts use the replacement variables in build.properties, you can typically use them "as is", without modifying the generated scripts. If you do modify them, you must not recreate the scripts every time you want to rebuild the component.

To create scripts, you can simply select Create Ant Build File while a suitable manifest file (plugin.xml, fragment.xml or feature.xml) is selected in the Navigator or Package Explorer views. The command will generate the build script. After selecting Run Ant... from the pop-up menu while the newly generated script file is selected, the following wizard will open:

Ant execution wizard showing available targets

The standard Ant wizard allows customization in two ways: by providing the execution arguments and by selecting one or more build targets.

Properties

Ant arguments are typically used to provide property values that override default values and control the build process. Arguments are set using "-Dproperty=value". The following properties are recognized:

bootclasspath - if set, it replaces the default boot classpath. Used when compiling cross-platform plug-ins (e.g. building a UI plug-in for Windows using Linux)

build.result.folder - where the temporary files for the update JAR creation should be placed. These files are usually the plug-in library JARs.

plugin.destination - where plug-in and fragment update JARs should be put. These JARs represent entire plug-ins and fragments in a format suitable for publishing on an Install/Update server and referencing by a feature. The typical layout of an Update site is to have all the plug-in and fragment JARs in one place and all the features in another. This argument is useful for placing plug-ins and fragment directly into the desired directory (or the staging place on the local machine before pushing the features onto the remote server).

feature.destination - where feature update JARs should be put.

To adapt the behavior of the compiler, the following properties are recognized:

javacFailOnError - stop the build when an error occurs when set to true. Default is false.

javacDebugInfo - compile source with debug information when set to true. Default is true.

javacVerbose - produce verbose output when set to true. Default is true.

javacSource - value of the -source command-line switch.

javacTarget - generate class files for specific VM version.

compilerArg - additional command line arguments for the compiler.

Targets

When executing feature build scripts, the following targets are used to call individual targets of plug-ins or fragments. In order to specify what target to execute, the property target should be set (e.g. -Dtarget=refresh). One of the all.\* targets serves as an iterator, whereas the actual target to execute is specified via the property target.

all.plugins - for all listed plug-ins

all.fragments - for all listed fragments

all.children - for all listed plug-ins and fragments

build.jars - build JARs for all feature children;

build.sources - build source for all feature children;

build.update.jar - generate a feature JAR in the format used by the install/update mechanism. The above mentioned property feature.destination can be used to define where to put the JAR;

zip.distribution - creates a zip file with the feature and its plug-ins and fragments in an SDK-like structure but does not include source code;

zip.sources - creates a zip file with the feature and its plug-ins and fragments in an SDK-like structure which only includes the source;

clean - delete everything produced by running any of the target;

refresh - performs a "Refresh" action in the current project, thus making the newly generated resources visible in the Navigator or Package Explorer.

zip.plugin - creates a zip file with the binary and source contents of a plug-in with the following structure:

id\_version/

contents

where 'id' is the plug-in unique identifier and 'version' is the plug-in version. This zip file can be directly unzipped into the Eclipse installation directory as a form of a quick manual deployment.

3.1.5. Generating Ant scripts from the command line

Ant scripts are typically generated using the Plug-in Development Environment (PDE), but it is also possible to generate them by hand or from other scripts.

Indeed PDE exposes Ant tasks to generate the various build scripts. Build script generation facilities reside in the following tasks. Arguments are also listed for each task.

eclipse.fetch: generates an Ant script that fetches content from a CVS repository. The eclipse fetch is driven by a file whose format is described below (see Directory file format).

elements:

the entry that will be fetched. The format expected is of the form type@id[,version] as specified in the directory file format;

buildDirectory:

the directory into which fetch scripts will be generated and into which features and plug-in projects will be checked out;

directory:

the path to a directory file;

children:

optional, specifies whether the script generation for contained plug-ins and fragments should be invoked. Default is set to true;

cvspassfile:

optional, the name of a CVS password file;

fetchTag:

optional, overrides the tag provided in directory file by the given value;

configInfo:

optional, an ampersand separated list of configuration indicating the targeted configuration. The default is set to be platform independent;

recursiveGeneration:

optional, specify whether or not fetch scripts should be generated for nested features. The default is set to true.

eclipse.buildScript: generates a build.xml file for the given elements as well as additional scripts used by the pde build infrastructure.

elements :

the entry to be built. Entry is expected to be of the form type@id[,version] as specified in the directory file format;

buildDirectory :

the directory where the features and plug-ins to build are located. Plug-ins and features must respectively be located in plugins and features folders;

children :

optional, specifies whether the script generation for contained plug-ins and fragments should be invoked. Default is set to true;

recursiveGeneration :

optional, specified whether the script generation for contained features should be invoked. Default is set to true;

devEntries :

optional, a comma separated list of directories to be given to the compile classpath;

buildingOSGi :

optional, indicates if the target is 3.x. or 2.1. Default is true;

baseLocation :

optional, indicates a folder which containing a target eclipse install with features/ and plugins/ subdirectories;

configInfo :

optional, an ampersand separated list of configuration indicates the targeted configuration. The default is set to be platform independent;

pluginPath :

optional, a platform file separator separated list of locations from which to find additional plug-ins;

archivesFormat :

optional, an ampersand separated list of configs and the expected output format for each of those. The format is separated by a dash (-) from the configuration. The values supported are: folder, tar, zip, antZip, respectively meaning don't archive, use tar to create the archive, use the version of info zip available on the platform, use ant zip . The default value is antZip;

product :

optional, '/' separated path to the location of an RCP product being built. The first segment of the path must refer to the plug-in id of a plug-in containing the .product file;

signJars :

optional, indicates if the scripts generated must sign jars for features and plug-ins. The default value is false. The parameters to the sign task are controlled by the following ant properties: sign.alias, sign.keystore and sign.storepass respectively being passed to the alias, keystore and storepass parameters from the ant signJar task. The default value is false;

generateJnlp :

optional, indicates if a jnlp file should be generated for all the features being built;

outputUpdateJars :

optional, generates plug-ins and features in the update site format when set. The default value is false. Note that the site.xml is not generated nor updated;

groupConfiguration :

optional, whether or not to group all configurations being built into a single archive;

generateVersionsLists :

optional, whether or not to generate properties files listing the versions of features and plug-ins that were built;

filteredDependencyCheck :

optional, constrain the set of plug-ins and features to be considered during the build to those reachable from the features;

filterP2Base :

optional, constrain the set of plug-ins and features to be considered from the baseLocation to those considered to be installed by p2;

generateFeatureVersionSuffix: optional, whether or not to generate the feature version suffix. See also Version Qualifiers;

generatedVersionLength :

optional, the length of the generated feature version suffix. By default it is set to 28. See also Version Qualifiers;

significantVersionDigits :

optional, the number of significant digits used when generating feature version suffixes. See also Version Qualifiers;

forceContextQualifier :

optional, uses the given value to replace the .qualifier being by plug-ins and features. See also Version Qualifiers;

eclipse.generateFeature: generates a feature that will contain the provided elements. This task is used to build an RCP application from a product configuration file.

featureId :

The id to give to the generated Feature.

buildDirectory:

the directory where the features and plug-ins (in /features and /plugins subdirectories) to include are located. The generated feature will be placed in the /features subdirectory.

baseLocation :

optional, indicates a folder which contains installed features and plug-ins

productFile :

optional, a product file on which to base the generated feature.

verify :

Whether or not to use the resolver to verify that the provided plug-ins and features are available. This also determines whether or not the the feature will be able to correctly handle platform specific fragments and plug-ins that will be JARed. If all the elements to be included in the feature are available locally, then verify should be set to True. If the elements need to be fetched from a repository, set verify to False, use the resulting feature to fetch the required plug-ins and features, then regenerate the container feature using verify = True.

pluginPath :

optional, a platform file separator separated list of locations from which to find additional plug-ins

configInfo :

optional, an ampersand separated list of configuration indicates the targeted configuration. The default is set to be platform independent;

pluginList :

optional, a comma separated list of plug-in ids to include in the generated feature. (Supports attributes, see below)

featureList :

optional, a comma separated list of feature ids to include in the generated feature. (Supports attributes, see below)

fragmentList :

optional, a comma separated list of fragment ids to include in the generated feature. (Supports attributes, see below)

buildPropertiesFile :

optional, the location of a properties file whose contents should be used as the generated feature's build.properties.

includeLaunchers :

optional, whether or not to include the equinox launchers, default value is "true".

The pluginList, featureList, and fragmentList arguments of the eclipse.generateFeature task all support attributes. These arguments all take the form:

List = <element-id>[;<attribute>=<value>]\* [,<element-id>[;<attribute>=<value>]\*]\*

Each attribute=value pair will be inserted into the generated xml corresponding to the given element. Example:

featureList = org.eclipse.feature;optional=true

pluginList = org.eclipse.plugin.A;unpack=true

fragmentList=org.eclipse.fragment.a;os=win32,org.eclipse.fragment.b;os=linux;ws=gtk;arch=x86

<feature ...>

...

<includes id="org.eclipse.feature" version="0.0.0" optional="true" />

...

<plugin id="org.eclipse.plugin.A" version="0.0.0" unpack="true" />

<plugin id="org.eclipse.fragment.a" version="0.0.0" os="win32" fragment="true" />

<plugin id="org.eclipse.fragment.b" version="0.0.0" os="linux" ws="gtk" arch="x86" fragment="true" />

...

</feature>

Examples

<eclipse.fetch elements="bundle@org.eclipse.osgi"

buildDirectory="c:\toBuild"

directory="directory.txt"

configInfo="win32,win32,x86 & linux, motif, x86"

/>

<eclipse.buildScript elements="bundle@org.eclipse.osgi"

buildDirectory="c:\toBuild"

archivesFormat="macosx, carbon, ppc - tar"/>

Directory file format

Directory files are used to indicate where the plug-ins and features can be obtained, as well as their versions. It is a Java property file whose keys are always of the form type@id[,version] but where the value is open ended. When using map files to fetch your elements, the directory file is a concatenation of all the map files, for more details on map files see Fetching From Repositories.

Using the targets

The tasks previously described only work if Eclipse is running. In the particular scenario of executing Ant scripts using Eclipse tasks,the scripts must be run using the Eclipse Ant Runner application. The command line for this particular case is the following:

java -jar plugins/org.eclipse.equinox.launcher\_<version>.jar -application org.eclipse.ant.core.antRunner -buildfile build.xml

Note that the parameters appearing after the application are the parameters that are passed to Ant.

3.1.6. Using Shared Licenses

Many Eclipse features use the same license. For example, all features in the Eclipse SDK use the Eclipse Foundation Software User Agreement. If individual licenses are added to each feature built with PDE, updating them will be very tedious. Using the shared license support allows multiple features to reference license text and files from a single location. This makes updating licenses across multiple features easier and less error prone.

To create a license feature, create a feature that does not include any bundles or features. Add the necessary license files and ensure they are included in your build.properties. The feature.properties should have two properties, licenseURL and license.

To make use of the shared license facility, simply open your feature.xml file in the feature editor. Select the Information tab and then the License Agreement section. Selecting Shared license will allow you to specify the feature and version that contains your license text and optional license files. The license-feature and license-feature-version attributes will be added to the feature element of your feature.xml. If you are converting a single license to a shared license, you should remove the license files from the individual features. The license information should be removed from feature.properties and build.properties.

Setting a shared license in your feature

Contents of feature using a shared license

The new shared license feature will have to be included in the build, so update your map files appropriately. You do not need to nest the feature in another feature to fetch it, it happens automatically. An IU will not be created in the p2 repository for the license feature, but the license text will be included in the p2 metadata.

3.2.1. Customizing a Headless Build

There are multiple methods for customizing a headless build:

Using a customTargets.xml file

Using an allElements.xml file

Using a customAssembly.xml file

Using feature or plug-in custom callbacks. (See Customizing feature and plug-in builds)

The exact format of the contents of these files all depend on each other. Therefore, when using customTargets, allElements or customAssembly customizations, it is recommended to ensure that the templates used for these files all come from the same version of PDE/Build. New versions of PDE/Build are backward compatible with old versions of these customization scripts.

Template versions of these scripts are available in the org.eclipse.pde.build plug-in under the templates/headless-build folder. To use the customization, simply copy the template into your build configuration directory and edit the appropriate targets.

customTargets.xml

The customTargets.xml script is for top level customization of the build process. It provides ant targets that are called between the major phases of the build. There are pre and post targets available around each of the following build steps:

Process p2 repositories

Get Map files

Fetch source

Generate build scripts

Process and run the build scripts

Assemble the compiled plug-ins and features

Package pre-compiled binary plug-ins and features

To use the customTargets.xml script, simply copy it from the PDE/Build templates directory into your build configuration directory and edit the appropriate ant targets. The script will be automatically detected and used by the headless build.

The customTargets.xml script also provides delegation to the allElements.xml script for each of the build phases.

allElements.xml

The allElements.xml script is a kind of delegation script that is used to run the various build steps on a given element (usually a feature). As with the customTargets.xml, to use these customization steps, copy the template from PDE/Build into your build configuration directory. In Eclipse 3.4, the target element that will be built can be specified in the top level build configuration build.properties file using the properties topLevelElementType and topLevelElementId. The elementType should be set to feature and the elementId should be set to the feature-id you wish to build.

<target name="allElementsDelegator">

<ant antfile="${genericTargets}" target="${target}">

<property name="type" value="${topLevelElementType}" />

<property name="id" value="${topLevelElementId}" />

</ant>

</target>

The allElements.xml also provides targets used during the assembly and packaging phase of the build. The defaultAssemble target will be called to assemble the archives for all platforms. If desired, custom targets can be defined for individual platforms.

To use a custom target for a given platform, define a property "assemble.<element-id>[.<config>] and a corresponding target. The property tells PDE/Build to use this custom target instead of the default target. This custom target can then be used to give the archive a custom name or perform pre or post processing on the archive.

Example:

<property name="assemble.org.foo.feature.id.linux.gtk.x86" value="true" />

<target name="assemble.org.foo.feature.id.linux.gtk.x86">

<property name="archiveName" value="acme-RCP-${buildId}-linux.gtk.x86.tar.gz" />

<-- custom pre assemble steps for linux.gtk.x86-->

<ant antfile="${assembleScriptName}" dir="${buildDirectory}"/>

<-- custom post assemble steps for linux.gtk.x86-->

</target>

customAssembly.xml

The customAssembly.xml script is new in 3.4. It provides customization points that will be called during the assembly and packaging phases of the build. As with customTargets.xml and allElements.xml, to use these customization, copy the template from PDE/Build into your build configuration directory.

The assemble and packaging scripts generated by PDE/Build perform the following actions:

Call the gather.bin.parts target in the generated build.xml for all features and plug-ins being built.

Jar up (and optionally sign) any plug-ins and features that are not shipped as folders

Copy the final results into a staging area and create the final archive

The assemble scripts deal with collecting the features and plug-ins that were compiled during the build and the package scripts deal with collecting any pre-compiled binary features and plug-ins. The customAssembly.xml script provides the following targets before or after each of the above steps:

gather.bin.parts: Called after each invocation of the gather.bin.parts target on the individual plug-ins and features.

post.gather.bin.parts: Called after invoking the gather.bin.parts targets for all plug-ins and features. Results exist as folders and have not yet been jarred.

pre.jarsigning: Called just prior to signing a jar.

post.jarup: Called after all plug-ins and features have been jarred (and potentially signed).

pre.archive: Called just before the archive file is built

3.2.2. Feature and Plug-in custom build steps

New in 3.2 is the ability for a feature or plug-in to contribute custom ant targets to its own build. These custom steps can be used for things like instrumenting byte-code after compilation, indexing help files for documentation, or copying extra files into the bundle.

To use custom targets, a feature or plug-in must define the property customBuildCallbacks in its build.properties file; the value of this property is the location of an xml file, relative to the root of the feature/plug-in, containing the custom targets. If the xml file is located elsewhere, use the property customBuildCallbacks.buildpath to set the directory to use.

Templates named customBuildCallbacks.xml containing the supported custom targets are provided in org.eclipse.pde.build/templates/plugins and org.eclipse.pde.build/templates/features. These templates can be copied into your feature/plug-in and modified as needed.

The custom targets will be invoked using the subant task; by default, failonerror is set to false. To change this set the property customBuildCallbacks.failonerror to true. Similarly, the inheritall parameter defaults to false. Set the property customBuildCallbacks.inheritall to change this.

Plug-ins

The generated build.xml that builds a plug-in contains several targets that perform specific tasks. PDE Build allows the plug-in to provide custom pre and post targets that will be called before and after specific tasks. In addition to any ant properties that may already be defined, each custom target may also be passed a couple of specific properties. The following targets support custom callbacks:

build.jars: The build.jars target will invoke in turn each of the compilation targets that will be built for this bundle. It supports pre.build.jars and post.build.jars targets which are called before and after all the compilation targets are called. Both the pre and post targets are provided with the property build.result.folder which is the folder to which the results of the build will be copied.

Compilation Targets: The build.xml defines a compilation target for each jar being built (eg @dot for the bundle itself, or library.jar). Each compilation target supports 3 custom callbacks:

pre.<compilation target>: Called before source files are compiled. It is provided with the following properties:

target.folder: The folder where the compiled class files will be placed

source.folder<N>: The source folder(s) containing the source to be compiled. (e.g. source.folder1, source.folder2, etc).

<compilation target>.classpath: A reference to the ant path structure containing the classpath that will be used in the compilation. (e.g. @dot.classpath, library.jar.classpath). The reference to the classpath should be used with a refid.

Example:

<classpath refid="library.jar.classpath"/>

<property name="mypath" refid="library.jar.classpath" />

post.compile.<compilation target>: Called after the source is compiled, but before the class files are jared or copied into the build result folder. It is provided with the same properties as the pre.<compilation target> target.

post.<compilation target>: Called at the end of the compilation target after the results are copied to the build result folder. It is provided with the following properties:

jar.Location: The jar or folder containing the compilation results.

<compilation target>.classpath: A reference to the ant path structure containing the classpath that was used in the compilation.

build.sources: This target builds the source zips for each of the compilation targets. It supports pre.build.sources and post.build.sources targets. Both of these targets are provided with the property build.result.folder which is the folder to which the source zip files will be copied.

gather.bin.parts: This target copies the build results (i.e. the jars and any other files being included in the bundle) to the final destination directory. It is also the target where the version qualifiers are replaced in the manifest file (see Version Qualifiers). It supports pre.gather.bin.parts and post.gather.bin.parts targets. Two properties are provided to the pre and post targets:

build.result.folder: the folder containing the results of the build

target.folder: the destination folder to which the build results and bundle contents are being copied.

gather.sources: This target gathers the source zips for the bundle. It supports pre.gather.sources and post.gather.sources targets. Both the pre and post targets are provided with the target.folder property which is the folder that the sources are being copied to.

gather.logs: This target collects the log files from the build. It supports pre.gather.logs and post.gather.logs. Both the pre and post targets are provided with the destination.temp.folder which is the temporary folder that the logs files are being copied to.

clean: This target performs a clean, it supports pre.clean and post.clean targets. The pre and post targets have the following properties:

plugin.destination: The folder that the final bundle was copied to.

build.result.folder: The folder that the results of the build were copied to

temp.folder: The temporary folder where the results were staged before being copied to the final destination.

Features

The generated build.xml for features only supports custom build callbacks for the gather.bin.parts target. This target will invoke the gather.bin.parts target of all the included features and plug-ins. It supports pre.gather.bin.parts and post.gather.bin.parts. Both of these targets are provided with the following properties:

destination.temp.folder: The directory to which the plug-ins will be copied.

feature.directory: The directory to which the files for this feature will be copied.

3.2.3. Fetching from Repositories

The infrastructure offered by PDE Build provides steps to fetch the source code and pre-built plug-ins from a repository as part of the build process (see the eclipse.fetch task and the properties to control fetching). Two possibilities are available: let PDE build automatically fetch the source, or fetch the source code manually.

First step to automation, the map file

Given the identifier of a feature or a plug-in that needs to be built, the build infrastructure is able to fetch from a repository all the included features and plug-ins.

In order to know where to get things from, PDE uses map files. A map file is a java property file which maps feature and plug-in ids to a location and a tag in a repository. The format of map files is open and but the key part is fixed by PDE Build:

<elementType>@<elementID>[,<elementVersion>] = <repo specific content>

where elementType is one of bundle, feature, plugin or fragment, and elementId and elementVersion give the specifics of the element.

Map file entry for CVS

The format of a map file entry to fetch content from CVS is the following:

<key> = CVS, [,args]

where args is a comma-separated list of key/value pairs described below:

tag, the CVS tag to check out;

cvsRoot, a CVS connection string;

path, an optional CVS module and path where to get the folder containing the element. By default the element name is used as module name;

prebuilt, an optional boolean indicating if the element downloaded is prebuilt. By default this is false;

cvsPassFile, an option path to a cvs pass file;

password, an optional password to use to connect to CVS;

Example:

plugin@com.ibm.icu,3.6.1=CVS,tag=v200704191630,cvsRoot=:pserver:anonymous@dev.eclipse.org:/cvsroot/tools,path=org.eclipse.orbit/com.ibm.icu/bin,prebuilt=true

Old style Map Entries

In Eclipse 3.2 and earlier, the map file format for CVS was:

<elementType>@<elementID> = CVS, <TAG>, <CVSROOT>[,<PASSWORD>[,<PATH>[,<CVSPASSFILE>]]]

If any of the optional arguments aren't specified, then the commas are still required to delimit any following arguments. This format should be considered deprecated and the key/value format is prefered.

Map file entry for Ant GET

The format of a map file entry to fetch content from any URL supported by Ant GET is the following:

<key> = GET, <url> [,args]

where url is the url to retrieve the data from and args is an optional comma-separated list of key/value pairs described below:

unpack, set to true the indicate that the downloaded element should be unzipped;

verbose, set to true to show verbose progress information when downloading. The default value is false;

ignoreerrors, set to true to log errors during the transport. The default value is false;

usetimestamp, conditionally download a file based on the timestamp of the local copy (for http URLs only) when set to true. The default value is false;

username, the username for "BASIC" http authentication;

password, the password to perform the download.

Example:

plugin@com.ibm.icu,3.4.5=GET,http://download.eclipse.org/tools/orbit/downloads/drops/S200705301823/bundles/com.ibm.icu\_3.4.5.jar, unpack=true

Map file entry for a p2 fetch

A new extension has been added to PDE/Build that enables users to fetch artifacts from p2 repositories. You can now add an entry to your map file for build input and PDE/Build will retrieve the artifact and include it in the build. The map file entry needs to include the id and version of the installable unit (IU), and the URI of the repository where the IU can be found. The metadata and artifact repositories are assumed to be co-located.

The format of a map file entry to fetch content via p2 is as follows:

<elementType>@<elementID> = p2IU, id=<id>, version=<version>, repository=<repository>

id, the identifier of the Installable Unit;

version, the version of the Installable (optional). If not specified, the highest version available will be fetched.

repository, the location (URI) of the repository where the IU will be found.

Fetching features with p2

In the Helios release, features can be fetched with p2 by specifying the Installable Unit representing the feature jar. See the feature metadata page for details on the structure of p2 metadata for features.

Examples:

plugin@my.bundle.id,1.0.0=p2IU,id=my.bundle.id,version=1.0.0,repository=http://example.eclipse.org/repo

feature@my.feature.id = p2IU,id=my.feature.id.feature.jar, version=1.0.0,repository=http://example.eclipse.org/repo

Map file entry for other repositories

PDE Build provides an extension point where fetch script generators for different repositories can be plugged in. When your eclipse install contains a bundle that provides a script generator extension for a different type of repository, you can use that repository by specifying it in your map file entries and adhering to the format specified by the extension provider.

Setting up for CVS source fetching

To set up fetching your source from CVS, you should do the following:

Make sure you have a cvs.exe on your system search path. (Download CVS from here).

Create a folder called maps in the build directory.

Create a file with extension .map in the maps folder.

Fill in this map file with all the elements that need to be fetched from a repository.

Edit the build.properties file from the configuration folder and comment out the line skipFetch=true. If that property is set, then PDE build will skip the entire fetch phase. If you want the tag specified in the map files to be ignored you can set the property fetchTag to a specific value. This is useful when doing a nightly build from HEAD instead of from the tagged versions.

Getting the map files from CVS

Map files are usually stored in a repository. PDE Build offers default infrastructure to get those map files from a CVS repository. To enable this function change the following properties in your configuration's build.properties from your configuration directory:

skipMaps: Comment out this property. If this property is set, then PDE build will not fetch the map files.

mapsRepo = :pserver:anonymous@example.com/path/to/repo. The CVS repository from which to get the map files.

mapsRoot = /path/to/maps. The path in the CVS repository to the directory containing the map files.

mapsCheckoutTag = HEAD. The CVS tag to use to checkout the map files.

Fetching the map files from CVS occurs during the Pre-Build phase of the build. Scripts to fetch all the features and plug-ins included in the feature you are building will be generated and run during the fetch phase of the build.

Automatic fetching from other repositories

If you are automatically fetching your map files from the repository, you will need to copy the customTarget.xml file from org.eclipse.pde.build/templates/headless-build into your configuration directory. The target getMapFiles is used to fetch the map files, this should be modified to fetch from your repository.

Getting source code manually

You can write custom fetch targets to retrieve your plug-ins and features which can be invoked from the preSetup or postSetup targets in the customTargets.xml file. Features and plug-ins should be fetched to ${buildDirectory}/features and ${buildDirectory}/plugins respectively.

3.2.4. Controlling the output format of the build

This section describes how to control the output of a feature or product build.

Controlling the output format

The archivesFormat property allows specifying the output format (zip, tar, and folder) of the build on a configuration basis. The format of the property is <configuration> - <format>, where configuration is the same value as specified in the configs property and format is one of the following.

folder - the output will be stored in a folder located in ${buildDirectory}/tmp/${archivePrefix}. The targets specified in the allElements.xml are not used in this case. Due to current restrictions, only one configuration can be build at a time when using folder as output format.

zip - the output will be stored in a zip file. The zip file will be created by the zip.exe found on the machine running the build. This is useful if you have non jar'ed plug-ins that contains files requiring specific permissions (mostly on UNIX based systems). The property zipargs allows passing additional options to the zip command.

antZip - the output will be stored in a zip file. The zip file will be created by Ant's implementation of zip. This is the default value when nothing is specified.

tar - the output will be stored in a gzip'ed tar file. The OS implementation of tar and gzip will be used. The property tarargs allows passing additional options to the tar command.

Note that changing the format of the output will not change the name of the created file. It needs to be changed in the allElements.xml file.

Generating JNLP manifests

To enable the generation of JNLP manifests at build time, set generateJnlp to true, set jnlp.codebase to be the value where the jnlp file will be served from (see codebase attribute in jnlp tag), and set jnlp.j2se. Note that this will not package your plug-ins and features for an update site, see the "preparing the output for an update site or jnlp site" below. Due to current restrictions, the creation of the jnlp manifest only works when producing folders as an output format.

Preparing the output for an update site or jnlp site

When put on an update site or a java web start site, plug-ins and features need to be packaged properly. To do this, simply set the outputUpdateJars property to true. Note that this will not generate the site.xml for the eclipse update site and it will not generate jnlp files (see generating jnlp files).

Grouping multiple configurations in one archive

In order to produce an output that contains the plug-ins and features of multiple configurations, set groupConfigurations to true. This will create an archive that contains all the plug-ins and features that match the configurations listed in the configs property. When using this, don't forget to add a target named <feature.id>.group.group.group in the allElements.xml file. Due to current restrictions, the usage of this option will cause the value specified in archivesFormat to be ignored and antZip will be used by default.

Signing jars

In order to sign jars during the build, set the property signJars to true and set values for the properties sign.alias, sign.keystore, and sign.storepass.

3.2.5. Compilation Order and Parallelization

By default, PDE/Build compiles plug-ins by delegating through the feature structure. Features are visited depth first, the plug-ins within each feature are compiled in dependency-order (as determined by the OSGi resolver).

This means that all dependencies of a given bundle must contained in the same feature, or in a feature that appears earlier in the depth-first traversal of the feature inclusion hierarchy. This can make it difficult to organize your features.

Flattening Dependencies

New in 3.5 is a builder property flattenDependencies. Setting this property to true will cause PDE/Build to instead sort all plug-ins regardless accross feature boundaries. PDE/Build will then generate a new build script compile.<feature>.xml which lists all plug-ins in order sorted by their dependencies. This allows you to partition your bundles into different features according to their functionality without worrying about their inter-dependencies.

Parallel Compilation

If flattenDependencies is specified, then PDE/Build is then able to compile bundles in parallel. It does this by partitioning the sorted list of bundles into groups where each group contains bundles that depend only on bundles in earlier groups.

Each group is then wrapped in an Ant <parallel> task. The following properties control the parallelization behaviour:

parallelCompilation Set to true to enable parallel compilation. (Requires flattenDependencies=true)

parallelThreadCount The maximum number of threads to use. Default is 3. Corresponds to threadCount on the ant parallel task.

parallelThreadsPerProcessor The maximum number of threads to use per available processor. Corresponds to threadsPerProcessor on the ant parallel task.

The resulting compile script that gets generated then looks something like this:

<project name="Compile master" default="main">

<target name="main">

<parallel threadsPerProcessor='3'>

<ant antfile="build.xml" dir="plugins/org.eclipse.osgi" target="build.jars"/>

</parallel>

<parallel threadsPerProcessor='3'>

<ant antfile="build.xml" dir="plugins/org.eclipse.equinox.common" target="build.jars"/>

<ant antfile="build.xml" dir="plugins/org.eclipse.equinox.p2.jarprocessor" target="build.jars"/>

</parallel>

<parallel threadsPerProcessor='3'>

<ant antfile="build.xml" dir="plugins/org.eclipse.update.configurator" target="build.jars"/>

<ant antfile="build.xml" dir="plugins/org.eclipse.equinox.simpleconfigurator" target="build.jars"/>

<ant antfile="build.xml" dir="plugins/org.eclipse.equinox.frameworkadmin" target="build.jars"/>

<ant antfile="build.xml" dir="plugins/org.eclipse.core.jobs" target="build.jars"/>

<ant antfile="build.xml" dir="plugins/org.eclipse.core.databinding.observable" target="build.jars"/>

<ant antfile="build.xml" dir="plugins/org.eclipse.swt" target="build.jars"/>

...

3.2.6. Using Version Qualifiers

Eclipse bundles and features have version numbers of the form major.minor.micro.qualifier. The qualifier segment of the version is often used to indicated a specific build. If you set your feature or plug-in version qualifier to "qualifier" (i.e. a version of 1.0.0.qualifier), then PDE build will automatically replace the word "qualifier" with a generated qualifier.

Controlling the Generated Qualifier

A feature or plug-in can control its qualifier by setting the property "qualifier" in its build.properties file. There are are a couple of different cases for the value of this property:

qualifier = none Sets the qualifier to be empty. (i.e. "1.2.3.qualifier" becomes "1.2.3")

qualifier = context Sets the qualifier to be the context qualifier. See below for the value of the context qualifier.

not set If the "qualifier" property is not set, this is equivalent to setting qualifier = context.

qualifier = v12345 Any other value sets the qualifier to be that value.

The Context Qualifier

The value of the generated context qualifier is determined by the first of the following items that apply:

If the property forceContextQualifier is set in your builder's configuration build.properties file, then the value of this property is used as the qualifier. This property can also be set on the command line with -D when invoking ant.

When exporting from the UI, forceContextQualifier is set according to the "Qualifier replacement" value on the options tab of the "Deployable plug-ins and fragments" export wizard.

If you are using map files to fetch your bundles from a repository, then the tag that was used to fetch the bundle will be used as the qualifier.

If neither forceContextQualifier nor map files are used, then the qualifier will be a time stamp in the form YYYYMMDDHHMM (i.e. 200605121600)

Feature Version Suffixes

You can turn on the generation of qualifier suffixes for features by setting the property generateFeatureVersionSuffix=true in the build configuration build.properties file.

The idea of a version is that it should increment each time something changes, the version qualifier increments and captures changes from build to build. Features rarely change, however, they do contain plug-ins which may change often. When version suffixes are turned on, a feature version will have the form major.minor.micro.qualifier-suffix. The suffix is in effect the sum of the version qualifiers of all the features and plug-ins included by the feature. This means that when one of the included plug-ins increments its version, then the version of the feature will automatically increment as well. This is particularly useful when the feature is deployed via an update site.

Final Versions

Occasionally there may be custom build steps that may require the actual final version used in the build. When the property generateVersionsLists is set, PDE Build will generate properties files containing the versions of the features and plug-ins that were built. These properties files can be loaded using the ant property task.

The files are named: finalPluginsVersions.properties, finalPluginsVersions.<config>.properties, finalFeaturesVersions.properties, and finalFeaturesVersions.<config>.properties. finalPluginsVersions.properties and finalFeaturesVersions.properties contain the versions for all of the plug-ins and features that were built. The <config> properties files contain the versions of plug-ins and features for specific build configurations.

The properties in these files have the form:

<bundleSymbolicName> = <version>

<bundleSymbolicName>\_<first 3 segments of version> = <version>

If more than one versions of a plug-in or feature exists, then <bundleSymbolicName>=<version> refers to the plug-in or feature with the highest version. Example:

org.eclipse.pde.build = 3.3.0.v20070524

org.eclipse.pde.build\_3.3.0=3.3.0.v20070524

org.eclipse.pde.build\_3.2.1=3.2.1.r321\_v20060823

3.2.7. Setting the Compilation Environment

By default, your bundles will be compiled with whichever JRE is being used to run the build and with source and target levels set to 1.3 and 1.2.

PDE build is capable of configuring the JRE used during compilation on a per bundle basis. To take advantage of this, the first thing to do is to tell PDE build which JREs you have available. In your build configuration build.properties file, set properties to point to each of your installed JREs. The template build.properties provided in org.eclipse.pde.build/templates/headless-build has a list of environments commented out. Simply uncomment the properties for which you have JREs and set the value to be the set of jars that belong on the boot classpath. For example, setting the property:

J2SE-1.5=C:/Java/1.5/jre/lib/rt.jar;C:/Java/1.5/jre/lib/jsse.jar

will compile bundles requiring java 1.5 against rt.jar and jsse.jar. The names of the properties should match the names of the environments that PDE Build knows about. PDE Build will automatically set the source and target levels for each environment. The environments that PDE builds knows and their associated source/target levels are as follows:

Environment

Source

Target

CDC-1.0/Foundation-1.0

1.3

1.1

CDC-1.1/Foundation-1.1

1.3

1.2

OSGi/Minimum-1.0

1.3

1.1

OSGi/Minimum-1.1

1.3

1.2

JRE-1.1

1.1

1.1

J2SE-1.2

1.2

1.1

J2SE-1.3

1.3

1.1

J2SE-1.4

1.3

1.2

J2SE-1.5

1.5

1.5

JavaSE-1.6

1.6

1.6

PersonalJava-1.1

1.1

1.1

PersonalJava-1.2

1.1

1.1

CDC-1.0/PersonalBasis-1.0

1.3

1.1

CDC-1.0/PersonalJava-1.0

1.3

1.1

CDC-1.1/PersonalBasis-1.1

1.3

1.2

CDC-1.1/PersonalJava-1.1

1.3

1.2

Choosing the Environment for your Bundle

There are two settings that affect the environment used to compile your bundle. One is the manifest header Bundle-RequiredExecutionEnvironment. This header is a list of environments on which your bundle is able to run. Your bundle should only use methods that are in the proper subset of all the environments listed. It is desirable to have your compilation environment match your runtime environment as closely as possible.

The second setting is the property jre.compilation.profile from your bundle's build.properties file. The value of this property should be the preferred environment for compiling this bundle if the Bundle-RequiredExecutionEnvironment list is not adequate.

These two settings together form a list of environments, jre.compilation.profile first, then the environments from Bundle-RequiredExecutionEnvironment. The first environment on this list that is available in your build configuration (see above section) will be used to compile your bundle.

Example:

<buildDirectory>/plugins/bundleA/

META-INF/manifest.mf: Bundle-RequiredExecutionEnvironment: CDC-1.0/Foundation-1.0, J2SE-1.3

build.properties: jre.compilation.profile=J2SE-1.4

<buildConfiguration>/

build.properties: J2SE-1.3=C:/Java1.3/jre/lib/rt.jar;C:/Java1.3/jre/lib/i18n.jar

J2SE-1.4=C:/Java1.4/jre/lib/rt.jar;C:/Java1.4/jre/lib/jsse.jar

In the above setup, the list of environments considered (in order) is: J2SE-1.4, CDC-1.0/Foundation-1.0, J2SE-1.3. The bundle can run on CDC/Foundation and J2SE-1.3, but for some reason, it should be compiled with J2SE-1.4. In this case, J2SE-1.4 is set in the build configuration and comes first in the list of environments, so it is used to compile the bundle. The source and target levels used in this case are 1.3 and 1.2. If instead, the build configuration defined only J2SE-1.3 and not J2SE-1.4, then J2SE-1.3 would be the only environment on the list for which a JRE is available, so it would be used to compile the bundle. In that case, the source and target levels would be 1.3 and 1.1.

Explicitly Setting Compilation values

If, for some reason, using the environments as outlined above does not result in the desired setup, then it is possible for a bundle to explicitly list the settings it requires. For example, if you have a bundle that uses J2SE-1.4 with asserts, then the source and target levels would need to be 1.4, instead of the default 1.3 and 1.2.

To explicitly set the java source and target levels, the bundle should set the properties javacSource and javacTarget in its build.properties file. These properties will override the values derived from the environment settings.

Similarly, setting the property bootClasspath in the bundle's build.properties file will override the set of jars the bundle is compiled against.

3.2.8. Compiler options

PDE/Build supports a number of options to control the compiler for individual plug-ins. For settings related to source and target levels and bootclasspath, see "Setting the Compilation Environment".

Warnings and Errors

PDE/Build allows specifying custom compile warnings and errors in the build.properties file. These can be set on a per library basis (a single plug-in can contain multiple jar libraries) with the javacWarnings.<library> and javacErrors.<library> options. The values for these properties are given in the "-warn:..." section on the using the batch compiler page (from the Java development user guide).

Example:

javacErrors.. = forbidden,discouraged,

javacWarnings.library.jar = deprecation,nullDereference

Warnings and Errors from Project preferences

The JDT project preferences can now also be used during PDE/Build. PDE/Build itself is not able to read the preference files, but it is able to pass them to the JDT compiler which does understand them. To use this, a plug-in should specify the javacProjectSettings property in its build.properties file. The value should be:

javacProjectSettings=true : Use the project setting specified in the default .settings/org.eclipse.jdt.core.prefs preferences file.

javacProjectSettings=path/to/jdt.prefs : Use the specified preference file.

Some projects may wish to use different warning and error settings in the workspace as compared to a releng build. This allows a more convenient method of specifying these settings compared to the "javacWarnings.<library>" property.

File Encoding

PDE/Build can accept custom file encodings on a per library, folder or file basis.

javacDefaultEncoding.<library> - sets default encoding to use when compiling the given library.

javacCustomEncodings.<library> - a comma separated list of paths and encodings to use when compiling the given library. When specifying a folder, the encoding is used for everything contained in that folder. When specifying a file, the encoding is used for just that file. Example:

javacCustomEncodings.library.jar = src/org/foo[US-ASCII], src/org/foo/bar.java[ISO-8859-1]

Custom Compiler Arguments

Custom compiler arguments can now be specified per bundle using the compilerArg property in the bundle's build.properties file. This property is similar to the compilerArg property in the builder configuration except that it affects only the single plug-in instead of all plug-ins. The specific arguments here depend on the compiler being used. The JDT compiler arguments are listed here.

Compiler Option Precedence

There are a number of difference ways to specify different compiler options. As a general rule, option precedence is as follows:

javacDefaultEncoding.<library> has precedence over a custom -encoding command line option.

For general options, an option appearing later on the compiler command line overrides options specified earlier. The options are provided to the JDT in the following order:

The compilerArg property from the builder configuration. (Affects all bundles).

The compilerArg property from this plug-in's build.properties.

The compiler preferences file specified with javacProjectSettings.

The generated compiler arguments file (javaCompiler.<library>.args) containing file encodings, javacWarnings, javacErrors and access rules.

Using a Custom Compiler

A bundle can now use a custom compiler by setting the compilerAdapter property. The specified compiler adapter must be available in the ant environment. The JDT compiler adapter used by default is org.eclipse.jdt.core.JDTCompilerAdapter. One particular reason for specifying an alternate compiler is for Aspect J or Groovy bundles. Bundles specifying a custom compiler adapter may need to also specify some of the following properties:

sourceFileExtensions : A list of extensions that should be considered source. (eg \*.java, \*.aj, \*.groovy).

compilerAdapter.useLog : whether or not to pass "-log" argument to the compiler

compilerAdapter.useArgFile : whether or not to pass the generated javaCompiler.<library>.args argument file. (Contains access rules, file encodings and warning/error levels).

3.2.9. Generating Source Features and Plug-ins

Starting in 3.4, Eclipse has shipped with individual source bundles which allow for more flexible delivery of source. See the "Individual Source Bundles" page for details. Traditional folder-shaped source plug-ins can still be built as outlined below.

Consider the following SDK feature:

features/

org.foo.sdk/

feature.xml: <includes id="org.foo.rcp" />

<includes id="org.foo.rcp.source" />

build.properties: generate.feature@org.foo.rcp.source = org.foo.rcp

org.foo.rcp/

feature.xml: <plugin id="org.foo.BundleA" />

<plugin id="org.foo.BundleB" />

<plugin id="org.foo.BundleB.win32" fragment="true" os="win32" />

The idea of the SDK is that it includes a feature as well as the source for that feature. Notice the generate.feature property in the feature's build.properties file. This property tells PDE Build to generate a feature named "org.foo.rcp.source" based on the contents of the feature "org.foo.rcp". PDE build will generate an org.foo.rcp.source feature, a plug-in that will contain the source code, and fragments containing the source of any platform specific bundles that were included in the feature:

features/

org.foo.rcp.source/

feature.xml: <plugin id="org.foo.rcp.source"/>

<plugin id="org.foo.rcp.source.win32" fragment="true" os="win32" />

plugins/

org.foo.rcp.source/src/

org.foo.BundleA/src.zip

org.foo.BundleB/src.zip

org.foo.rcp.source.win32/src/

org.foo.BundleB.win32/src.zip

Customizing the Generated Source Feature

The generate.feature property has the following form:

generate.feature@<source feature id> = <feature id> [, feature@<feature id>[;attribute=value]\*]\* [, plugin@<plugin id>[;attribute=value]\*]\*

<source feature id>: This is the id of the source feature and plug-in to be generated. Normally it is something like <feature id>.source

<feature id>: The feature on which to base the source feature, the source for all contained plug-ins will be collected into the <source feature id> plug-in.

feature@<feature id>: This will include the named feature in the generated source feature.

plugin@<plug-in id>: This will include the named plug-in in the generated source feature. If the plug-in is JARed specify unpack="false". If multiple versions of the plug-in are available, specify the requested version using version=<pluginVersion> followed by qualifier.

attribute=value : Both feature@ and plugin@ entries can be qualified with additional attributes that should be added in the generated source feature.

Possible attributes (with example values) for feaure@ entries are:

version = 1.0.0 : specify the version of the feature to include

optional = true : set the included feature as optional

ws=gtk : set the window system for the included feature

os=linux : set the operating system for the included feature

arch=x86 : set the architecture for the included feature

Possible attributes for plugin@ entries are:

version = 1.0.0 : specify the version of the plug-in to include

unpack = true : set the included plug-in to be unpacked (ie the plug-in should be installed as a folder).

ws=gtk : set the window system for the included plug-in

os=linux : set the operating system for the included plug-in

arch=x86 : set the architecture for the included plug-in

Use plugin@ to add additional plug-ins that weren't part of the original feature to the generated source feature. This is useful for documentation plug-ins. The feature@ together with a source template can be used to nest source features (see below).

Source Templates

The feature from which the source feature is being generated can provide template files to be included in the generated source feature:

features/

org.foo.rcp/

sourceTemplateFeature/<files to be included in generated source feature>

sourceTemplatePlugin/<files to be included in generated source plugin>

sourceTemplateFragment/<files to be included in generated platform specific fragments>

Any files located in these sourceTemplate folder will be included in the appropriate generated feature/plug-in/fragment. Specifically, files from these directories will replace files generated by PDE build. This can be used to provide a custom feature.xml if there are requirements for your source feature that PDE build does not support.

Nesting Generated Source Features

You can nest source features by providing a sourceTemplateFeature/build.properties file for your generated source feature that contains a generate.feature property for the nested source feature. You will also need to ensure that your top source feature.xml includes the nested source feature, do this either by using feature@ or by providing a template feature.xml:

features/

org.foo.sdk/

sourceTemplateFeature/

build.properties: generate.feature@org.foo.nested.source = org.foo.nested

feature.xml: <includes id="org.foo.rcp" />

<includes id="org.foo.rcp.source" />

build.properties: generate.feature@org.foo.rcp.source = org.foo.rcp, feature@org.foo.nested.source

org.foo.rcp/...

org.foo.nested/...

In the above example, org.foo.sdk contains a generated source feature named org.foo.rcp.source. This generated source feature will get the template sourceTemplateFeature/build.properties file. It will also include org.foo.nested.source in its feature.xml. So when PDE build is processing the generated org.foo.rcp.source, it sees the inclusion of the org.foo.nested.source feature and the generate.feature property and then generates the org.foo.nested.source.

Generating a Source Plug-in

It is also possible to generate a single source plug-in based on a feature instead of generating an entire source feature. The property to do this is:

generate.plugin@<source plug-in id>=<feature-in id>

Be aware that these old-style source plug-ins (one plug-in containing the source for all plug-ins of the feature) must be installed in folder form. Therefore you have to set the unpack="true" attribute in the file feature.xml.

Example:

features/

org.foo.sdk/

feature.xml: <plugin id="org.foo.rcp" />

<plugin id="org.foo.sdk.source" unpack="true" />

build.properties: generate.plugin@org.foo.sdk.source = org.foo.sdk

3.2.10. Individual Source Bundles

The traditional format of source bundles in Eclipse was a folder containing src.zip's for all the plug-ins included in a given feature. Starting in 3.4, Eclipse now ships with individual source bundles. For each bundle there is a corresponding source bundle which is a jar containing the source for that bundle. This allows for more flexible delivery of source, and it also alleviates some path length issues that may be experienced on some platforms.

Eclipse-SourceBundle

A source bundle is identified by the presence of the Eclipse-SourceBundle header in its manifest. The format of this header is:

Eclipse-SourceBundle: <bundle-id>;version=<version>;roots:="root1, root2"

The bundle-id and version indicate the bundle that the included source code corresponds to. The roots directive indicates the folders within the source bundle that actually contain source code. If no roots directive is specified, then a value of "." is assumed.

Class file libraries are mapped to source locations in a predictable manner. The root of the plug-in maps to the root of the source bundle. This is the main case for most jarred plug-ins. If the plug-in contains additional libraries "foo/lib.jar", the source is expected to be in corresponding "foo/libsrc/" folders in the source bundles.

Example 1 : The org.eclipse.pde.core bundle has class files in the root of the jar, plus a nested jar "ant\_tasks/pde-ant.jar". Therefore, the source bundle should have java files in the root and java files in a "ant\_tasks/pde-antsrc" folder. The roots directive lists both the root of the source bundle "." and the "ant\_tasks/pde-antsrc" folder.

org.eclipse.pde.core.jar org.eclipse.pde.core.source.jar

- org/\*\*/\*.class - org/\*\*/\*.java

- ant\_tasks/pde-ant.jar - ant\_tasks/pde-antsrc/org/\*\*/\*.java

Eclipse-SourceBundle: org.eclipse.pde.core;version="3.4.0.N20071128-0010";roots:="ant\_tasks/pde-antsrc,."

Example 2 : The org.eclipse.pde.build bundle is a folder on disk. It contains a pdebuild.jar and a lib/pdebuild-ant.jar. The source bundle will be a jar that contains a "pdebuildsrc" folder and a "lib/pdebuild-antsrc" folder.

org.eclipse.pde.build/ org.eclipse.pde.build.source.jar

- pdebuild.jar - pdebuildsrc/org/\*\*/\*.java

- lib/pdebuild-ant.jar - lib/pdebuild-antsrc/org/\*\*/\*.java

Eclipse-SourceBundle: org.eclipse.pde.build;version="3.4.0.N20071128-0010";roots:="pdebuildsrc,lib/pdebuild-antsrc"

Generating Individual Source Bundles

PDE/Build can automatically generate individual source bundles, but only in a headless build. Individual soure bundle geneation can be turned on by specifying:

individualSourceBundles=true

This must be specified in the build configuration's top level build.properties file and it controls all source generation for that build. See Generating Source Features and Plug-ins for details on generating traditional source plug-ins and features, the remainder of this page assumes familiarity with traditional source generation.

generate.feature

In a feature's build.properties file, the generate.feature property tells pde.build to generate a source feature.

generate.feature@<source feature id> = <feature id> [, feature@<feature id>]\* [, plugin@<plugin id>]\* [, exclude@<plugin id>]\*

When generating individual source bundles, this property remains as before and supports the same attributes (eg version, unpack, optional, etc), the difference will be noticed in the resulting source feature. Before, the source feature would have included 1 source plug-in + a source fragment for each platform being built. In the new format, the source feature will include a source bundle for each plug-in/fragment listed in the original feature.

Plug-ins that were included in the source feature via the plugin@ syntax will not get corresponding source bundles. This is useful for adding doc plug-ins to the source feature.

The exclude@ entry is new for individual source generation. Some plug-ins that were included in the originating feature (ie doc plug-ins or fragments that contain only native code) may not have source and should be excluded from the generated source feature. The exclude@ entry supports a version attribute.

Example:

generate.feature@org.eclipse.jdt.source=org.eclipse.jdt, plugin@org.eclipse.jdt.doc.isv;unpack="false",exclude@org.eclipse.jdt.doc.user

generate.plugin

In the old format, the generate.plugin property generates a source plug-in based on the contents of a given feature. When generating individual source bundles, this changes to be based on a given plug-in:

generate.plugin@<source plug-in id>=<plug-in id>

The generate.plugin property was used by features to include source without having a source feature (even though behind the scenes an entire source feature was generated). When generating individual source bundle, features will need to include a \*.source bundle for each plug-in along with a corresponding generate.plugin property for each one.

Example: The sdk.examples feature used to look like this:

<feature id="org.eclipse.sdk.examples" ... >

....

<plugin id="org.eclipse.sdk.examples.source" version="0.0.0"/>

<plugin id="org.eclipse.sdk.examples.source.win32.win32.x86" version="0.0.0"/>

</feature>

generate.plugin@org.eclipse.sdk.examples.source=org.eclipse.sdk.examples

This changes to:

<feature id="org.eclipse.sdk.examples" ...>

...

<plugin id="org.eclipse.compare.examples.source" version="0.0.0"/>

<plugin id="org.eclipse.debug.examples.core.source" version="0.0.0"/>

<plugin id="org.eclipse.swt.examples.source" version="0.0.0"/>

</feature>

generate.plugin@org.eclipse.compare.examples.source=org.eclipse.compare.examples

generate.plugin@org.eclipse.debug.examples.core.source=org.eclipse.debug.examples.core

generate.plugin@org.eclipse.swt.examples.source=org.eclipse.swt.examples

generateSourceBundle

Specific plug-ins may not require source bundles because they don't actually contain source. This may occur with platform specific fragments that only contain a native library. In this case, the bundles may be excluded by the feature as outlined above in generate.feature.

Or, bundles can explicitly specify in their own build.properties file that no source bundle should be generated for them:

generateSourceBundle=false

Custom Content

Custom content can always be added to generated source bundles using the post.gather.source custom callback in the originating bundle. (See Feature and Plug-in custom build steps).

When doing this, set the property "src.additionalRoots" in the plugin's build.properties file so that the generate source bundle has the correct roots directive on the Eclipse-SourceBundle header.

Example:

org.junit4 :

build.properties

src.includes = about.html,junitsrc.zip

src.additionalRoots=junitsrc

customBuildCallbacks=customBuildCallbacks.xml

customBuildCallbacks.xml

<target name="post.gather.sources" >

<mkdir dir="${target.folder}/junitsrc"/>

<unzip src="../topic/org.eclipse.pde.doc.user/tasks/${target.folder}/junitsrc.zip" dest="${target.folder}/junitsrc" overwrite="false"/>

<delete file="${destination.temp.folder}/junitsrc.zip" />

</target>

Branding the Source Feature

Previously, the generated source plug-in served as a branding plug-in for the generated source feature. Branding files (about.properties, eclipse32.gif, etc) were provided using the sourceTemplatePlugin directory in the original feature.

When generating individual source bundles, the branding plug-in for the source feature will be the source bundle corresponding to the original branding plug-in. This means that branding files for the source feature's branding plug-in can be contributed using the src.includes property in the original branding plug-in.

The sourceTemplatePlugin folder of the feature for which we are generating source also contributes files to the branding plug-in.

sourceTemplatePlugin vs sourceTemplateBundle

Previously, the contents of the sourceTemplatePlugin directory of the original feature was copied into the source plug-in. When generating individual source bundles, this remains true only for the branding source bundle (see above).

For all the other source bundles, the contents of a sourceTemplateBundle directory will be copied over.

3.2.11. Packaging Eclipse Components

Eclipse components are delivered as archive files. Each archive contains a collection of features and plug-ins. Typically the archives contain all the functions that a particular component has to offer. It is often the case that consumers either need more than one component and/or only need parts of some components. Without assistance, consumers would have to manually fetch the archives containing the superset of the functionality they need and then manually extract the required features and plug-ins. This can be a laborious and error prone process.

Fortunately, PDE contains a batch oriented mechanism, the packager, which can help. In short, the packager takes as input a list of archives containing features and a list of interesting features. It then fetches the zips, extracts the features (and their plug-ins) and repackages them into an output zip.

Quick Start

Below is a set of basic steps to get started with the packager. It is suggested that you build \*nix packages on a \*nix machine to ensure that file permissions are preserved.

Create a packaging configuration directory and copy the template files (packager.properties, packaging.properties, and customTargets.xml) from org.eclipse.pde.build/templates/packager here.

Edit the copied packager.properties files and set the following properties:

baseDirectory : the directory in which the actual packaging work will take place.

featureList : a comma separated list of feature ids that you wish to include in your archive.

componentFilter : A comma separated list of components in which your features can be found. Specifying this will avoid unnecessary downloads. Use \* if you don't know the components. (See the componentName in the Packager Map File Format below).

contentFilter : A comma separated list of content types you are interested in. Common content types are runtime or sdk. This is used to optimize the downloading of the archives. Leave this blank to not filter on content type. (See the contentDescription in the Packager Map File Format below).

config : The configurations to package. This is an "&" separated list of comma separated triples of operating system, windowing system and architecture (eg win32,win32,x86 & linux,gtk,ppc)

Get packager map files. The following properties control downloading the map files:

localMaps : Set this property if you have map files locally, put the map files in ${downloadDirectory} (by default this is ${baseDirectory}/toPackage). Comment out this property to automatically download the map file.

packagerMapURL : The URL from which to download the map file.

The packager script will concatenate all \*.map files found in ${downloadDirectory}. To download more than one map file, or to fetch them from CVS edit the customTargets.xml file and change the getMapFiles target.

Run the packager using the following command:

java -jar <eclipse install>/plugins/org.eclipse.equinox.launcher\_<version>.jar -application org.eclipse.ant.core.antRunner

-buildfile <<eclipse install>/plugins/org.eclipse.pde.build\_<version>/scripts/package.xml>

-DpackagingInfo=<path to your packaging configuration directory>

Packager Map File Format

The packager map files are used to describe the various archives, their content and where they can be downloaded from. They are similar to the map files used in the build process to fetch bundles from repositories. The format of a packager map file entry is as follows:

archiveName = url | [configList] | [directory] | contentDescription | componentName

archiveName : This is the name of the archive

url : This is the url where the archive can be found. The concatenation of url and archiveName should point to the archive to download.

configList : This is an "&" separated list of configs that this archive contains. (eg: win32,win32,x86 & linux,gtk,x86). If no value is given the archive is considered to be platform independent.

directory : The directory where the content of the archive should be extracted relative to a standard eclipse installation. When no value is specified it means that the archive contains an eclipse installation directory at its root.

contentDescription : The content flags indicating the type of content in this archive. Normally this is something like runtime, sdk, source, doc, etc.

componentName : The canonical name of the component this archive is for. For example eclipse, jdt, cdt, gef, emf, etc.

Note that configList and directory are optional, it is important that the appropriate number "|" separators are still used.

Examples:

eclipse-SDK-3.1-win32.zip=http://download.eclipse.org/downloads/drops/R-3.1-200506271435/ | win32,win32,x86 | | sdk | eclipse

eclipse-PDE-3.1.zip=http://download.eclipse.org/downloads/drops/R-3.1-200506271435/ | | | runtime | pde

Packager Configuration Properties

The following properties can be set in the packager configuration packager.properties file:

baseDirectory

The directory in which all the packaging will take place

workingDirectory

The directory in which the scripts will be generated. The value in the template is ${baseDirectory}/workingPlace

downloadDirectory

The folder to which the archives will be downloaded. The value in the template is ${baseDirectory}/toPackage

tempDirectory

The folder that the archives will be extracted to. The value in the template is ${baseDirectory}/temp

featurePaths

The name of the root of the archives downloaded. The value in the template is eclipse.

featureList

A comma separated list of feature ids that you wish to repackage.

componentFilter

A comma separated list of the components from which the features in featureList can be found. This filters the available archives and allows the packager to optimize the set of files downloaded.

contentFilter

A comma separated list of content types to fetch. This filters the set of available archives and allows the packager to optimize the set of files downloaded.

config

An "&" separated list of configs (comma separated triples of operating system, windowing system, architecture) to repackage. (eg: win32, win32, x86 & linux, gtk, x86)

archivesFormat

The formats of the archives. An "&" separated list of config - format. (eg: win32, win32, x86 - antZip & linux, gtk, ppc - tar). If no archive format is specified for a given config, the default format is antZip.

zipargs

Extra arguments to be passed to zip

unzipargs

Extra arguments to be passed to unzip

archivePrefix

The prefix of the resulting archive(s)

collectingFolder

The name of the root folder of the resulting archive.

buildId

Controls the build id in the default name of the archive.

buildType

Type of the build, used in naming the build output. Typically one of I, N, M, S, ...

buildLabel

Label for the build. The template value is ${buildType}.${buildId}

archiveNamePrefix

Control the name of the resulting archive. The default value of this is ${buildId}

packagingProperties

A relative path to the file containing the properties packaging root files.. By default this points to packaging.properties.

deltaPack

Whether or not to generate a delta pack. This will exclude platform independent pieces and will group all configs into one archive.

unzipOrder

a comma separated list of archive name prefixes setting the order in which archives should be extracted.

Packaging Root Files

In the packager configuration packager.properties file, the property packagingProperties points to a property file which specifies the root files that need to be packaged. By default this is the file packaging.properties.

This property file should contain the properties specifying the required root files. See the Feature specific properties page for details on the root properties.

3.2.12. Adding Files to the Root of a Build

Root files are files that must be packaged with an Eclipse install but are not features or plug-ins. Common examples are the eclipse executable and ini files as well as licensing files.

Root files are contributed to a build by features containing the appropriate root properties.

Root file Properties

root

Specify a list comma separated list of files or folders to be included in all configurations being built.

Relative paths are relative to the containing feature. Use "file:" to specify a file and "absolute:" to use an absolute path.

Examples:

root=rootfiles: A relative path to a folder, the contents of the folder are included

root=file:license.html,licenses: A relative path to a "license.html" file, and a relative path to a "licenses" folder

root=absolute:/eclipse/rootfiles: An absolute path to a folder, the contents of the folder are included

root=absolute:file:/eclipse/about.html,file:lib/\*.so\*: An absolute path to include "about.html", as well as a relative path to include all the \*.so\* files under the lib folder.

Each comma separated entry becomes an ant fileset. If the entry refers to a folder, then that folder is the "dir" parameter of the fileset and a "\*\*" is used as the "includes" parameter. If the entry is a file, then the containing directory of the file becomes the "dir" parameter and the file is specified in the "includes" parameter.

For all relative files, the property ${baseDir} will be used in the directory. This ${baseDir} property will be feature's root directory where the generated build.xml file will be located.

root.<config>

Specify a list of files or folders to be included for the given configuration. Configurations specified here should correspond to configurations in the configs property for the build. Relative paths are relative to the containing feature. The same "file:" and "absolute:" prefixes used in the "root" property also apply here.

Examples:

root.win32.win32.x86=rootfiles: A relative path to a folder, the contents of the folder are included in the win32.win32.x86 configuration

root.linux.gtk.x86=absolute:file:about.html: An absolute path to a file to include in the linux.gtk.x86 configuration.

root.folder.<subfolder>

root.<config>.folder.<subfolder>

Similar to the "root" and "root.<config>" properties, except that instead of being copied into the root of the eclipse install, files and folders are instea copied into the given subfolder.

Relative paths are relative to the containing feature. The same "file:" and "absolute:" prefixes used in the "root" property also apply here.

Examples:

root.folder.docs/html=html\_files: The contents of the feature's "html\_files" folder are copied into the docs/html folder in the eclipse install.

root.linux.gtk.x86.folder.jre=absolute:/build/jres/linux: An absolute path, the contents of the /build/jres/linux folder are copied into the jre folder of the eclipse install.

These properties are used by PDE/UI when a JRE is specified in a .product file for export. PDE/Build does not currently support including a JRE using the .product file, and these properties would need to be specified in a feature in to include a JRE in a headless product build.

root.permissions.<rights>

root.<config>.permissions.<rights>

Specify a list of files to on which to run the ant chmod task. The value of this property will be passed to the ant chmod task as the "includes" parameter. This means that ant wildcard patterns can be used.

Examples:

root.permissions.755=eclipse: Set the 'eclipse' file to have executable permissions.

root.linux.gtk.x86.permissions.755=lib/\*\*: Set all files under the lib directory to be executable

The <rights> portion of the property is actually passed to the ant chmod task as the "perm" parameter. If the final archive format for the configuration is antTar, then the <rights> format is also passed to an ant tarfileset as the "filemode" parameter. Similarly, for antZip and zipfileset. While the ant chmod task supports permissions such as "og-rwx" (make non-owners unable to touch the file), the tar and zip filesets only support 3 digit octal strings.

root.link

root.<config>.link

List by pairs (separated by comma) the files and folders that need to be symbolically linked. The first entry of the pair is the target and the second entry is the link name..

Example:

root.macosx.carbon.ppc.link=Eclipse.app/Contents/MacOS/eclipse,eclipse: In the macosx.carbon.ppc configuration create a softlink in the root named "eclipse" that points to the "Eclipse.app/Contents/MacOS/eclipse" file.

This results in an ant exec task that executes an "ln -sf" command. Given a value of "targetA, nameA, targetB, nameB, targetC, nameC", a set of "ln" commands will be executed in the folder where the eclipse rootfiles are collected:

ln -sf targetA nameA

ln -sf targetB nameB

ln -sf targetC nameC

Including Root Files in a Product Build

When using a .product file based on features, then rootfiles can be added to the product simply by specifying the above rootfile properties in any of the included features.

It is possible to include a feature solely for the purpose of contributing rootfiles. If this "rootfile" feature does not specify a "bin.includes" property in its build.properties file, then it will not be included in the final build results, but its contributed rootfiles will be.

When a .product file is based on plug-ins instead of features, rootfiles can still be added to the build. In a product build, PDE/Build automatically generates a container feature which includes all the plug-ins specified in the .product file (See the eclipse.generateFeature task). There are two ways to add rootfiles to this product build:

Create a feature that contributes rootfiles, but does not specify a bin.includes property. Add this feature to the automatically generated container feature by setting the featureList property:

${buildDirectory}/features/org.foo.rootfile.feature/rootfiles/\*

/build.properties : root = rootfiles

${buildDirectory}/build.properties: featureList = org.foo.rootfile.feature

Add a build.properties file containing absolute root properties to the automatically generated container feature:

${buildDirectory}/build.properties: generatedBuildProperties = ${buildDirectory}/root.properties

${buildDirectory/root.properties: root = absolute:${buildDirectory}/rootfiles

${buildDirectory}/rootfiles/\*

3.3.1. Building p2 Repositories and Products

PDE/Build has built in support for publishing p2 metadata using the new p2 Publisher. 3.4-style integration with the old metadata generator still works, but does not support newer features like the customization of metadata.

The new publisher integration gathers your features and bundles from source and publishes them directly to a p2 repository. To use the new functionality, the builder should define the property:

p2.gathering = true

Setting this property will change the build in significant manner:

Feature builds produce a single p2 repository that is a group of all the configurations being built.

Product builds produce a properly installed fully enabled p2 product. (And optionally the corresponding repository.)

See the separate feature and product build help pages for more detail on those types of builds.

Overview of the changes

The build's local repository

During the build, all metadata and artifacts will be published into a build repository defined by the property p2.build.repo. This property should specify a URI, the default value for this is file:${buildDirectory}/buildRepo.

Once all the metadata and artifacts are published into this repository, the final assemble and packaging scripts will mirror and/or install from this repository into the locations that will become the archives produced by the build. This final mirroring and installation can be skipped using skipMirroring and skipDirector properties, in which case the build results would all just be in the build repository.

Generated build.xml : gathering vs publishing

In the generated build.xml for features and plugins there is a target gather.bin.parts which was responsible for gathering the contents of the feature/plug-in and copying them to a final directory to be jared (or not) and included in the build results. When using p2.gathering=true, we instead publish the feature/plug-in directly into a p2 repository. That is, we are using p2 to gather the binary artifacts directly from the source (this is where the name of the property comes from).

This is done using a new target named publish.bin.parts which uses a new ant tasks contributed by PDE/Build. In old style builds, calling gather.bin.parts was done by delegating through the feature structure. Calling publish.bin.parts is instead done directly from a new assemble p2 script that is generated.

Rootfiles contributed by features

Old style integrations grouped all the rootfiles contributed by all the features in the build into a single root file installable unit and artifact (which may have been named for a product). With the new publishing support, root files are instead associated with the feature that contributed them. See the feature metadata layout page for more details.

The assemble scripts

Old style integration was done by calling the metadata generator on the binary results during the assembly and packaging process. When using p2.gathering=true there is a new assemble.<feature>.p2.xml script.

This script will invoke the publish.bin.parts for all features and bundles being built. It will also publish metadata for all pre-existing binary features and plug-ins, as well as publishing metadata for products.

At the end of the p2 assemble script, before the remaining assemble and package scripts are called, we have a complete repository with the full results of the build. The remaining assemble and package scripts are simply slicing up the repository and/or performing installs using the p2 director to create the final archives which are the results of the build.

3.3.2. Feature Builds with p2

Defining the new property:

p2.gathering = true

will cause a few changes for feature builds. See also the page for feature metadata layout.

In particular, instead of producing per-platform archives, the build will produce a single p2 repository which is a group of all the platforms. Effectively this is forcing an implicit groupConfigurations=true.

The following is a list of related properties (old and new):

p2.gathering Set to true to turn on p2 publisher based builds.

p2.build.repo A URI to the local build time p2 repository, default is file:${buildDirectory}/buildRepo. Results will be mirrored from here to the final archive location.

groupConfigurations p2.gathering=true has the implicit effect of setting groupConfigurations=true. To control the output format of the archive use the group.group.group configuration.

generate.p2.metadata This is property for the old metadata generator integration. It has no effect when p2.gathering=true.

p2.metadata.repo

p2.artifact.repo These properties were associated with generate.p2.metadata and have no effect on feature builds when p2.gathering=true because the default behaviour in this case is to create a p2 repository. (However, these properties do affect product builds).

p2.metadata.repo.name

p2.artifact.repo.name Optional, these properties will be used to name the final feature repository.

p2.compress Set to true to compress the final feature repository xml into a jar.

p2.flavor No effect for feature builds, but does affect product builds.

p2.publish.artifacts No effect when p2.gathering=true (Old property)

p2.root.name

p2.root.version No effect when p2.gathering=true (Old property)

p2.context.repos Define context repositories. See reusing metadata.

repoBaseLocation A folder containing repositories to transform using <p2.repo2runnable>. See also reusing metadata.

transformedRepoLocation The folder containing the output of <p2.repo2runnable>. See also reusing metadata.

p2.category.site A URL to a site.xml file used to define categories.

p2.category.definition A URL to a category.xml file used to define categories.

p2.category.prefix Define a prefix to ensure unique ids for category IUs generated from site/category files that don't use unique names.

skipMirroring Skip the mirroring step, no final archive is created. Build results are found in ${p2.build.repo}.

3.3.3. Product Builds with p2

Defining the new property:

p2.gathering = true

will cause a few changes for product builds. In particular, the build will produce properly installed fully p2 enabled products.

The following is a list of related properties (old and new):

p2.gathering Set to true to turn on p2 publisher based builds.

p2.build.repo A URI to the local build time p2 repository, default is file:${buildDirectory}/buildRepo. Results will be mirrored from here to the final archive location.

generate.p2.metadata This is property for the old metadata generator integration. It has no effect when p2.gathering=true.

p2.metadata.repo

p2.artifact.repo These properties are be URIs. By default for product builds, the final archives are the installed products and metadata and artifacts are left in the ${p2.build.repo}. If p2.metadata.repo and p2.artifact.repo are defined, then the artifacts and metadata for the product will be mirrored from the build repository.

p2.metadata.repo.name

p2.artifact.repo.name Optional, these properties will be used to name the final repository when p2.metadata.repo and p2.artifact.repo are used.

p2.compress Set to true to compress the final repository xml into a jar.

p2.flavor The flavor of the product, used as a qualifier on the configuration metadata for the product. See below.

p2.product.qualifier The qualifier to use when replacing "1.0.0.qualifier" in a product's version. If not set, the qualifier will be based on forceContextQualifier or the timestamp.

p2.publish.artifacts No effect when p2.gathering=true (Old property)

p2.root.name

p2.root.version No effect when p2.gathering=true (Old property)

p2.context.repos Define context repositories. See reusing metadata.

repoBaseLocation A folder containing repositories to transform using <p2.repo2runnable>. See also reusing metadata.

transformedRepoLocation The folder containing the output of <p2.repo2runnable>. See also reusing metadata.

p2.category.site A URL to a site.xml file used to define categories.

p2.category.definition A URL to a category.xml file used to define categories.

p2.category.prefix Define a prefix to ensure unique ids for category IUs generated from site/category files that don't use unique names.

skipMirroring Skip the final mirroring from ${p2.build.repo} to ${p2.metadata.repo}.

skipDirector Skip the call to the director. No installed products will be produced. If p2.metadata.repo and p2.artifact.repo are defined, those repositories will contain the product metadata and artifacts, otherwise ${p2.build.repo} will contain the results.

p2.director.log Location of a log file to log the results of the director call.

p2.director.profile The name to use for the p2 profile created by the director. Generally it is a good idea to name this something related to your product. Default is "profile".

p2.director.extraArgs Extra arguments to pass to the directory. Default is "-profileProperties org.eclipse.update.install.features=true".

Mirroring Properties When building products or features, PDE/Build mirrors dependencies from context repositories to be included in the build. These properties control what dependencies are included in this operation. See the p2.mirror ant task for details.

p2.mirror.raw Default value: false. Set to true to reuse artifact descriptors from the source repositories.

p2.mirror.slicing.filter Default is no filter. Set additional filter properties IUs must satisfy in order to be included.

p2.mirror.slicing.followOnlyFilteredRequirements Default is false. Set to true to invert the filters and only include IUs that don't match.

p2.mirror.slicing.followStrict Default is false. Set to true to only include dependencies whose versions are exactly specified.

p2.mirror.slicing.includeFeatures Default is true. Whether or not to include features.

p2.mirror.slicing.includeNonGreedy Default is false. Whether or not to include non-greedy dependencies.

p2.mirror.slicing.includeOptional Default is true, whether or not to include optional dependencies.

p2.mirror.slicing.latestVersionOnly Default is false, whether or not to only include the latest version of a dependency when the requirement is a range and multiple versions are available.

p2.mirror.slicing.platformFilter Default is no filter, An "os,ws,arch" triplet to set as the platform against which IU LDAP filters will be matched. IUs not matching the filter will not be mirrored.

Product Flavor

As part of a product build, PDE/Build automatically generates default configuration metadata to set start levels and config.ini property. This metadata is commonly referred to as Configuration Units (CUs). In particular, start levels are set using CU fragments on the IU for the bundle being started. The flavor is used as a qualifier when generating the CU's name based on the IU.

For example, with "p2.flavor = tooling", 'toolingwin32.win32.x86org.eclipse.core.runtime' will be the name of the CU that configures the org.eclipse.core.runtime bundle on windows. It may be a good idea to use a flavor based on your product id to avoid conflicts with other metadata, particularly if your product has particular needs with respect to start levels.

See also the configuring products page.

The director call

PDE/Build includes a runDirector target that is called to perform installs during a product build. Currently, this is a fork of the director application in a new process. In more advanced releng scenarios, it is possible to reuse this task to perform additional installs. The task requires that the location of the equinox launcher for the builder is defined. An example call would look something like this:

<property name="equinoxLauncherJar" value="/builder/eclipse/plugins/org.eclipse.equinox.launcher\_1.0.200.v20090520.jar"/>

<ant antfile="${eclipse.pdebuild.scripts}/genericTargets.xml" target="runDirector" inheritAll="true">

<property name="ws" value="gtk"/>

<property name="os" value="linux"/>

<property name="arch" value="x86"/>

<property name="p2.director.installPath" value="${installFolder}"/>

<property name="p2.repo" value="${p2.build.repo}"/>

<property name="p2.director.iu" value="my.rcp.product"/>

<property name="p2.director.version" value="1.0.0"/>

</ant>

3.3.4. PDE Build Extensions to the p2 Publisher

PDE Build contributes several ant tasks that extend the p2 publisher. These tasks allow PDE Build to publish metadata and artifacts directly into a p2 repository.

The generated build scripts use these tasks in the context of a full build where things like compiling class files and replacing versions are done. However, some of the tasks may be usefull in special cases to publish metadata directly without running a full build.

eclipse.publish.featuresAndBundles

This task is very similar to the p2.publish.featuresAndBundles task contributed by the p2 publisher. This task will publish metadata for pre-existing binary features and plug-ins which are not being built from source. The task will also publish categories based on a provided site.xml or category.xml file.

This task supports the following attributes:

metadataRepository A URI specifying the metadata repository to publish to.

artifactRepository A URI specifying the artifact repository to publish to.

repository Sets both metadataRepository and artifactRepository.

site A URL to a site.xml specifying category information.

category A URL to a category.xml specifyiny category information.

siteQualifier A qualifier used to ensure resulting category IUs have unique names.

<features> A nested fileset element specifying the locations of binary features to publish.

<bundles> A nested fileset element specifying the locations of binary plug-ins to publish.

Example:

<eclipse.publish.featuresAndBundles repository="file:C:/build/repository" category="file:C:/build/category.xml" >

<features dir="C:\eclipse\features\" includes="org.eclipse.cvs\_1.1.100.v20090514-7E79FEc9BJ99r9XGQ3CICF" />

<bundles dir="C:\eclipse\plugins" includes="org.eclipse.cvs\_1.0.300.v20090520.jar" />

<bundles dir="C:\eclipse\plugins" includes="org.eclipse.team.cvs.core\_3.3.200.I20090430-0408.jar" />

<bundles dir="C:\eclipse\plugins" includes="org.eclipse.team.cvs.ssh\_3.2.100.I20090508-2000.jar" />

<bundles dir="C:\eclipse\plugins" includes="org.eclipse.team.cvs.ssh2\_3.2.200.I20090508-2000.jar" />

<bundles dir="C:\eclipse\plugins" includes="org.eclipse.team.cvs.ui\_3.3.200.I20090521-1750.jar" />

</eclipse.publish.featuresAndBundles>

eclipse.gatherBundle

Publish a plug-in directly from source. Once the .class files for a bundle are compiled, this task will gather up all the files that make up the binary result and publish them as a p2 artifact with metadata. This task is used in the publish.bin.parts task in the plug-in's generated build.xml, this task replaces the gather.bin.parts task when you are building with the p2 publisher.

This task supports the following attributes:

metadataRepository A URI to the metadata repository to publish to.

artifactRepository A URI to the artifact repository to publish to.

buildResultFolder Folder containing built .class files, the Manifest and potentially .api\_description files.

targetFolder Content when running with customBuildCallbacks.

baseDirectory Project location to gather content from when not using customBuildCallbacks

unpack Whether or not the plug-in should be unpacked into folder form when installed.

gatheredSource For source bundles, the location of the gather source.

<outputFolder> Nested elements specifying alternate locations to get .class files from, used when reusing .class files from the workspace. This is an ant FileSet with an additional "library" attribute specifying which library the class files belong to

Example:

<eclipse.gatherBundle metadataRepository="file:/build/repo" artifactRepository="file:/build/repo"

buildResultFolder="${basedir}" baseDirectory="${basedir}" unpack="true" >

<outputFolder library="lib/pdebuild-ant.jar" dir="${basedir}" includes="bin\_ant/\*\*" />

<outputFolder library="pdebuild.jar" dir="${basedir}" includes="bin/\*\*" />

</eclipse.gatherBundle>

eclipse.gatherFeature

Publish a feature directly from source. This task will gather up all the files that make up the feature jar and publish them as a p2 artifact with metadata. This task will also publish artifacts containing any root files that are contributed by this feature. If the feature does not specify a bin.includes property, then the feature jar will not be generated, but there will still be root file artifacts and feature group metadata.

This task supports the following attributes:

metadataRepository A URI to the metadata repository to publish to.

artifactRepository A URI to the artifact repository to publish to.

buildResultFolder Folder containing a modified feature.xml with versions replaced.

targetFolder Content when running with customBuildCallbacks.

baseDirectory The feature project location

Example:

<eclipse.gatherFeature

metadataRepository="file:/build/repo"

artifactRepository="file:/build/repo"

buildResultFolder="${basedir}"

baseDirectory="${basedir}"

/>

eclipse.brand.p2.artifacts

Brand launcher artifacts for a product and republish them with a new name for inclusion in the given product.

This task supports the following attributes:

metadataRepository A URI to the metadata repository to publish to.

artifactRepository A URI to the artifact repository to publish to.

config The configuration for which the launchers will be used. (eg "gtk.linux.x86")

iconsList Locations to find the icons to brand the executable with.

tempDirectory The location of a temporary directory that can be be used.

productId The id of the product we are publishing for.

productVersion The version of the product we are publishing for.

launcherName The name to give the new launcher.

launcherProvider The name of the IU which is providing the original launcher artifacts. By default this is "org.eclipse.equinox.executable", the task will be looking for artifacts named <launcherProvider>\_root.<config>

3.3.5. PDE Build Integration with p2

This page describes a deprecated old way of integrating with p2. See the "Building p2 Repositories and Products" page for the recommended methods to build with p2. The properties described below work using a backward compability layer and do not support any new features that may be added to the p2 publisher.

See the Generating p2 metadata page in the Platform Plugin-in Developer Guide for details on p2 metadata generation. Generating metadata by itself it not sufficient to create a p2-enabled product. To do this you must perform an actual p2 install from the generated metadata. See the p2 director documentation for details on how to perform this install from a headless script.

It is possible to generate p2 metadata with manual calls to the p2 metadata generator in any of PDE/Build's custom tasks (see Customizing a Headless Build). However, PDE/Build does provide integration with p2 via the following properties:

generate.p2.metadata

(DEPRECATED, use p2.gathering instead) Set to true to turn on p2 metadata generation. The ant scripts are based on the existence of this property and not the value. To turn off metadata generation, the property must be removed and not just set to false.

In the build scripts, calls to p2 targets are generated when the p2 bundle org.eclipse.equinox.p2.metadata.generator and its dependencies are present in the Eclipse that is running the build. These p2 targets are conditional on the generate.p2.metadata property.

p2.metadata.repo

p2.artifact.repo

Specify a file: URI giving the location of the p2 metadata and artifact repositories to publish to. It is common, but not necessary for the artifact repository to use the same location as the metadata repository.

Example:

p2.metadata.repo=file:${buildDirectory}/repository

p2.artifact.repo=file:${buildDirectory}/repository

p2.metadata.repo.name

p2.artifact.repo.name

Optional. Specify a names for the generator p2 repositories.

When using a repository that already exists, the existing name of the repository will not be changed. If not specified, p2 will name the repositories according to their location.

p2.flavor

Set the flavor for the p2 metadata. When metadata is generated for the Eclipse SDK, a flavor of "tooling" is used.

p2.publish.artifacts

Set to true to publish jars into the artifact repository. If set to false, then artifact metadata is generated but the jars are not copied to the artifact repository. WARNING: when set to true, make sure that the p2.artifact.repo is not the same location as the assembly directories used by PDE/Build, otherwise jars might end up deleted!

p2.root.name

p2.root.version

Specify the name and version for the root IU that will contain everything that was built.

If a .product file is specified in the build, then these properties will be ignored. Instead, when a .product file is available the root name and version will be the product id and version from the .product file. See below for more information.

Metadata Generation for Products

When running a build with a product file set in the top level build configuration build.properties file (both in builds using the productBuild.xml and the normal build.xml), then PDE/Build will automatically generate metadata for that product.

In addition to creating Installable Units for all the features and bundles, this will also create the following product Installable Units:

A top level Product IU using the product-id and the product version. If no version is set in the .product file, "1.0.0" is used.

Configuration Units that set start level information for individual bundles as specified in the config.ini used for the product.

Configuration Units that set program and vm arguments for the product's launcher .ini file.

Configuration Units that add entries to the config.ini based on the config.ini used for the product.

Installable Units to deliver the branded launcher as well as any root files.

Configuration Units that set arguments in the launcher .ini file specifying -startup and --launcher.library to enable installing the product with a bundle pool.

With all of these generated Installable Units, it is possible to use p2 to install the complete product out of the repository.

The Details

PDE/Build is essentially calling the metadata generation task provided by p2 at certain points during the build. It is likely that during a build there is never a single point in time where all build artifacts exist on disk at the same place. Therefore, it would be necessary to either call the p2 metadata generater on the final archives or, as an optimization, call it multiple times throughout the build.

Metadata generation will occur throughout the assemble and packaging phases. This is true for both normal builds and packager builds. To illustrate, for a product build using productBuild.xml, the following scripts will be generated (assuming configs=win32,win32,x86 & linux,gtk,ppc):

assemble.org.eclipse.pde.build.container.feature.all.xml

assemble.org.eclipse.pde.build.container.feature.win32.win32.x86.xml

assemble.org.eclipse.pde.build.container.feature.linux.gtk.ppc.xml

package.org.eclipse.pde.build.container.feature.all.xml

package.org.eclipse.pde.build.container.feature.win32.win32.x86.xml

package.org.eclipse.pde.build.container.feature.linux.gtk.ppc.xml

Assemble scripts collect features and plug-ins that were built along with associated root files. Package scripts collect pre built binary features and plug-ins. The platform specific scripts collect the pieces for those particular configurations and the "all" scripts delegate to each of the platform specific scripts.

p2 generation calls will occur as follows:

assemble win32.win32.x86 : Generate metadata for collected features and plug-ins for windows.

assemble win32.win32.x86 : Second call to generated metadata for collected root files for windows.

assemble linux.gtk.ppc : Generate metadata for collected features and plug-ins for linux.

assemble linux.gtk.ppc : Second call to generated metadata for collected root files for linux.

assemble all : If not running the packager: Generate metadata for a root installable unit that includes everything above.

package win32.win32.x86 : Generate metadata for packaged features and plug-ins for windows.

package win32.win32.x86 : Second call to generated metadata on rootfiles from packaging windows.

package linux.gtk.ppc : Generate metadata for packaged features and plug-ins for linux.

package linux.gtk.ppc : Second call to generated metadata on rootfiles from packaging for linux.

package all : Generate metadata for a root installable unit that includes everything above.

The p2 metadata generator task supports incremental generation, each call to the p2 metadata generated is done with mode=incremental, the final call to generate the root installable unit will run with mode="final"

3.3.6. Layout of Feature Metadata

PDE Build has always used features as a kind of grouping mechanism specifying what exactly should be built. With p2, this idea of a feature as a group has been extended, resulting a more than a single installable unit (IU) being generated per feature.

From build's perspective, a feature contributes three things:

A grouping of nested features and plug-ins.

The (optional) feature jar itself that contains the feature.xml, license files, etc.

A mechanism to contribute root files to the install.

When using the metadata generator introduced in 3.4, we end up with the following structure for feature "org.example.platform":

org.example.platform.feature.group This is the top level grouping IU for the feature, it will have requirements on all the features and plug-ins that were included and required by the feature.xml. It also includes a requirement on the nested org.example.platform.feature.jar

org.example.platform.feature.jar This is the IU representing the feature jar itself. It has an LDAP filter

"(org.eclipse.update.install.features=true)"

which causes the feature jar to only be installed if the profile defines that property. This IU also has a requirement on the actual org.example.platform\_1.0.0.jar artifact.

If the feature does not define "bin.includes" in its build.properties file, then this feature jar IU will not be generated.

Notice this IU structure does not include anything for the root files contributed by the feature. Instead a build using the metadata generation placed all rootfiles together into a single IU and artifact.

New in 3.5 is the p2 publisher. If we use PDE Build's eclipse.gatherFeature task to publish the feature from source, we instead get root file IUs corresponding to the feature that contributed them. In this case, we end up with metadata as follows:

org.example.platform.feature.group The top level grouping IU for the feature

org.example.platform.feature.jar The feature jar IU.

org.example.platform\_root

org.example.platform\_root.<ws>.<os>.<arch> These are the root file IUs. The feature.group will include a root IU per platform for which the feature contributes files. The root IU itself has a requirement on the actual binary artifact from a p2 artifact repository (eg binary/org.example.platform\_root.gtk.linux.x86\_1.0.0) that contains the files.

The use of the new p2 publisher instead of the old metadata generator allows for much finer grain control over how root files are delivered for products.

3.3.7. Reusing p2 Metadata

Metadata for a given feature or bundle can be different depending on how it was generated. Particularily when customizing metadata, or when features contribute root files.

Because of this, it is always a good idea to reuse metadata when possible instead of regenerating it based on binary features and bundles that you may not own.

The repo2runnable task

There is an Ant task <p2.repo2runnable> which will transform feature and bundle artifacts into their installed form. Generally speaking, this is the shape we need the plug-ins to be in if we want to be able to compile against them. This task allows us to easily reconsume repositories from other projects in our own build. PDE/Build has built-in integration with this task through the following properties:

repoBaseLocation A folder containing repositories to transform. This folder can contain zipped repositories as well as sub-folders which are repositories.

transformedRepoLocation The folder where the tranformed artifacts will be placed. This folder will then become a p2 repository itself.

New in 3.5 are custom targets preProcessRepos and postProcessRepos which occur before and after PDE/Build's call to <p2.repo2runnable>. This allows for automatically mirroring or download zipped repositories:

build.properties:

repoBaseLocation=${buildDirectory}/inputRepositories

transformedRepoLocation=${buildDirectory}/transformedRepo

customTargets.xml:

<target name="preProcessRepos">

<p2.mirror source="http://download.eclipse.org/releases/galileo" destination="file:${repoBaseLocation}/mirrored">

<iu id="org.eclipse.equinox.p2.user.ui.feature.group" />

<iu id="org.eclipse.cdt.feature.group"/>

</p2.mirror>

<property name="RCP.Repo-3.5RC3" value="http://download.eclipse.org/eclipse/downloads/drops/S-3.5RC3-200905282000/org.eclipse.rcp-p2repo-3.5RC3.zip" />

<property name="Equinox.Repo-3.5RC3" value="http://download.eclipse.org/equinox/drops/S-3.5RC3-200905282000/equinox-SDK-3.5RC3.zip" />

<get src="../topic/org.eclipse.pde.doc.user/tasks/${RCP.Repo-3.5RC3" dest="${repoBaseLocation}/org.eclipse.rcp-p2repo-3.5RC3.zip" />

<get src="../topic/org.eclipse.pde.doc.user/tasks/${Equinox.Repo-3.5RC3" dest=""${repoBaseLocation}/eclipse-equinox-3.5RC3.zip" />

</target>

(At the time of of this writing, the location of the 3.5 final zips was not yet known, the URLs here serve as an example only and may not exist at any later date.)

PDE/Build will first call the preProcessRepos custom target, then it will call the repo2runnable ant task to transform those downloaded repositories into a form that can be reused by the build.

The transformedRepoLocation property which defines the location for the output of this transformation also serves as a context repository.

Context repositories

The property p2.context.repos is a comma separated list of repositories that serve as context to the build.

When generating metadata, build will first consult the context repositories to see if there is already existing metadata for the feature or plug-in. If metadata already exists, then it will be copied into the build local repository (${p2.build.repo}) instead of new metadata being generated.

As well, if features, bundles, or products have customized their metadata to depend on something that wasn't included in the build, then PDE/Build will perform a mirror out of the context repositories. This mirror operation will include any IUs that were added to feature or plug-in metadata.

3.3.8. Configuring p2 Products

By default, PDE/Build will automatically generate start levels and other configuration information for you. It does this by generating properties into the p2.inf file. If your product provides its own p2.inf file, then PDE/Build will append to a copy of that file. It is possible to suppress this behaviour by defining some properties in your p2.inf:

org.eclipse.pde.build.append Set to false to disable appending any defaults to the p2.inf file.

org.eclipse.pde.build.append.startlevels Set to false to disable appending start level information to the p2.inf file.

org.eclipse.pde.build.append.launchers Set to false to disable including launchers in the p2.inf file.

Start Levels

Start level information can be set manually on the configuration tab of the product editor. If any start level information is set in the .product file, then PDE/Build will not generate any defaults. This means that if you decide to set any custom start levels, then you must set start levels for all bundles that will require them. The default start levels generated by PDE/Build for p2 products are:

Bundle Start Level Auto-Start

org.eclipse.equinox.simpleconfigurator 1 true

org.eclipse.equinox.common 2 true

org.eclipse.update.configurator 4 (default) true

org.eclipse.core.runtime 4 (default) true

org.eclipse.equinox.ds 2 true

When setting the start level for org.eclipse.update.configurator, PDE/Build will also automatically set In addition to these properties, org.eclipse.update.reconcile=false.

Config.ini properties

The p2 product publisher used by PDE/Build will automatically set a number of config.ini properties based on settings in your .product file:

Property Based On

eclipse.product The product set on the overview tab of the product editor.

eclipse.application The application set on the overview tab of the product editor.

osgi.splashPath The plug-in set on the Splash tab of the product editor.

It is also possibly to add arbitrary config.ini properties to your .product file, however there is no tab the editor for this, so it must be done manually by editing the .product file as xml.

An example of setting properties this way would look like this:

...

<configurations>

<-- The product editor will generate start level information like this:

<plugin id="org.eclipse.core.runtime" autoStart="true" startLevel="4" />

-->

<property name="osgi.bundles.defaultStartLevel" value="4" />

<property name="osgi.requiredJavaVersion" value="1.5.0"/>

</configurations>

...

Launchers

PDE/Build will automatically brand launchers (which it gets from the org.eclipse.equinox.executable feature). These launchers will be automatically included in the product along with generated metadata that will set the appropriate -startup and --launcher.library arguments in the product ini file.

Because of changes to the structure of feature metadata with respect to root files, the org.eclipse.equinox.executable feature should not be included directly in your product.

3.4.1. Setting up API Tools

Configure existing plug-ins

This section explains how to enable your existing plug-ins for API analysis.

Select the project you want to convert.

Right-click and select PDE Tools > API Tools Setup...

Once the API Tools Setup wizard is open, check that the project is selected.

If your project contains a file component.xml, you can uncheck the checkbox to preserve it after conversion.

Click Next if you want to preview the changes, or Finish otherwise.

Your plug-in is now converted to use API Tools.

Configure a new plug-in

This section explains how to enable a new plug-ins for API analysis.

Click on File>New>Project...

Select Plug-in Development>Plug-in Project and click Next.

Fill the first page as you would do for any other plug-in and click Next.

Check the option Enable API Analysis.

Plug-in Creation Wizard

Complete your Plug-in setup

Click Finish.

Related tasks

Setting up a baseline

Related references

API Tools Setup Wizard

PDE Tools Menu

API Tools Setup Command

Defining API Restrictions

3.4.2. Setting up an API baseline

Once your plug-in is converted to use API Tools, you need to set up a baseline in order to get the compatibility analysis performed for your plug-in.

To create an API baseline do the following:

Open the Opens the API Baselines preference page Plug-in Development > API Baselines preference page.

Select the Add Baseline... button to open the API baseline wizard.

Give a meaningful name for your baseline

Click Browse... and select the root folder of the Eclipse SDK you want to choose as your baseline

Click OK in the folder dialog

Once the baseline is initialized, click Finish

Click OK and you are all set. If your workspace contains some existing projects, you will be prompted for a rebuild.

Related references

API Baselines Preferences

API Baseline Wizard

Missing API Baseline Quick Fix

3.4.3. Creating API Problem Filters

Filters can be used to remove known or expected API problems that cannot be readily corrected. Problem filters can be created easily using the Create API Problem Filter quick fix.

Using the quick fix let's see how we can add a new filter.

The Problems view

Open the Problems view.

Select an API Tools problem.

Use Ctrl + 1 to trigger the quick fix from the Problems view.

Select Finish to create the new filter.

Editing the resource

Open the resource the API problem exists in.

Use Ctrl + 1 on the corresponding problem.

One of the quick fix proposals will be the creation of a problem filter for the corresponding problem.

Select it and the problem filter is added for you.

Once the problem filter has been created the corresponding resource is rebuilt and the problem is removed from the Problems view and any open editors.

Related references

API Problem Filters Property Page

Add API Problem Filter Quick Fix

3.4.4. Removing filters

If you want to remove an existing API filter, do the following steps:

Select the project that has an existing API filter.

Right-click on the project and select Properties.

Go to Plug-in Development > API Problem Filters

API Filter Properties

From the list of existing filters, select the one you want to remove

Select Remove.

Select OK or Apply.

Related tasks

Creating filters

Related references

API Problem Filters Property Page

Add API Problem Filter Quick Fix

Remove Unused API Problem Filter Quick Fix

3.4.5. Changing API Tools options

The options can be changed per project or per workspace.

First you need to open the preference page. Do one of the following according to the desired scope:

for the project scope, right-click a project and select Properties. Then go to Plug-in Development > API Errors/Warnings

for the workspace scope, go to the Opens the Console preference page Plug-in Development > API Errors/Warnings preference page.

Select the tab for which you want to modify an option:

API Use: for options related to API usage (unsupported tags, API restrictions, API leaks)

API Compatibility: for options related to compatibility issues. They are sorted per type of element

Version Management: for options related to invalid, missing, malformed @since tags or incorrect plug-in versions)

Analysis Options - for general options

All options of a page can be reset to the default values by clicking on Restore Defaults.

Use the buttons insider the Set All As group to modify all options of the page to a specific value.

Related references

API Errors and Warnings Preferences

3.4.6. Comparing to an API baseline

Any API Tools enabled project in your workspace can be compared to any one of the API baselines set up in your workspace. In addition to selecting projects to compare to a baseline you can also select (or multi-select) any of the following within an API tools enabled project:

source folders

package fragments

compilation units

NOTE: all selections must be made from the Package Explorer because the compare wizard works only for Java elements.

To compare to an API baseline do the following:

Select the elements to compare using the criteria mentioned above.

Select the API baseline to compare to in the Compare Wizard

Click Finish and you are all set. Any results from the compare will be shown in the API Tools view.

Related tasks

Setting up a baseline

Related references

API Baselines Preferences

API Baseline Wizard

Missing API Baseline Quick Fix

3.5.1. Export and Install into the Running Host

PDE provides several tools for launching plug-ins and features in your workspace. The Eclipse Application Launcher can be used to create a second running instance of Eclipse that runs the code you have in your workspace. This is commonly referred to as self-hosting. Self-hosting is a powerful debugging and testing tool. However it is sometimes necessary for code to be tested in your host, your currently running Eclipse instance. To test code in the host, you need to export and install into the running Eclipse instance.

To export plug-ins and features and install them into the host Eclipse do the following:

Develop your plug-ins and features in the workspace

Open the export wizard, either Open the plug-in export wizard File > Export... > Plug-in Development > Deployable plug-ins and fragments or Open the feature export wizard File > Export... > Plug-in Development > Deployable features

Select your plug-ins or features to export and install

Select the last option on the Destination tab Install into host. Repository. Then choose a directory to create the repository in

Hit Finish. The export operation will run followed by the installation operation.

If the operations completed succesfully, you will be prompted to restart. Choose to restart now

Your plug-ins will be installed and running after the restart. You can see what has been installed using the Installation Details button on the About Dialog (available by going to Help > About Eclipse SDK)

Related references

Eclipse Application Launcher

Export Wizards

Plug-in Export Wizard

Feature Export Wizard

3.5.2. Import Projects from a Repository

The Plug-in Import Wizard allows you to create projects in the workspace by importing plug-ins from the active target platform, a directory, or a target definition. If the plug-ins you are importing contain appropriate source reference information, the import wizard allows you to import the projects associated with those bundles from a repository.

For example, the plug-ins in the Eclipse SDK contain Eclipse-SourceReference: headers identifying the CVS repository and projects the plug-ins originated from. You can import the specific version of a project that a plug-in was built from or you can import the associated project from HEAD.

To import plug-ins from a repository do the following:

Open the import wizard: Open the plug-in import wizard File > Import... > Plug-in Development > Plug-ins and Fragments

In the Import From section, select a source of plug-ins such as the active target platform.

Import from the active target platform

In the Import As section select Projects from a repository.

Import as projects from a repository

Press Next. A list of plug-ins in the target platform that can be imported from a repository will be shown. Select the plug-ins you want to import and press Next.

The next page allows you to import specific versions of the associated projects (the versions used to build the target platform), or from HEAD. Select the desired option and press Finish. The projects will be imported and appear as projects in the workspace.

Import specific versions or from HEAD

You can also import plug-ins from a repository from the Plug-ins View. Select the plug-ins you want to import and use the Import As > Project from a Repository... action.

Import projects from a repository

Related references

Import Wizards

Plug-in Import

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Reference

Contents

1. API Tools

1.1. API Tools Ant Tasks

1.1.1. Analysis Ant Task

1.1.2. Analysis Report Conversion Ant Task

1.1.3. API Freeze Ant Task

1.1.4. API Freeze Report Conversion Ant Task

1.1.5. API Deprecation Ant Task

1.1.6. API Deprecation Report Conversion Ant Task

1.1.7. API Use Ant Task

1.1.8. API Use Report Conversion Task (Producer)

1.1.9. API Use Report Conversion Task (Consumer)

1.1.10. API Use Migration Ant Task

1.1.11. API Use Migration Report Conversion Task

1.1.12. File Generation Ant Task

1.1.13. Compare Ant Task

1.1.14. API Use Scan Problems Ant Task

1.1.15. API Use Scan Problems Report Conversion Ant Task

1.2. API Tools Command Line Applications

1.2.1. API Analysis application

1.3. API Evolution

1.4. Defining API Restrictions

1.5. Javadoc @since Tag Management

2. Build Configuration

2.1. Builder Configuration properties

2.2. Feature and Plug-in Build configuration

3. Editors

3.1. Plug-in Manifest Editor

3.1.1. Plug-in Overview

3.1.2. Plug-in Dependencies

3.1.3. Plug-in Runtime

3.1.3.1. Access Rules

3.1.4. Plug-in Extensions

3.1.4.1. New Extension Wizard

3.1.5. Plug-in Extension Points

3.1.6. Plug-in Build

3.2. Feature Manifest Editor

3.2.1. Feature Overview

3.2.2. Feature Information

3.2.3. Included Plug-ins

3.2.4. Included Features

3.2.5. Feature Dependencies

3.2.6. Feature Installation

3.3. Site Manifest Editor

3.3.1. Site Map

3.3.2. Site Information

3.4. Extension Point Schema Editor

3.4.1. Schema Overview Page

3.4.2. Schema Definition Page

3.5. Product Configuration Editor

3.5.1. Product Overview

3.5.2. Product Content

3.5.3. Product Configuration

3.5.4. Product Launching

3.5.5. Product Splash

3.5.6. Product Branding

3.5.7. Product Licensing

3.6. Target Definition Editor

3.6.1. Target Definition

3.6.2. Target Content

3.6.3. Target Environment

3.7. Simple Cheat Sheet Editor

3.7.1. Simple Cheat Sheet Page

3.7.1.1. Content Section

3.7.1.2. Title Section

3.7.1.3. Introduction Section

3.7.1.4. Step Section

3.7.1.5. Sub-Step Section

3.7.1.6. Command Section

3.7.1.6.1. Command Composer Dialog

3.7.1.6.1.1. Commands Section

3.7.1.6.1.2. Command Details

3.7.1.7. Help Section

3.7.2. Register Cheat Sheet Wizard

3.8. Composite Cheat Sheet Editor

3.8.1. Composite Cheat Sheet Page

3.9. Table of Contents Editor

3.9.1. Table of Contents Page

3.10. Context Help Editor

3.11. Category Definition Editor

4. Launchers

4.1. Eclipse Application Launcher

4.1.1. Main Tab

4.1.2. Arguments Tab

4.1.3. Plug-ins Tab

4.1.4. Configuration Tab

4.1.5. Tracing Tab

4.1.6. Environment Tab

4.1.7. Common Tab

4.1.8. Prototype Tab

4.2. OSGi Framework Launcher

4.2.1. Bundles Tab

4.2.2. Arguments Tab

4.2.3. Settings Tab

4.2.4. Tracing Tab

4.2.5. Environment Tab

4.2.6. Common Tab

4.3. JUnit Plug-in Test Launcher

4.3.1. Test Tab

4.3.2. Main Tab

4.3.3. Arguments Tab

4.3.4. Plug-ins Tab

4.3.5. Configuration Tab

4.3.6. Tracing Tab

4.3.7. Environment Tab

4.3.8. Common Tab

5. Menus and Actions

5.1. API Tools Setup

5.2. Compare with API baseline

6. Preferences

6.1. Plug-in Development

6.2. API Baselines

6.3. API Errors and Warnings

6.4. Compilers

6.4.1. Plug-in Compiler Preferences

6.4.2. Extension Point Schema Compiler Preferences

6.4.3. Feature and Update Site Compiler Preferences

6.5. Editors

6.6. OSGi Frameworks

6.7. Target Platform

6.8. API Use Scans

7. Property Pages

7.1. API Errors and Warnings

7.2. API Problem Filters

7.3. Runtime Classpath

7.4. Plug-in Manifest Compiler

8. Quickfixes

8.1. Create API Problem Filter

8.2. Missing API Baseline

8.3. Since Tags

8.4. Version Numbering

8.5. Remove Unused API Problem Filters

9. Views

9.1. Plug-in Dependencies

9.2. Plug-in Image Browser

9.3. Plug-in Registry

9.4. Plug-ins

9.5. Target Platform State

10. Wizards and Dialogs

10.1. New Project Creation Wizards

10.1.1. Plug-in Project

10.1.1.1. Plug-in Project Structure

10.1.1.2. Plug-in Content

10.1.1.3. Templates

10.1.2. Plug-in from existing JAR archives

10.1.3. Fragment Project

10.1.3.1. Fragment Project Structure

10.1.3.2. Fragment Content

10.1.4. Feature Project

10.1.5. Feature Patch

10.1.6. Update Site Project

10.2. New File Creation Wizards

10.2.1. Category Definition

10.2.2. Extension Point Schema

10.2.3. Product Configuration

10.2.4. Target Definition

10.2.5. Cheat Sheet

10.2.6. Context Help

10.2.7. Table of Contents

10.3. Import Wizards

10.3.1. Plug-in Import

10.3.2. Feature Import

10.4. Export Wizards

10.4.1. Plug-in Export

10.4.2. Feature Export

10.4.3. Product Export

10.4.4. Target Export

10.5. Task Wizards

10.5.1. Converting Java Projects to Plug-in Projects

10.5.2. Externalizing Strings

10.5.3. Organizing Manifest Files

10.5.4. Updating Classpaths

10.6. API Tools

10.6.1. API Tools Setup Wizard

10.6.2. API Baseline Wizard

10.6.3. API Compare Wizard

10.6.4. API Use Reports

10.6.5. API Use Patterns Wizard

10.7. Target Wizards

10.7.1. New Target Wizard

10.7.2. Edit Target Wizard

10.7.2.1. Locations Tab

10.7.2.2. Content Tab

10.7.2.3. Environment Tab

10.7.2.4. Arguments Tab

10.7.2.4.1. Import Arguments Dialog

10.7.2.5. Implicit Dependencies Tab

10.7.2.5.1. Implicit Dependencies Selection Dialog

10.7.3. Move Target Wizard

11. API Reference

11.1. org.eclipse.pde.api.tools.annotations

11.2. org.eclipse.pde.build

11.3. org.eclipse.pde.core

11.4. org.eclipse.pde.core.build

11.5. org.eclipse.pde.core.plugin

11.6. org.eclipse.pde.core.project

11.7. org.eclipse.pde.core.target

11.8. org.eclipse.pde.launching

11.9. org.eclipse.pde.ui

11.10. org.eclipse.pde.ui.launcher

11.11. org.eclipse.pde.ui.target

11.12. org.eclipse.pde.ui.templates

12. Extension Points Reference

12.1. org.eclipse.ui.trace.traceComponents

12.2. org.eclipse.pde.build.fetchFactories

12.3. org.eclipse.pde.core.bundleClasspathResolvers

12.4. org.eclipse.pde.core.javadoc

12.5. org.eclipse.pde.core.pluginClasspathContributors

12.6. org.eclipse.pde.core.source

12.7. org.eclipse.pde.core.targetLocations

12.8. org.eclipse.pde.core.targets

12.9. org.eclipse.pde.ui.launchShortcuts

12.10. org.eclipse.pde.ui.newExtension

12.11. org.eclipse.pde.ui.osgiFrameworks

12.12. org.eclipse.pde.ui.pluginContent

12.13. org.eclipse.pde.ui.samples

12.14. org.eclipse.pde.ui.targetLocationProvisioners

12.15. org.eclipse.pde.ui.targetProvisioners

12.16. org.eclipse.pde.ui.templates

13. Other Reference Information

13.1. PDE Plug-ins Map

1.1.1. API Analysis Ant Task

Purpose

This task runs a complete API analysis of an API profile relative to a baseline - including API use, binary compatibility, and bundle version number validation. The profile is the current state of a product under development. The profile is compared to an API baseline for binary compatibility (usually the previous release of a product).

The analysis does not include @since tag validation as all verification is performed on binary class files (source is not analyzed).

Usage

Description

The name of the Ant task is: apitooling.analysis. To be used, the jar file apitooling-ant.jar has to be on the Ant classpath.

<apitooling.analysis

baseline="..."

profile="..."

report="..."

filters="..."

excludelist="..."

includelist="..."

preferences="..."

debug="..."

eefile="..."

processunresolvedbundles="..."

/>

Parameters

Attribute Description Required

baseline This attribute specifies the location of the reference baseline.

It can be a .zip, .jar, .tgz, .tar.gz file, or a directory that corresponds to the Eclipse installation folder. This is the directory is which you can find the Eclipse executable.

The location is specified using an absolute path. Yes

profile This attribute specifies the location of the current product or profile that you want to compare against the reference baseline.

It can be a .zip, .jar, .tgz, .tar.gz file, or a directory that corresponds to the Eclipse installation folder. This is the directory is which you can find the Eclipse executable.

The location is specified using an absolute path. Yes

report Set the output location where the reports will be generated.

Once the task is completed, reports are available in this directory using a structure similar to the filter root. A sub-folder is created for each component that has problems to be reported. Each sub-folder contains a file called "report.xml".

A special folder called "allNonApiBundles" is also created in this folder which also contains a file called "report.xml". This file lists all the bundles that are not using the API Tools nature.

There is also a "counts.xml" file created in the report directory that contains a summary of the problems found.

The location is specified using an absolute path. Yes

filters Set the root directory of API filters to use during the analysis.

The argument is the root directory of the .api\_filters files that should be used to filter potential problems created by the API Tools analysis.

The root is specified using an absolute path.

The root needs to contain the following structure:

root

|

+-- component name (i.e. org.eclipse.jface)

|

+--- .api\_filters

No

excludelist Set the exclude list location.

The exclude list location specifies a text file listing bundles to be excluded from the analysis. By default all bundles in the profile are analyzed. The include and exclude list attributes can be used to selectively include and exclude bundles. The include list seeds the set of bundles to be analyzed (when omitted, all bundles are included), and the exclude list is then applied. Each line of the file specifies one of:

a specific bundle name

a regular expression to match against bundle names (lines being with "R:")

a comment (lines being with '#')

The location is specified using an absolute path.

Following is an example exclude list:

# DOC BUNDLES

org.eclipse.jdt.doc.isv

org.eclipse.jdt.doc.user

org.eclipse.pde.doc.user

org.eclipse.platform.doc.isv

org.eclipse.platform.doc.user

# NON-ECLIPSE BUNDLES

com.ibm.icu

com.jcraft.jsch

R:javax\..\*

...

No

includelist Set the include list location.

The include list location specifies a text file listing bundles to be included in the analysis. By default all bundles in the profile are analyzed. The include and exclude list attributes can be used to selectively include and exclude bundles. The include list seeds the set of bundles to be analyzed (when omitted, all bundles are included), and the exclude list is then applied. Each line of the file specifies one of:

a specific bundle name

a regular expression to match against bundle names (lines being with "R:")

a comment (lines being with '#')

The location is specified using an absolute path.

Following is an example include list:

# DOC BUNDLES

org.eclipse.jdt.doc.isv

org.eclipse.jdt.doc.user

org.eclipse.pde.doc.user

R:org.eclipse.platform.doc.\*

# NON-ECLIPSE BUNDLES

com.ibm.icu

com.jcraft.jsch

R:javax\..\*

...

No

preferences Set the preferences for the task.

The preferences are used to configure problem severities. Problem severities have three possible values: Ignore, Warning, or Error. The set of problems detected is defined by corresponding problem preference keys in API tools.

The location is specified using an absolute path. If the given location doesn't exist, the preferences won't be set.

Lines starting with '#' are ignored. The format of the preferences file looks like this:

#Thu Nov 20 17:35:06 EST 2008

ANNOTATION\_ELEMENT\_TYPE\_ADDED\_METHOD\_WITHOUT\_DEFAULT\_VALUE=Ignore

ANNOTATION\_ELEMENT\_TYPE\_CHANGED\_TYPE\_CONVERSION=Ignore

ANNOTATION\_ELEMENT\_TYPE\_REMOVED\_FIELD=Ignore

ANNOTATION\_ELEMENT\_TYPE\_REMOVED\_METHOD=Ignore

ANNOTATION\_ELEMENT\_TYPE\_REMOVED\_TYPE\_MEMBER=Warning

API\_COMPONENT\_ELEMENT\_TYPE\_REMOVED\_API\_TYPE=Ignore

API\_COMPONENT\_ELEMENT\_TYPE\_REMOVED\_TYPE=Ignore

CLASS\_ELEMENT\_TYPE\_ADDED\_METHOD=Error

CLASS\_ELEMENT\_TYPE\_ADDED\_RESTRICTIONS=Ignore

CLASS\_ELEMENT\_TYPE\_ADDED\_TYPE\_PARAMETER=Ignore

CLASS\_ELEMENT\_TYPE\_CHANGED\_CONTRACTED\_SUPERINTERFACES\_SET=Ignore

...

The keys can be found in org.eclipse.pde.api.tools.internal.provisional.problems.IApiProblemTypes. No

debug Set the debug value.

The possible values are: true, false

Default is false. No

eefile Set the execution environment file to use.

By default, an execution environment file corresponding to a JavaSE-1.6 execution environment is used.

The format of the file is described in this wiki page.

The file is specified using an absolute path. No

processunresolvedbundles Set whether bundles with resolver errors should be analyzed.

If set, bundles that have resolver errors (such as missing dependencies) will be included in the analysis. The errors may affect the results so a list of resolver errors is included in the xml output and warnings will be added to the html during report conversion.

The possible values are: true, false

Default is true. No

Examples

<apitooling.analysis

baseline="D:\eclipse\3.4.1\eclipse"

profile="D:\eclipse-SDK-I20081118-0800-linux-gtk.tar.gz"

report="D:\reports\xml"

filters="D:\filters"

excludelist="D:\exclude\_list\_external.txt"

preferences="D:\tests\_api\org.eclipse.pde.api.tools.prefs"

debug="true"

/>

This will run the task creating report.xml files inside the folder D:\reports\xml. It will use the exclude list and the .api\_filter files located in D:\exclude\_list\_external.txt and D:\filters to reduce the number of problems to report.

Problem severities will be generated as specified by D:\tests\_api\org.eclipse.pde.api.tools.prefs.

Since debug is enabled, some debug tracing will show up in the Ant console.

Related references

API Freeze Ant Task

File Generation Ant Task

API Use Ant Task

Analysis Report Conversion Ant Task

API Freeze Report Conversion Ant Task

API Use Report Conversion Ant Task

API Deprecation Ant Task

API Deprecation Report Conversion Ant Task

1.1.2. API Tools Analysis Report Conversion Ant Task

Purpose

This task converts XML reports created by the apitooling.analysis Ant task into HTML files.

Usage

Description

The name of the Ant task is: apitooling.analysis\_reportconversion. To be used, the jar file apitooling-ant.jar has to be on the Ant classpath.

<apitooling.analysis\_reportconversion

xmlfiles="..."

htmlfiles="..."

debug="..."

/>

Parameters

Attribute Description Required

xmlfiles Set the location where the xml reports are retrieved.

The location is set using an absolute path. Yes

htmlfiles Set the location where the html reports are generated.

If not set, the html files are created in the same folder as the xml files.

The location is set using an absolute path. No

debug Set the debug value.

The possible values are: true, false

Default is false. No

Examples

<apitooling.analysis\_reportconversion

xmlfiles="D:\reports\xml"

htmlfiles="D:\reports\html"

debug="true"

/>

This will create an index.html inside the folder D:\reports\html. A report for each bundle will be created inside a sub-folder of D:\reports\html. Each sub-folder will contain a report.html.

Since debug is enabled, some debug tracing will show up in the Ant console.

Related references

Analysis Ant Task

File Generation Ant Task

API Use Ant Task

API Freeze Ant Task

API Freeze Report Conversion Ant Task

API Deprecation Ant Task

API Deprecation Report Conversion Ant Task

API Use Report Conversion Ant Task

1.1.3. API Freeze Ant Task

Purpose

This task identifies APIs that have been added, modified, or removed relative to an API baseline. This task is intended to validate no API changes occur after an API freeze.

Usage

Description

The name of the Ant task is: apitooling.apifreeze. To be used, the jar file apitooling-ant.jar has to be on the Ant classpath.

<apitooling.apifreeze

baseline="..."

profile="..."

report="..."

excludelist="..."

includelist="..."

debug="..."

eefile="..."

processunresolvedbundles="..."

/>

Parameters

Attribute Description Required

baseline This attribute specifies the location of the reference baseline.

It can be a .zip, .jar, .tgz, .tar.gz file, or a directory that corresponds to the Eclipse installation folder. This is the directory is which you can find the Eclipse executable.

The location is specified using an absolute path. Yes

profile This attribute specifies the location of the current product or profile that you want to compare against the reference baseline.

It can be a .zip, .jar, .tgz, .tar.gz file, or a directory that corresponds to the Eclipse installation folder. This is the directory is which you can find the Eclipse executable.

The location is specified using an absolute path. Yes

report Set the name of the report file to be generated.

It should be specified using an absolute path.

This must be a file location, not a directory. Yes

excludelist Set the exclude list location.

The exclude list location specifies a text file listing bundles to be excluded from the analysis. By default all bundles in the profile are analyzed. The include and exclude list attributes can be used to selectively include and exclude bundles. In addition, this task supports listing members in the include and exclude lists. The include list seeds the set of bundles and members to be analyzed (when omitted, all bundles and members are included), and the exclude list is then applied. Each line of the file specifies one of:

a specific bundle name

a specific member name

a regular expression to match against bundle or member names (lines being with "R:")

a comment (lines being with '#')

The location is specified using an absolute path.

Following is an example exclude list:

# ECLIPSE MEMBERS

org.eclipse.jface.databinding\_1.2.0:org.eclipse.jface.databinding.viewers.ObservableListContentProvider#getElements(Ljava/lang/Object;)[Ljava/lang/Object;

# ECLIPSE BUNDLES

org.eclipse.ui.workbench\_3.4.0:org.eclipse.ui.ISharedImages#IMG\_DEC\_FIELD\_ERROR

org.eclipse.ui.workbench\_3.4.0:org.eclipse.ui.ISharedImages#IMG\_DEC\_FIELD\_WARNING

org.eclipse.ui.workbench\_3.4.0:org.eclipse.ui.ISharedImages#IMG\_ELCL\_COLLAPSEALL

# NON-ECLIPSE BUNDLES

com.ibm.icu

R:javax\..\*

...

No

includelist Set the include list location.

The include list location specifies a text file listing bundles to be included in the analysis. By default all bundles in the profile are analyzed. The include and exclude list attributes can be used to selectively include and exclude bundles. In addition, this task supports listing members in the include and exclude lists. The include list seeds the set of bundles and members to be analyzed (when omitted, all bundles and members are included), and the exclude list is then applied. Each line of the file specifies one of:

a specific bundle name

a specific member name

a regular expression to match against bundle or member names (lines being with "R:")

a comment (lines being with '#')

The location is specified using an absolute path.

Following is an example include list:

# ECLIPSE MEMBERS

org.eclipse.jface.databinding\_1.2.0:org.eclipse.jface.databinding.viewers.ObservableListContentProvider#getElements(Ljava/lang/Object;)[Ljava/lang/Object;

# ECLIPSE BUNDLES

org.eclipse.ui.workbench\_3.4.0:org.eclipse.ui.ISharedImages#IMG\_DEC\_FIELD\_ERROR

org.eclipse.ui.workbench\_3.4.0:org.eclipse.ui.ISharedImages#IMG\_DEC\_FIELD\_WARNING

org.eclipse.ui.workbench\_3.4.0:org.eclipse.ui.ISharedImages#IMG\_ELCL\_COLLAPSEALL

# NON-ECLIPSE BUNDLES

com.ibm.icu

R:javax\..\*

...

No

debug Set the debug value.

The possible values are: true, false

Default is false. No

eefile Set the execution environment file to use.

By default, an execution environment file corresponding to a JavaSE-1.6 execution environment is used.

The format of the file is described in this wiki page.

The file is specified using an absolute path. No

processunresolvedbundles Set whether bundles with resolver errors should be analyzed.

If set, bundles that have resolver errors (such as missing dependencies) will be included in the analysis. The errors may affect the results so a list of resolver errors is included in the xml output and warnings will be added to the html during report conversion.

The possible values are: true, false

Default is true. No

Examples

<apitooling.apifreeze

baseline="D:\eclipse\3.4.1\eclipse"

profile="D:\eclipse-SDK-I20081118-0800-linux-gtk.tar.gz"

report="D:\report\report.xml"

excludelist="D:\exclude\_list.txt"

debug="true"

/>

This will run the task creating report.xml inside the folder D:\report. It will use the exclude list D:\exclude\_list\_external.txt to reduce the number of problems to report.

Since debug is enabled, some debug tracing will show up in the Ant console.

Related references

Analysis Ant Task

File Generation Ant Task

API Use Ant Task

Analysis Report Conversion Ant Task

API Freeze Report Conversion Ant Task

API Deprecation Ant Task

API Deprecation Report Conversion Ant Task

API Use Report Conversion Ant Task

1.1.4. API Tools API Freeze Report Conversion Ant Task

Purpose

This task converts the XML report created by the apitooling.apifreeze Ant task into an HTML file.

Usage

Description

The name of the Ant task is: apitooling.apifreeze\_reportconversion. To be used, the jar file apitooling-ant.jar has to be on the Ant classpath.

<apitooling.apifreeze\_reportconversion

xmlfile="..."

htmlfile="..."

debug="..."

/>

Parameters

Attribute Description Required

xmlfile This attribute specifies the location of the xml file to convert.

The location is specified using an absolute path.

This must be a file location, not a directory. Yes

htmlfile This attribute specifies the location of the html file to generate.

If not set, the html file name is retrieved from the xml file name by replacing ".xml" in ".html".

The location is specified using an absolute path.

This must be a file location, not a directory. No

debug Set the debug value.

The possible values are: true, false

Default is false. No

Examples

<apitooling.apifreeze\_reportconversion

xmlfile=="D:\report\report.xml"

debug="true"

/>

This will run the task creating report.html inside the folder D:\report as the htmlfile attribute is not set.

Since debug is enabled, some debug tracing will show up in the Ant console.

Related references

Analysis Ant Task

File Generation Ant Task

API Use Ant Task

AAPI Freeze Ant Task

Analysis Report Conversion Ant Task

API Deprecation Ant Task

API Deprecation Report Conversion Ant Task

API Use Report Conversion Ant Task

1.1.5. API Deprecation Ant Task

Purpose

This task runs a complete API analysis of an API profile relative to a baseline - looking only for members (classes, fields, methods) that have been deprecated / un-deprecated. The profile is the current state of a product under development. The profile is compared to an API baseline for deprecation changes (usually the previous release of a product).

Usage

Description

The name of the Ant task is: apitooling.apideprecation. To be used, the jar file apitooling-ant.jar has to be on the Ant classpath.

<apitooling.apideprecation

baseline="..."

profile="..."

report="..."

excludelist="..."

includelist="..."

debug="..."

eefile="..."

/>

Parameters

Attribute Description Required

baseline This attribute specifies the location of the reference baseline.

It can be a .zip, .jar, .tgz, .tar.gz file, or a directory that corresponds to the Eclipse installation folder. This is the directory is which you can find the Eclipse executable.

The location is specified using an absolute path. Yes

profile This attribute specifies the location of the current product or profile that you want to compare against the reference baseline.

It can be a .zip, .jar, .tgz, .tar.gz file, or a directory that corresponds to the Eclipse installation folder. This is the directory is which you can find the Eclipse executable.

The location is specified using an absolute path. Yes

report Set the output location where the reports will be generated.

Once the task is completed, reports are available in this directory using a structure similar to the filter root. A sub-folder is created for each component that has problems to be reported. Each sub-folder contains a file called "report.xml".

A special folder called "allNonApiBundles" is also created in this folder which also contains a file called "report.xml". This file lists all the bundles that are not using the API Tools nature.

The location is specified using an absolute path. Yes

excludelist Set the exclude list location.

The exclude list location specifies a text file listing bundles to be excluded from the analysis. By default all bundles in the profile are analyzed. The include and exclude list attributes can be used to selectively include and exclude bundles. In addition, this task supports listing members in the include and exclude lists. The include list seeds the set of bundles and members to be analyzed (when omitted, all bundles and members are included), and the exclude list is then applied. Each line of the file specifies one of:

a specific bundle name

a specific member name

a regular expression to match against bundle or member names (lines being with "R:")

a comment (lines being with '#')

The location is specified using an absolute path.

Following is an example exclude list:

# ECLIPSE MEMBERS

org.eclipse.jface.databinding\_1.2.0:org.eclipse.jface.databinding.viewers.ObservableListContentProvider#getElements(Ljava/lang/Object;)[Ljava/lang/Object;

# DOC BUNDLES

org.eclipse.jdt.doc.isv

org.eclipse.jdt.doc.user

org.eclipse.pde.doc.user

org.eclipse.platform.doc.isv

org.eclipse.platform.doc.user

# NON-ECLIPSE BUNDLES

com.ibm.icu

com.jcraft.jsch

javax.servlet

javax.servlet.jsp

...

The exclude list can contain regular expressions if the line starts with "R:". No

includelist Set the include list location.

The include list location specifies a text file listing bundles to be included in the analysis. By default all bundles in the profile are analyzed. The include and exclude list attributes can be used to selectively include and exclude bundles. In addition, this task supports listing members in the include and exclude lists. The include list seeds the set of bundles and members to be analyzed (when omitted, all bundles and members are included), and the exclude list is then applied. Each line of the file specifies one of:

a specific bundle name

a specific member name

a regular expression to match against bundle or member names (lines being with "R:")

a comment (lines being with '#')

The location is specified using an absolute path.

Following is an example include list:

# ECLIPSE MEMBERS

org.eclipse.jface.databinding\_1.2.0:org.eclipse.jface.databinding.viewers.ObservableListContentProvider#getElements(Ljava/lang/Object;)[Ljava/lang/Object;

# DOC BUNDLES

org.eclipse.jdt.doc.isv

org.eclipse.jdt.doc.user

org.eclipse.pde.doc.user

org.eclipse.platform.doc.isv

org.eclipse.platform.doc.user

# NON-ECLIPSE BUNDLES

com.ibm.icu

com.jcraft.jsch

javax.servlet

javax.servlet.jsp

...

No

debug Set the debug value.

The possible values are: true, false

Default is false. No

eefile Set the execution environment file to use.

By default, an execution environment file corresponding to a JavaSE-1.6 execution environment is used.

The format of the file is described in this wiki page.

The file is specified using an absolute path. No

Examples

<apitooling.apideprecation

baseline="D:\eclipse\3.4.1\eclipse"

profile="D:\eclipse-SDK-I20081118-0800-linux-gtk.tar.gz"

report="D:\reports\xml"

excludelist="D:\exclude\_list\_external.txt"

debug="true"

/>

This will run the task creating report.xml files inside the folder D:\reports\xml. It will use the exclude list to reduce the number of problems to report.

Since debug is enabled, some debug tracing will show up in the Ant console.

Related references

API Freeze Ant Task

File Generation Ant Task

API Use Ant Task

Analysis Report Conversion Ant Task

API Freeze Report Conversion Ant Task

API Use Report Conversion Ant Task

API Deprecation Report Conversion Ant Task

1.1.6. API Deprecation Report Conversion Ant Task

Purpose

This task converts the XML report created by the apitooling.apideprecation Ant task into HTML files.

Usage

Description

The name of the Ant task is: apitooling.apideprecation\_reportconversion. To be used, the jar file apitooling-ant.jar has to be on the Ant classpath.

<apitooling.apideprecation\_reportconversion

xmlfile="..."

htmlfile="..."

debug="..."

/>

Parameters

Attribute Description Required

xmlfile This attribute specifies the location of the xml file to convert.

The location is specified using an absolute path. Yes

htmlfile This attribute specifies the location of the html file to generate.

If omitted, the html file name will be the given xml file name where the extension is replaced with html.

The location is specified using an absolute path. No

debug Set the debug value.

The possible values are: true, false

Default is false. No

Examples

<apitooling.apideprecation\_reportconversion

xmlfile="/eclipse/deprecation/deprecation.xml"

debug="true"

/>

This will run the task creating the deprecation.html file inside the folder /eclipse/deprecation.

If debug is enabled, some debug tracing will show up in the Ant console.

Related references

Analysis Ant Task

File Generation Ant Task

API Use Ant Task

API Freeze Ant Task

API Deprecation Ant Task

Analysis Report Conversion Ant Task

1.1.7. API Use Ant Task

Purpose

This task runs a complete search of a given baseline to create a report of API usage.

The analysis does not include bundles that are not API Tools enabled.

Usage

Description

The name of the Ant task is: apitooling.apiuse. To be used, the jar file apitooling-ant.jar has to be on the Ant classpath.

<apitooling.apiuse

location="..."

scopepattern="..."

referencepattern="..."

report="..."

considerinternal="..."

considerapi="..."

considerillegaluse="..."

apipatterns="..."

internalpatterns="..."

archivepatterns="..."

excludelist="..."

includelist="..."

filters="..."

debug="..."

/>

Parameters

Attribute Description Required

location This attribute specifies the location of the current product you want to search for API usage.

It can be a .zip, .jar, .tgz, .tar.gz file, or a directory containing a product or a collection of bundles.

The location is specified using an absolute path. Yes

scopepattern This attribute specifies a regular expression pattern used to select what bundles in the location will be searched.

For example, the following would only scan for API usage in Eclipse bundles:

org.eclipse.\*

The pattern must be a well-formatted regular expression as defined here. No

referencepattern This attribute specifies a regular expression pattern used to filter the search results. Only API usage that references a bundle matching the filter will be reported.

For example, the following would only report usage that references code in JDT:

org.eclipse.jdt.\*

The pattern must be a well-formatted regular expression as defined here. No

report Set the output location where the reports will be generated.

Once the task is completed, reports are available in this directory using a hierarchical structure. A sub-folder is created for each component that has another bundle that references it. Each sub-folder contains a file called "[bundlename].xml". Within this folder is another sub-folder (named "[referencing bundle name].xml") for each bundle that references the bundle of the containing folder. Inside the referencing bundle folder is one of three sub-folders (API, PRIVATE or OTHER) which contain type, method or field references of that kind.

Three other special files are written into the report directory, which are "counts.xml", "not\_searched.xml" and "no\_apidescription.xml" and they contain information about the number of references found, bundles from the baseline that were not search for usage information and those that did not have an .api\_description files in them, respectively.

The location is specified using an absolute path.

For example:

root

|

+-- "not\_searched.xml"

+-- "no\_apidescription.xml"

+-- "counts.xml"

+-- bundle name with usage

|

+--referencing bundle name

|

+-- [API or PRIVATE or OTHER]

|

+--[type\_ or method\_ or field\_]references.xml

Yes

considerinternal If internal references should be considered during the search.

Default is false. No

considerapi If API references should be considered during the search.

Default is false. No

considerillegaluse If illegal API use references should be considered during the search.

Default is false. No

apipatterns A comma separated list of package name patterns (regular expressions) to consider as API packages.

By default API descriptions are used in the scanned bundles, but this can be used to override or add API packages. No

internalpatterns A comma separated list of package name patterns (regular expressions) to consider as private packages.

By default API descriptions are used in the scanned bundles, but this can be used to override or add private packages. No

archivepatterns A comma separated list of archive identifiers to ignore during the scan.

An archive identifier is of the form: [bundle id]:[path to archive]. No

excludelist Set the exclude list location.

The exclude list location specifies a text file listing bundles to be excluded from the analysis. By default all bundles in the profile are analyzed. The include and exclude list attributes can be used to selectively include and exclude bundles. The include list seeds the set of bundles to be analyzed (when omitted, all bundles are included), and the exclude list is then applied. Each line of the file specifies one of:

a specific bundle name

a regular expression to match against bundle names (lines being with "R:")

a comment (lines being with '#')

The location is specified using an absolute path.

Following is an example exclude list:

# EXCLUDED BUNDLES

R:org\.eclipse\..\*\.doc\..\*

...

No

includelist Set the include list location.

The include list location specifies a text file listing bundles to be included in the analysis. By default all bundles in the profile are analyzed. The include and exclude list attributes can be used to selectively include and exclude bundles. The include list seeds the set of bundles to be analyzed (when omitted, all bundles are included), and the exclude list is then applied. Each line of the file specifies one of:

a specific bundle name

a regular expression to match against bundle names (lines being with "R:")

a comment (lines being with '#')

The location is specified using an absolute path.

Following is an example include list:

# INCLUDED BUNDLES

R:org\.eclipse\..\*

...

No

filters Set the root directory of API filters to use during the use scan.

The argument is the root directory of the .api\_filters files that should be used to filter references.

The .api\_filters files specify specific problems to ignore during api analysis. During the use scan, the problem filters will be converted to a list of references that will be filtered from the use scan results.

The root is specified using an absolute path.

The root needs to contain the following structure:

root

|

+-- component name (i.e. org.eclipse.jface)

|

+--- .api\_filters

No

debug Set the debug value.

The possible values are: true, false

Default is false. No

Examples

<apitooling.apiuse

location="/eclipse/eclipse.tar.gz"

scopepattern=".\*"

referencepattern="org\.eclipse.\*"

report="/eclipse/apiuse-both/XML"

considerinternal="true"

considerapi="true"

considerillegaluse="true"

archivepatterns="org.eclipse.test.bundle:/libs/contributed.jar"

excludelist="D:\exclude\_list\_external.txt"

debug="true"

/>

This will run the task creating \*.xml files inside the folder /eclipse/apiuse-both/XML. The task will search all bundles in the product location - defined by the regular expression .\* - searching for all references to any bundles whose identifier begins with org.eclipse (defined by the org\.eclipse.\* regular expression). It will use the exclude list file located in D:\exclude\_list\_external.txt to reduce the number of problems to report.

While scanning, any class files found in the /libs/contributed.jar archive inside org.eclipse.test.bundle will be ignored.

If debug is enabled, some debug tracing will show up in the Ant console.

Related references

API Freeze Ant Task

File Generation Ant Task

Analysis Report Conversion Ant Task

API Freeze Report Conversion Ant Task

API Deprecation Ant Task

API Deprecation Report Conversion Ant Task

API Use Report Conversion Ant Task

1.1.8. API Use Report Conversion Ant Task

Purpose

This task converts the XML report created by the apitooling.apiuse Ant task into HTML files. The HTML report will organize the references by the bundle that produced them (i.e. the bundle that contains the api/type being referenced).

Usage

Description

The name of the Ant task is: apitooling.apiuse\_reportconversion. To be used, the jar file apitooling-ant.jar has to be on the Ant classpath.

<apitooling.apiuse\_reportconversion

xmlfiles="..."

htmlfiles="..."

xsltfile="..."

filterpatterns="..."

tofilterpatterns="..."

debug="..."

/>

Parameters

Attribute Description Required

xmlfiles This attribute specifies the location of the xml files to convert.

The location is specified using an absolute path. Yes

htmlfiles This attribute specifies the location of the html files to generate.

The location is specified using an absolute path. Yes

xsltfile This attribute specifies the location of an XSLT file to be used to generate the reference details pages in a desired ordering.

A default XSLT file is available in the apitooling-ant.jar JAR file.

The location is specified using an absolute path. No

filterpatterns This attribute specifies a comma separated listing of regular expression patterns used to prune references from matching name patterns from the final report.

The pattern must be a well-formatted regular expression as defined here. No

tofilterpatterns This attribute specifies a comma separated listing of regular expression patterns used to prune references to matching name patterns from the final report.

The pattern must be a well-formatted regular expression as defined here. No

debug Set the debug value.

The possible values are: true, false

Default is false. No

Examples

<apitooling.apiuse\_reportconversion

xmlfiles="/eclipse/apiuse-both/XML"

htmlfiles="/eclipse/apiuse-both/HTML"

xsltfile="/eclipse/references.xsl"

debug="true"

/>

This will run the task creating \*.html files inside the folder /eclipse/apiuse-both/HTML corresponding to the folder structure of the XML file location

If debug is enabled, some debug tracing will show up in the Ant console.

Related references

API Consumer Use Report Conversion Ant Task

Analysis Ant Task

File Generation Ant Task

API Use Ant Task

API Freeze Ant Task

API Deprecation Ant Task

API Deprecation Report Conversion Ant Task

Analysis Report Conversion Ant Task

1.1.9. API Consumer Use Report Conversion Ant Task

Purpose

This task converts the XML report created by the apitooling.apiuse Ant task into HTML files. The HTML report organizes references by the bundle that consumes the reference (i.e. contains the code that references the api/type).

Usage

Description

The name of the Ant task is: apitooling.apiconsumeruse\_reportconversion. To be used, the jar file apitooling-ant.jar has to be on the Ant classpath.

<apitooling.apiconsumeruse\_reportconversion

xmlfiles="..."

htmlfiles="..."

xsltfile="..."

filterpatterns="..."

tofilterpatterns="..."

debug="..."

/>

Parameters

Attribute Description Required

xmlfiles This attribute specifies the location of the xml files to convert.

The location is specified using an absolute path. Yes

htmlfiles This attribute specifies the location of the html files to generate.

The location is specified using an absolute path. Yes

xsltfile This attribute specifies the location of an XSLT file to be used to generate the reference details pages in a desired ordering.

A default XSLT file is available in the apitooling-ant.jar JAR file.

The location is specified using an absolute path. No

filterpatterns This attribute specifies a comma separated listing of regular expression patterns used to prune references from matching name patterns from the final report.

The pattern must be a well-formatted regular expression as defined here. No

tofilterpatterns This attribute specifies a comma separated listing of regular expression patterns used to prune references to matching name patterns from the final report.

The pattern must be a well-formatted regular expression as defined here. No

debug Set the debug value.

The possible values are: true, false

Default is false. No

Examples

<apitooling.apiuse\_reportconversion

xmlfiles="/eclipse/apiuse-both/XML"

htmlfiles="/eclipse/apiuse-both/HTML"

xsltfile="/eclipse/references.xsl"

debug="true"

/>

This will run the task creating \*.html files inside the folder /eclipse/apiuse-both/HTML corresponding to the folder structure of the XML file location

If debug is enabled, some debug tracing will show up in the Ant console.

Related references

API Use Report Conversion Ant Task

Analysis Ant Task

File Generation Ant Task

API Use Ant Task

API Freeze Ant Task

API Deprecation Ant Task

API Deprecation Report Conversion Ant Task

Analysis Report Conversion Ant Task

1.1.10. API Use Migration Ant Task

Purpose

This task takes a given API use scan and tries to re-resolve it within a given candidate product release and reports any unresolved references.

Usage

Description

The name of the Ant task is: apitooling.apimigration. To be used, the jar file apitooling-ant.jar has to be on the Ant classpath.

<apitooling.apimigration

candidate="..."

usescan="..."

scopepattern="..."

referencepattern="..."

excludelist="..."

includelist="..."

report="..."

debug="..."

/>

Parameters

Attribute Description Required

candidate This attribute specifies the product you want to use as the migration candidate.

It can be a .zip, .jar, .tgz, .tar.gz file, or a directory that corresponds to the Eclipse installation folder. This is the directory is which you can find the Eclipse executable.

Yes

usescan This attribute specifies the product use scan you want to compare to the migration candidate.

The usescan is specified using an absolute path to the root of the use scan. Yes

scopepattern This attribute specifies the regular expression pattern used to build the scope of elements to search for references from in the product location.

The pattern must be a well-formatted regular expression as defined here. No

referencepattern Set the regular expression pattern used to build the scope of elements to search for references to in the product location.

The pattern must be a well-formatted regular expression as defined here. No

excludelist Set the exclude list location.

The exclude list location specifies a text file listing bundles to be excluded from the analysis. By default all bundles in the profile are analyzed. The include and exclude list attributes can be used to selectively include and exclude bundles. The include list seeds the set of bundles to be analyzed (when omitted, all bundles are included), and the exclude list is then applied. Each line of the file specifies one of:

a specific bundle name

a regular expression to match against bundle names (lines being with "R:")

a comment (lines being with '#')

The location is specified using an absolute path.

Following is an example exclude list:

# EXCLUDED BUNDLES

R:org\.eclipse\..\*\.doc\..\*

...

No

includelist Set the include list location.

The include list location specifies a text file listing bundles to be included in the analysis. By default all bundles in the profile are analyzed. The include and exclude list attributes can be used to selectively include and exclude bundles. The include list seeds the set of bundles to be analyzed (when omitted, all bundles are included), and the exclude list is then applied. Each line of the file specifies one of:

a specific bundle name

a regular expression to match against bundle names (lines being with "R:")

a comment (lines being with '#')

The location is specified using an absolute path.

Following is an example include list:

# INCLUDED BUNDLES

R:org\.eclipse\..\*

...

No

report Set the output location where the reports will be generated.

Once the task is completed, reports are available in this directory using a hierarchical structure. A sub-folder is created for each component that has another bundle that references it. Each sub-folder contains a file called "[bundlename].xml". Within this folder is another sub-folder (named "[referencing bundle name].xml") for each bundle that references the bundle of the containing folder. Inside the referencing bundle folder is one of three sub-folders (API, PRIVATE or OTHER) which contain type, method or field references of that kind.

Two other special files are written into the report directory, which are "not\_searched.xml" and "no\_apidescription.xml" and they contain information about bundles from the baseline that were not search for usage information and those that did not have an .api\_description files in them, respectively.

The location is specified using an absolute path.

For example:

root

|

+-- "not\_searched.xml"

+-- "no\_apidescription.xml"

+-- bundle name with usage

|

+--referencing bundle name

|

+-- [API or PRIVATE or OTHER]

|

+--[type\_ or method\_ or field\_]references.xml

Yes

debug Set the debug value.

The possible values are: true, false

Default is false. No

Examples

<apitooling.apimigration

candidate="/eclipse/sdk3.5"

usescan="/eclipse/scans/sdk3.4"

scopepattern=".\*"

referencepattern="org\.eclipse.\*"

excludelist="D:\exclude\_list\_external.txt"

report="/eclipse/apimigration/xml"

debug="true"

/>

This will run the task creating \*.xml files inside the folder /eclipse/apimigration/xml. The task will re-resolve all references in the usescan location - defined by the regular expression .\* - against the candidate product whose identifier begins with org.eclipse (defined by the org\.eclipse.\* regular expression). It will use the exclude list file located in D:\exclude\_list\_external.txt to reduce the number of problems to report.

If debug is enabled, some debug tracing will show up in the Ant console.

Related references

API Freeze Ant Task

File Generation Ant Task

Analysis Report Conversion Ant Task

API Freeze Report Conversion Ant Task

API Deprecation Ant Task

API Deprecation Report Conversion Ant Task

API Use Report Conversion Ant Task

API Use Migration Report Conversion Ant Task

1.1.11. API Use Migration Report Conversion Ant Task

Purpose

This task converts the XML report created by the apitooling.apimigration Ant task into HTML files.

Usage

Description

The name of the Ant task is: apitooling.apimigration\_reportconversion. To be used, the jar file apitooling-ant.jar has to be on the Ant classpath.

<apitooling.apimigration\_reportconversion

xmlfiles="..."

htmlfiles="..."

xsltfile="..."

filterpatterns="..."

tofilterpatterns="..."

debug="..."

/>

Parameters

Attribute Description Required

xmlfiles This attribute specifies the location of the xml files to convert.

The location is specified using an absolute path. Yes

htmlfiles This attribute specifies the location of the html files to generate.

The location is specified using an absolute path. Yes

xsltfile This attribute specifies the location of an XSLT file to be used to generate the reference details pages in a desired ordering.

A default XSLT file is available in the apitooling-ant.jar JAR file.

The location is specified using an absolute path. No

filterpatterns This attribute specifies a comma separated listing of regular expression patterns used to prune references from matching name patterns from the final report.

The patterns must be a well-formatted regular expression as defined here. No

tofilterpatterns This attribute specifies a comma separated listing of regular expression patterns used to prune references to matching name patterns from the final report.

The pattern must be a well-formatted regular expression as defined here. No

debug Set the debug value.

The possible values are: true, false

Default is false. No

Examples

<apitooling.apimigration\_reportconversion

xmlfiles="/eclipse/apimigration-both/XML"

htmlfiles="/eclipse/apimigration-both/HTML"

xsltfile="/eclipse/references.xsl"

debug="true"

/>

This will run the task creating \*.html files inside the folder /eclipse/apimigration-both/HTML corresponding to the folder structure of the XML file location

If debug is enabled, some debug tracing will show up in the Ant console.

Related references

Analysis Ant Task

File Generation Ant Task

API Use Ant Task

API Use Report Conversion Ant Task

API Use Migration Ant Task

API Freeze Ant Task

API Deprecation Ant Task

API Deprecation Report Conversion Ant Task

Analysis Report Conversion Ant Task

1.1.12. API Tools File Generation Ant Task

Purpose

This task runs to generate all files required by API Tools inside a binary bundle. This task is run during the Eclipse builds or during the bundle export for all projects that have an API Tools nature.

Right now, only the file called .api\_description is created.

Usage

Description

The name of the Ant task is: apitooling.apigeneration. To be used, the jar file apitooling-ant.jar has to be on the Ant classpath.

<apitooling.apigeneration

projectname="..."

project="..."

binary="..."

target="..."

encoding="..."

debug="..."

/>

Parameters

Attribute Description Required

projectname Set the project name Yes

project Set the project location.

This is the folder that contains all the source files for the given project.

The location is set using an absolute path. Yes

binary Set the binary locations.

This is a list of folders or jar files that contain all the .class files for the given project. They are separated by the platform path separator. Each entry must exist.

They should be specified using absolute paths. Yes

target Set the target location.

This is the folder in which the generated files are generated.

The location is set using an absolute path. Yes

encoding Set the file encoding.

This is the file encoding to use while reading source files during the generation. If an encoding is not specified the system property file.encoding will be used.

The specified encoding is not checked to ensure it is valid in any way and an invalid encoding will cause an UnsupportedEncodingException to be thrown from the Ant task. No

debug Set the debug value.

The possible values are: true, false

Default is false. No

extramanifests Set the extra manifest files' locations.

This is a list of extra MANIFEST.MF files' locations that can be set to provide more api packages to scan. They are separated by the platform path separator. Each entry must exist.

If the path is not absolute, it will be resolved relative to the current working directory.

Jar files can be specified instead of MANIFEST.MF file. If a jar file is specified, its MANIFEST.MF file will be read if it exists. No

extrasourcelocations Set the extra source locations.

This is a list of locations for source files that will be scanned. They are separated by the platform path separator. Each entry must exist.

They should be specified using absolute paths. No

allownonapiproject Allow the task to run on non-API Tools enabled projects.

If this task is run on a project that does not have an API Tools nature, no api\_description file will be generated as it is assumed that the project does not support API Tools. In some cases there is a need to generate the description file on projects without the nature. Setting this parameter to true will cause the task to not check for a nature. No

Examples

<apitooling.apigeneration

projectname="org.eclipse.jdt.core\_3.5.0.v\_927"

project="d:\eclipse\org.eclipse.jdt.core"

binary="d:\eclipse\org.eclipse.jdt.core\bin;d:\eclipse\org.eclipse.jdt.core\antbin"

target="d:\plugins\org.eclipse.jdt.core"

encoding="UTF-8"

allownonapiproject="true"

debug="true"

/>

This will create a file called .api\_description inside the folder d:\plugins\org.eclipse.jdt.core based on the source contained in d:\eclipse\org.eclipse.jdt.core and the .class files contained in d:\eclipse\org.eclipse.jdt.core\bin.

Related references

Analysis Ant Task

API Freeze Ant Task

API Use Ant Task

Analysis Report Conversion Ant Task

API Freeze Report Conversion Ant Task

API Deprecation Ant Task

API Deprecation Report Conversion Ant Task

API Use Report Conversion Ant Task

1.1.13. Compare Ant Task

Purpose

This task runs a complete comparison of an API profile relative to a baseline.

Usage

Description

The name of the Ant task is: apitooling.compare. To be used, the jar file apitooling-ant.jar has to be on the Ant classpath.

<apitooling.compare

baseline="..."

profile="..."

report="..."

excludelist="..."

includelist="..."

components="..."

visibility="..."

debug="..."

/>

Parameters

Attribute Description Required

baseline This attribute specifies the location of the reference baseline.

It can be a .zip, .jar, .tgz, .tar.gz file, or a directory that corresponds to the Eclipse installation folder. This is the directory is which you can find the Eclipse executable.

The location is specified using an absolute path. Yes

profile This attribute specifies the location of the current product or profile that you want to compare against the reference baseline.

It can be a .zip, .jar, .tgz, .tar.gz file, or a directory that corresponds to the Eclipse installation folder. This is the directory is which you can find the Eclipse executable.

The location is specified using an absolute path. Yes

report Set the output location where the report will be generated.

Once the task is completed, the report called "compare.xml" is available in this directory.

The location is specified using an absolute path. Yes

excludelist Set the exclude list location.

The exclude list location specifies a text file listing bundles to be excluded from the analysis. By default all bundles in the profile are analyzed. The include and exclude list attributes can be used to selectively include and exclude bundles. The include list seeds the set of bundles to be analyzed (when omitted, all bundles are included), and the exclude list is then applied. Each line of the file specifies one of:

a specific bundle name

a regular expression to match against bundle names (lines being with "R:")

a comment (lines being with '#')

The location is specified using an absolute path.

Following is an example exclude list:

# DOC BUNDLES

org.eclipse.jdt.doc.isv

org.eclipse.jdt.doc.user

org.eclipse.pde.doc.user

org.eclipse.platform.doc.isv

org.eclipse.platform.doc.user

# NON-ECLIPSE BUNDLES

com.ibm.icu

com.jcraft.jsch

javax.servlet

javax.servlet.jsp

# Exclude all pde bundles

R:org\.eclipse\.pde\..\*

...

The exclude list can contain regular expressions if the line starts with "R:". No

includelist Set the include list location.

The include list location specifies a text file listing bundles to be included in the analysis. By default all bundles in the profile are analyzed. The include and exclude list attributes can be used to selectively include and exclude bundles. The include list seeds the set of bundles to be analyzed (when omitted, all bundles are included), and the exclude list is then applied. Each line of the file specifies one of:

a specific bundle name

a regular expression to match against bundle names (lines being with "R:")

a comment (lines being with '#')

The location is specified using an absolute path.

Following is an example include list:

# DOC BUNDLES

org.eclipse.jdt.doc.isv

org.eclipse.jdt.doc.user

org.eclipse.pde.doc.user

org.eclipse.platform.doc.isv

org.eclipse.platform.doc.user

# NON-ECLIPSE BUNDLES

com.ibm.icu

com.jcraft.jsch

javax.servlet

javax.servlet.jsp

# Exclude all pde bundles

R:org\.eclipse\.pde\..\*

...

No

debug Set the debug value.

The possible values are: true, false

Default is false. No

components Set the components' list location.

The components' list contains all the components' names that must be compared. If omitted, all components from the given baseline are compared.

The location is specified using an absolute path.

The format of the components' list file looks like this:

# DOC BUNDLES

org.eclipse.jdt.doc.isv

org.eclipse.jdt.doc.user

org.eclipse.pde.doc.user

org.eclipse.platform.doc.isv

org.eclipse.platform.doc.user

# Include all pde bundles

R:org\.eclipse\.pde\..\*

...

The components' list can contain regular expressions if the line starts with "R:". No

visibility Set the visibility value.

The possible values are: API, ALL

Default is API. No

Examples

<apitooling.compare

baseline="D:\eclipse\3.4.1\eclipse"

profile="D:\eclipse-SDK-I20081118-0800-linux-gtk.tar.gz"

report="D:\reports\xml"

excludelist="D:\exclude\_list\_external.txt"

debug="true"

/>

This will run the task creating compare.xml files inside the folder D:\reports\xml. It will use the exclude list to reduce the number of problems to report.

Since debug is enabled, some debug tracing will show up in the Ant console.

Related references

API Freeze Ant Task

File Generation Ant Task

API Use Ant Task

Analysis Report Conversion Ant Task

API Freeze Report Conversion Ant Task

API Use Report Conversion Ant Task

API Deprecation Report Conversion Ant Task

1.1.14. API Use Scan Problems Ant Task

Purpose

This task resolves the API Use Scans in a profile relative to a baseline. It reports the references to the types, methods and fields which are now missing from the profile.

Usage

Description

The name of the Ant task is: apitooling.apiusescanproblems. To be used, the jar file apitooling-ant.jar has to be on the Ant classpath.

<apitooling.apiusescanproblems

profile="..."

report="..."

apiusescans="..."

excludelist="..."

includelist="..."

preferences="..."

debug="..."

/>

Parameters

Attribute Description Required

profile This attribute specifies the location of the current product or profile that you want to resolved the API Use Scans against.

It can be a .zip, .jar, .tgz, .tar.gz file, or a directory that corresponds to the Eclipse installation folder. This is the directory is which you can find the Eclipse executable.

The location is specified using an absolute path. Yes

report Set the output location where the reports will be generated.

Once the task is completed, reports are available in this directory. A sub-folder is created for each component that has problems to be reported. Each sub-folder contains a file called "report.xml".

The location is specified using an absolute path. Yes

apiusescans This attribute specifies the comma-separated list of the locations of the API Use Scans that you want to check against the profile.

It can be a .zip file or a directory that corresponds to the API Use Scan report. This is the directory is which you can find the XML folder in the Use Scan. Yes

excludelist Set the exclude list location.

The exclude list location specifies a text file listing bundles to be excluded from the analysis. By default all bundles in the profile are analyzed. The include and exclude list attributes can be used to selectively include and exclude bundles. The include list seeds the set of bundles to be analyzed (when omitted, all bundles are included), and the exclude list is then applied. Each line of the file specifies one of:

a specific bundle name

a regular expression to match against bundle names (lines being with "R:")

a comment (lines being with '#')

The location is specified using an absolute path.

Following is an example exclude list:

# DOC BUNDLES

org.eclipse.jdt.doc.isv

org.eclipse.jdt.doc.user

org.eclipse.pde.doc.user

org.eclipse.platform.doc.isv

org.eclipse.platform.doc.user

# NON-ECLIPSE BUNDLES

com.ibm.icu

com.jcraft.jsch

R:javax\..\*

...

No

includelist Set the include list location.

The include list location specifies a text file listing bundles to be included in the analysis. By default all bundles in the profile are analyzed. The include and exclude list attributes can be used to selectively include and exclude bundles. The include list seeds the set of bundles to be analyzed (when omitted, all bundles are included), and the exclude list is then applied. Each line of the file specifies one of:

a specific bundle name

a regular expression to match against bundle names (lines being with "R:")

a comment (lines being with '#')

The location is specified using an absolute path.

Following is an example include list:

# DOC BUNDLES

org.eclipse.jdt.doc.isv

org.eclipse.jdt.doc.user

org.eclipse.pde.doc.user

R:org.eclipse.platform.doc.\*

# NON-ECLIPSE BUNDLES

com.ibm.icu

com.jcraft.jsch

R:javax\..\*

...

No

preferences Set the preferences for the task.

The preferences are used to configure problem severities. Problem severities have three possible values: Ignore, Warning, or Error. The set of problems detected is defined by corresponding problem preference keys in API tools.

The location is specified using an absolute path. If the given location doesn't exist, the preferences won't be set.

Lines starting with '#' are ignored. The format of the preferences file looks like this:

#Thu Nov 20 17:35:06 EST 2008

API\_USE\_SCAN\_TYPE\_SEVERITY=Error

API\_USE\_SCAN\_METHOD\_SEVERITY=Error

API\_USE\_SCAN\_FIELD\_SEVERITY=Error

The keys can be found in org.eclipse.pde.api.tools.internal.provisional.problems.IApiProblemTypes. No

debug Set the debug value.

The possible values are: true, false

Default is false. No

Examples

<apitooling.apiusescanproblems

profile="D:\eclipse\3.7\eclipse"

report="D:\reports\xml"

apiusescans="D:\APIUseScan\ProductOne,D:\APIUseScan\ProductTwo"

excludelist="D:\exclude\_list\_external.txt"

preferences="D:\tests\_api\org.eclipse.pde.api.tools.prefs"

debug="true"

/>

This will run the task creating report.xml files inside the folder D:\reports\xml. The types, methods and fields from the use scans will be resolved in the profile at D:\eclipse\3.7\eclipse and the missing ones will be reported. It will use the exclude list from the file D:\exclude\_list\_external.txt to reduce the number of bundles for which problems to report.

Problem severities will be generated as specified by D:\tests\_api\org.eclipse.pde.api.tools.prefs.

Since debug is enabled, some debug tracing will show up in the Ant console.

Related references

API Analysis Ant Task

API Freeze Ant Task

File Generation Ant Task

API Use Ant Task

Analysis Report Conversion Ant Task

API Freeze Report Conversion Ant Task

API Use Report Conversion Ant Task

API Deprecation Ant Task

API Deprecation Report Conversion Ant Task

1.1.15. API Use Scan Problems Report Conversion Ant Task

Purpose

This task converts XML reports created by the apitooling.apiusescanproblems Ant task into HTML files.

Usage

Description

The name of the Ant task is: apitooling.apiusescanproblem\_reportconversion. To be used, the jar file apitooling-ant.jar has to be on the Ant classpath.

<apitooling.apiusescanproblem\_reportconversion

xmlfiles="..."

htmlfiles="..."

debug="..."

/>

Parameters

Attribute Description Required

xmlfiles Set the location where the xml reports are retrieved.

The location is set using an absolute path. Yes

htmlfiles Set the location where the html reports are generated.

If not set, the html files are created in the same folder as the xml files.

The location is set using an absolute path. No

debug Set the debug value.

The possible values are: true, false

Default is false. No

Examples

<apitooling.analysis\_reportconversion

xmlfiles="D:\reports\xml"

htmlfiles="D:\reports\html"

debug="true"

/>

This will create an index.html inside the folder D:\reports\html. A report for each bundle will be created inside a sub-folder of D:\reports\html. Each sub-folder will contain a report.html.

Since debug is enabled, some debug tracing will show up in the Ant console.

Related references

API Use Scan Problems Ant Task

Analysis Ant Task

File Generation Ant Task

API Use Ant Task

API Freeze Ant Task

API Freeze Report Conversion Ant Task

API Deprecation Ant Task

API Deprecation Report Conversion Ant Task

API Use Report Conversion Ant Task

1.2.1. API Analysis Command Line Application

Purpose

This application runs a complete API analysis of a Plugin project relative to a baseline - including API use, binary compatibility, and bundle version number validation. The profile is compared to an API baseline for binary compatibility (usually the previous release of a product).

Usage

Description

The name of the application task is: org.eclipse.pde.api.tools.apiAnalyzer. To be used, the bundle file org.eclipse.pde.api.tools in version superior or equals to 1.1.800 has to be installed in the Eclipse Platform instance. Once the Platform is properly configured, the application can be started with commands like:

eclipse -application org.eclipse.pde.api.tools -project /path/to/project -depednencies /path/to/list/of/dependencies.txt -baseline default -failOnError

Application Arguments

Attribute Description Required

project This attribute specifies the location of the project to analyze. The project must be the a valid Eclipse Plugin project, that is a project with typical .project, MANIFEST.MF... files. Yes

dependencyList This attribute specifies a path to a file containing a list of depenencies that will be used as default target platform.

The file must list the absolute path of dependencies (as jar files), either separated by new lines or colon (:). Tokens that are not absolute path to jar files are ignored. Yes

baseline This attribute specifies the location of the reference baseline.

It can be the absolute path to a directory or a to a .target file, or default. If default, the current platform running the API Tools will be used as baseline. No. If omitted, similar to default

failOnError Report API compliance errors as a failure (non-0 exit code) No

Examples

In a Tycho-enabled pom.xml file:

<project>

[...]

<build>

[...]

<plugins>

[...]

<plugin>

<artifactId>maven-dependency-plugin</artifactId>

<executions>

<execution>

<id>list-dependencies</id>

<goals>

<goal>list</goal>

</goals>

<phase>verify</phase>

<configuration>

<outputAbsoluteArtifactFilename>true</outputAbsoluteArtifactFilename>

<outputScope>false</outputScope>

<outputFile>${project.build.directory}/dependencies.txt</outputFile>

</configuration>

</execution>

</executions>

</plugin>

<plugin>

<groupId>org.eclipse.tycho.extras</groupId>

<artifactId>tycho-eclipserun-plugin</artifactId>

<version>1.4.0</version>

<executions>

<execution>

<id>api-analysis</id>

<goals>

<goal>eclipse-run</goal>

</goals>

<phase>verify</phase>

<configuration>

<applicationsArgs>

<!-- need to set workspace to a dir that's not a child of the project -->

<arg>-data</arg>

<args>${project.basedir}/../target/${project.artifactId}-apiAnalyzer-workspace</args>

<args>-application</args>

<args>org.eclipse.pde.api.tools.apiAnalysis</args>

<args>-project</args>

<args>${project.basedir}</args>

<args>-baseline</args>

<args>default</args>

<args>-dependencyList</args>

<args>${project.build.directory}/dependencies.txt</args>

<args>-failOnError</args>

</applicationsArgs>

<repositories>

<repository>

<id>eclipse-4.12</id>

<layout>p2</layout>

<url>https://download.eclipse.org/eclipse/updates/4.12-I-builds/</url>

</repository>

</repositories>

<dependencies>

<!-- This will constitute the default baseline -->

<dependency>

<artifactId>org.eclipse.sdk.ide</artifactId>

<type>p2-installable-unit</type>

</dependency>

</dependencies>

</configuration>

</execution>

</executions>

</plugin>

</plugins>

</build>

</project>

This will run API analysis and fail the build in case an error is found.

Related references

API Analysis Ant Task

1.3. API Evolution

API tooling is about controlling API Evolution and Version Numbering.

Read Achieving API Binary Compatibility for understanding the various incompatibilities.

The following API incompatibilities have been explained in more detail below :

Field Addition to Class

Field Addition to Interface

Default Method Addition to Interface

Field Addition to Class

Adding an API field to an API class that is extended by Clients is dangerous in two respects:

It may break API contract compatibility similar to Evolving\_Java-based\_APIs#Example\_4\_-\_Adding\_an\_API\_method in case of name clashes.

It may cause linkage errors in case an instance (respectively static) field hides a static (respectively instance) field, see JLS8 13.4.8.

Example - Adding a Field to Class

public class Base {String h = "Base";}

public class Derived extends Base { }

public class Test {

public static void main(String[] args) {

String s = new Derived().h;

System.out.println(s);

}

}

This program produces the output:

Base

Suppose that a public static field h is added to Derived:

public class Derived extends Base { public static String h = "humongous"; }

If Derived is recompiled but not Test, then running the new binary with the existing binary for Test will cause an IncompatibleClassChangeError.

Furthermore, if char type h is added to Derived instead, there would be compilation error.

This can not happen if all field declarations follow the usual naming conventions from JLS8 6.1 and classes only have API fields that are either

"static final" (and hence have a name that doesn't contain any lowercase letters), or

non-static and non-final (and hence have a name that contains a least one lowercase letter)

Apart from the binary compatibility issues, it is generally good software engineering practice that API classes should not expose any non-constant fields.

Field Addition to Interface

Adding an API field to an API interface that is implemented by Clients is dangerous in two respects:

It may break API contract compatibility similar to Evolving\_Java-based\_APIs#Example\_4\_-\_Adding\_an\_API\_method in case of name clashes.

It may cause linkage errors in case an instance (respectively static) field hides a static (respectively instance) field, see JLS8 13.4.8. This can not happen if all field declarations follow the usual naming conventions from JLS8 6.1 and classes only have API fields that are either

"static final" (and hence have a name that doesn't contain any lowercase letters), or

non-static and non-final (and hence have a name that contains a least one lowercase letter)

Default Method Addition to Interface

Adding a default method will break an existing client type if it already implements another interface that declares a default method with a matching signature, and the client type already refers to the default method from the other interface (except when using the Interface.super.method() notation). The added default method will cause an IncompatibleClassChangeError at run time, see JLS8 13.5.6. Furthermore, re-compiling the type will result in a compile error.

Example - Adding A Default Method

API

public interface interfaceAPI {}

Client Code

public interface clientInterface {

default void doSomething() {

System.out.println("Do something client...") ;

}

}

public class ClientCode implements clientInterface, interfaceAPI {

public static void main(String[] args) {

new ClientCode().doSomething();

}

}

This program produces the output:

Do something client...

Suppose that a default method is added to interfaceAPI:

public interface interfaceAPI {

default void doSomething() {

System.out.println("Do something API...");

}

}

If interfaceAPI is recompiled but not ClientCode, then running the new binary with the existing binary for ClientCode will cause an IncompatibleClassChangeError.

1.4. Defining API Restrictions

API Tools provides Javadoc tags to explicitly document and restrict the use of API. The following tables summarizes the Javadoc tags supported by each member and the semantics of each tag.

A client refers to a plug-in or bundle that requires the bundle where the associated API is defined. Restrictions are not applied in the same bundle where API is defined. For example, a bundle that defines an interface as @noimplement is also allowed to provide an implementation of that interface.

Supported Restriction Tags

Class Interface Enum Annotation Method Constructor

Final Field

Non-Final Field

@noimplement

-

Supported

-

-

-

-

-

-

@noextend

Supported

Supported

-

-

-

-

-

-

@noinstantiate

Supported

-

-

-

-

-

-

-

@nooverride

-

-

-

-

Supported

-

-

-

@noreference

Supported

Supported

Supported

Supported

Supported

Supported

-

Supported

Restriction Semantics

@noimplement

Indicates that clients must not implement this interface. Any class using the implements keyword for the associated interface or parent of the associated interface where there is no implementing superclass will be flagged with problem.

@noextend

Indicates that clients must not extend this class or interface. Any class or interface using the extends keyword for the associated type will be flagged with a problem.

@noinstantiate

Indicates that clients must not instantiate this class. Any code that instantiates the associated class with any constructor will be flagged with a problem.

@nooverride

Indicates that clients must not redeclare this method. Any subclass that defines a method that overrides the associated method will be flagged with a problem.

@noreference

Indicates that clients must not reference this type (class, interface, enum, or annotation), method, constructor, or non-final field. Any code that directly invokes the associated method or constructor or references the associated non-final field will be flagged with a problem.

When the tag is used on a type, it behaves as though the tag was added to any of the types members. For example, adding the tag to class will flag any references to any methods or non-final fields from that class.

Related tasks

Setting up a baseline

Related references

Javadoc @since tag Management

API Baselines Preferences

API Errors and Warnings Preferences

1.5. Javadoc @since Tag Management

API Tools provides support for management of @since Javadoc tags on new elements that have been added to API (types, methods, fields, etc). New API elements could be a new type added to an API package, a new type added to an API type, a new method added to an API type or a new field added to an API type. Method addition is a special case, where a method addition could be adding a method to a type, overriding a parent class method, implementing a parent interface method or changing an existing methods' signature.

Note: @since tag information is not propagated via implementation or sub-classing. Each element that is added to API is expected to have its' own @since tag and version information.

The tooling provides the following validation for @since tags:

Missing @since tags

Malformed @since tags

Invalid @since tag versions

The preferences for @since tag management can be changed on the Opens the Console preference page Plug-in Development > API Errors/Warnings preference page.

Missing @since tags

Every new API element detected will be checked by the tooling to ensure it has an @since tag. If the new element does not have an @since tag, it will be flagged as needing one, and a version for the tag will be proposed. The proposed version for the new tag will be the current version of the bundle - except in the case of a breaking API change where the bundle version also needs to be updated. If the bundle version also needs to be updated, the proposed version for the missing @since tag will be that of the proposed bundle version.

Consider the following example: we have a bundle A with version 1.0.0 and we have a class C that was added in version 1.1.0 of A which contains method m1().

If we add a new method to C, say m2(), the tooling will report a missing @since tag problem on m2() and suggest that a new tag of @since 1.1 be placed on m2().

If we add a new method to C, say m2() again and we change m1() - introduce a breaking API change - the tooling will report a missing @since tag problem on m2() and suggest that a new tag of @since 2.0 be place on m2(). Where 2.0.0 is the new proposed bundle version.

Malformed @since tags

New elements can have their @since tags checked for consistency to ensure they are properly formulated. API tools checks that all @since tags follow the general format of:

[@since] [pre-amble] [2 part version] [post-amble]

Consider the following @since tag examples:

@since A 1 added m2(): would be flagged as malformed because the version is missing the second segment

@since: would flagged as malformed since there is no version information

@since A: would be flagged since there is no version information

@since 1.0.0: would be flagged since the version has too many segments

@since A 1.0 added m2(): would not be flagged

Invalid @since tag versions

New elements can also have their @since tags checked for validity. An @since tag is considered to be valid if the version information in the tag matches the version of the bundle.

Consider the following example where we have added a new method m2() to an API class in bundle A whose version is 1.0.0:

@since A 1.0 added method m2(): is considered valid

@since A 2.0 added method m2(): is invalid since the version of bundle A is 1.0.0

@since A 0.1 added method m2(): is invalid since the version of bundle A is 1.0.0

Related tasks

Setting up a baseline

Related references

API Baselines Preferences

API Errors and Warnings Preferences

2.1. Builder Configuration Properties

The overall build mechanism is driven by a builder configuration which includes a build.properties file.

Basic Settings

topLevelElementId The id of the feature being built.

This property is new in 3.4 and depends on the use of the 3.4 allElements.xml file. Build configurations using older style customTargets.xml and/or allElements.xml files should not use this property.

buildDirectory

The directory where the build will take place.

configs

A list of configurations to build. A configuration is an ampersand separated list of comma separated operating system, windowing system and architecture. If no config is specified, the platform independent config will be built. Example:

configs = win32, win32, x86 & linux, gtk, ppc

p2 Integration

See the p2 Integration documentation for properties controlling feature builds, and product builds.

Base Control: The base is a target eclipse install containing plug-ins and features that you require but are not building yourself.

baseLocation

The location of the target eclipse install. In the default template, the value of this is ${base}/eclipse. (See the base property below). This directory is expected to contain plugins/ and features/ subdirectories which contain the binary plug-ins and features respectively.

baseos

The operating system of the eclipse specified by baseLocation

basews

The windowing system of the eclipse specified by baseLocation

basearch

The architecture of the eclipse specified by baseLocation

pluginPath

A platform path-separator separated list of locations to find additional plug-ins and features.

skipBase

If set, automatically download a base eclipse to use as a target

base

The location the base eclipse will be downloaded to. This should be the parent of baseLocation, such that baseLocation = ${base}/eclipse.

eclipseURL

The URL for the eclipse download site

eclipseBuildId

The build ID of the eclipse to download

eclipseBaseURL

The actual URL of the zip to download. In the default template, the value is:

eclipseBaseURL = ${eclipseURL}/eclipse-platform-${eclipseBuildId}-win32.zip

filteredDependencyCheck

Constraint the set of plug-ins and features to be considered during the build to the one reachable from the features / plugins being built when set to true

resolution.devMode

When set to true, the validation done on the set of plug-ins being built will be less strict. For example it will allow multiple versions of a singleton bundle to be resolved.

Output Control: The following properties affect the shape of the build results.

runPackager

Set if the packager needs to be run. Set this if your build results need to contain binary features and plug-ins that come from the baseLocation.

product

The product configuration file. Set this if you are building a product. It controls branding of your build results.

archiveNamePrefix

The prefix of the name of the final archive files. This is used in allElements.xml when setting the archive name on a per config basis. By default, the value will be ${buildId}

archivePrefix

The name of the archive root folder, this is "eclipse" for eclipse builds.

buildType Type of build, normally something like I, N, M, etc.

buildLabel The label for the build. In the default template, the value is ${buildType}.${buildId}

collectingFolder

The folder where the build output will be collected. In the default template, the value is ${archivePrefix}.

groupConfigurations

Whether or not to group all configuration into one archive file.

archivesFormat

The archive format for the different configs being built. This is an ampersand separated list of configs and their archive format. The format is separated by a dash (-). Example:

archivesFormat = win32, win32, x86 - antzip & linux, gtk, ppc - tar

outputUpdateJars

When set, generates plug-ins and features in JAR format for an update site.

zipArgs

Arguments to be sent to the zip executable when the archive format is zip.

tarArgs

Arguments to be sent to the tar executable when the archive format is tar.

Signing and Jnlp control:

generateJnlp

Set to true to generate JNLP manifests for the JAR archives

jnlp.codebase

The site URL. This should be a URL that will be used as the root of all relative URLs in the output.

jnlp.j2se

The J2SE version required to run the JNLP app being exported

jnlp.locale

The locale in which the generated jnlp files should be generated.

jnlp.generateOfflineAllowed

Generate the <offlineAllowed/> in the generated files when set to true.

jnlp.configs

Filter the content of the generated jnlp files based on the configuration being built. The configurations need to be listed.

signJars

Set to true to sign jars for features and plug-ins.

sign.alias

The alias passed to the ant signJar task.

sign.keystore

The keystore passed to the ant signJar task.

sign.storepass

The storepass passed to the ant signJar task.

sign.keypass

The keypass passed to the ant signJar task.

Version Control. (See also Version Qualifiers)

forceContextQualifier

The value of this property is used when replacing qualifiers in plug-in and feature versions. If set it is used instead of the tags from the map files or the timestamp.

generateFeatureVersionSuffix

Set to true to generate feature version suffixes

generateVersionsLists

Set to true to generate properties files listing the final version numbers of features and plug-ins being built. This will create finalFeaturesVersions.properties and finalPluginsVersions.properties files in the builder directory.

Fetch phase Control (See also Fetching from Repositories)

skipMaps

If set, do not attempt to fetch map files from CVS

mapsRepo

The CVS repository to fetch map files from. Example:

:pserver:anonymous@example.com/path/to/repo

mapsRoot

The path in the CVS repository to the directory containing the map files

mapsCheckoutTag

The CVS tag to use to checkout the map files

tagMaps

Set to true to tag the map files after fetching them.

mapsTagTag

The tag with which to tag the map files after fetching them.

skipFetch

Set to true to skip the entire fetch phase.

fetchTag

The tag to use when fetching features and plug-ins from CVS. This property overrides the values in the map files.

Compilation Control (See also Controlling Compilation Environment)

logExtension

The extension of the generated log file. This only applies when using the default JDT compiler. Default value is ".log".

javacDebugInfo

Whether or not to include debug info in the output jars. This is passed to the ant javac task. If not set, default value is on.

javacFailOnError

Whether or not to fail the build if there are compilation errors. This is passed to the ant javac task. If not set, default value is true.

javacVerbose

Whether or not to enable the verbose mode of the compiler. This is passed to the ant javac task. If not set, default value is false.

compilerArg

Set specific arguments to pass directly to the compiler.

javacSource

The java source compatibility level to use if a plug-in does not specify any settings. If not set, 1.3 is the default value.

javacTarget

The target .class compatibility level to use if a plug-in does not specify any settings. If not set, 1.2 is the default value.

flattenDependencies

Sort bundle dependencies across feature boundaries. (See also Compilation Order and Parallelization)

parallelCompilation

Set to true to enable parallel compilation. (Requires flattenDependencies=true)

parallelThreadCount

The maximum number of threads to use. Corresponds to threadCount on the ant parallel task.

parallelThreadsPerProcessor

The maximum number of threads to use per available processor. Corresponds to threadsPerProcessor on the ant parallel task.

bootclasspath

The bootclasspath to use if the plug-in does not specify any settings itself. This is a platform path separator separated list of jars. If not set, the file set matching ${java.home}/jre/lib/\*.jar will be used.

<Execution Environment>

Set properties for each JRE that can be used to build. These properties will match up against Execution Environments specified by individual bundles. The values should specify the jars to compile against when using that environment. See Controlling Compilation Environment.

Other properties

generateAPIDescription

Enable automatic generation of API description files for bundles.

generateSourceReferences Enable generation of Eclipse-SourceReferences headers into bundle manifests.

2.2. Feature and Plug-in Build Configuration Properties

The build mechanism is driven by a build configuration. The build configuration for an individual plug-in, fragment, or feature is found in a build.properties file for the corresponding element.

PDE project creation wizards generate the build.properties file when plug-in projects are created. The file contains information on how to compile source folders into JARs. This information can be indirectly updated in the Runtime page of the manifest editor. It can also be directly modified using the appropriate editor.

PDE provides a simple editor for the build.properties that has form and source views. The file itself follows the Java properties format. You need to provide a number of keys and their corresponding values. Multiple values are separated using a comma as the delimiter.

Common properties

bin.includes - lists files that will included in the binary version of the plug-in being built;

bin.excludes - lists files to exclude from the binary build;

qualifier - when the element version number ends with .qualifier this indicates by which value ".qualifier" must be replaced. The value of the property can either be context, <value> or none. Context will generate a date according to the system date, or use the CVS tags when the built is automated. Value is an actual value. None will remove ".qualifier". If the property is omitted, context is used. (See Version Qualifiers.)

custom=true - indicates that the build script is hand-crafted as opposed to automatically generated. Therefore no other value is consulted.

customBuildCallbacks - indicates an xml file containing custom targets to use when using this feature/plug-in. (See Feature and Plug-in Custom Build Steps)

customBuildCallbacks.failonerror - sets the subant task's failonerror parameter. The default is "false".

customBuildCallbacks.buildpath - sets the subant task's buildpath parameter. Use when the location of the custom callbacks file is not relative to the root of the plug-in.

customBuildCallbacks.inheritall - sets the subant task's inheritall parameter. Ant's default is false.

Plug-in specific properties

source.<library> - lists source folders that will be compiled (e.g. source.xyz.jar=src/, src-ant/). If the library is specified in your plug-in.xml or manifest.mf, the value should match it;

output.<library> - lists the output folder receiving the result of the compilation;

exclude.<library> - lists the files that should not be copied into the library by the compiler;

extra.<library> - extra classpaths used to perform automated build. Classpath can either be relative paths, or platform urls referring to plug-ins and fragments of your development environment (e.g. ../someplugin/xyz.jar, platform:/plugins/org.apache.ant/ant.jar). Platform urls are recommended over relative paths;

manifest.<library> - indicate the file that will be used as a manifest for the library. The file must be located in one of the source folder being used as input of the jar.

src.includes - lists files to include in the source build;

src.excludes - lists files to exclude from the source build;

jars.extra.classpath - (deprecated) same effect than extra.<library> except that the entries are applied to all libraries;

jars.compile.order - defines the order in which jars should be compiled (in case there are multiple libraries).

Compiler related options. See also the Controlling the Compilation Environment and Compiler Options pages.

jre.compilation.profile - set the Environment used to compile this bundle.

javacSource - set the source compatibility level for compiling this bundle. Overrides jre.compilation.profile.

javacTarget - set the target .class compatibility level for compiling this bundle. Overrides jre.compilation.profile.

bootClasspath - set the boot classpath to compile this bundle against, Overrides jre.compilation.profile.

javacWarnings.<library> - set the warning options to pass to the compiler. (See the -warn option in Using the batch compiler from the Java development user guide).

javacErrors.<library> - set the error options to pass to the compiler. (See the -err option in Using the batch compiler from the Java development user guide).

javacDefaultEncoding.<library> - sets default encoding to use when compiling the given library.

javacCustomEncodings.<library> - a comma separated list of paths and encodings to use when compiling the given library. Example:

javacCustomEncodings.library.jar = src/org/foo[US-ASCII], src/org/foo/bar.java[ISO-8859-1]

javacProjectSettings - A relative path to a preference file specifying properties for the JDT Compiler. Set to true to use the default ".settings/org.eclipse.jdt.core.prefs".

compilerArg - specify custom compiler arguments

compilerAdapter - specify a custom compiler adapter to use instead of the default org.eclipse.jdt.core.JDTCompilerAdapter

compilerAdapter.useLog - true/false: whether or not the custom compiler adapter should receive the -log argument.

compilerAdapter.useArgFile - true/false: whether or not the custom compiler adapter should receive the generated javaCompiler.<library>.args argument file.

sourceFileExtensions - a list of extensions (\*.java, \*.aj) that should be considered as source when using a custom compiler adapter.

The values defined for these keys ending with "includes" or "excludes" are expressed as Ant "patterns". Standard formats give the expected results. For example, "\*.jar" indicates all jar files in the top level directory. The patterns are not deep by default. If you want to describe all Java files for example, you should use the pattern "\*\*/\*.java". The pattern "\*\*" matches any number of directory levels. Similarly, to describe whole sub-trees, use "xyz/".

Feature specific properties

root - List the files and folders that must be included in the root of the product. See the Rootfiles help page for more detail, incuding the following additional properties:

root.<config>: per configuration rootfiles.

root.folder : rootfiles placed in subfolders.

root.permissions : perform chmods on rootfiles.

root.link : generate softlinks for rootfiles.

root.<config> - list the files and folders that must be included in the root of the product when it is built for the specified configuration. config is composed of the three (3) segments of a configuration separated with a dot;

root.permissions.<permissionValue> - list the files and folders to chmod to the given value. Reference to folders must ends with a '/';

root.permissions.<config>.<permissionValue> - list the files and folders to chmod to the given value for a specific configuration. Reference to folders must ends with '/';

root.link - list by pairs (separated by a comma) the files and folders that need to be symbolicly linked. The first entry indicate the source (target in the unix terminology) and the second entry the link name;

root.link.<config> - a comma separated list of pairs of files and folders that need to be symbolicly linked for a specific configuration. The first entry indicate the source (target in the unix terminology) and the second entry the link name;

generate.feature@<featureId> - indicates that the source feature featureId will be the source feature for the feature indicated as value of this property. The values listed after the first comma indicates elements to be fetched from the repository; (See Source Plugin and Feature Generation)

generate.plugin@<pluginId> - indicates that the source plug-in pluginId will be the source plug-in for the indicated as value of this property.

significantVersionDigits - The number of significant digits used when generating feature version suffixes (See Version Qualifiers)

generatedVersionLength - The length of the generated feature version suffix.

The following example has been extracted from the build.properties of the org.eclipse.platform feature.

bin.includes=epl-v10.html,feature.xml,feature.properties,license.html

root=rootfiles,configuration/

root.permissions.755=eclipse

root.linux.motif.x86=../../plugins/platform-launcher/bin/linux/motif,linux.motif

root.linux.motif.x86.link=libXm.so.2.1,libXm.so.2,libXm.so.2.1,libXm.so

root.linux.motif.x86.permissions.755=\*.so\*

3.1. Plug-in Manifest Editor

PDE provides a single form-based multi-page plug-in manifest editor that manages all plug-in files: MANIFEST.MF, plugin.xml, fragment.xml and build.properties. The plug-in manifest editor is used as though a single file is being edited; while, PDE handles the task of writing the data to the different files on disk.

For details on the individual editor pages, refer to the following documents:

Plug-in Overview

Plug-in Dependencies

Plug-in Runtime

Plug-in Extensions

Plug-in Extension Points

Plug-in Build

Manifest Editor Presentation

By default, the plug-in manifest editor shows information and tabs on extensions, extension points and Equinox-specific MANIFEST.MF headers and attributes. If you do not wish to use these Eclipse-specific features, you can filter out these pages from the editor on the Plug-in Development property page of your plug-in project via Open the main PDE preference page Properties > Plug-in Development from the context menu of plug-in projects.

Editor Presentation

Relevant Links

OSGi Bundle Manifest Headers

Related references

Overview Page

Dependencies Page

Runtime Page

Extensions Page

Extension Points Page

Build Page

3.1.1. Plug-in Overview

The Overview page serves a dual purpose:

It contains two main sections that define important plug-in properties: General Information and Execution Environments.

It functions as a quick reference on how to develop, test and deploy plug-ins by providing the Plug-in Content, Extensions, Testing and Exporting sections. These sections provide hyperlinks that, when clicked, navigate to other pages or invoke commands.

General Information

General Overview

An ID is mandatory and must comply with the plugin.dtd.

A Version is mandatory and must be of the form major.minor.micro.qualifier (e.g. 1.3.0).

A Name is the translatable presentation name of the plug-in. This field is required.

A Vendor is the translatable name of the plug-in vendor. This field is optional.

A Platform filter is a valid LDAP string that must evaluate to true in a running system for the the plug-in to run. For example, the following filter indicates that the plug-in is designed to only run on platforms with a win32 windowing system: Eclipse-PlatformFilter: (ws=win32). If a user attempts to run Eclipse on a platform that does not meet this requirement, the plug-in will be silently ignored by the runtime.

An Activator is a Java class that controls the plug-in's life cycle. It is only needed if you require work to be done on the startup or shutdown of your plug-in.

Execution Environments

A bundle execution environment specifies the minimum level of JRE required for the plug-in to run. If the JRE used to run Eclipse does not meet the requirement, the plug-in will not run.

Execution Environments

Press the Add... button to browse a list of available execution environments and add one that is most suitable for your plug-in. For example, if you declare J2SE-1.4 as your plug-in's bundle execution environment, your plug-in will only run with a JRE version >= 1.4.

If your plug-in can run in an execution environment that is not a proper subset of the execution environment specified (e.g. J2SE-1.4 and CDC-1.1/Foundation-1.1), both execution environments should be specified.

During a plug-in export, the plug-in code is compiled against the JRE associated with the first execution environment listed in the MANIFEST.MF file . Click on the Configure JRE associations... link to open the Open the JRE profiles preference page Windows > Preferences... > Java > Installed JREs > Execution Environments preference page for a list of OSGi execution environments and their corresponding list of compatible JREs installed on the system.

Whenever you update the list of execution environments for your plug-in, you should Update the classpath settings via the link to ensure you are compiling against the right JRE and have the settings that best match your first execution environment.

Related references

Plug-in Editor

Dependencies Page

Runtime Page

Extensions Page

Extension Points Page

Build Page

3.1.2. Plug-in Dependencies

The Dependencies page shows the dependencies that your plug-in has on other plug-ins. On this page, you must list all plug-ins that contribute code to your plug-in project and that are required to be on your project's classpath in order to compile. When you modify the list of dependencies and save the file, PDE will automatically update your classpath.

Required Plug-ins

The Required Plug-ins section specifies your plug-in project's dependencies on other plug-ins by explicitly enumerating these plug-ins. Your plug-in will not resolve or run at runtime if any of your plug-in dependencies are missing or unresolved.

Require-Bundle

Press the Add... button to browse the list of all plug-ins that your plug-in may make a dependency on.

The order of the plug-ins in the list is important because it dictates the class loading order at runtime; therefore, use the Up and Down buttons to organize the list as appropriate.

You can set several properties on each plug-in dependency by selecting a plug-in and clicking the Properties button:

You can make a plug-in dependency optional by selecting the Optional option. When this option is enabled, it indicates to the runtime that your plug-in should run even if an optional plug-in dependency is missing or not resolved.

You can select the Re-export this dependency option to re-export your dependency in order to make it automatically visible to any downstream plug-in that require it. Please note that you should only re-export a given plug-in dependency only if your plug-in exposes one ore more API's from that dependency.

You can also specify the Minimum Version and Maximum Version of the plug-in dependency.

Imported Packages

The Eclipse runtime provides the flexibility of specifying a dependency on a package without explicitly identifying its originating plug-in. These packages are enumerated in the Imported Packages section.

Import-Package

Press the Add... button to browse the list of packages available to your plug-in for importing. PDE filters out packages contributed by plug-ins that are already on your plug-in classpath.

You can set several properties on each package dependency:

You can make the dependency optional by selecting the Optional option. When this option is enabled, this indicates to the runtime that your plug-in should run even if a package dependency is missing.

You can specify the Minimum Version and Maximum Version of the package you require.

Automated Management of Dependencies

The Automated Management of Dependencies section provides a new flexible workflow. This workflow allows you to code your plug-in first and then have PDE analyze your code and automatically generate the list of plug-in dependencies.

Automated Management

Press the Add... button to add plug-ins to the list. Consequently, this action will augment your development build path and your content assist scope accordingly.

These dependencies do not get added to the MANIFEST.MF file immediately; however, you can start coding right away as if they were.

At any time, you can click the add dependencies hyperlink to have PDE analyze your code and generate the correct dependencies in your MANIFEST.MF file via either the Require-Bundle or Import-Package headers.

Dependency Analysis

The Dependency Analysis section contains several useful features that help you examine and optimize your plug-in dependencies to make your plug-in run better.

Dependency Analysis

Since JARs from all the plug-ins listed as dependencies will be on your plug-in runtime classpath, it is very important not to list dependencies that are not required. Otherwise, class loading performance will be detrimentally affected. If you click the Find unused dependencies hyperlink, PDE will find unused and redundant entries in the list of required plug-ins and imported packages and offer to remove them.

Related references

Plug-in Editor

Overview Page

Runtime Page

Extensions Page

Extension Points Page

Build Page

3.1.3. Plug-in Runtime

The Runtime page shows all the packages that your plug-in makes visible to other plug-ins and the libraries and folders that constitute the plug-in runtime classpath.

Exported Packages

The Exported Packages section lists all packages that your plug-in intends to expose to downstream clients.

Exported Packages

You can set a Version on each exported package when you press Properties.... The version is of the form major.minor.micro (e.g. 1.3.0).

Press Add... to see the list of packages available for export that are not already in your list.

PDE can calculate the uses directive, as defined in the OSGi Specification, for all of your publicly exported packages. Press the Calculate Uses button to begin. Because this process involves code introspection, this might be a long running operation.

You can control the visibility of each package to downstream clients in the Package Visibility section. Refer to the Access Rules document for full details.

When the visiblity of a package is changed the icon beside the package will change to reflect its status as follows:

Public Package Visible to all downstream plug-ins (public)

Internal Package Not visible to any downstream plug-ins (internal)

X-Friend Package Visible to a specific list of downstream plug-ins (internal with friends)

Classpath

The Classpath section lists all the locations the runtime will search when loading classes from this plug-in.

Classpath

Since JAR'd plug-ins are the recommended format for Eclipse 3.1 and up built plug-ins, it is recommended to leave this section blank.

Press New... to add the name of a library that has not been built yet.

Press Add... to browse the current plug-in project and add an existing JAR or folder to the classpath.

The order of the table entries dictate the class loading order; therefore, use the Up and Down buttons to organize your classpath accordingly.

Related references

Plug-in Editor

Access Rules

Overview Page

Dependencies Page

Extensions Page

Extension Points Page

Build Page

3.1.3.1. Access Rules

The Eclipse 3.1 runtime gives the plug-in developer the option to control plug-in code visibility to downstream plug-ins on a per-package basis.

A package may be classified as one of the following:

Accessible

Forbidden

Internal

Internal with friends

PDE translates these runtime visibility rules into compiler access restriction rules at compile time. As a result, a violation of a visibility rule is flagged by the compiler as a warning or an error - depending on the severity of that violation.

With this type of support at compile time, one is never caught by surprise by runtime class loading errors and is always aware of internal type references.

Accessible Packages

Accessible packages are visible to downstream plug-ins unconditionally. While API packages must clearly fall in this category, it is completely up to the developer to decide what other packages exported by the plug-in ought to be given this level of visibility.

In order to declare a package as accessible, you must list it in the Exported Packages section on the Runtime of the plug-in manifest editor and leave the default visibility setting as-is.

Accessible Packages

Forbidden Packages

You can hide a package from downstream plug-ins at all times by excluding it from the list in the Exported Packages section on the Runtime page of the plug-in manifest editor.

References to types from a forbidden package result in class loading errors at runtime.

To avoid such unpleasant situations:

The compiler will flag references to forbidden packages with an error.

Types from forbidden packages are NOT available as proposals in the content assist.

Notes:

All plug-ins in the Eclipse SDK enumerate all their packages in the Exported Packages section; therefore, none of the packages in the SDK have forbidden access.

The severity level for forbidden references is set on the Open problem severities preference page Java > Compiler > Errors/Warnings > Deprecated and restricted API preference page.

It is strongly recommended that the severity of a forbidden reference is kept at error.

Forbidden preferences

Internal Packages

Internal packages are packages that are not intended for use by downstream plug-ins. These packages are visible to downstream plug-ins by default.

Internal packages are hidden from downstream plug-ins only when Eclipse is launched in strict mode (i.e. when you launch with the -Dosgi.resolverMode=strict VM argument).

Internal packages must be listed in the Exported Packages section on the Runtime page of the plug-in manifest editor with the hidden option selected.

discouraged access

Two measures are taken to discourage downstream plug-ins from referencing internal packages:

The compiler flags references to internal packages with a warning.

discouraged access

Types from discouraged packages are available as content assist proposals; but, with a lower priority.

discouraged content assist

The severity level for discouraged references can be set on the Open the problem severities preference page Java > Compiler > Errors/Warnings > Deprecated and restricted API preference page.

Discouraged preferences

Internal packages with friends

It is important for a plug-in to be able to grant full access to its internal packages to designated "friend" plug-ins. For instance, the PDE code is split across multiple plug-ins, and the org.eclipse.pde.ui plug-in should have full access to org.eclipse.pde.core's internal packages.

In the example below, the the org.eclipse.pde.ui friend plug-in has full access to the org.eclipse.pde.internal.core.bundle package from the org.eclipse.pde.core plug-in.

Friends

The friends are free to reference any type from the org.eclipse.pde.internal.core.bundle package with the compiler's blessing.

On the other hand, if any other plug-in references a type from the org.eclipse.pde.internal.core.bundle package, the compiler flags the reference as a discouraged reference as described in the previous section.

How to enable access restrictions

To take advantage of the PDE access restriction support, the only requirement is that the plug-ins in question contain an OSGi bundle MANIFEST.MF. PDE will take care of the rest including the management of the plug-in classpath.

If the plug-in does not contain a MANIFEST.MF file, that file can be created as follows:

Open the plugin.xml in the plug-in manifest editor.

In the Plug-in Content section of the Overview page, click on the 'create an OSGi bundle manifest' link.

convert to manifest.mf

Inspecting access rules

You can inspect the access restriction rules imposed on each classpath entry by PDE on the Java Build Path property page of your plug-in project.

Java Build Path Properties

Related references

Plug-in Editor

Overview Page

Dependencies Page

Runtime Page

Extensions Page

Extension Points Page

Build Page

3.1.4. Plug-in Extensions

Extensions are the central mechanism for contributing behavior to the platform. Unless your plug-in is a documentation plug-in or a simple Java API library made available to other plug-ins, new behavior is contributed as an extension.

The Extensions page is where you can add, remove and modify the extensions your plug-in contributes to the platform.

All Extensions

In the All Extensions section, press Add... to view all the available extension points that you can extend.

Extension Details

Each extension point comes with an XML schema specifying its grammar. When you select a new extension, PDE extracts the grammar for the corresponding extension point and populates the context menu of each element selected in the All Extensions viewer with a list of valid child elements that you can create.

For each selected element in the body of an extension, PDE populates the Extension Element Details section with all the valid attributes for that element. Required attributes are denoted with an asterisk.

When you hover your mouse over an attribute name, a tool tip box appears showing the description of that attribute.

When an attribute expects the name of a Java class as a value (e.g. the class attribute above), clicking on the name of the attribute will open the Java file specified - if it exists. If the file does not exist, then clicking on the class link will bring up the JDT New Class wizard to create a new Java class on the fly.

Some extension elements do not have attributes. Instead they require a description which can be specified in the Body Text section:

Body Text

Related references

Plug-in Editor

New Extension Wizard

Overview Page

Dependencies Page

Runtime Page

Extension Points Page

Build Page

3.1.4.1. New Extension

The New Extension wizard allows you to add new extensions to your plugin.xml from the Extensions Editor. New extensions can be added by selecting an existing extension point or using a template wizard.

From Extension Point

New Extension Wizard Points Tab

Select an extension point from the list of contributed points. By default Show only extension points from the required plug-ins is turned on so the list will only show points contributed by plug-ins in your project's required bundle list.

From Extension Template

New Extension Wizard Wizards Tab

Select a template from the list and press Next to use a wizard to create a new extension.

Related references

Plug-in Editor

Overview Page

Dependencies Page

Runtime Page

Extensions Page

Extension Points Page

Build Page

3.1.5. Plug-in Extension Points

Extension points define new function points for the platform that other plug-ins can plug into. The Extension Points page is the place to add, remove and edit extension points declared by your plug-in.

Extension Points

An extension point has three attributes:

ID - a required attribute whose value is a simple name

Name - a required attribute whose value is a translatable string

Schema - an optional attribute whose value is a relative path to the schema corresponding to this extension point

Although the Schema attribute is optional, you are strongly encouraged to provide a schema; since, PDE utilizes it to assist other developers in using your extension point.

Related references

Plug-in Editor

Overview Page

Dependencies Page

Runtime Page

Extensions Page

Build Page

3.1.6. Plug-in Build

The Build page contains all the information needed to build, package and export the plug-in. Although it appears as a page in the plug-in manifest editor, changes made to it are written by PDE to the plug-in's build.properties file. The build.properties file solely guides the build process.

Custom Build

The Custom Build option prevents the build.xml file from being overwritten with a generated build file. Use this option when the plug-in provides a custom build.xml that is being manually updated.

Runtime Information

The Runtime Information section lists all the libraries that you want to build. For each library, you must list the source folder(s) that will be compiled into the library.

Runtime

If your plug-in declares more than one library, order them correctly using the Up and Down button. This will ensure that they get compiled in the correct order.

Binary and Source Builds

Binary and Source Builds

The Binary Build section is where you select all the files and folders that will make it into the packaged plug-in.

The Source Build section has a specialized purpose and is not commonly used or needed by the general population. It is only needed when you need to ship source in separate plug-ins and features rather than in binary plug-ins. See the org.eclipse.pde.core.source extension point for details.

Extra Classpath Entries

If you need extra libraries on the build path for your source code to compile, you can add these required JARs in the Extra Classpath Entries section.

Extra Classpath

Please note that this section should be used to add libraries that are not required to be on the plug-in runtime classpath.

Related references

Plug-in Editor

Overview Page

Dependencies Page

Runtime Page

Extensions Page

Extension Points Page

3.2. Feature Manifest Editor

PDE provides a form-based multi-page feature manifest editor that manages and edits the feature.xml file.

For details on the individual editor pages, refer to the following documents:

Feature Overview

Feature Information

Included Plug-ins

Included Features

Feature Dependencies

Feature Installation

3.2.1. Feature Overview

The Overview page serves a dual purpose:

It contains two main sections that define important properties of the feature: General Information and Supported Environments.

It functions as a quick reference on how to define, export and publish a feature by providing the Feature Content, Exporting and Publishing sections. These sections provide hyperlinks that, when clicked, navigate to other pages or invoke commands.

General Information

Feature Overview

An ID is mandatory and must comply with the specifications.

A Version is mandatory and must be of the form major.minor.micro.qualifier (e.g. 1.3.0).

A Name is the translatable presentation name of the feature. This field is required.

A Vendor is the translatable name of the feature vendor. This field is optional.

A Branding Plug-in stores branding information for primary features. If not explicitly set, Eclipse will assume that the branding plug-in has the same identifier as the feature.

The Update Site URL denotes a URL that is used by the installer when searching for feature updates.

The Update Site Name is the human-readable label of the referenced update site.

Supported Environments

By default, your feature is treated as universally portable. You can add constraints by providing supported Operating Systems, Window Systems, Languages and/or System Architectures. This information will be used to ensure that your feature is not installed or shown in the context that does not match these constraints.

Feature Environments

3.2.2. Feature Information

The main purpose of the Feature Information page is to specify information that is required to install the feature.

Feature Description

A Feature Description is a brief component description.

Description

Typically, a description is plain text and intended to be translated. Alternatively, you can provide the description in HTML format and point to it via a URL. The URL can be specified as absolute or relative. If a relative URL is specified, it is assumed to be relative to (and packaged in) the feature archive.

Copyright Notice

The feature Copyright Notice is typically provided as plain text and is intended to be translated.

Copyright

Alternatively, you can provide the copyright notice in HTML format and point to it via a URL. The URL can be specified as absolute or relative. If a relative URL is specified, it is assumed to be relative to (and packaged in) the feature archive.

License Agreement

The License Agreement is a feature click-through license. It is typically simple text and is intended to be translated.

The license is displayed in a standard dialog with Accept and Reject actions during the download and installation process.

License

Note that a click-through license must be specified for any feature that will be selected for installation or update within Eclipse. When using nested features, only the nesting parent (i.e. the feature selected for installation or update) must have click-through license Text defined.

Please note that the license text is required even if the Optional URL attribute is specified.

Sites to Visit

Sites to Visit specify the name and URL of sites to search for new features.

Sites To Visit

In general, a vendor can use Sites to Visit to reference its own site(s), or partner site(s) that offer complementary features.

This data is simply used 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 as Available Software Sites within the install wizard.

3.2.3. Included Plug-ins

The Plug-ins and Fragments page lists all the plug-ins and fragments that should be packaged in the feature and specifies their properties.

Plug-in List

The Plug-ins and Fragments list enumerates all the plug-ins and fragments that are included directly in the feature.

Plug-ins

Press the Add... button to browse the list of available plug-ins to add to the list.

The preferred versioning scheme for referenced plug-ins is to give them a 0.0.0 version. This is a special value that will be replaced by the actual plug-in version when the feature is exported. This option is especially convenient when plug-in versions contain a qualifier segment.

Plug-in Details

Plug-in Details are used as guidelines to determine what, when and how to install the plug-ins included by a feature.

Plug-in Details

The Download Size and Installation Size are optional hints specified to enable useful user feedback about the feature being installed.

The Unpack the plug-in archive after installation option determines whether a plug-in is to be unpacked (i.e. flattened) or left in a JAR after installation.

If a plug-in or fragment contains platform-specific code, then environment conditions should be specified and associated with that plug-in or fragment.

3.2.4. Included Features

The Included Features section references all features nested inside your feature and their associated properties.

Feature List

The Included Features list enumerates all the features that are directly nested in your feature.

Included Features

When your feature is built, all of its included features are recursively built and packaged as well.

Included Feature Details

Included Feature Details

An included feature may be marked as optional using the The feature is optional option. Users may not choose to install optional features. However, if users do choose to install optional features, they may safely disable them later on. Please note that a missing optional feature is not treated as an error.

An included feature can be updated by patches. By default, the installer searches for updates to an included feature at the update site for the parent feature; but, this setting can be overridden.

If an included feature contains platform-specific code or resources, then environment conditions should be specified and associated with that feature.

3.2.5. Feature Dependencies

The Dependencies page lists all Required Features and Plug-ins that must be present in the product before the feature can be installed. If any of these pre-requisites are missing, the feature will not be installed.

Dependencies

The requirement can be based solely on plug-in IDs, or further constrained using expected versions and match rules.

Properties

Version to match: Optional version specification.

Match Rule: Optional matching rule. Valid values and processing are as follows:

If the 'Version to match' attribute is not specified, the 'Match Rule' attribute (if specified) is ignored.

Equivalent - The dependency 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 - The dependency 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).

Perfect - The dependency version must match exactly the specified version.

Greater or Equal - The dependency version must be at least at the version specified, or at a higher service, minor or major level.

3.2.6. Feature Installation

The Installation page contains information that was used by the legacy Eclipse Update Manager prior to Eclipse 3.4. The features on this page should be avoided when using newer versions of Eclipse.

3.3. Site Manifest Editor

PDE provides a form-based multi-page site manifest editor that manages and builds a software site (repository) for p2.

For details on the individual editor sections, refer to the following documents:

Site Map

Site Information

3.3.1. Site Map

The Site Map page of the site manifest editor lists, categorizes and builds the features hosted on the update site.

Managing the Site

Building an update site is a relatively simple task with the

Managing the Site

To publish a feature on the site, you must first add it via the Add Feature... button.

For easier browsing of the site, it is recommended to create categories and organize the features under those categories. A category has a name and a translatable label for display. A feature may appear in >= 0 categories. Categorization enables the installer to present a hierarchical view of features.

Features are built recursively when you press Build All. This means that the features and all the plug-ins and fragments they include will be built in a single batch operation. The feature JARs end up in the features/ folder and the plug-in JARs are placed in the plugins/ folder of the update site project.

If you are dealing with features that include other features, only the root feature needs to be listed, since the child features are automatically built.

To build a single feature, select the feature and press the Build button..

Feature Environments

Features that do not carry such environment constraints are portable and can be installed on all platforms. If a feature has environment constraints that does not match the user's current platform, you can configure them to be filtered out.

For faster filtering, feature environments should be associated with a feature entry in the site.xml.

To copy feature environments from the feature.xml to the site.xml, press the Synchronize... button in the Managing the Site section.

Feature Environment

3.3.2. Site Information

Site Description

A site description may be as simple as a brief text description. Alternately, you may specify a URL pointing to an HTML file. The URL can be specified as absolute of relative. If relative, the URL is assumed to be relative to site.xml.

Description

Site Mirrors

An update site may point to the URL of a file listing update site mirror definitions. This URL can be absolute or relative to this site.

Mirrors

Mirrors File

The updates mirror file (the one specified in the Site Mirrors section above) contains a list of mirror locations. Its 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.

3.4. Extension Point Schema Editor

About Extension Point Schemas

An extension point schema is an XML schema file that defines a contract for extension points that extensions must conform to in order to plug-in and extend the platform. More specifically, an extension point schema specifies a formal grammar for an extension point's elements and attributes. This grammar dictates the structure, content and semantics of any contributed extension's XML mark-up.

Extension point schemas are used during plug-in development to do the following:

Validate extensions defined in workspace plug-ins.

Generate reference extension point documentation.

Assist the user in extension creation.

About This Editor

The Extension Point Schema Editor allows you to do the following:

Create and edit an extension point schema's grammar.

Include and use the grammar defined by other shared extension point schemas.

Document an extension point schema's grammar.

Specify schema metadata used by PDE to automate common tasks.

Preview the extension point schema's reference document.

This editor abstracts the extension point schema's file mark-up away as an implementation detail that you do not have to learn to use.

Editor Pages

This editor is comprised of the following pages:

Schema Overview Page

Schema Definition Page

Global Editor Actions

Click the Preview Reference Document hyperlink to preview the reference document pertaining to the saved or unsaved contents of the edited extension point schema. This action generates an HTML representation of the schema and opens it in a browser.

Schema Header

PDE XML Schema Support

Extension point schemas are XML schemas. As a result, they are written in XML and are able to take advantage of schema annotations to improve ease of processing and readability. However, extension point schemas only use a small subset of XML schema features defined by the W3C XML Schema specification.

Use of a small XML schema feature subset dramatically simplifies the structure and syntax of extension point schemas while still addressing the required needs of the extension point problem domain. It also provides an aesthetic one-to-one mapping of the schema grammar to an equivalent DTD grammar.

The main extension point schema limitations are as follows:

Only global element declarations are allowed.

Only local attribute declarations are allowed.

Only sequence and choice compositors are supported.

Only string and boolean attribute types are allowed.

Only the enumeration restriction is supported for attributes of type string.

Only local element type declarations are allowed.

XML namespaces are not supported.

An element can be declared to specify either element content or character content; but, not mixed content (both element and character content).

Relevant Links

New Extension Point Schema Wizard

What is an extension point schema?

Plug-in Manifest Editor

3.4.1. Schema Overview Page

About This Page

This form page allows you to identify and document an extension point schema. It also allows you to include and use grammars defined by other shared extension point schemas.

Page Sections

This page is comprised of the following sections:

General Information Section

Schema Inclusions Section

Documentation Section

General Information Section

This section uniquely identifies the edited extension point schema. The format of an extension point schema ID is the same as the fully qualified ID of its corresponding extension point: <plugin-id>.<point-id>

General Information Section

General Information Section Fields

Edit the Plug-in ID field to specify the qualifier portion of the unique extension point schema ID. This field's value should match the the ID of the plug-in that the schema's corresponding extension point is defined in.

Edit the Point ID field to specify the ID portion of the unique extension point schema ID. This field's value should match the ID used to define the schema's corresponding extension point.

Edit the Point Name field to specify a human-readable name for the extension point schema. This field's value should match the name used to define the schema's corresponding extension point.

Schema Inclusions Section

This section allows you to include and use the grammar from other shared extension point schemas.

Schema Inclusions Section

Schema Inclusions Section Fields

The viewer field lists the shared extension point schemas whose grammars are included and used by the edited extension point schema.

Shared extension point schemas located in the same plug-in as the edited extension point schema may be included. These schemas are represented in the viewer by a relative path to their file name: <relative-path>/<schema-file-name>

Shared extension point schemas located in the another workspace plug-in may be included. These schemas are represented in the viewer by a special absolute path to their file name: schema://<defining-plugin-id>/<path>/<schema-file-name>

Schema Inclusions Section Actions

Click the Add... button, to open the File Selection dialog in order to include and use the grammar from an existing shared extension point schema in the edited schema. The dialog presents a filtered hierarchical view of all workspace extension point schemas. If an extension point schema contained in a different plug-in than the edited schema is selected, a special absolute path will be constructed and entered into the viewer field. If an extension point schema contained in the same plug-in as the edited schema is selected, a relative path from the edited schema to the selected schema is computed and entered into the viewer field.

Click the Remove button, to remove the selected shared extension point schema's grammar from inclusion and use by the edited extension point schema. This action is available only when a schema is selected in the viewer field.

Double-click an extension point schema in the viewer field to open that schema in a new Extension Point Schema Editor.

Documentation Section

This section allows you to document the edited extension point schema. It's contents will be used to compose the edited extension point schema's reference document. As a result, valid HTML mark-up may be used in any of the section's fields.

This section contains the following tabs:

Description

Since

Examples

API Information

Supplied Implementation

Copyright

Documentation Section

Documentation Section Fields

Edit the Description tab's field to specify a detailed description of the defining extension point for this edited schema. Typically, this field is used to describe the extension point's purpose, utility and associated concepts.

Edit the Since tab's field to specify the version of Eclipse this edited schema was first added to. For example, if this edited schema was first added to Eclipse 3.0, the field's value should be 3.0. The assumption is that the edited schema's grammar is API that persists release to release once introduced and is maintained for backwards compatibility. This field is analogous to Java's Javadoc @since tag.

Edit the Examples tab's field to specify a valid XML snippet of the extension that conforms to this edited schema's grammar. Try to use an example that illustrates a typical use case of the extension.

Edit the API Information tab's field to describe the structure, content and semantics of the extension point's elements and attributes.

Edit the Supplied Implementation tab's field to specify whether the plug-in that defines the extension point for this edited schema also contributes extensions for it. This field points the user to an actual extension implementation that they can use as a template or reference for defining their own extensions.

Edit the Copyright tab's field to outline the exclusive legal rights to reproduce, publish, sell, or distribute the edited schema.

Relevant Links

Extension Point Schema Editor

Plug-in Manifest Editor

New Extension Point Schema Wizard

3.4.2. Schema Definition Page

About This Page

This form page allows you to edit the structure and properties of schema objects that constitute the edited schema's grammar.

The structure of schema objects are edited in the Extension Point Elements section. This section occupies the upper left-hand portion of the editor area.

The properties of schema objects are edited in the <schema-object> Details sections. These sections are activated in the upper right-hand portion of the editor area depending on the type of schema element selected in the Extension Point Elements section.

Page Sections

This page is comprised of the following sections:

Extension Point Elements Section

Description Section

Element Details Section

Compositor Details Section

Element Reference Details Section

Attribute Details Section

Extension Point Elements Section

This section allows you to do the following:

Visualize and edit the structure of schema objects that constitute the edited schema's grammar.

Activate the corresponding <schema-object> Details section used to edit the properties of a selected schema object.

This section is directly accessible through the Schema Definition form page.

Extension Point Elements Section Fields

Use the extension point element viewer field to do the following:

Visualize and edit the structure of the elements, attributes and compositors that constitute the edited schema's grammar.

Activate the corresponding <schema-object> Details section used to edit the properties of a selected element, attribute or compositor.

This viewer contains the following nodes:

Elements:

Element Object - Select the Element node, to activate the Element Details section.

Element Reference Object - Select the Element Reference node, to activate the Element Reference Details section.

Compositors:

Sequence Object - Select the Sequence Compositor node, to activate the Compositor Details section.

Choice Object - Select the Choice Compositor node, to activate the Compositor Details section.

Attributes:

Required Atribute Object - Select the Required Attribute node, to activate the Attribute Details section.

Optional Atribute Object - Select the Optional Attribute node, to activate the Attribute Details section.

Attribute Class Object - Select the Class Attribute node, to activate the Attribute Details section.

Attribute Resource Object - Select the Resource Attribute node, to activate the Attribute Details section.

Example: The extension element specifies that a sequence of one or more shortcut elements are allowed as its legal children. An element reference is used to represent the shortcut element. A Sequence compositor is used to represent a sequence of elements.

Extension Point Elements Section 2

Example: The shortcut element specifies that either a description or category element is allowed as its legal child. Element references are used to represent the description and category elements. A Choice compositor is used to represent a choice of elements.

Extension Point Elements Section 1

Extension Point Elements Section Actions

Click the Add Element button to add a new global element declaration to the extension point schema grammar. This action is available when any node is selected. Note: Element declaration order is not important

Click the Add Attribute button to add a new local attribute declaration to a selected element in the extension point schema grammar. This action is available for all selected element nodes except for the extension element node. The extension element node always has the same attributes: id, name, point. Note: Attribute declaration order is not important.

To delete a schema object from the extension point schema grammar, right-click on a selected node and select Delete. This action is available for all selected nodes except for the extension element node. For non-shared schemas, the extension element node is the special root element used to define the entry point to the edited schema's grammar. Any extension defined in the Plug-in Manifest Editor must use an extension parent element to specify its XML related mark-up.

To specify that an element's children is composed of a sequence or choice of other elements in the extension point schema grammar, do the following:

Select and right-click the element node you wish to designate as the parent element.

Select New > Compositor > Sequence to define a sequence of elements. Select New > Compositor > Choice to define a choice of elements.

Select and right-click the new compositor node.

Select the element from the New > Reference sub-menu that you wish to designate as a child of the parent element. The Reference sub-menu is populated with all the elements declared globally in the extension point schema grammar.

Repeat steps (3) and (4) to specify additional child elements. If a Sequence compositor was specified, this amounts to specifying additional elements in the sequence. If a Choice compositor was specified, this amounts to specifying additional element choices.

Note: To specify that an element is the child of another element, an intermediate Sequence or Choice compositor must always be used by the parent element. The compositor nodes are connector nodes; whereas, the element reference nodes are leaf nodes. Therefore, compositor nodes should always specify at least one child element reference node.

Note: Sequence and Choice compositors may be nested any number of times to create more complex grammars.

Note: A grammar must not contain self-referent cycles. The most basic form of a prohibited cycle occurs when Element A requires Element B as a child and Element B requires Element A as a child.

To specify that an element has no children and allows text to be specified within its start and end XML tags (character content), simply leave the element's structure as is by not adding attributes, compositors and element references to it (element content). Note: An element cannot specify mixed content (both character content and element content).

Description Section

This section allows you to provide a brief description for any element, attribute or compositor within the edited schema's grammar.

This section is directly accessible through the Schema Definition form page.

Description Section

Description Section Fields

Edit the description text field to provide a brief description for the current element, attribute or compositor node selected in the Extension Point Elements section. Valid HTML mark-up may be used within this field as needed.

Note: Updating an element reference description also updates the description of the global element declaration it refers to and vice versa.

Element Details Section

This section allows you to edit the properties of an element. There are two kinds of elements: extension element, non-extension element. An extension element is the special root element declaration used to define the entry point to the edited schema's grammar. A non-extension element is a global element declaration used as a basic structural building block within the edited schema's grammar.

This section is activated by selecting either a extension element node or non-extension element node in the Extension Point Elements section.

The Element Details section for an extension element.

Element Details Section 1

The Element Details section for a non-extension element.

Element Details Section 2

Element Details Section Fields

Edit the Name field to specify the element name.

Click either the true button or false button within the Internal field to indicate whether the element is internal or not. The Plug-in Manifest Editor will flag any extension XML mark-up that is internal with a warning (by default). This field is analogous to x-internal attribute on exported packages.

Click either the true button or false button within the Deprecated field to indicate whether the element is deprecated or not. The Plug-in Manifest Editor will flag any extension XML mark-up that is deprecated with a warning (by default). This field is analogous to Java's Javadoc @deprecated tag.

If an extension element is marked as deprecated, edit the Replacement field to specify the non-deprecated extension point the user should use in its place to achieve the same functionality.

Select one of the element's defined attributes from the Label Property field to specify the attribute value to use as the element's label in the Plug-in Manifest Editor. For instance, if a value is selected, the element's node label in the All Extensions section of the Extensions page in the Plug-in Manifest Editor will be as follows: selected-attribute-value (element-name). Otherwise, if no value is selected, the element's node label will be as follows: (element-name).

Select one of the element's defined resource type attributes from the Icon field to specify the icon resource to use to represent the element's node in the Plug-in Manifest Editor. In order for the element's node icon to show up correctly in the All Extensions section of the Extensions page in the Plug-in Manifest Editor, the following conditions must be met:

The element must define a resource type attribute.

That attribute must be selected as the value of this field.

The contributed XML mark-up supplying the attribute value must point to a valid icon resource file.

Click either the true button or false button within the Translatable field to indicate whether the element's character content is translatable or not. This field is only applicable for elements that have no specified children (element content) or attributes and allow text to be specified within its start and end XML tags (character content). If an element's character content is marked as translatable, then PDE tools can be used on contributed extension XML mark-up to externalize the content into a properties file for multi-language support.

Use the read-only DTD approximation field to view an approximate one-to-one mapping of the element schema grammar to the equivalent element DTD grammar. A DTD (Document Type Definition) is an easier to read, less-powerful alternative to XML Schema for specifying an XML document's formal grammar. The following notation is used to specify the DTD approximation:

element-name - Represents an element in the defined grammar.

EMPTY - Indicates that no element content or character content is allowed (only attribute declarations).

#PCDATA - Indicates that only character content is allowed (Parsed Character Data).

() - Used to group an element's defined sequence and choice compositors.

| - Used to delimit element choices. Represents a logical or disjunction.

, - Used to delimit element sequences. Represents a logical and conjunction.

+ - A qualifier used to indicate one or more instances are allowed.

\* - A qualifier used to indicate zero or more instances are allowed.

Compositor Details Section

This section allows you to edit the properties of a compositor. There are two types of compositors: sequence compositor, choice compositor. A sequence compositor allows you specify that an element's children is composed of a sequence of other elements in the edited schema's grammar. A choice compositor allows you specify a that an element's children is composed of a choice of other elements in the edited schema's grammar.

This section is activated by selecting either a Sequence or Choice node in the Extension Point Elements section.

Compositor Details Section

Compositor Details Section Fields

Edit the Min Occurrences field to specify the minimum number of occurrences this compositor of other elements and / or compositors is allowed to appear in the contributed extension XML mark-up. This field's value is used by PDE to validate multiplicity constraints within the Plug-in Manifest Editor. If Element A requires exactly one Element B or Element C as its child and it uses a Choice compositor specifying 2 minimum occurrences, then any 2 instances of Element B in combination with Element C must be defined in the contributed extension XML mark-up in order to be valid.

Edit the Max Occurrences field to specify the maximum number of occurrences this compositor of other elements and / or compositors is allowed to appear in the contributed extension XML mark-up. This field's value is used by PDE to validate multiplicity constraints within the Plug-in Manifest Editor. If Element A requires exactly one Element B or Element C as its child and it uses a Choice compositor specifying 2 maximum occurrences, then no more than 2 instances of Element B in combination with Element C can be defined in the contributed extension XML mark-up to remain valid.

Check the Unbounded field to specify that there is no constraint on the maximum number of allowed occurrences this compositor may appear within the contributed extension XML mark-up.

Select a value from the Type field to specify whether this compositor is a sequence or choice.

Use the read-only DTD approximation field to view an approximate one-to-one mapping of the compositor schema grammar to the equivalent compositor DTD grammar. The notation used to specify the DTD approximation is described under the Element Details Section Fields heading in this document.

Element Reference Details Section

This section allows you to edit the properties of an element reference. An element reference is a reference to a global element declaration within the edited schema's grammar. Element references are used by compositors as atomic building blocks to define the grammar for global element declarations.

This section is activated by selecting an element reference node in the Extension Point Elements section.

Element Reference Details Section

Element Reference Details Section Fields

Edit the Min Occurrences field to specify the minimum number of occurrences this element is allowed to appear in the contributed extension XML mark-up. This field's value is used by PDE to validate multiplicity constraints within the Plug-in Manifest Editor. If Element A specifies 2 minimum occurrences, then 2 instances of Element A must be defined in the contributed extension XML mark-up in order to be valid.

Edit the Max Occurrences field to specify the maximum number of occurrences this element is allowed to appear in the contributed extension XML mark-up. This field's value is used by PDE to validate multiplicity constraints within the Plug-in Manifest Editor. If Element A specifies 2 maximum occurrences, then no more than 2 instances of Element A can be defined in the contributed extension XML mark-up to remain valid.

Check the Unbounded field to specify that there is no constraint on the maximum number of allowed occurrences this element may appear within the contributed extension XML mark-up.

Use the read-only DTD approximation field to view an approximate one-to-one mapping of the element schema grammar to the equivalent element DTD grammar. The notation used to specify the DTD approximation is described under the Element Details Section Fields heading in this document.

Element Reference Details Section Actions

Click the element-name hyperlink within the Reference field to edit the properties of the global element declaration this local element reference refers to. When this action is complete, the referenced element node is selected in the Extension Point Elements section and the corresponding Element Details section is activated.

Attribute Details Section

This section allows you to edit the properties of an attribute. Attributes are used to provide additional information about elements.

This section is activated by selecting either a required attribute, optional attribute, class attribute or resource attribute node in the Extension Point Elements section.

The Attribute Details section for a required attribute of type string that specifies an enumeration restriction.

Attribute Details Section 1

The Attribute Details section for an optional attribute of type boolean that specifies a default value.

Attribute Details Section 2

The Attribute Details section for a class attribute that indicates that the class represented by the attribute value must extend the specified class and / or implement the specified interface.

Attribute Details Section 3

The Attribute Details section for a resource attribute.

Attribute Details Section 4

Using the Schema Indentifier Attribute

Extension point authors can define attributes that reference other attributes. This is accomplished by setting your extension point attributes as type identifier and choosing the attribute it references. An example of this would be the org.eclipse.ui.handlers extension point. The commandId attribute references commands defined in the org.eclipse.ui.commands extension point.

Schema Identifier Usage

Extension developers contributing a handler can then select from all valid identifiers for the commandId attribute. Warnings are issued for identifiers that don't exist to prevent the common mistake of a misspelled identifier.

Attribute Details Section Fields

Edit the Name field to specify the attribute name.

Click either the true button or false button within the Deprecated field to indicate whether the attribute is deprecated or not. The Plug-in Manifest Editor will flag any extension XML mark-up that is deprecated with a warning (by default). This field is analogous to Java's Javadoc @deprecated tag.

Select a value from the Use field to indicate whether the attribute must be specified on its defining element within contributed extension XML mark-up. The following selections are available:

optional - The attribute may be optionally specified on its defining element.

required - The attribute must be specified on its defining element.

default - The attribute may be optionally specified on its defining element. If the attribute is not specified, then its value will assume the defined default value.

Edit the Default Value field to specify the default value to use for an attribute when it is not specified on its defining element within contributed extension XML mark-up. This field is only applicable when the default value from the Use field is selected.

Select a value from the Type field to specify the attribute value type. The following selections are available:

string - The attribute value can be any valid string.

boolean - The attribute value can be either true or false (case insensitive).

java - The attribute value can be any fully-qualified class name on the target platform's classpath (e.g. java.lang.String).

resource - The attribute value can be any project relative path to an existing file or folder resource.

Click either the true button or false button within the Translatable field to indicate whether the attributes's value is translatable or not. This field is only applicable when the string value from the Type field is selected. If an attribute's value is marked as translatable, then PDE tools can be used on contributed extension XML mark-up to externalize these values into a properties file for multi-language support.

Use the Restrictions field to view a list of string values the attribute is restricted to using as its own value. If the field contains no values, then there are no restrictions on the attribute value. This field is only applicable when the string value from the Type field is selected. Note: An enumeration restriction on the string base type is used in the edited schema's grammar to accomplish this functionality.

Edit the Extends field to specify a fully-qualified class name on the target platform's classpath (e.g. org.eclipse.ui.part.EditorPart) that must be sub-classed by the class specified as the attribute value. PDE will validate the class attribute value in the contributed extension XML mark-up to ensure it extends the super-class. This field is only applicable when the java value from the Type field is selected.

Edit the Implements field to specify a fully-qualified interface name on the target platform's classpath (e.g. org.eclipse.ui.IEditorPart) that must be implemented by the class specified as the attribute value. PDE will validate the class attribute value in the contributed extension XML mark-up to ensure it implements the interface. This field is only applicable when the java value from the Type field is selected.

Attribute Details Section Actions

Click the Add... button in the Restrictions field to open the New Restriction dialog in order to add a new restriction to the attribute value. Once the dialog is completed, the new restriction is added to the list of string values the attribute is restricted to using as its own value. This action is only available when the string value from the Type field is selected.

Select a string value within the Restrictions field and click the Remove button to remove a restriction from the list of string values the attribute is restricted to using as its own value. This action is only available when the string value from the Type field is selected.

If the value specified in the Extends field refers to a fully-qualified class name on the target platform's classpath, then clicking on the Extends hyperlink will open that class in a new Java source editor. Otherwise, clicking on the Extends hyperlink will open the New Java Class wizard allowing you to create the new class. This action is only available when the java value from the Type field is selected.

If the value specified in the Implements field refers to a fully-qualified interface name on the target platform's classpath, then clicking on the Implements hyperlink will open that interface in a new Java source editor. Otherwise, clicking on the Implements hyperlink will open the New Java Class wizard allowing you to create the new interface. This action is only available when the java value from the Type field is selected.

Click the Browse... button to open the Select Type dialog in order to select an existing class on the target platform's classpath to use as the value for the Extends field. This action is only available when the java value from the Type field is selected.

Click the Browse... button to open the Select Type dialog in order to select an existing interface on the target platform's classpath to use as the value for the Implements field. This action is only available when the java value from the Type field is selected.

Relevant Links

Extension Point Schema Editor

New Extension Point Schema Wizard

Plug-in Manifest Editor

3.5. Product Configuration Editor

Building an Eclipse product using PDE is centered around a product configuration, a .product file. This file manages all aspects of a product definition from its constituent plug-ins to configuration files to branding.

The product configuration is entirely managed by PDE to simplify Eclipse product development. Its format is internal and is not intended to be read by tools other than PDE.

PDE provides a form-based multi-page editor to manage the product configuration.

For details on the Product Configuration Editor, refer to the following documents:

Product Overview

Product Contents

Product Configuration

Product Launching

Product Splash

Product Branding

Product Licensing

Product Updates

3.5.1. Product Overview

The Overview page defines the product and provides hot links to test and export it.

Product Definition

A product is an org.eclipse.core.runtime.products extension. At minimum, it must have a name, an ID, a version and an application.

Product Definition

The product name is a translatable string that appears in the title bar of the application. It is defined as a property in the org.eclipse.core.runtime.products extension.

The product identifier combo box is populated with the identifiers of all product extensions defined in the workspace and the target. You can select one of the defined products, or you can create a new product definition on the fly using the New... button.

The product version is a string that is used to version your product file.

A product must be associated with an application, which acts as the main entry point when the product is launched. An application is an org.eclipse.core.runtime.applications extension. The application combo box is populated with the identifiers of all application extensions defined in the workspace and the target.

A product configuration can be plugin-based or feature-based. A plugin-based configuration is the default because it is simple and does not force you to understand concepts like features before building a product. A feature-based configuration is more powerful though, since it enables you to deliver a product that is updatable. That is, you can later publish patches or upgrades to your product on an update site and have your customers use the Eclipse installer to download and install these updates.

Testing

To test your product configuration, just click one of the hot links in the Testing section.

Product Testing

Similar hot links are also available in the plug-in manifest editor, but there is a noticeable difference in this instance. There is a Synchronize step.

So why is a Synchronize needed? Remember that the product configuration is for PDE use only and is not interpreted nor read by the runtime. Some of the data that enter in this file (e.g. product ID, application, window images, about image, etc.) must be copied to the plugin.xml file of your branding plug-in for these changes to take real effect at runtime. That's what the synchronize action does. It makes sure your plugin.xml contains up-to-date data and in sync with the product configuration.

If you use the hot link in this section, PDE does an implicit synchronization of files. If the product is launched via other means, e.g. a context menu launch shortcut, then automatic synchronization does not take place.

Exporting

To export an Eclipse product, you can invoke the Eclipse Product Export wizard via the hot link in the Exporting section.

Product Exporting

To export a single product configuration to multiple platforms, you need to have the org.eclipse.equinox.executable feature installed in the target platform. It is available from the Eclipse project's p2 repository. The org.eclipse.equinox.executable feature contains all the different platform-specific fragments needed for all supported platforms.

Once the org.eclipse.equinox.executable feature is in the target platform, PDE detects the presence of the org.eclipse.equinox.launchers feature and enables the cross-platform export mode.

Note, to use the org.eclipse.equinox.executable feature, you have to ensure that the 'Build target platform based on the target's installed plug-ins' option remains unchecked in the Target Platform preference page.

Note, the previously provided "Delta Pack" is no longer provided, as same functionality can be achieved by using p2 repositories directly. For more information, see Cross-platform build.

3.5.2. Product Contents

The Contents page defines the content of the product.

Content

If the product is plugin-based, the Contents page shows a Plug-ins and Fragments section where you list all the plug-ins that will make up the product. This section also provides New Plugin... and New Fragment... buttons to enable you to create new plug-ins and fragments on the fly.

Content Plug-ins

If the product is feature-based, the Contents page shows a Features section where you list all the features that will make up the product. This section also provides a New Feature... button to enable you to create new features on the fly.

Content Features

3.5.3. Product Configuration

The Configuration page defines information that builds the configuration file needed to run the product.

Configuration File

Upon startup, the runtime reads a configuration file named config.ini located in a configuration/ subdirectory of the configuration area of an Eclipse product. This file is a standard properties file that can configure many aspects of the runtime. For a full list of the supported properties, refer to the Eclipse runtime options document.

Configuration File

The Generate a default config.ini file is the default option and is recommended unless you have some very specific needs. When this option is selected, PDE generates a config.ini file with the following properties:

osgi.bundles - lists all the bundles that should be installed and optionally started once the system is up and running. This list is based on the osgi.bundles key found in the ${target\_home}/configuration/config.ini file. (Note, the ${target\_home} variable is the location of the target platform).

osgi.bundles.defaultStartLevel - denotes the default OSGi start level at which a bundle should start. The default value is set to 4.

eclipse.product - specifies the ID of the product to launch.

osgi.splashPath - specifies one or more locations of the splash screen.

You also have the option to use an existing config.ini file which will be copied by PDE to your final product as-is.

Start Levels

In the start level section, you can list plug-ins to be automatically started or started with a specific start level.

Start Levels

Properties

In the properties section, you can set configuration properties that will be added to any config.ini files generated by the editor. The properties can be qualified for a specific platform and/or OS architecture, or can apply to all configurations.

Properties

3.5.4. Product Launching

The Launching page customizes the native launcher of your product and the launching arguments.

Execution Environment

The Execution Environment section allows you to specify an execution environment (EE) that the product runs on. The default JRE associated with the chosen EE can be bundles with the product if the option is turned on.

launcher jre

The JRE will be placed in the jre directory at the root of your product.

Program Launcher

The Program Launcher section customizes the name and icon of the native launcher of the product.

Launcher Exe

To customize the name of the launcher, just enter the name in the Launcher Name field. If left unspecifed, the name will default to eclipse.

Customizing the launcher icon varies per platform. The product editor indicates the platform requirements, and flags warnings at the top of the page if the supplied icons do not match the size or depth requirements.

Launching Arguments

The Launching Arguments section lets you specify the program and VM arguments that should be used for your product. These arguments are stored in a file named <launcher\_name>.ini at the root of the product. PDE creates this file based on the values entered in this section. You can specify arguments for all Platforms, or choose a tab to specify arguments that apply only to a particular platform. You may use the combo box below the Platform tabs to specify arguments for a specific OS architecture. The Complete Arguments Preview text shows you what arguments will be used for the selected Platform and OS architecture, including arguments that apply to all platforms or architectures.

launcher arguments

3.5.5. Product Splash

The Splash page provides the ability to customize your product with a splash screen and a template or a progress bar and message.

Location

The splash screen is typically located at the root of the product-defining plug-in and must be called splash.bmp.

Splash screen location

Alternately, you may choose to place it at the root of any other plug-in in the product (the name restriction stands).

Customization

To customize the startup of a product, you can use either a splash screen template or a progress bar and message.

Splash screen location

Splash screen templates contribute custom behavior to the splash screen that appears during startup. The Template combo contains three templates available for your convenience. During the export process the selected template's required code will be added and a org.eclipse.ui.splashHandlers extension will be created. Please be aware that you may only use a splash screen template if your platform is built with 3.3 or higher.

Another option to customize your product is to use the integrated progress bar in the Eclipse splash screen by specifying the co-ordinates of the progress bar and progress message, as well as the text color. The integrated progress data is specified using properties in the org.eclipse.core.runtime.products extension.

3.5.6. Product Branding

The Branding page gives the product its identity by customizing the window images, About dialog and the welcome experience.

Window Images

The Window Images section specifies the images associated with a product. If the images supplied by you do not match the size requirements, PDE flags a warning at the top of the page.

Window Images

About Dialog

The About Dialog is customiable by providing a custom image and text.

About Image

The About dialog text and image are also properties in the org.eclipse.core.runtime.products extension.

Welcome Page

It is recommended to ship a product with its own welcome page to introduce the features of the product to first-time users.

Intro

Providing a welcome page (Intro) requires the binding of the product to an org.eclipse.ui.intro extension.

The Intro ID combo box is populated with existing Intro identifiers defined by plug-ins in the workspace and in the target.

You may also create a new Intro on the fly by pressing the New.. button.

3.5.7. Product Licensing

The Licensing page allows you to add a URL and license text for your product.

3.6. Target Definition Editor

The Target Definition Editor is used to edit target definition files in the workspace, which should end with the extension ".target". Target definition files can be created using the New Target Definition Wizard or by moving them to the workspace on the Target Platform Preference Page.

The editor consists of four pages:

Definition Page - Give your definition a descriptive name and provide the locations for plug-ins

Content Page - View the plug-ins available in your target and select a subset to include in the target platform

Environment Page - Modify other settings for your target defiition including target environment, JRE, arguments and implicit dependencies

Source Page - View and edit the source code of your target definition with assistance from completion-assist, error diagnostics, and an update command

At the top of each of the three first pages in the editor there is a link Set as Target Platform. Activating this link will set this target definition as the active target platform. To see what your active target platform is, use the Target Platform Preference Page.

Resolving Targets

Before the contents of a target can be edited and before a target definition can be set as the active target platform it must be resolved. When a target definition is resolved, the definition's locations are searched for plug-ins. Depending on the type of locations in the target, this can involved searching a directory, reading the metadata for an installation, reading a feature, or downloading from a remote site. During resolution a background job will be run. When it completes the editor will be updated with any problems that occurred.

For up to the minute target troubleshooting help see the Target Definitions Wiki Page.

Related references

Definition Page (Target Definition Editor)

Content Page (Target Definition Editor)

Environment Page (Target Definition Editor)

Source Page (Target Definition Editor)

New Target Wizard

Edit Target Wizard

Move Target Wizard

Related concepts

Target Definitions

3.6.1. Definition Page

The Definition Page in the Target Definition Editor is used to edit the most important settings in a target definition file. The Target Name is used to display your targets on the Target Platform Preference Page. The Locations section is used to add locations that contain plug-ins and works the same as the Location Tab on the Edit Target Wizard

Definition Page

The locations in the target definition will be displayed in the list. Each type of location will have a different icon and text. If the target has been resolved the count of plug-ins will be displayed in blue (the number of plug-ins included from the content page and the total number of plug-ins found. If there are problems resolving the target, they will be displayed underneath the location with the problem. The Show Plug-in Content option can be turned on to display the included plug-ins underneath each location.

Pressing Add will open the Add Location Wizard. It will provide a choice of location types to add. Selecting a location and pressing Edit will open a location type specific wizard to view and modify the location.

Related references

Target Definition Editor

Content Page (Target Definition Editor)

Environment Page (Target Definition Editor)

Source Page (Target Definition Editor)

Edit Target Wizard

Related concepts

Target Definitions

3.6.2. Content Page

The Content Page in the Target Definition Editor is used to view and edit the set of plug-ins to be included in the target definition when set as the active target platform. The definition must contain one or more locations containing plug-ins and the definition must be resolved to edit the content. Only the plug-ins that are checked on this page will be included in the target. There are multiple options available to filter and sort the list to quickly check what you require.

This page edits the same information as the Content Tab on the Edit Target Wizard

Content Page

Select / Deselect will check or uncheck the selected plug-ins.

Select All / Deselect All will check or uncheck all plug-ins in the target definition.

Add Required takes the set of currently checked plug-ins and determines all plug-ins required by them. This will also include any implicit dependencies set on this target. The required plug-ins will be checked in the list.

Use the Show options along with the filter text box at the top of the list to filter what is shown in the list. Showing plug-ins will show any non-source plug-ins in the target, while showing source bundles will display any source. Even if a plug-in is filtered from the list, it may be included in the target. You can see a count of included plug-ins underneath the list.

The Group By option is used to organize the list into different groups. The options are:

None - No grouping, plug-ins sorted by name

File Path - Grouped by the directory path the plug-ins are stored in on the local file system

Location - Grouped by the locations set on the locations tab

Managing Content with Features

On the right hand side, you can choose between two different modes to manage the content of the target. By default targets are managed using a list of plug-ins. Selecting Features will change your target contents to a list of features.

Content Page Feature Mode

You can select features in the same way as when managing by plug-ins. Add Required will select any features that the currently selected features have in their include or require entries. If there are plug-ins in the target that do not belong to a feature, they will be listed under the Other Plug-ins item and can be selected individually.

Related references

Target Definition Editor

Definition Page (Target Definition Editor)

Environment Page (Target Definition Editor)

Source Page (Target Definition Editor)

Edit Target Wizard

Related concepts

Target Definitions

3.6.3. Environment Page

The Enviroment Page in the Target Definition Editor is used to edit many settings in the definition affecting how the target will be compiled and run.

Environment Page

The environment settings on this page can also be edited using the Environment Tab, Arguments Tab and Implicit Dependencies Tab on the Edit Target Wizard

The Environment section has four settings that describe the system that this target is built for. If left blank, the settings for your current running environment are used. Changing these settings will affect how Software Site Locations resolve as well as how plug-ins are built and exported.

You can specify a JRE or execution environment for this target definition in the Target JRE section. If you specify a specific JRE or EE here, when this target is set as the active target platform, your workspace default JRE will be changed.

The Arguments section is used to set the default program and vm arguments for new PDE launch configurations.Pressing Variables will open the Variable Selection Wizard. You can browse through all known variables and get descriptions of what each does. Under the VM tab you can also press Import. This will open the Import Arguments Dialog. You can use it to import vm arguments used in the configuration data of any installation locations specified on the definition page.

The Implicit Dependencies section is used to manage the implicit dependencies of the target. Any plug-in marked as an implicit dependency will always be added as a required plug-in when PDE determines requirements. Pressing Add will open the Implicit Dependencies Selection Dialog allowing you to select one or more plug-ins from your target platform to add as implicit dependencies.

Related references

Target Definition Editor

Definition Page (Target Definition Editor)

Content Page (Target Definition Editor)

Source Page (Target Definition Editor)

Edit Target Wizard

Related concepts

Target Definitions

3.7. Simple Cheat Sheet Editor

About Cheat Sheets

In general, cheat sheets help guide users through a complex set of instructions. There are two types of cheat sheets: simple and composite. Simple cheat sheets are used to organize and present logical steps and sub-steps to achieve a certain task.

You can view and run cheat sheets using the Cheat Sheet View. To open a cheat sheet in this view, click Help > Cheat Sheets...

About This Editor

The Simple Cheat Sheet Editor allows you to compose simple cheat sheets and preview them in the Cheat Sheet View. This editor also allows you to embed workbench commands and help document links in your cheat sheets.

Editor Pages

This editor is comprised of the following pages:

Page Description

Simple Cheat Sheet Page A form page that allows you to edit the structure, order and properties of cheat sheet elements

Global Editor Actions

This editor provides the following global actions:

Label Type Name Description

Register this cheat sheet Hyperlink Register Cheat Sheet Wizard This wizard allows you to register a simple or composite cheat sheet with the Eclipse workbench to make it accessible through the Help menu.

Relevant Links

Cheat sheet example: Open the plug-in cheat sheet Creating an Eclipse Plug-in

Cheat sheet documentation: Cheat Sheets

Composite Cheat Sheet Editor

New Cheat Sheet Wizard

Recommended Work Flow for Cheat Sheet Development

Commands

3.7.1. Simple Cheat Sheet Page

About This Page

This form page allows you to edit the structure, order and properties of cheat sheet elements.

The structure and order of cheat sheet elements are edited in the Content section. This section occupies the entire left-hand portion of the editor area.

The properties of cheat sheet elements are edited in the Definition sections. These sections are activated in the right-hand portion of the editor area depending on the type of cheat sheet element selected in the Content section.

Cheat Sheet Elements

A simple cheat sheet is composed of the following elements:

Element Description Parent(s) Allowed Occurrences

Root The cheat sheet itself. None 1

Title The title of the task being solved by the cheat sheet. Root 1

Introduction The introduction of the task being solved by the cheat sheet. Root 1

Step A step in the task being solved by the cheat sheet. Root 1 or more

Sub-Step A sub-step of a step in the task being solved by the cheat sheet. Step 0 or more

Command An executable workbench command that automates the work required by a step or sub-step. Step,

Sub-Step 0 or 1

Help Document Link An associated help link added to a step or introduction. Step,

Introduction 0 or 1

Help Context ID An associated help link added to a step or introduction. Step,

Introduction 0 or 1

Page Sections

This page is comprised of the following sections:

Section Associated Element(s) Description

Content Section Title,

Introduction,

Step,

Sub-Step This section allows you to do the following:

Visualize the structure and order of the cheat sheet's title, introduction, steps and sub-steps as they appear in the Cheat Sheet view.

Edit the structure and order of the cheat sheet's steps and sub-steps.

Activate the corresponding Definition section used to edit the properties of a selected title, introduction, step or sub-step.

Preview and test the edited cheat sheet within the Cheat Sheet view.

Title Section Title This Definition section allows you to edit the title of the cheat sheet.

Introduction Section Introduction This Definition section allows you to edit the introduction of the cheat sheet.

Step Section Step This Definition section allows you to edit the properties of a step.

Sub-Step Section Sub-Step This Definition section allows you to edit the properties of a sub-step.

Command Section Command This section allows you to do the following:

Add an executable workbench command to a step or sub-step

Remove an executable workbench command from a step or sub-step

Edit the properties of an executable workbench command added to a step or sub-step

Help Section Help Document Link,

Help Context ID This collapsible section allows you to do the following:

Add a help link to a step or introduction using either a help document link or help context ID as input.

Remove a help link from a step or introduction.

Edit the help document link or help context ID used as input to add a help link to a step or introduction.

Relevant Links

Simple Cheat Sheet Editor

3.7.1.1. Content Section

About This Section

This section allows you to do the following:

Visualize the structure and order of the cheat sheet's title, introduction, steps and sub-steps as they appear in the Cheat Sheet view.

Edit the structure and order of the cheat sheet's steps and sub-steps.

Activate the corresponding Definition section used to edit the properties of a selected title, introduction, step or sub-step.

Preview and test the edited cheat sheet within the Cheat Sheet view.

This section is accessible through the Cheat Sheet form page.

Section Fields

This section contains the following fields:

Label Type Name Description Required

None Tree Viewer Cheat Sheet Element Viewer Use this field this field to do the following:

Visualize the structure and order of the cheat sheet's title, introduction, steps and sub-steps as they appear in the Cheat Sheet view.

Edit the structure and order of the cheat sheet's steps and sub-steps.

Activate the corresponding Definition section used to edit the properties of a selected title, introduction, step or sub-step.

N/A

Cheat Sheet Element Viewer

This viewer contains the following nodes:

Node Representation Description

Root Cheat Sheet Node Root Select this node, to activate the title Definition section.

Introduction Node Introduction Select this node, to activate the introduction Definition section.

Step Node Step Select this node, to activate the step Definition section.

Sub-Step Node Sub-Step Select this node, to activate the sub-step Definition section.

Content Section

Section Actions

This section provides the following actions:

Label Type Name Description

Add Step Button Add Step to Cheat Sheet Click this button to add a new step to the cheat sheet. This action is available when any of following nodes are selected: root, introduction, step.

If a step or introduction node is selected, the new step will be added after the selected node. If the root node is selected, the new step will be added as the last step of the cheat sheet.

Add Sub-Step Button Add Sub-Step to Step Click this button to add a new sub-step to a step. This action is available when any of following nodes are selected: step\*, sub-step.

Note\*: Any step that either is optional or contains a command cannot contain sub-steps.

If a sub-step node is selected, the new sub-step will be added after the selected node. If a step node is selected, the new sub-step will be added as the last sub-step of that step.

Remove Button Remove Cheat Sheet Element Click this button to remove a step or sub-step. This action is available when any of following nodes are selected: step, sub-step.

Note: In order to maintain cheat sheet validity, the following nodes cannot be removed: root, introduction, step (if it is the last remaining step in the cheat sheet).

Up Button Move Cheat Sheet Element Up Click this button to move a step or sub-step up one position in order above their respective siblings. This action is available when any of following nodes are selected: step, sub-step.

Down Button Move Cheat Sheet Element Down Click this button to move a step or sub-step down one position in order below their respective siblings. This action is available when any of following nodes are selected: step, sub-step.

Preview Button Preview Cheat Sheet Click this button to preview and test the saved or unsaved contents of the edited cheat sheet in the Cheat Sheet view.

This convenient action allows you to observe how the edited cheat sheet actually appears and operates within its deployed environment without having to register the cheat sheet and launch another Eclipse runtime instance.

Representation

A cheat sheet previewed in the Cheat Sheet view:

Title Cheat Sheet View

Relevant Links

Simple Cheat Sheet Page

Title Section

Introduction Section

Step Section

Sub-Step Section

3.7.1.2. Title Section

About This Section

This Definition section allows you to edit the title of the cheat sheet. It is activated by selecting the root node in the Content section.

Title Content Section

Section Fields

This section contains the following fields:

Label Type Name Description Required

Title Text Cheat Sheet Title Edit this field to specify the cheat sheet title. Yes

Title Definition Section

Representation

The title is represented in the Cheat Sheet view as follows:

Title Cheat Sheet View

Relevant Links

Content Section

3.7.1.3. Introduction Section

About This Section

This Definition section allows you to edit the introduction of the cheat sheet. It is activated by selecting the Introduction node in the Content section.

Introduction Content Section

Section Fields

This section contains the following fields:

Label Type Name Description Required

Body Text Cheat Sheet Introduction Edit this field to specify the introduction of the cheat sheet.

The text may be formatted using bold and break tags: To bold text, enclose with HTML-style bold tags: <b></b>. To break text, use break tags: <br/>. Yes

Introduction Definition Section

Representation

The introduction is represented in the Cheat Sheet view as follows:

Introduction Cheat Sheet View

Relevant Links

Content Section

3.7.1.4. Step Section

About This Section

This Definition section allows you to edit the properties of a step. It is activated by selecting a step node in the Content section.

Selection of a step without sub-steps:

Step Content Section

Selection of a step with sub-steps:

Step Content Section

Section Fields

This section contains the following fields:

Label Type Name Description Required

Title Text Step Title Edit this field to specify the title of the step. Yes

Body Text Step Body Edit this field to specify the instructional content of the step.

The text may be formatted using bold and break tags: To bold text, enclose with HTML-style bold tags: <b></b>. To break text, use HTML-style break tags: <br/>. Yes

This step is optional Checkbox Optional Step Check this field to make the completion of this step in the Cheat Sheet view optional.

A non-optional step includes a Click when complete action in the Cheat Sheet view to complete the step. An optional step includes a Click to skip action in the Cheat Sheet view to skip the step.

Note: A step that contains sub-steps cannot be made optional. No

Properties of a step without sub-steps:

Step Definition Section

Properties of a step with sub-steps:

Step Definition Section

Representation

A step without sub-steps is represented in the Cheat Sheet view as follows:

Step Cheat Sheet View

A step with sub-steps is represented in the Cheat Sheet view as follows:

Step Cheat Sheet View

Relevant Links

Content Section

Sub-Step Section

3.7.1.5. Sub-Step Section

About This Section

This Definition section allows you to edit the properties of a sub-step. It is activated by selecting a sub-step node in the Content section.

Sub-Step Content Section

Section Fields

This section contains the following fields:

Label Type Name Description Required

Body Text Sub-Step Body Edit this field to specify the instructional content of the sub-step. Yes

This sub-step is optional Checkbox Optional Sub-Step Check this field to make the completion of this sub-step in the Cheat Sheet view optional.

A non-optional sub-step includes a Click when complete action in the Cheat Sheet view to complete the sub-step. An optional sub-step includes a Click to skip action in the Cheat Sheet view to skip the sub-step. No

Sub-Step Definition Section

Representation

A sub-step is represented in the Cheat Sheet view as follows:

Sub-Step Cheat Sheet View

Relevant Links

Content Section

Step Section

3.7.1.6. Command Section

About This Section

This section allows you to do the following:

Add an executable workbench command to a step or sub-step

Remove an executable workbench command from a step or sub-step

Edit the properties of an executable workbench command added to a step or sub-step

This section is accessible by selecting a step or sub-step node in the Content section.

Step Content Section

Section Fields

This section contains the following fields:

Label Type Name Description Required

Command Combo Box Workbench Command Select a recently added workbench command from this field to add a command to a step or sub-step. Select <none> to remove a command from a step or sub-step.

Note: A command cannot be added to a step that contains sub-steps.

This field is globally populated and synchronized with all commands recently added to all steps and sub-steps.

Use the Browse... button to select new workbench commands. No

Parameters Table Workbench Command Parameters This non-editable field shows the applicable parameters for a workbench command selected in the Command combo box.

Each row represents one command parameter or key-value pair. The first column lists command parameter keys and the second column lists command parameter values.

To edit the parameters of a selected workbench commands, click the Browse... button. No

Command Definition Section

Section Actions

This section provides the following actions:

Label Type Name Description

Browse... Button Browse Workbench Commands Click this button to launch the Command Composer Dialog.

This dialog allows you to browse, test and select new executable workbench commands to add to a step or sub-step. The dialog also allows you to edit applicable parameters for selected workbench commands.

Representation

A step with a command is represented in the Cheat Sheet view as follows:

Step Cheat Sheet View

A sub-step with a command is represented in the Cheat Sheet view as follows:

Step Cheat Sheet View

Relevant Links

Commands

Command Composer Dialog

Content Section

Step Section

Sub-Step Section

3.7.1.6.1. Command Composer Dialog

About This Dialog

This dialog allows you to browse, test and select new executable workbench commands to add to a step or sub-step. The dialog also allows you to edit applicable parameters for selected workbench commands.

This dialog is accessible by clicking the Browse... button in the Command section.

Command Definition Section

Dialog Sections

This dialog is comprised of the following sections:

Section Description

Commands Section This section allows you to browse and filter executable workbench commands by category.

Command Details Section This section allows you to do the following:

Test an executable workbench command

Edit an executable workbench command's applicable parameters

Copy an executable workbench command's string representation to the clipboard.

Relevant Links

Commands

Command Section

3.7.1.6.1.1. Command Section

About This Section

This section allows you to browse and filter executable workbench commands by category. It is accessible through the Command Composer Dialog.

Section Fields

This section contains the following fields:

Label Type Name Description Required

None Text Workbench Command Filter Field Type a partial string in this field to filter the Workbench Command Viewer categories and commands by matching occurrences of that string in their names. N/A

None Tree Viewer Workbench Command Viewer Browse this field to select executable workbench commands. All root nodes represent categories and all child nodes represent workbench commands. N/A

Command Composer Content Section

Relevant Links

Commands

Command Composer Dialog

Command Details Section

3.7.1.6.1.2. Command Details Section

About This Section

This section allows you to do the following:

Test an executable workbench command

Edit an executable workbench command's applicable parameters

Copy an executable workbench command's string representation to the clipboard.

This section is activated by selecting a command node in the Commands section.

Command Composer Content Section

Section Fields

This section contains the following fields:

Label Type Name Description Required

Command ID Text Workbench Command ID This non-editable field shows the workbench command ID. N/A

Command Parameter (0 or more) Text or Combo Box Command Parameter A workbench command may have 0 or more configurable parameters. One input field is generated for each command parameter.

Enter applicable values into these fields to modify the behavior of the workbench command accordingly. N/A

An executable workbench command with parameters:

Command Composer Details Section 1

An executable workbench command without parameters:

Command Composer Details Section 2

Section Actions

This section provides the following actions:

Label Type Name Description

Execute Hyperlink Execute Workbench Command Click this link to execute the workbench command from within the Command Composer Dialog.

Note: This action is not available for workbench commands that cannot be executed from within a dialog.

Copy To Clipboard Hyperlink Copy Command String To Clipboard Click this link to copy an executable workbench command's string representation to the clipboard.

The string representation can be used to directly embed workbench commands into simple cheat sheets and user assistance documentation.

Command Composer Details Section 2

Relevant Links

Commands

Command Composer Dialog

Commands Section

3.7.1.7. Help Section

About This Section

This collapsible section allows you to do the following:

Add a help link to a step or introduction using either a help document link or help context ID as input.

Remove a help link from a step or introduction.

Edit the help document link or help context ID used as input to add a help link to a step or introduction.

This section is accessible by selecting a step or introduction node in the Content section.

Step Content Section

Section Fields

This section contains the following fields:

Label Type Name Description Required

Type Combo Box Help Link Type Select <none> to remove a help link from a step or introduction. Consequently, the Value input field will no longer be visible.

Select Help Context ID to make the Value input field visible right under this field and ready to accept a help context ID as input.

Select Help Document Link to make the Value input field visible right under this field and ready to accept a help document link as input. No

Value Text Help Document Link Value or Help Context ID Value If Help Context ID is selected in the Type field, specify a valid help context ID in this field to add a help link to a step or introduction.

In the Cheat Sheet view, clicking on a step or introduction's help link created using a help context ID will open that help context ID in the Help view.

The format of a valid help context ID is the following: <plugin-id>.<help-context-id>

If Help Document Link is selected in the Type field, specify a valid help document link in this field to add a help link to a step or introduction.

In the Cheat Sheet view, clicking on a step or introduction's help link created using a help document link will open that help document link in the Help Infocenter.

The format of a valid help document link is the following: /<plugin-id>/<path>/<html-file> No

Example of a help link removed from a step or introduction. Note: This section is collapsed by default when no help context ID or help document link is specified.

Definition Section: None

Example of a help link added to a step or introduction defined using a help context ID as input. Note: This section is expanded by default when a help context ID is specified.

Definition Section: Help Context ID

Example of a help link added to a step or introduction defined using a help document link as input. Note: This section is expanded by default when a help document link is specified.

Definition Section: Help Document Link

Representation

An introduction with a help link and various steps with help links are represented in the Cheat Sheet view as follows:

Title Cheat Sheet View

Note: A help link created using a help context ID is indistinguishable from one created using a help document link in the Cheat Sheet view.

Relevant Links

Content Section

Step Section

Introduction Section