

# Practical Eclipse Rich Client Platform Projects



Vladimir Silva

## **Practical Eclipse Rich Client Platform Projects**

**Copyright © 2009 by Vladimir Silva**

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13 (pbk): 978-1-4302-1827-2

ISBN-13 (electronic): 978-1-4302-1828-9

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Java™ and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc., in the US and other countries. Apress, Inc., is not affiliated with Sun Microsystems, Inc., and this book was written without endorsement from Sun Microsystems, Inc.

Lead Editor: Tom Welsh

Technical Reviewer: Sumit Pal

Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Tony Campbell,

Gary Cornell, Jonathan Gennick, Michelle Lowman, Matthew Moodie, Jeffrey Pepper, Frank Pohlmann,  
Ben Renow-Clarke, Dominic Shashaft, Matt Wade, Tom Welsh

Project Managers: Douglas Sulenta, Susannah Davidson Pfalzer

Senior Copy Editor: Marilyn Smith

Associate Production Director: Kari Brooks-Copony

Production Editor: Ellie Fountain

Compositor: Molly Sharp

Proofreader: Linda Seifert

Indexer: Broccoli Information Management

Artist: Kinetic Publishing Services, LLC

Cover Designer: Kurt Krames

Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit <http://www.springeronline.com>.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail [info@apress.com](mailto:info@apress.com), or visit <http://www.apress.com>.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales—eBook Licensing web page at <http://www.apress.com/info/bulksales>.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at <http://www.apress.com>.

# Contents at a Glance

About the Author.....	.xi
About the Technical Reviewer.....	xiii
Introduction.....	xv
<b>CHAPTER 1</b> Foundations of Eclipse RCP.....	1
<b>CHAPTER 2</b> Plug-ins: A First Glimpse.....	21
<b>CHAPTER 3</b> RCP Basics.....	53
<b>CHAPTER 4</b> User Interface Concepts.....	77
<b>CHAPTER 5</b> Forms API and Presentation Framework.....	107
<b>CHAPTER 6</b> Help Support.....	141
<b>CHAPTER 7</b> 2D Graphics with GEF and Zest.....	173
<b>CHAPTER 8</b> 3D Graphics for RCP with OpenGL.....	209
<b>CHAPTER 9</b> Professional Reports with the Business Intelligence and Report Toolkit.....	261
<b>CHAPTER 10</b> Automated Updates.....	291
<b>INDEX</b> .....	325

# Contents

About the Author .....	.xi
About the Technical Reviewer .....	xii
Introduction .....	xv
<b>CHAPTER 1 Foundations of Eclipse RCP .....</b>	<b>1</b>
Benefits of Eclipse .....	1
How Is RCP Different from the Eclipse Workbench? .....	2
Eclipse RCP Architecture .....	2
Equinox OSGi .....	3
Core Platform .....	3
Standard Widget Toolkit .....	4
JFace .....	5
The Eclipse Workbench .....	6
Hands-on Exercise: Getting Your Feet Wet with the OSGi Console .....	6
Starting a New Plug-in Project .....	6
Creating the Plug-in .....	9
Testing the Plug-in .....	12
Using OSGi Console Commands .....	15
Using Logging Services .....	16
Summary .....	19
<b>CHAPTER 2 Plug-ins: A First Glimpse .....</b>	<b>21</b>
Introducing the Eclipse Plug-in Model .....	21
The Plug-in Class and BundleContext .....	22
Manifests .....	23
Plug-in Fragments and Features .....	24
Adding Extension Points .....	24
Perspectives .....	25
Views .....	28
View Actions .....	29
Editors .....	31
Pop-up Menus .....	33
Commands .....	35

Hands-on Exercise: Fun with a Web Browser Plug-in . . . . .	38
Adding a Perspective Extension Point . . . . .	39
Adding a Perspective Factory . . . . .	40
Adding Views and Content . . . . .	41
Testing the Plug-in . . . . .	48
Enhancing the Web Browser . . . . .	49
Summary . . . . .	50
<b>CHAPTER 3 RCP Basics . . . . .</b>	<b>53</b>
Components of an RCP Application . . . . .	53
Extension Points for an RCP Application . . . . .	55
OSGi Manifest . . . . .	59
Plug-in Manifest . . . . .	60
Advisor Classes . . . . .	61
Plug-in Class . . . . .	63
Defining and Branding Products . . . . .	65
Using Features . . . . .	67
Product Testing and Packaging . . . . .	67
Hands-on Exercise: An RCP Application for the Web Browser Plug-in . . . . .	68
Adding an Application Extension Point . . . . .	68
Changing the Default Perspective . . . . .	70
Modifying Advisor Classes . . . . .	70
Adding Menu and Toolbar Extension Points . . . . .	72
Adding Commands, Key Bindings, and Handlers . . . . .	73
Creating the Product Configuration File . . . . .	75
Summary . . . . .	76
<b>CHAPTER 4 User Interface Concepts . . . . .</b>	<b>77</b>
Hierarchical Navigation with the Common Navigator Framework . . . . .	77
CNF Basics . . . . .	78
Using CNF Within RCP . . . . .	79
Concurrency Infrastructure . . . . .	83
Jobs API Basics . . . . .	84
Using the Concurrency Infrastructure . . . . .	86
Hands-on Exercise: A CNF File System Navigator . . . . .	93
Creating an RCP Project Template . . . . .	93
Adding CNF Extension Points . . . . .	94
Implementation Classes . . . . .	96
Summary . . . . .	106

<b>CHAPTER 5</b>	<b>Forms API and Presentation Framework</b>	107
	Forms API Basics	107
	Common Controls	109
	Form Look and Feel	111
	Custom Layouts	115
	Complex Controls	115
	Complex Forms	124
	Managed Forms	124
	Master/Details Form	125
	Multipage Editors	125
	Hands-on Exercise: A Web Look for the Mail Template	127
	Customizing the Workbench Window	128
	Customizing the Window Contents	129
	Modifying the Navigation View	132
	Modifying the Mail View	134
	Summary	138
<b>CHAPTER 6</b>	<b>Help Support</b>	141
	Configuring a Product to Use the Help System	141
	Adding the Dependency Plug-ins	142
	Updating the Menu Bar	142
	Adding Help Content	144
	Help System Extension Points	146
	TOC File	147
	Index File	148
	Internationalization	150
	Adding Context Help Support	150
	Product Plug-in Modifications	151
	Help Plug-in Modifications	152
	Customizing the Help System	154
	Hands-on Exercise: Create an Infocenter from Custom Documentation	156
	Splitting the Documentation into Topic HTML/XHTML Files	156
	Creating the Help Contents Plug-in	157
	Creating the Infocenter Plug-in	158
	Adding a Product Configuration File to the Infocenter Plug-in	159
	Adding a TOC to the Help Contents Plug-in	160
	Adding a Help Menu to the Infocenter Plug-in	162
	Adding Help System Dependencies to the Product Configuration	163
	Testing the Infocenter Plug-in	164

	Deploying the Infocenter Plug-in . . . . .	166
	Starting the Infocenter from the Command Line . . . . .	166
	Customizing the Infocenter . . . . .	168
	Summary . . . . .	170
<b>CHAPTER 7</b>	<b>2D Graphics with GEF and Zest . . . . .</b>	<b>173</b>
	Draw2d—The Big Picture . . . . .	173
	Using GEF . . . . .	175
	Displaying Figures . . . . .	176
	Exploring the GEF Shapes Example . . . . .	176
	Adding EditPolicies . . . . .	185
	Adding a Palette . . . . .	187
	Using Zest . . . . .	190
	Zest Components . . . . .	191
	Zest Layouts . . . . .	193
	Hands-on Exercise: Build Your Own Advanced 2D Graphics Editor . . . . .	195
	Creating the RCP Product . . . . .	196
	Building a Zest Plug-in . . . . .	200
	Testing the Final Product . . . . .	206
	Summary . . . . .	207
<b>CHAPTER 8</b>	<b>3D Graphics for RCP with OpenGL . . . . .</b>	<b>209</b>
	OpenGL and SWT . . . . .	209
	The Device-Independent Package . . . . .	210
	OpenGL Bindings for SWT . . . . .	211
	Creating OpenGL Scenes with JOGL and SWT . . . . .	211
	Setting Up for the OpenGL Scenes . . . . .	212
	Creating the Wire Cubes Scene . . . . .	220
	Creating the 3D Chart Scene . . . . .	228
	Rotating and Moving the Scene . . . . .	236
	Refreshing the Scene . . . . .	241
	Putting the Scene into an RCP View . . . . .	241
	Hands-on Exercise: Build a Powerful 3D Earth Navigator . . . . .	242
	WWJ Basics . . . . .	242
	Setting Up the Earth Navigator Project . . . . .	245
	Creating the Earth Navigator View . . . . .	248
	Flying to a Location Within a Globe . . . . .	250
	Finding Latitude and Longitude with the Yahoo Geocoding API . . . . .	251
	Creating the Layer Navigator View with Geocoding . . . . .	256
	Summary . . . . .	258

<b>CHAPTER 9</b>	<b>Professional Reports with the Business Intelligence and Report Toolkit</b> .....	261
	Using the Report Designer Within the Eclipse IDE .....	261
	Installing BIRT Using the Software Updates Manager .....	262
	Report Anatomy .....	263
	Getting Your Feet Wet with the Report Designer .....	263
	Using BIRT Within a Servlet Container .....	269
	Deploying the BIRT Runtime .....	269
	Using the Report Viewer Servlet .....	270
	Using the JSP Tag Library .....	272
	Using the Report Engine API .....	275
	Configuring and Creating a Report Engine .....	276
	What Kinds of Operations Can Be Done with the Report Engine? .....	277
	Hands-on Exercise: Report Generation from the OSGi Console .....	283
	Extending the OSGi Console .....	283
	Generating the Report .....	285
	Running the Report Generator Plug-in .....	289
	Summary .....	290
<b>CHAPTER 10</b>	<b>Automated Updates</b> .....	291
	Updating and Installing Software the Eclipse Way .....	291
	Defining and Configuring a Product .....	292
	Grouping Plug-ins in Features .....	295
	Grouping Plug-ins Within Fragments .....	296
	Building an Update Site Project .....	296
	Software Update UI Tools .....	296
	Using the Software Updates and Add-ons Dialog .....	297
	Installing Software from the Command Line .....	298
	Product Build Automation with the Headless Build System .....	300
	Build Configuration .....	302
	Build Phases .....	304
	Hands-on Exercise: Automated Updates and Builds for RCP .....	305
	Creating a Feature .....	305
	Creating an Update Site .....	306
	Testing and Publishing .....	308
	Building the Product Headless .....	314
	Building the Product Headless from a CVS Repository .....	318
	Summary .....	323
<b>INDEX</b> .....		325



# About the Author

■ **VLADIMIR SILVA** was born in Quito, Ecuador. He received a System's Analyst degree from the Polytechnic Institute of the Army in 1994. In the same year, he came to the United States as an exchange student pursuing a Master's degree in Computer Science at Middle Tennessee State University. After graduation, he joined IBM's Web-Ahead technology think tank. His interests include grid computing, neural nets, and artificial intelligence. Vladimir also holds numerous IT certifications, including Oracle Certified Professional (OCP), Microsoft Certified Solution Developer (MCSD), and Microsoft Certified Professional (MCP).

# About the Technical Reviewer



■ **SUMIT PAL** has about 15 years of experience with software design and development and architecture on a variety of platforms, including Java and J2EE. Sumit worked in the Microsoft SQL Server Replication group for 2 years, and with Oracle's OLAP Server group for 7 years.

Currently, he works as an OLAP architect for LeapFrogRx, which provides advanced analytics to pharmaceutical companies.

Along with certifications like IEEE Certified Software Development Professional (CDSP) and J2EE Architect, Sumit has a Master's degree in Computer Science.

Sumit has a keen interest in search engine internals, data mining, database internals, and algorithms. He has invented some basic generalized algorithms to find divisibility between numbers and also to find divisibility rules for prime numbers less than 100.

In his spare time, Sumit loves to play badminton and swim, and also help organizations like Akshaya Patra Foundation (<http://foodforeducation.org/>) raise funds.

# Introduction

**E**clipse Rich Client Platform (RCP) has become the leading open development platform, capturing close to 70% of the open integrated development environment (IDE) market. I wrote this book to give you a clear and technical guide to Eclipse development, and to help you achieve your goals quickly. If you use Eclipse, you must become familiar with RCP. It gives you all the tools you need to build commercial-quality applications and deploy them quickly, thus saving time and increasing the return on investment.

In *Practical Eclipse Rich Client Platform Projects*, I explain the necessary technical concepts approachably, with plenty of source code and images, in a detailed and engaging (I hope) way. This book will show you how to apply modern graphical user interface (GUI) concepts to your applications using real-world examples. Each chapter explains the concepts carefully, and then puts them to the test with a hands-on exercise.

We start with the architecture and foundations of Eclipse RCP, taking a tour of Equinox, the core platform, Standard Widget Toolkit (SWT), and the Eclipse IDE workbench. Next, you learn about the details of the plug-in architecture, always with a focus on RCP components: perspectives, advisor classes, basic branding, and product configuration. In Chapter 4, we look at common concurrency concepts used in modern GUI development with the Jobs API: job classes, scheduling rules, resource management, and more.

In Chapter 5, you learn how to spice up your GUI with the powerful Forms API. Among the areas covered are look and feel, form controls, and advanced topics—in short, everything you need to improve the look of your RCP application.

A good help system is an important component of any application. Chapter 6 tackles this subject with detailed descriptions of how to build your help files, required help plug-ins, dependencies, and configuration.

If you ever work on graphics-enabled applications, you will find Chapters 7 and 8 of special interest. They cover 2D and 3D graphics in depth. In the 2D arena, you learn how to use Draw2D, Graphical Editing Framework (GEF), and the Zest visualization toolkit. If 3D is your thing, you'll want to take a look at how OpenGL can be used to build a powerful Earth navigator (Google style) in a snap.

Chapter 9 shows you how to create powerful reports using the Business Intelligence and Report Toolkit (BIRT), which you will find particularly useful if your application is targeted to a business environment.

Finally, Chapter 10 explains how to pack your work and deploy it automatically to an update site using the automated build system.

*Practical Eclipse Rich Client Projects* covers all the major needs of a modern application. It will help you get things done. If you are interested in the source code, you can download it from this book's details page at the Apress web site (<http://www.apress.com>).



# Foundations of Eclipse RCP

**T**he Eclipse philosophy is simple and has been critical to its success. The Eclipse Platform was designed from the ground up as an integration framework for development tools. Eclipse also enables developers to easily extend products built on it with the latest object-oriented technologies.

Although Eclipse was designed to serve as an open development platform, it is architected so that its components can be used to build just about any client application. The minimal set of modules needed to build a rich client is collectively known as the *Rich Client Platform* (RCP).

This chapter focuses on the foundations of RCP. It begins with a summary of the benefits of Eclipse, and then discusses the architecture of RCP. Finally, you'll work through a practical exercise that demonstrates the power of this dynamic modular technology.

## Benefits of Eclipse

Eclipse is an integrated development environment (IDE) written primarily in Java. However, it goes well beyond a Java development platform in the following ways:

- It is open and extensible. Extensible software can function as a component of a larger system. Eclipse's openness permits greater interoperability, opportunity, and choice.
- It provides multilanguage support. Eclipse supports an army of programming languages, including Java, Java Platform, Enterprise Edition (Java EE), AspectJ, C/C++, Ruby, Perl, COBOL, and many others.
- It provides a consistent feature set across all platforms. This allows developers to concentrate on the problem rather than the specific platform. More important, it functions the same way on each of these platforms.
- It provides a native look and feel, which is required by today's professional applications.
- A very active community is willing to help with any problem. Moreover, since Eclipse is the foundation for a number of commercial software products, many vendors offer additional support.
- Eclipse is at the forefront of the software tools industry. This means that you can depend on it as a viable, industrial-strength tool for the foreseeable future.

The bottom line is that Eclipse is extensible, configurable, free, and fully supported. It is so well designed for these purposes that many developers find it a pleasure to work with. Newcomers from other languages, especially C/C++ on Unix, will discover this after learning the basics.

## How Is RCP Different from the Eclipse Workbench?

Many people struggle to understand the difference between the Eclipse IDE workbench and RCP. The answer is simple: there is no difference—well almost no difference. Both are based on a dynamic plug-in model, and the user interface (UI) for the workbench and RCP is built using the same toolkits and extension points. However, RCP has the following distinguishing features:

- In RCP, the layout and function of the Eclipse IDE workbench is under fine-grained control of the plug-in developer. In fact, the Eclipse IDE workbench itself is an RCP application for software development. Here is where the line between these two becomes thin.
- In RCP, the developer is responsible for defining the application and customizing the look and feel of the Eclipse IDE workbench to fit the needs of the application.
- In RCP, the platform application needs only the plug-ins `org.eclipse.ui` and `org.eclipse.core.runtime` to run. However, RCP applications are free to use any platform plug-ins they need to provide their feature set.

## Eclipse RCP Architecture

RCP employs a lightweight software component framework based on plug-ins. This architecture provides extensibility and seamless integration. Everything in RCP (and Eclipse, for that matter), with the exception of the runtime kernel, is a plug-in. It could be said that all features are created equal, as each plug-in integrates with Eclipse in exactly the same way. A plug-in can be anything: a dialog, a view, a web browser, a database explorer, a project explorer, and so forth.

RCP is architected so that its components can be put together to build just about any client application using a dynamic plug-in model, toolkits, and extension points. The layout and function of the workbench is under the fine-grained control of the plug-in developer. Under the covers, the following components constitute RCP:

- Equinox
- Core platform
- Standard Widget Toolkit
- JFace
- Eclipse IDE workbench

Let's take a closer look at each of these components.

## Equinox OSGi

According to its developers, OSGi<sup>1</sup> is a dynamic module system for Java. OSGi was designed as a technology to tackle software complexity created by monolithic software products. Its focus is the development of new software, as well as the integration of existing software into new systems. By providing standards for the integration of software, the OSGi framework improves reusability and reliability, and reduces development costs.

At its core, OSGi provides a software framework that allows applications to be constructed from small, reusable, and collaborative components. These components, in turn, can be included in a bigger application and deployed.

Equinox is Eclipse's implementation of the OSGi framework. It defines an application life-cycle management model, a service registry, an execution environment, and modules. On top of this framework, a large number of OSGi layers, application program interfaces (APIs), and services have been defined.

An important concept in the OSGi framework is the *bundle*. A bundle is a dynamic component that can be remotely installed, started, stopped, updated, and uninstalled without requiring a reboot.

Life-cycle management is done via APIs, which allow for remote downloading of management policies. Such a dynamic component model is missing from today's stand-alone Java Virtual Machine (JVM) environments.

OSGi provides a powerful dynamic component model, which is why the Eclipse Foundation selected it as the underlying runtime for Eclipse RCP and the IDE.

## Core Platform

The core platform includes a runtime engine that starts the platform base and dynamically discovers and runs plug-ins.

### Core Platform Responsibilities

The core platform is responsible for the following:

- Defining a structure for plug-ins and the implementation details: bundles and class-loaders
- Finding and executing the main application, and maintaining a registry of plug-ins, their extensions, and extension points
- Providing miscellaneous utilities, such as logging, debug trace options, adapters, a preference store, and a concurrency infrastructure

The runtime is defined by the plug-ins `org.eclipse.osgi` and `org.eclipse.core.runtime` on which all other plug-ins depend. It effectively holds all the pieces together.

---

1. OSGi originally stood for Open Services Gateway initiative, but that name is now obsolete. Visit <http://www.osgi.org> for more information about OSGi.

**Note** Because plug-ins are implemented using the OSGi framework, a plug-in is essentially the same thing as an OSGi bundle. I will use these terms interchangeably, unless discussing particular framework classes.

---

## Runtime Plug-in Model

The plug-in model is structured around the following concepts:

*Plug-in:* A plug-in is a structured bundle of code and/or data that contributes functionality to the system. Some plug-ins can contribute to the UI using an extension point model. Others supply class libraries that can be used to implement system extensions.

*Extension points:* An extension point is a well-defined place where other plug-ins can add functionality. Plug-ins can add extensions to the platform by implementing an extension point. Defining an extension point can be thought of as defining an API, with the difference that the extension point is declared in Extensible Markup Language (XML) instead of code.

*OSGi manifest and plug-in manifest:* These manifests allow the plug-in to describe itself to the system. The extensions and extension points are declared in the plug-in manifest file, which is called `plugin.xml`. The platform maintains a registry of installed plug-ins and the functions they provide in the `MANIFEST.MF` file.

*Dynamic loading:* In the OSGi services model, software bundles do not pay a memory or performance penalty for components that are installed but not used. A plug-in can be installed and added to the registry, but it will not be activated unless a function that it provides is requested at runtime.

*Resource management:* Resources within the user's workspace are managed by the plug-in `org.eclipse.core.resources`. This plug-in provides services for accessing the projects, folders, and files stored in the user's workspace or alternate file systems, such as network file systems or a database. This plug-in is most useful for Eclipse IDE applications.

The overall philosophy of the core platform revolves around the idea of building plug-ins to extend the system. For example, the Eclipse Software Development Kit (SDK) includes the basic platform plus two major tools: the full-featured Java development tools (JDT) and a Plug-in Developer Environment (PDE) to facilitate the development of plug-ins and extensions. These tools provide an example of how new tools can be composed by building plug-ins that extend the system.

## Standard Widget Toolkit

The Standard Widget Toolkit (SWT) is the graphical widget toolkit used by Eclipse. Originally developed by IBM, it was created to overcome the limitations of the Swing graphical user interface (GUI) toolkit introduced by Sun. Swing is 100% Java and employs a lowest common

denominator to draw its components by using Java 2D to call low-level operating system primitives. SWT, on the other hand, implements a common widget layer with fast native access to multiple platforms.

SWT's goal is to provide a common API, but avoid the lowest common denominator problem typical of other portable GUI toolkits. SWT was designed for the following:

*Performance:* SWT claims higher performance and responsiveness, and lower system resource usage than Swing.<sup>2</sup>

*Native look and feel:* Because SWT is a wrapper around native window systems such as GTK+ and Motif, SWT widgets have the exact same look and feel as native ones. This is in contrast to the Swing toolkit, where widgets are close copies of native ones. This is clearly evident just by looking at Swing applications.

*Extensibility:* Critics of SWT may claim that the use of native code does not allow for easy inheritance and hurts extensibility. However, both Swing and SWT support writing new widgets using Java code only.

Perhaps a shortcoming is that, unlike Swing, SWT requires manual object deallocation, as opposed to the standard automatic garbage collection of Swing. SWT objects must be explicitly disposed of; otherwise, memory leaks or other unintended behavior may result. This is due to the native nature of SWT, as widgets are not tracked by the JVM, which is unable to garbage-collect them. Some claim that this increases development time and costs for the average Java developer. But the truth of the matter is that the only SWT objects a developer must explicitly dispose of are the subclasses of `Image`, `Color`, and `Font` objects.

## JFace

JFace is a window-system-independent GUI toolkit for handling many common programming tasks. JFace is designed to work with SWT without hiding it, and implements a model-view-controller (MVC) architecture.

The following are some of the UI components in JFace:

- Image and font registries
- Text, dialog, preference, and wizard frameworks
- Viewers
- Actions

Viewers are used to simplify the interaction between an underlying data model and the widgets used to present that model. Table and tree viewers are the most typical examples.

Actions are essential for the developer. They may fire when a toolbar button or a menu item is clicked or when a defined key sequence is invoked. They are most useful when contributed to the workbench declaratively in `plugin.xml`.

---

2. See “Why I choose SWT against Swing” (November 19, 2004), on Ozgur Akan’s blog. ([http://weblogs.java.net/blog/aiqa/archive/2004/11/why\\_i\\_choose\\_sw.html](http://weblogs.java.net/blog/aiqa/archive/2004/11/why_i_choose_sw.html)).



## The Eclipse Workbench

The Eclipse IDE workbench is the basic development environment in Eclipse. It is built around the following concepts:

*Perspectives:* A perspective defines the initial set and layout of the views in your workbench window. Perspectives are focused on a specific development task, such as Java, Java EE, plug-in, and so on.

*Views:* Views are the small windows and sidebars around the edges of the workbench. Views are used to navigate the workbench and present information in different ways.

*Editors:* Editors are used to do the actual coding. For example, you might use editors to code in Java, JavaScript, Hypertext Markup Language (HTML), or Cascading Style Sheets (CSS).

*Workspaces:* A workspace is the disk folder where the actual work will be stored.

*Projects:* A project is a container used by the workbench to group associated folders and files.

---

**Note** All the exercises in this book were written using Eclipse 3.4 (Ganymede). This is important, as the UI is somewhat different from that of version 3.3.

---

## Hands-on Exercise: Getting Your Feet Wet with the OSGi Console

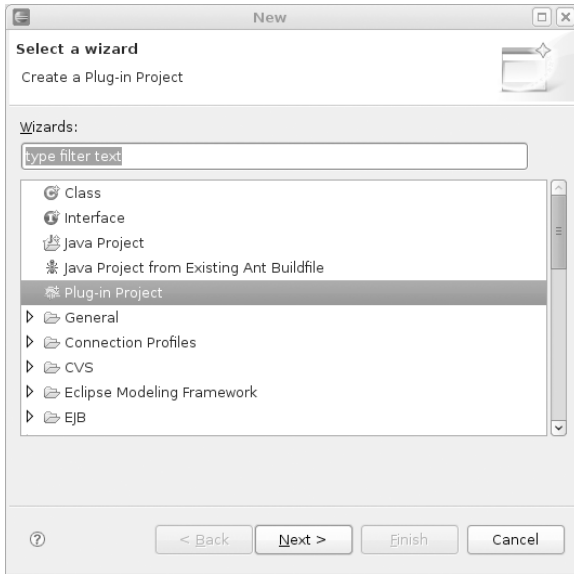
Programming with Eclipse can be thought of as a game. The more you practice, the better you get at it. The goal of this exercise is to get you started by building a plug-in project that uses the OSGi console. We'll go beyond of the typical Hello World example.

In this exercise, you will write a plug-in to embed a tiny Jetty web server that uses Equinox to define a simple servlet class that returns the headers of the HTTP request. This plug-in will use the extension point `org.eclipse.equinox.http.registry.servlets` to define the servlet alias `/servlet1`, which will be accessed through the browser as `http://localhost:8080/servlet1`.

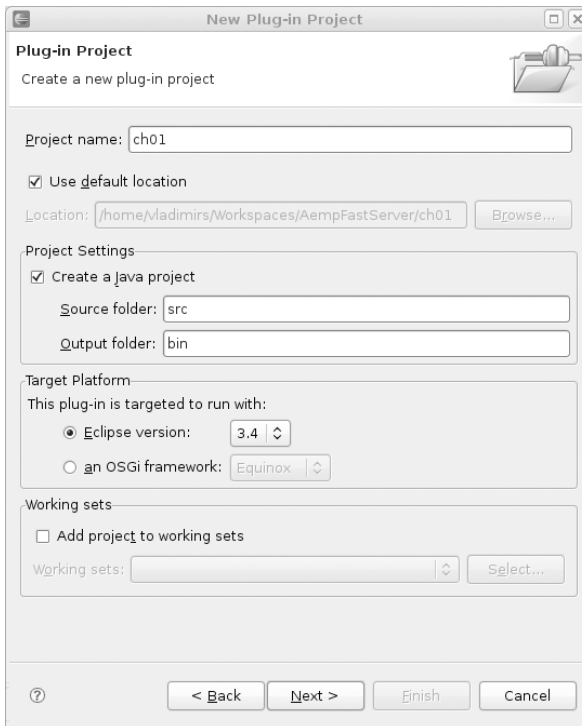
### Starting a New Plug-in Project

Starting a new plug-in project is easy with the Plug-in Project wizard.

1. From the Eclipse IDE main menu, select **File** ► **New Project** (or click the **New Project** icon on the toolbar) and choose **Plug-in Project**, as shown in Figure 1-1. Then click **Next**.
2. Enter a project name and use the default target platform, as shown in Figure 1-2. Click **Next** to continue.



**Figure 1-1.** *Selecting to create a plug-in project*



**Figure 1-2.** *Naming and targeting the plug-in project*

3. Enter the plug-in information. The plug-in ID uniquely identifies the plug-in within the core runtime. In the Plug-in Options section, you need to choose to generate an activator class to control the plug-in life cycle. Leave the option “This plug-in will make contributions to the UI” unchecked, as the plug-in will not display a UI. You do not want to create a rich client application, so leave the final option set to No, as shown in Figure 1-3. Click Finish to create the plug-in project.



**Figure 1-3.** *Specifying plug-in content*

The wizard builds the project, and then presents the plug-in manifest editor, as shown in Figure 1-4. The two most important files are `Activator.java` and `MANIFEST.MF`.

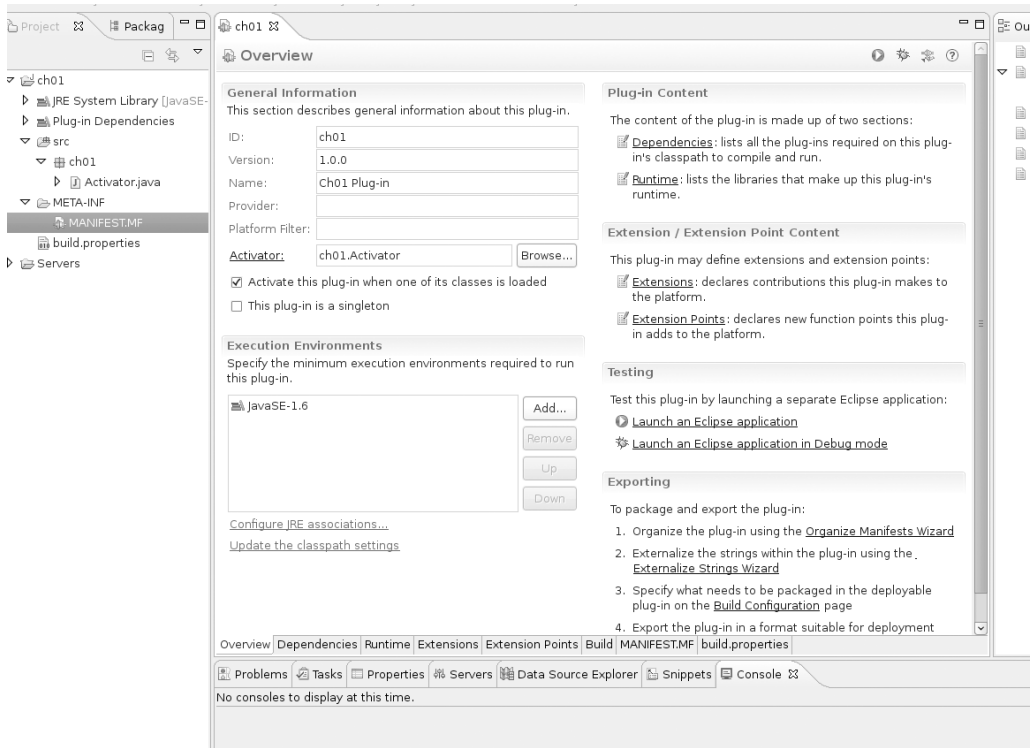


Figure 1-4. Plug-in manifest editor for this exercise

## Creating the Plug-in

The activator class controls the life-cycle aspects and overall semantics of a plug-in. A plug-in can implement specialized functions for the start and stop aspects of its life cycle. Each life-cycle method includes a reference to a `BundleContext`, as follows:

```
public void start(BundleContext context) throws Exception {
    super.start(context);
    plugin = this;

    log.info("Activator Start");
}
```

```
public void stop(BundleContext context) throws Exception {
    plugin = null;
    super.stop(context);

    log.info("Activator Stop");
}
```

BundleContext is a reference that contains information related to the plug-in and other bundles/plug-ins in the system. Chapter 2 provides more information about the BundleContext methods.

The Dependencies tab of the plug-in manifest editor is used to add references to other bundles. You also need to add an extension point and implement the servlet, which you can do through the Extensions tab of the editor.

1. To add references to other bundles, click the Dependencies tab, and then click the Add button in the Required Plug-ins section. This displays the Plug-in Selection dialog, as shown in Figure 1-5. For this exercise, add the following references, which are required by the servlet extension point:

- javax.servlet
- org.eclipse.equinox.http.jetty
- org.eclipse.equinox.http.registry
- org.eclipse.equinox.http.servlet

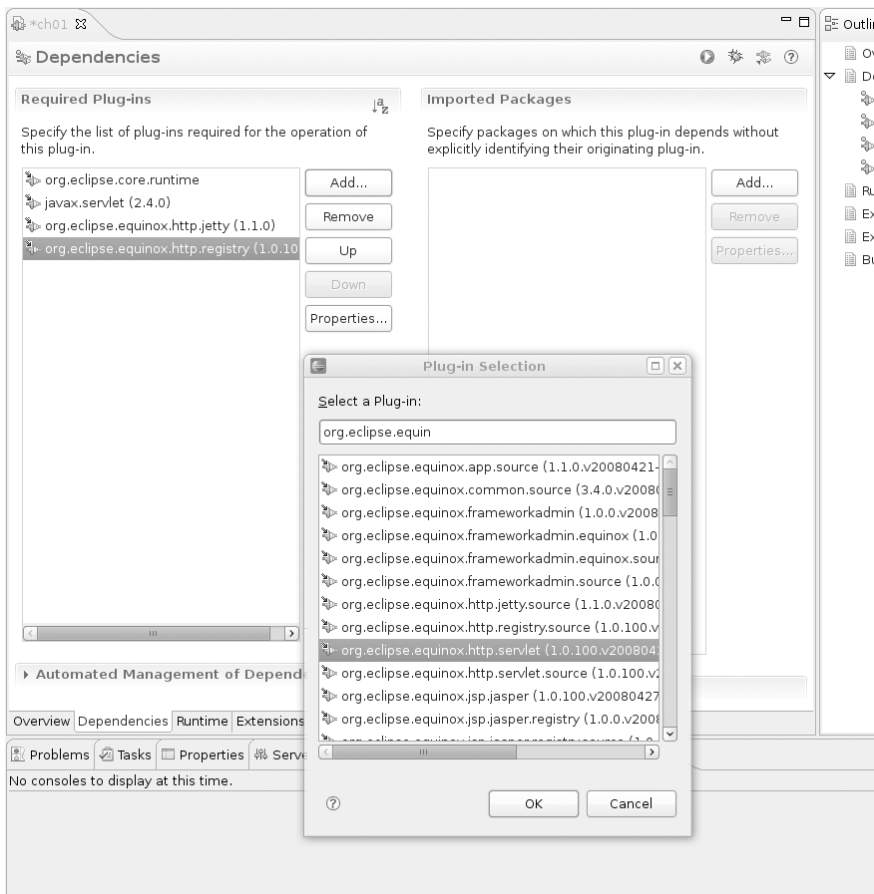


Figure 1-5. Adding dependencies

2. Click the Extensions tab. Click the Add button and select the extension point `org.eclipse.equinox.http.registry.servlets`. A servlet class name and alias will be inserted automatically. The servlet alias (`/servlet1`) will be used to reference the servlet from a web browser. Internally, the XML for this extension point looks like this:

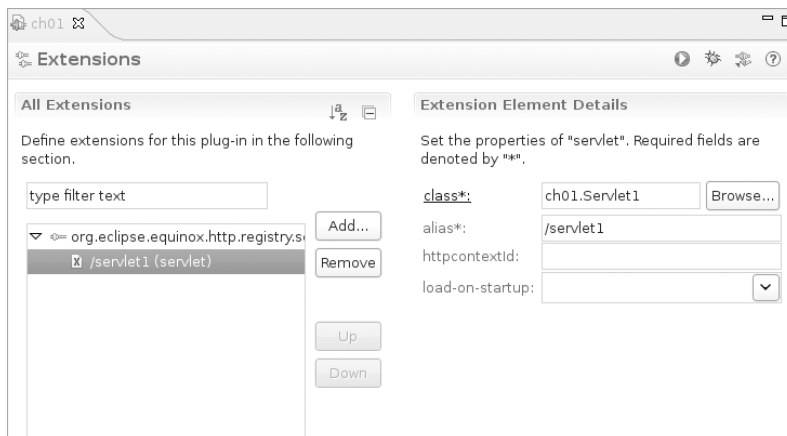
```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
  <extension
    point="org.eclipse.equinox.http.registry.servlets">
    <servlet
      alias="/servlet1"
      class="ch01.Servlet1">
    </servlet>
  </extension>
</plugin>
```

3. To implement the servlet class, click the class label link in the Extensions tab, as shown in Figure 1-6. This launches the New Java Class wizard.

---

**Note** You can also implement a new class manually by adding the class name (`ch01.Servlet1` in this example) to the plug-in manifest editor, and then right-clicking the plug-in project folder and selecting **New ▶ Java Class**.

---



**Figure 1-6.** Servlet extension point details

4. Enter the class information, select `javax.servlet.http.HttpServlet` as the superclass, and click Finish. The Java class will be created automatically.
5. Use the plug-in manifest editor to override the `doGet` method to return the headers of the HTTP request to the browser, as follows:

```
@Override
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException
{
    resp.setContentType("text/html");
    dumpHttpHeaders(req, resp.getWriter());
}

@SuppressWarnings("unchecked")
private void dumpHttpHeaders(HttpServletRequest req, PrintWriter out)
{
    out.println("URI:" + req.getRequestURI() + "<br/>");

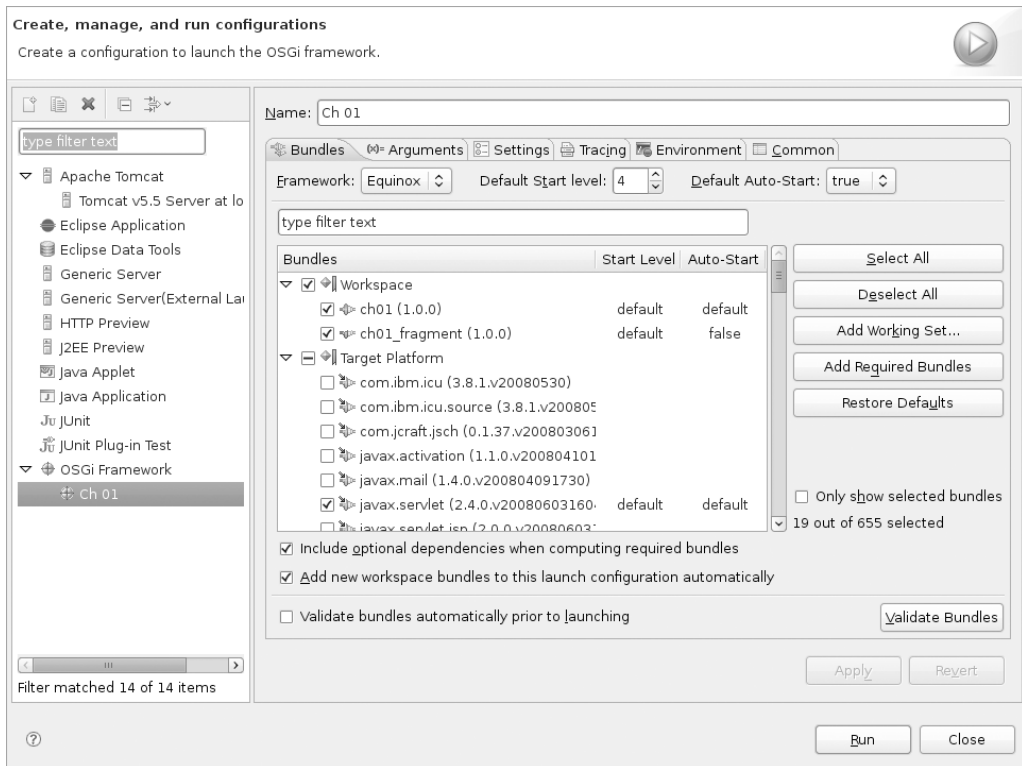
    Enumeration<String> names = req.getHeaderNames();

    while (names.hasMoreElements()) {
        final String name = names.nextElement();
        out.println(name + "=" + req.getHeader(name) + "<br/>");
    }
}
```

## Testing the Plug-in

Now that you've created the plug-in, you can test it. You'll see that the OSGi console is very useful for examining the OSGi framework and debugging missing dependencies.

1. From the main menu, select **Run** ► **Configurations** to open the Run Configurations dialog.
2. To create a new configuration under the OSGi framework, right-click and select **New**. Make sure your plug-in is selected in the Bundles list, under **Workspace**, as shown in Figure 1-7. You must also select all required bundles under **Target Platform**. To make sure all required bundles are selected, unselect all bundles under **Target Platform**, and then click **Add Required Bundles**. This will ensure only the required dependencies are used at runtime. Then click **Run**.



**Figure 1-7.** Run configuration dialog showing both the exercise plug-in (*ch01*) and the logging fragment (*ch01\_fragment*) discussed later in this chapter

### 3. Click the Arguments tab. Note the runtime arguments:

- `os ${target.os}`: The target operating system
- `ws ${target.ws}`: The target window system
- `arch ${target.arch}`: The target architecture
- `n1 ${target.n1}`: The locale
- `Console`: Start the OSGi console; handy for investigating the state of the system

Also note the VM argument:

- `osgi.noShutdown`: If true, the VM will not exit after the Eclipse application has ended; useful for examining the OSGi framework after the application has ended

When the plug-in runs, the console starts and is ready to receive user commands. This is a handy tool to inspect the state of the system. From the following output, you can see that Jetty started on port 80, which is the default in Windows.



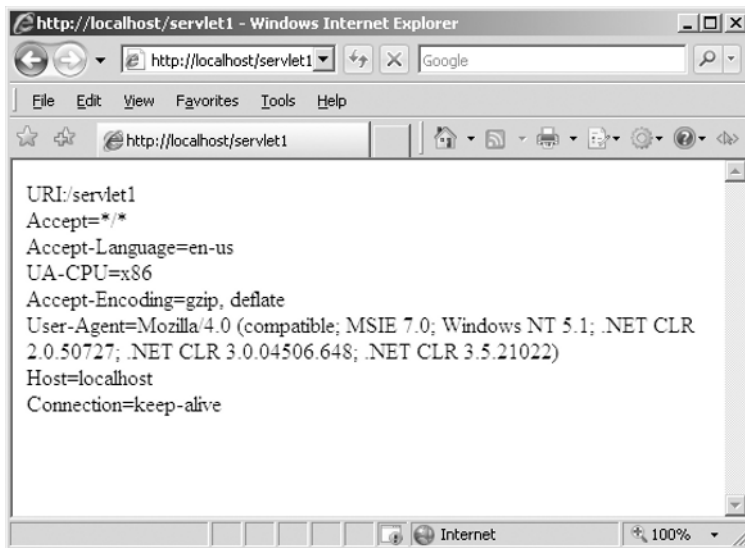
---

**Note** Under Linux environments, Jetty may fail to start on port 80, as ports lower than 1024 require `sysadmin` access. In that case, add the VM argument `-Dorg.eclipse.equinox.http.jetty.http.port=8080` to start Jetty on port 8080.

---

```
osgi> Jun 21, 2008 6:21:10 PM ch01.Activator start
INFO: Activator Start
Jun 21, 2008 6:21:10 PM org.mortbay.http.HttpServer doStart
INFO: Version Jetty/5.1.x
Jun 21, 2008 6:21:11 PM org.mortbay.util.Container start
INFO: Started org.eclipse.equinox.http.jetty.internal.Servlet25Handler@1a99561
Jun 21, 2008 6:21:11 PM org.mortbay.util.Container start
INFO: Started HttpContext[/,/]
Jun 21, 2008 6:21:11 PM org.mortbay.http.SocketListener start
INFO: Started SocketListener on 0.0.0.0:80
Jun 21, 2008 6:21:11 PM org.mortbay.util.Container start
INFO: Started org.mortbay.http.HttpServer@1ea0252
osgi>
```

4. Point the browser to `http://localhost/servlet1`. You should see the output shown in Figure 1-8.



**Figure 1-8.** *Output of the exercise*

## Using OSGi Console Commands

The console is a handy tool to inspect your plug-in and identify problems. The following are some of the most useful commands:

- `start [<id>|<name>]`: Starts a bundle given an ID or symbolic name
- `stop [<id>|<name>]`: Stops a bundle given an ID or symbolic name
- `install {URL}`: Adds a bundle given a URL for the current instance
- `uninstall [<id>|<name>]`: Removes a bundle given a URL for the current instance
- `ss`: Lists a short status of all the bundles registered in the current instance
- `help`: Shows information about all available commands

For example, to look at all the registered bundles, use the `ss` command, as follows:

```
osgi> ss
```

```
Framework is launched.
```

```
Id  State      Bundle
0  ACTIVE    org.eclipse.osgi_3.4.0.v20080605-1900
1  ACTIVE    org.eclipse.osgi.services_3.1.200.v20071203
2  ACTIVE    org.eclipse.core.jobs_3.4.0.v20080512
3  RESOLVED  ch01_fragment_1.0.0
                Master=12
4  ACTIVE    org.mortbay.jetty_5.1.14.v200806031611
5  ACTIVE    org.eclipse.core.runtime.compatibility.auth_3.2.100.v20070502
6  ACTIVE    org.eclipse.equinox.http.servlet_1.0.100.v20080427-0830
7  ACTIVE    org.eclipse.equinox.registry_3.4.0.v20080516-0950
                Fragments=16
8  ACTIVE    org.apache.commons.logging_1.0.4.v20080605-1930
9  ACTIVE    org.eclipse.core.runtime_3.4.0.v20080512
10 ACTIVE    org.eclipse.equinox.http.registry_1.0.100.v20080427-0830
11 ACTIVE    org.eclipse.core.contenttype_3.3.0.v20080604-1400
12 ACTIVE    org.apache.log4j_1.2.13.v200806030600
                Fragments=3
13 ACTIVE    javax.servlet_2.4.0.v200806031604
14 ACTIVE    org.eclipse.equinox.common_3.4.0.v20080421-2006
15 ACTIVE    ch01_1.0.0
16 RESOLVED  org.eclipse.core.runtime.compatibility.registry_3.2.200.v20070717
                Master=7
17 ACTIVE    org.eclipse.equinox.preferences_3.2.200.v20080421-2006
18 ACTIVE    org.eclipse.equinox.app_1.1.0.v20080421-2006
19 ACTIVE    org.eclipse.equinox.http.jetty_1.1.0.v20080425
```

```
osgi>
```

To start and stop your plug-in, simply use the bundle ID. (The bundle name can also be used, but who wants to type such long names?)

```
osgi> stop 15
271770 [OSGi Console] INFO  ch01.Activator  - Activator Stop
Jun 21, 2008 6:25:42 PM ch01.Activator stop
INFO: Activator Stop

osgi> start 15
Jun 21, 2008 6:25:47 PM ch01.Activator start
INFO: Activator Start
277188 [OSGi Console] INFO  ch01.Activator  - Activator Start
```

## Using Logging Services

Enabling a logging service within a plug-in is somewhat different from logging in a traditional Java application. It is a bit trickier because of the dynamic component nature of the runtime.

To enable log4j in a traditional Java application, for example, the developer would create a log4j.properties file in the project classpath, and then use statements such as the following:

```
// Log4J Logger
private static final Logger logger = Logger.getLogger(Activator.class);

public void start(BundleContext context) throws Exception {
    super.start(context);
    plugin = this;

    logger.info("Activator Start");
}
```

However, putting log4j.properties in the plug-in class will not work, because the OSGi framework manages a per-bundle classpath. It returns this message:

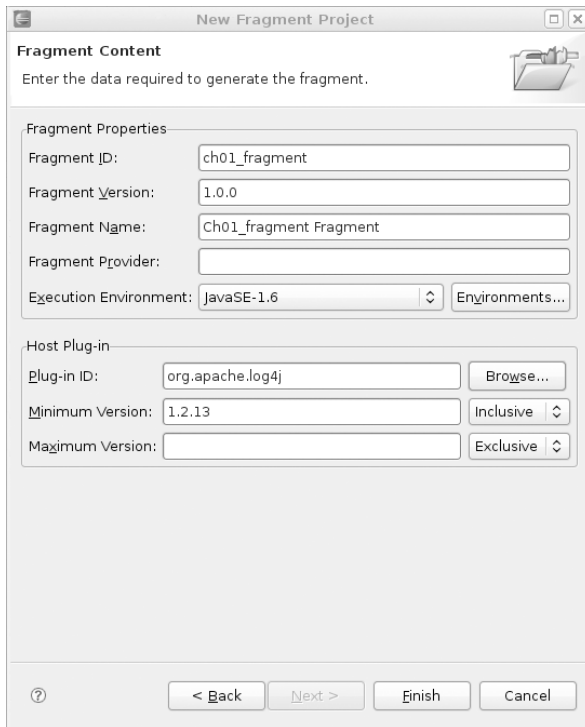
```
log4j:WARN No appenders could be found for logger (ch01.Activator).
log4j:WARN Please initialize the log4j system properly.
```

The solution is to have the plug-in find log4j.properties in the classpath at runtime and use it. However, this is a little tricky. One way to handle this is to create a plug-in fragment and set the host plug-in ID to org.apache.log4j, as shown in Figure 1-9. This fragment will have a log4j.properties file at the main level. Then, at runtime, the fragment will attach itself to the log4j bundle classpath, thus finding the required log4j.properties file. The fragment must also be included in the run configuration for the plug-in.

---

**Note** *Fragments* are separately packaged files whose contents are treated as if they were in the original plug-in archive file. They are useful for adding plug-in functionality, such as additional language translations, to an existing plug-in after it has been installed. Fragments are discussed further in Chapter 2.

---



**Figure 1-9.** Attaching a `log4j.properties` to the `log4j` bundle at runtime using a fragment

Here is the procedure to create the fragment for this example:

1. From the Eclipse IDE main menu, select **File** ► **New** ► **Other** ► **Plug-in Development** ► **Plug-in Fragment**.
2. In the New Fragment Project dialog, enter the plug-in information as shown in Figure 1-9. Make sure the host plug-in points to `org.apache.log4j`. You can click the Browse button to find and select that plug-in ID. Then click Finish.
3. In the fragment folder, add a `log4j.properties` file with the log configuration shown in the following fragment. To add a text file, right-click the fragment folder and select **New** ► **File**. Make sure the file name is `log4j.properties`.

```
# Set root logger level to debug and its only appender to default.
log4j.rootLogger=debug, default
```

```
# default is set to be a ConsoleAppender.
log4j.appender.default=org.apache.log4j.ConsoleAppender
```

```
# default uses PatternLayout.
log4j.appender.default.layout=org.apache.log4j.PatternLayout
log4j.appender.default.layout.ConversionPattern=%-4r [%t] %-5p %c %x - %m%n
```

This technique should enable the log4j logging service in your plug-in. However, if this seems too complicated, a simpler way is to use the Commons Logging service within the main plug-in, using this code:

```
// Commons Log
private static final Log log = LogFactory.getLog(Activator.class);

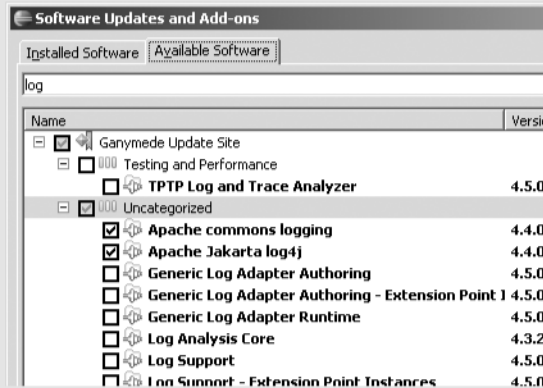
public void start(BundleContext context) throws Exception {
    super.start(context);
    plugin = this;
    log.info("Activator Start");
}
```

This fragment is much simpler; however, it will use the default Java logging service, which I personally dislike. It is up to you to choose the logging service that best fits your needs.

## USING THE ECLIPSE 3.4 SOFTWARE UPDATE MANAGER

The Eclipse 3.4 (Ganymede) distribution does not ship with a log service such as Apache log4j or Commons Logging. However, the new Software Update Manager can be used to quickly discover and install software, including logging plug-ins.

To use the Software Update Manager, from the Eclipse IDE main menu, select Help ► Software Updates. In the Software Updates and Add-ons dialog, click the Available Software tab. From here, you can search for and install the Jakarta log4j and Commons Logging plug-ins.



This concludes the exercise in this chapter. The goal of this exercise has been to provide an introduction to the power of the OSGi console and the basic plug-in life cycle, using a simple Jetty servlet extension point to listen for HTTP requests.

## Summary

This chapter introduced Eclipse RCP. The following are the important points to take away from this chapter:

- In today's heterogeneous software world, there is a quest for openness and extensibility. A platform that addresses interoperability challenges and supports collaboration is of critical importance.
- Eclipse provides a consistent feature set on multiple platforms. It allows developers to concentrate on the problem at hand, rather than the details of the specific platform.
- The plug-in architecture makes it possible for Eclipse to support many programming languages and development paradigms.
- Eclipse is open source, free, and fully supported.
- Eclipse is designed to be extensible and configurable.
- Eclipse is at the forefront of the software tools industry. This means that you can depend on it as a viable, industrial-strength tool for the foreseeable future.
- The foundation of RCP includes Equinox, the core platform, SWT, JFace, and the Eclipse workbench.
  - Equinox is an implementation of the OSGi framework, a dynamic component model for remote component management. This is something that is missing in stand-alone JVM environments.
  - The core runtime implements the basic plug-in model based on extension points declared in XML in a manifest file (`plugin.xml`). The 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.
  - SWT is a GUI toolkit with fast native access to multiple platform widget sets, providing a common API. It is designed for performance, native look and feel, and extensibility.
  - JFace is a window-system-independent GUI toolkit for handling many common programming tasks. It implements text, dialog, preference, and wizard frameworks, as well as actions and data viewers.
  - The workbench is the basic development environment in the Eclipse universe. It is divided into perspectives, viewers, editors, workspaces, and projects.