the eclipse series

Bestselling plug-in guide updated for Eclipse 3.4

Third Edition

eclipse Plug-ins

New chapter on GEF

Eric Clayberg ▪ Dan Rubel

SERIES EDITORS ▸ Erich Gamma · Lee Nackman · John Wiegand

Library of Congress Cataloging-in-Publication Data

Clayberg, Eric.
Eclipse : plug-ins / Eric Clayberg, Dan Rubel. -- 3rd ed.
p. cm.
Includes bibliographical references and index.
ISBN 0-321-55346-2 (pbk. : alk. paper)
1. Computer software--Development. 2. Eclipse (Electronic resource) 3. Java (Computer program language) I. Rubel, Dan. II. Title.

QA76.76.D47C574 2008
005.13'3--dc22 2008047409

ISBN-13: 978-0-321-55346-1
ISBN-10: 0-321-55346-2

Text printed in the United States on recycled paper at Courier in Stoughton, Massachusetts.

First printing, December 2008

Foreword

To the millions of developers, engineers, and users all over the world, Eclipse is an extensible platform for tool integration. To the hundreds of thousands of commercial customers using it to develop plug-ins or complete tool platforms, Eclipse represents a proven, reliable, scalable technology on which commercial products can be quickly designed, developed, and deployed.

To the thousands of students and researchers, Eclipse represents a stable platform for innovation, freedom, and experimentation. To all these individuals, groups, and organizations, Eclipse is a vendor-neutral platform for tool integration supported by a diverse Eclipse Ecosystem.

The Eclipse vendor-neutral platform is built on industry standards, which support a wide range of tools, platforms, and languages. The Eclipse Technology is royalty-free and has worldwide redistribution rights. The platform was designed from a clean slate to be extensible and to provide exemplary tools. Eclipse development is based on rules of open source engagements. This includes open, transparent, merit-based, and collaborative development. All individuals can participate and contribute. All plans are developed in the public arena. This platform and the open source development process creates an environment for creativity, originality, and freedom. Eclipse is unparalleled in today's software-tool environment.

The software-tool industry is undergoing massive changes from the commoditization of the technology to the company consolidation. New technology efforts are being redesigned, while a common set of tooling infrastructure is adopted as an industry standard. Successful developers and development paradigms are being challenged to adopt new skills and new, more efficient methods. Old business models are being challenged with free software, and new business models are being developed.

The software-tool industry is deeply appreciative of Eric Clayberg and Dan Rubel for this authoritative book. This book provides the knowledge base so that developers, engineers, and users can learn and use the Eclipse Technology. This enables them to respond to these technology and industry change agents.

Eric and Dan leverage long careers of building software tooling. They each have extensive experience with using Smalltalk for twenty years, Java for thirteen years, and Eclipse for nine years. They have developed extensive vendor and customer relationships that enable them to experience firsthand the necessary elements for building successful software. They are able to combine this direct knowledge of the technology with the experiences of the users to create a book that provides an in-depth description of the process to build commercial-quality Eclipse extensions.

This book provides an introduction and overview to the new developer of the entire process of plug-in development, including all the best practices to achieve high-quality results. This is a reference book for experienced Eclipse developers. It discusses the APIs and demonstrates many samples and examples. Detailed tutorials are provided for both new and experienced developers. Eric and Dan leverage their broad knowledge of user interface (UI) development and present the Eclipse SWT UI. This establishes the building blocks for all Eclipse UI development. These authors articulate the development challenges of building tool software and establish proven in-depth solutions to the problems.

If you are a developer, engineer, or user wishing to build or use Eclipse, this book provides both a foundation and reference. It also provides the intellectual foundation to contribute to the open source Eclipse project and to develop commercial software.

—Skip McGaughey

Foreword

In the 1990s, when Java was in its infancy, learning the Java class libraries involved studying a handful of classes in four or five packages. The Java class libraries have grown in size and complexity, presenting a significant problem to developers wishing to learn Java today. Just like Java, the Eclipse platform has necessarily grown over the years, and therefore considerably more time and effort is required to learn Eclipse 3.4 than its predecessors. One of the principles of the Eclipse platform is that a plug-in should integrate seamlessly with the workbench and with other plug-ins. To achieve seamless integration, it is necessary for plug-in developers to understand the best practices, conventions, and strategies related to building software for Eclipse. *Eclipse Plug-ins* covers everything you need to know to develop Eclipse plug-ins of which you will be proud.

Through the development of a **Favorites** plug-in, the Eclipse Standard Widget Toolkit (SWT) and JFace frameworks are thoroughly discussed, teaching you how to build professional-looking user interfaces such as views, editors, preferences pages, and dialogs. In addition to stock-in-trade subjects, such as user-interface design, lesser-understood Eclipse topics (for example, building features and product branding) are extensively covered, as well as the best discussion I have seen on using Ant to build a product from a single source that targets multiple versions of Eclipse.

Java developers new to Eclipse often have difficulty understanding the extension point mechanism and the critical link between a plug-in's declarative manifest and the Java code necessary to implement a plug-in's functional behavior. This book serves as a roadmap to using the Plug-in Development Environment (PDE) and the extension points defined by the Eclipse platform. It also provides the missing link that developers need to understand the

aspects of a plug-in that should be described in the manifest, how to develop a plug-in using existing extension points, and how to create new extension points to which other developers may further contribute.

When I first saw CodePro, I was both impressed with the productivity gains it brought to Eclipse and the extent to which its plug-ins integrated with the Eclipse platform. Having used CodePro for a while, it has become a part of my development toolkit that I cannot do without. By drawing on their extensive experience gained while developing CodePro, Eric and Dan have done an excellent job of capturing in this book those aspects of plug-in development necessary to create a high-quality and professional-looking Eclipse product.

—Simon Archer

Preface

When we were first exposed to Eclipse back in late 1999, we were struck by the magnitude of the problem IBM was trying to solve. IBM wanted to unify all its development environments on a single code base. At the time, the company was using a mix of technology composed of a hodgepodge of C/C++, Java, and Smalltalk.

Many of IBM's most important tools, including the award-winning VisualAge for Java IDE, were actually written in Smalltalk—a wonderful language for building sophisticated tools, but one that was rapidly losing market share to languages like Java. While IBM had one of the world's largest collections of Smalltalk developers, there wasn't a great deal of industry support for it outside of IBM, and there were very few independent software vendors (ISVs) qualified to create Smalltalk-based add-ons.

Meanwhile, Java was winning the hearts and minds of developers worldwide with its promise of easy portability across a wide range of platforms, while providing the rich application programming interface (API) needed to build the latest generation of Web-based business applications. More important, Java was an object-oriented (OO) language, which meant that IBM could leverage the large body of highly skilled object-oriented developers it had built up over the years of creating Smalltalk-based tools. In fact, IBM took its premiere Object Technology International (OTI) group, which had been responsible for creating IBM's VisualAge Smalltalk and VisualAge Java environments (VisualAge Smalltalk was the first of the VisualAge brand family and VisualAge Java was built using it), and tasked the group with creating a highly extensible integrated development environment (IDE) construction set based in Java. Eclipse was the happy result.

OTI was able to apply its highly evolved OO skills to produce an IDE unmatched in power, flexibility, and extensibility. The group was able to replicate most of the features that had made Smalltalk-based IDEs so popular the decade before, while simultaneously pushing the state of the art in IDE development ahead by an order of magnitude.

The Java world had never seen anything as powerful or as compelling as Eclipse, and it now stands, with Microsoft's .NET, as one of the world's premier development environments. That alone makes Eclipse a perfect platform for developers wishing to get their tools out to as wide an audience as possible. The fact that Eclipse is completely free and open source is icing on the cake. An open, extensible IDE base that is available for free to anyone with a computer is a powerful motivator to the prospective tool developer.

It certainly was to us. At Instantiations and earlier at ObjectShare, we had spent the better part of a decade as entrepreneurs focused on building add-on tools for various IDEs. We had started with building add-ons for Digitalk's Smalltalk/V, migrated to developing tools for IBM's VisualAge Smalltalk, and eventually ended up creating tools for IBM's VisualAge Java (including our award-winning VA Assist product and our jFactor product, one of the world's first Java refactoring tools). Every one of these environments provided a means to extend the IDE, but they were generally not well-documented and certainly not standardized in any way. Small market shares (relative to tools such as VisualBasic) and an eclectic user base also afflicted these environments and, by extension, us.

As an Advanced IBM Business Partner, we were fortunate to have built a long and trusted relationship with the folks at IBM responsible for the creation of Eclipse. That relationship meant that we were in a unique position to be briefed on the technology and to start using it on a daily basis nearly a year and half before the rest of the world even heard about it. When IBM finally announced Eclipse to the world in mid-2001, our team at Instantiations had built some of the first demo applications IBM had to show. Later that year when IBM released its first Eclipse-based commercial tool, WebSphere Studio Application Developer v4.0 (v4.0 so that it synchronized with its then current VisualAge for Java v4.0), our CodePro product became the very first commercial add-on available for it (and for Eclipse in general) on the same day.

Currently, the CodePro product adds hundreds of enhancements to Eclipse and any Eclipse-based IDE. Developing CodePro over the last several years has provided us with an opportunity to learn the details of Eclipse development at a level matched by very few others (with the obvious exception of the IBM and OTI developers, who eat, sleep, and breathe this stuff on a daily basis). CodePro has also served as a testbed for many of the ideas and tech-

niques presented in this book, providing us with a unique perspective from which to write.

Goals of the Book

This book, originally titled *Eclipse: Building Commercial-Quality Plug-ins*, provides an in-depth description of the process involved in building commercial-quality extensions for the Eclipse and the IBM Software Development Platform (SDP)—IBM's commercial version of Eclipse—development environments. To us, "commercial-quality" is synonymous with "commercial-grade" or "high-quality." Producing a *commercial-quality* plug-in means going above and beyond the minimal requirements needed to integrate with Eclipse. It means attending to all those details that contribute to the "fit and polish" of a commercial offering.

In the world of Eclipse plug-ins, very few people take the time to really go the extra mile, and most plug-ins fall into the open source, amateur category. For folks interested in producing high-quality plug-ins (which would certainly be the case for any software company wanting to develop Eclipse-based products), there are many details to consider. Our book is meant to encompass the entire process of plug-in development, including all the extra things that need to be done to achieve high-quality results. This book has several complementary goals:

- Provide a quick introduction to using Eclipse for new users
- Provide a reference for experienced Eclipse users wishing to expand their knowledge and improve the quality of their Eclipse-based products
- Provide a detailed tutorial on creating sophisticated Eclipse plug-ins suitable for new and experienced users

The first three chapters introduce the Eclipse development environment and outline the process of building a simple plug-in. The intention of these chapters is to help developers new to Eclipse quickly pull together a plug-in they can use to experiment with.

The first chapter, in particular, introduces the reader to the minimum set of Eclipse tools that he or she will need to build plug-ins. It is a fairly quick overview of the Eclipse IDE and relevant tools (one could write an entire book on that topic alone), and we would expect expert Eclipse users to skip that chapter entirely.

The second chapter introduces the example that we will use throughout most of the book and provides a very quick introduction to building a working plug-in from start to finish. The third chapter presents a high-level overview of the Eclipse architecture and the structure of plug-ins and extension points.

The fourth and fifth chapters cover the Standard Widget Toolkit (SWT) and JFace, which are the building blocks for all Eclipse user interfaces (UIs). These chapters can act as a stand-alone reference; they are intended to provide just enough detail to get you going. Both of these topics are rich enough to warrant entire books and several are currently available.

The subsequent chapters, comprising the bulk of this book, focus on describing each of the various aspects of plug-in development and providing the reader with in-depth knowledge of how to solve the various challenges involved. Each chapter focuses on a different aspect of the problem, and includes an overview, a detailed description, a discussion of challenges and solutions, diagrams, screenshots, cookbook-style code examples, relevant API listings, and a summary.

We have structured the book so that the most important material required for every plug-in project appears in the first half of it. Some of the packaging- and building-oriented material is placed at the end (for example, features and product builds). This organizational scheme left several topics that, while not critical to every plug-in, were important to the creation of commercial-quality plug-ins. These topics have been placed in the second half of the book in an order based on the importance of each and how it related to earlier material. Internationalization, for example, is one of those topics. It isn't critical, and it isn't even all that complicated when you get right down to it. It is, however, important to the book's premise, so we felt it was a topic we needed to include. Since we aren't assuming that the reader is an Eclipse expert (or even a plug-in developer), we have tried to take the reader through each of the important steps in as much detail as possible. While it is true that this is somewhat introductory, it is also an area that most plug-in developers totally ignore and have little or no experience with.

Sometimes a developer needs a quick solution, while at other times that same developer needs to gain in-depth knowledge about a particular aspect of development. The intent is to provide several different ways for the reader to absorb and use the information so that both needs can be addressed. Relevant APIs are included in several of the chapters so that the book can be used as a stand-alone reference during development without requiring the reader to look up those APIs in the IDE. Most API descriptions are copied or paraphrased from the Eclipse platform Javadoc.

As the originators of Eclipse and a major consumer of Eclipse-based technology, IBM is justifiably concerned that new plug-ins meet the same high-quality standards that IBM adheres to. To that end, IBM has established a rigorous *Ready for Rational Software* (RFRS) certification program meant to

ensure the availability of high-quality add-ons to Eclipse and the IBM Software Development Platform. RFRS certification should be one of the ultimate goals for anyone wishing to build and market Eclipse plug-ins. Every chapter covers any relevant RFRS certification criteria and strategies.

The examples provided as part of the chapters describe building various aspects of a concrete Eclipse plug-in that you will see evolve over the course of the book. When used as a reference rather than read cover-to-cover, you will typically start to look in one chapter for issues that are covered in another. To facilitate this type of searching, every chapter contains numerous cross-references to related material that appears in other chapters.

Intended Audience

The audience for this book includes Java tool developers wishing to build products that integrate with Eclipse and other Eclipse-based products, relatively advanced Eclipse users wishing to customize their environments, or anyone who is curious about what makes Eclipse tick. You do not need to be an expert Eclipse user to make use of this book because we introduce most of what you need to know to use Eclipse in Chapter 1, Using Eclipse Tools. While we don't assume any preexisting Eclipse knowledge, we do anticipate that the reader is a fairly seasoned developer with a good grasp of Java and at least a cursory knowledge of extensible markup language (XML).

Conventions Used in This Book

The following formatting conventions are used throughout the book.

Bold—the names of UI elements such as menus, buttons, field labels, tabs, and window titles

Italic—emphasize new terms and Web site addresses

`Courier`—code examples, references to class and method names, and filenames

`Courier Bold`—emphasize code fragments

"Quoted text"—quotation marks surrounding text indicates words to be entered by the user

What's New in the Third Edition

In this edition, we use the same **Favorites** view example as in the first and second editions, but have reworked much of the content and recreated the code from scratch. Some Eclipse concepts, such as views and editors, are similar but with additional functionality and capabilities; other areas, such as commands, GEF and PDE Build have been added. The following are some of the major changes in this third edition:

Eclipse Command Framework
The Eclipse command framework replaces the older Action framework. Throughout the book, use of the older Action framework has been replaced with new content describing how to accomplish the same thing with the new command framework. This is most obvious in Chapter 6 where the first half of the chapter is entirely devoted to the command framework. Chapter 7 and Chapter 8 also have lots of new material describing use of commands with views and editors.

Eclipse 3.4 and Java 5
All of the screen shots, text and code examples throughout the book have been updated to use the latest Eclipse 3.4 API and Java 5 syntax. Chapter 1 has been overhauled to include descriptions of new capabilities in Eclipse 3.4 including a new overview of using Mylyn and discussion of new preferences. Both Chapter 1 and Chapter 2 cover cool PDE and SWT tools available in Eclipse 3.4.

New GEF Chapter
GEF, the Graphical Editing Framework from Eclipse.org, provides a toolkit for building dynamic interactive graphical user interface elements. Chapter 20 takes you step by step through the process of building a GEF-based view for graphically presenting the relationships between the favorites items and their underlying resources. We then go further, building a GEF-based editor with the ability to add, move, resize, and delete the graphical elements representing those favorites items.

Put PDE Build through its paces
Over the past several years, the PDE build process has been steadily maturing. The proprietary Ant scripts in Chapter 19 of earlier versions of our book have been completely replaced with step-by-step instructions for getting an Eclipse PDE Build process up and running to automate assembly of your product for distribution.

New "p2" update site creation description
"p2" debuts in Eclipse 3.4, replacing the older Update Manager. We take you through the process of using update sites and then building your own in Section 18.3, Update Sites, on page 679.

Acknowledgments

The authors would like to thank all those who have had a hand in putting this book together or who gave us their support and encouragement throughout the many months it took to create.

To our comrades at Instantiations, who gave us the time and encouragement to work on this book: Rick Abbott, Brent Caldwell, Devon Carew, Jim Christensen, Taylor Corey, Dianne Engles, Marta George, Nick Gilman, Seth Hollyman, Mark Johnson, Ed Klimas, Tina Kvavle, Alexander Lobas, Warren Martin, David Masulis, Nancy McClure, Steve Messick, Alexander Mitin, Gina Nebling, John O'Keefe, Keerti Parthasarathy, Phil Quitslund, Mark Russell, Rob Ryan, Andrey Sablin, Balaji Saireddy, Konstantin Scheglov, Chuck Shawan, Bryan Shepherd, Julie Taylor, Mike Taylor, Solveig Viste, Brian Wilkerson, and Jaime Wren.

A special thanks to Jaime Wren, who provided much of the basic research and initial content for Chapter 20.

To our agent, Laura Lewin, and the staff at Studio B, who encouraged us from day one and worked tirelessly on our behalf.

To our editors, Greg Doench and John Neidhart, our production editors, Elizabeth Ryan and Kathleen Caren, our copy editors, Marilyn Rash and Camie Goffi, our editorial assistants, Michelle Housley and Mary Kate Murray, our art director, Sandra Schroeder, our marketing managers, Brandon Prebynski and Beth Wickenhiser, and the staff at Pearson, for their encouragement and tremendous efforts in preparing this book for production.

To Simon Archer and Robert Konigsberg who contributed an unparalleled number of changes and suggestions to various editions of the book, and helped us improve them in almost every dimension.

To Linda Barney who helped us polish and edit the second edition.

To our technical reviewers who helped us enhance the book in many ways: Matt Lavin, Kevin Hammond, Mark Russell, Keerti Parthasarathy, Jaime Wren, Joe Bowbeer, Brian Wilkerson, Joe Winchester, David Whiteman, Boris Pruesmann, Balaji Saireddy, and Raphael Enns.

To the many readers of the first and second editions who contributed errata that have gone into this third edition: Bruce Gruenbaum, Tony Saveski, James Carroll, Tony Weddle, Karen Ploski, Lori Kissell, Brian Vosburgh, Peter Nye, Chris Lott, David Watkins, Simon Archer, Mike Wilkins, Brian Penn, Bernd Essann, Eric Hein, Dave Hewitson, Frank Parrott, Catherine Suess,

William Beebe, Janine Kovack, David Masulis, Jim Norris, Jim Wingard, Roy Johnston, David Karr, Chris Gage, Paul Tamminga, Asim Ullah, and Ebru Aktuna.

To the series editors, Erich Gamma, Lee Nackman, and John Weigand, for their thoughtful comments and for their ongoing efforts to make Eclipse the best development environment in the world.

We would also like to thank our wives, Karen and Kathy, for their endless patience, and our children, Beth, Lauren, Lee, and David, for their endless inspiration.

About the Authors

Eric Clayberg is Senior Vice President for Product Development for Instantiations, Inc. Eric is a seasoned software technologist, product developer, entrepreneur, and manager with more than seventeen years of commercial software development experience, including twelve years of experience with Java and nine years with Eclipse. He is the primary author and architect of more than a dozen commercial Java and Smalltalk add-on products, including the popular WindowBuilder Pro, CodePro, and the award-winning VA Assist product lines. He has a B.S. from MIT, an MBA from Harvard, and has cofounded two successful software companies—ObjectShare and Instantiations.

Dan Rubel is Chief Technology Officer for Instantiations, Inc. He is an entrepreneur and an expert in the design and application of OO technologies with more than fifteen years of commercial software development experience, including thirteen years of experience with Java and nine years with Eclipse. He is the architect and product manager for several successful commercial products, including RCP Developer, WindowTester, jFactor and jKit, and has played key design and leadership roles in other commercial products such as VA Assist, and CodePro. He has a B.S. from Bucknell and is a cofounder of Instantiations.

Instantiations is an Advanced IBM Business Partner and developer of many commercial add-ons for Eclipse and IBM's VisualAge, WebSphere, and Rational product lines. Instantiations is a member of the Eclipse Foundation and a contributor to the Eclipse open source effort with responsibility for the Eclipse Collaboration Tools project known as Koi and for the Eclipse Pollinate project (Beehive).

How to Contact Us

While we have made every effort to make sure that the material in this book is timely and accurate, Eclipse is a rapidly moving target and it is quite possible that you may encounter differences between what we present here and what you experience using Eclipse. The Eclipse UI has evolved considerably over the years, and the latest 3.4 release and upcoming 3.5 release are no exceptions. While we have targeted it at Eclipse 3.4 and used it for all of our examples, this book was completed after Eclipse 3.4 was finished and during the initial phases of development of Eclipse 3.5. If you are using an older or newer version of Eclipse, this means that you may encounter various views, dialogs, and wizards that are subtly different from the screenshots herein.

- Questions about the book's technical content should be addressed to: *info@qualityeclipse.com*
- Sales questions should be addressed to Addison-Wesley at: *www.informit.com/store/sales.aspx*
- Source code for the projects presented can be found at: *www.qualityeclipse.com/projects*
- Errata can be found at: *www.qualityeclipse.com/errata*
- Tools used and described can be found at: *www.qualityeclipse.com/tools*

CHAPTER 2

A Simple Plug-in Example

Before covering the Eclipse infrastructure (see Chapter 3) and each area of plug-in construction in-depth, it is useful to create a simple plug-in on which discussion and examples can be based. This chapter takes a step-by-step approach to creating a simple but fully operational plug-in that will be enhanced bit-by-bit during the course of this book. This process provides valuable firsthand experience using the Eclipse IDE and touches on every aspect of building and maintaining a plug-in.

2.1 The Favorites Plug-in

The **Favorites** plug-in, which you'll build over the course of this book, displays a list of resources, lets you add and remove resources from the list, easily opens an editor on a selected resource, updates the list automatically as a result of events elsewhere in the system, and more. Subsequent chapters discuss aspects of plug-in development in terms of enhancements to the **Favorites** plug-in.

This chapter starts the process by covering the creation of the **Favorites** plug-in in its simplest form using the following steps:

- Creating a plug-in project
- Reviewing the generated code
- Building a product
- Installing and running the product

2.2 Creating a Plug-in Project

The first step is to create a plug-in project using the Eclipse **New Project** wizard. In addition to creating a new project, this wizard has a number of different code generation options, such as views, editors, and actions, for creating sample plug-in code. To keep things simple and focus only on the essentials of plug-in creation, select the **Plug-in with a view** option, which is discussed in the next subsection.

2.2.1 New Plug-in Project wizard

From the **File** menu, select **New > Project** to launch the **New Project** wizard (see Figure 2–1). On this first page of the wizard, select **Plug-in Project** from the list and then click the **Next** button.

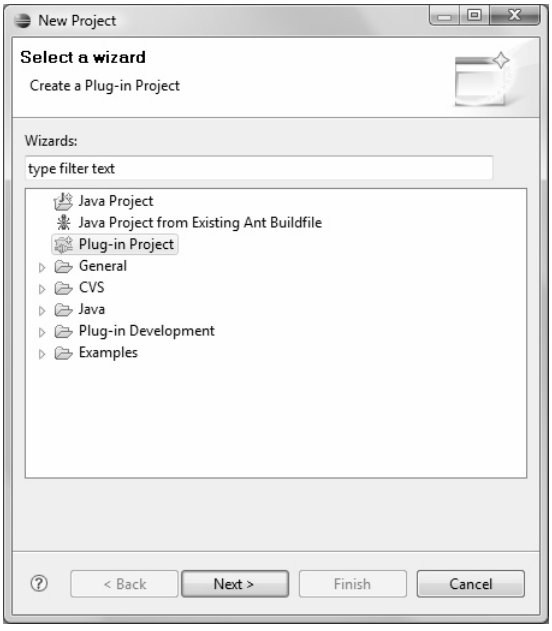

Figure 2–1 New Project wizard page 1—selecting a project type.

On the next page of the wizard (see Figure 2–2), enter the name of the project; in this case, it's `com.qualityeclipse.favorites`, which is the same as the Favorites plug-in identifier. Chapter 3, Eclipse Infrastructure, discusses

plug-in identifiers and other aspects of plug-in architecture in more detail. Fill in the other fields as shown and then click the **Next** button.

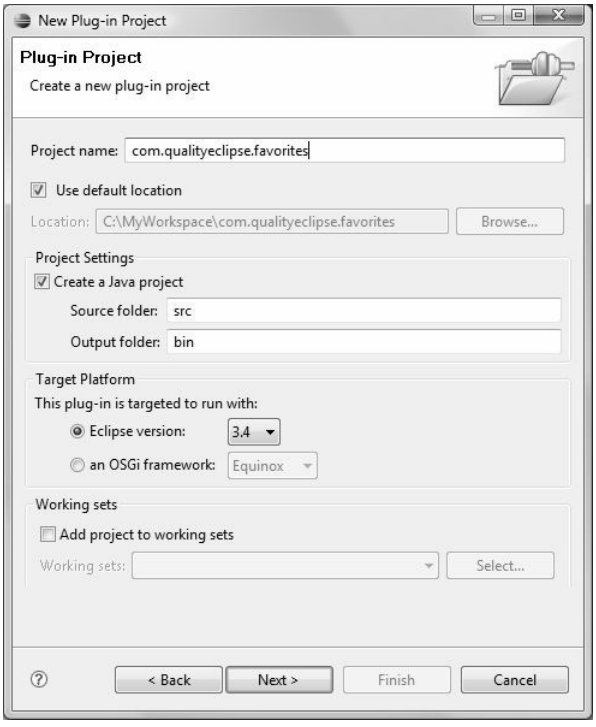

Figure 2–2 New Project wizard page 2—naming the project..

Tip: A project can be named anything, but it is easier to name it the same as the plug-in identifier. By convention, this is the plug-in project-naming scheme that the Eclipse organization uses for most of its work. Because of this, the **New Project** wizard assumes that the project name and the plug-in identifier are the same.

2.2.2 Define the plug-in

Every plug-in has a `META-INF/MANIFEST.MF` file. In addition, it may contain a `plugin.xml` file and/or a Java class that represents the plug-in programmatically. The next wizard page displays options for generating both the plug-in manifest and plug-in Java class. Supply the **Plug-in ID**, **Plug-in Version**, **Plug-in Name** and more for the plug-in as shown in Figure 2–3 then click the **Next** button.

Figure 2–3 New Project wizard page 3—describing the plug-in.

Next, the **New Plug-in Project** wizard next displays the various plug-in pieces that can be automatically generated by the wizard (see Figure 2–4). There are many different options on this page for generating quite a bit of sample code. It is useful to try out each option and review the code that is generated; however for this example, select **Plug-in with a view** and then click the **Next** button.

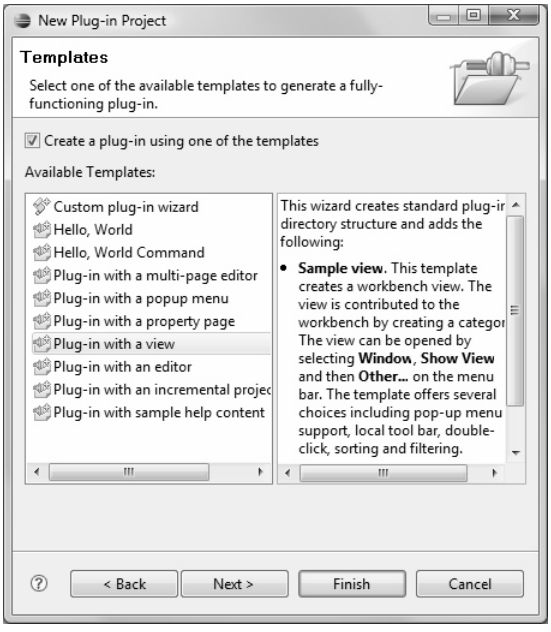

Figure 2–4 New Plug-in Project wizard page 4—selecting a plug-in type.

2.2.3 Define the view

Selecting view code generation options is the next step in this process. Enter the values for this page (see Figure 2–5), uncheck the **Add the view to the resource perspective** and **Add context help to the view** (Eclipse 3.4 only) checkboxes to simplify the generated plug-in manifest file.

If you are in Eclipse 3.3, then click the **Next** button and uncheck each of the code generation options (see Figure 2–6). Each of these checkboxes represents code that could be generated as part of the **Favorites** view. These are covered in subsequent chapters. This wizard page has been removed in Eclipse 3.4 and thus the `FavoritesView` class generated by the wizard will contain more code than is shown in the book (see Section 2.3.3, The Favorites view, on page 84).

When you click the **Finish** button, the new plug-in project is created and the plug-in manifest editor is automatically opened (see Figure 2–9).

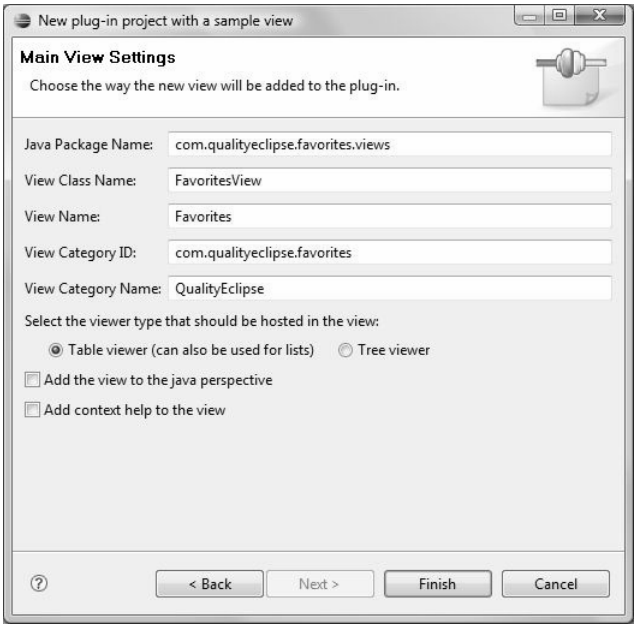

Figure 2–5 New Plug-in Project wizard page 5—defining the view.

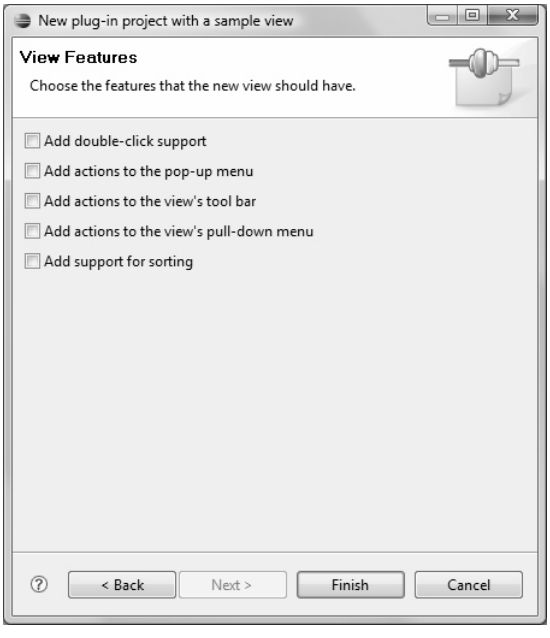

Figure 2–6 New Plug-in Project wizard page 6—code generation options for the view (Eclipse 3.3 only)

2.3 Reviewing the Generated Code

Reviewing the code generated by the **New Plug-in Project** wizard provides a brief look at the following major parts comprising the sample plug-in.

- The plug-in manifests
- The plug-in class
- The **Favorites** view

2.3.1 The Plug-in manifests

The plug-in manifest editor shows the contents of the two plug-in manifest files, `META-INF/MANIFEST.MF` and `plugin.xml`, which define how this plug-in relates to all the others in the system. This editor is automatically opened to its first page (see Figure 2–9) as a result of creating a new plug-in project. If the plug-in manifest editor is closed, double-clicking on either the `META-INF/MANIFEST.MF` or the `plugin.xml` file reopens the editor. The following is an overview of the manifest editor, while more detail on the plug-in manifest itself can be found in Chapter 3.

Although the editor is a convenient way to modify the plug-in's description, it's still useful to peek at the source behind the scenes to see how the editor's different parts relate to the underlying code. Click the **MANIFEST.MF** tab to display the source of the `META-INF/MANIFEST.MF` file that defines the runtime aspects of this plug-in (see Figure 2–7). The first two lines define it as an OSGi manifest file (see Section 3.3, Plug-in Manifest, on page 113). Subsequent lines specify plug-in name, version, identifier, classpath, and plug-ins on which this plug-in depends. All these aspects are editable using other pages in the plug-in manifest editor.

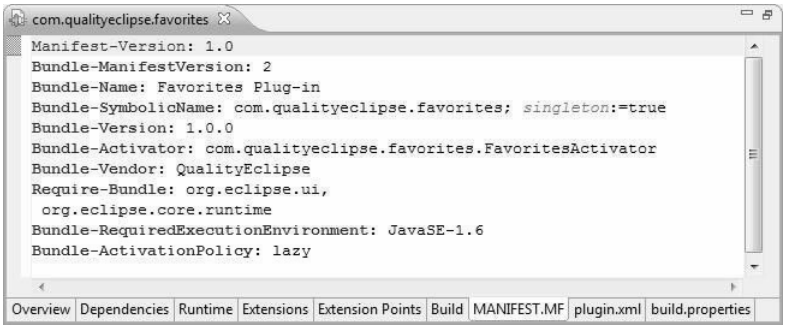

Figure 2–7 Plug-in manifest editor MANIFEST.MF page.

New in Eclipse 3.4 The `Eclipse-LazyStart: true` directive in the `MANIFEST.MF` file has been replaced with `Bundle-ActivationPolicy: lazy`. Both directives have the same semantics; only the name has changed.

Clicking on the **plugin.xml** tab of the editor displays the `plugin.xml` file that defines the extension aspects of this plug-in (see Figure 2–8). The first line declares this to be an XML file, while subsequent lines specify plug-in extensions.

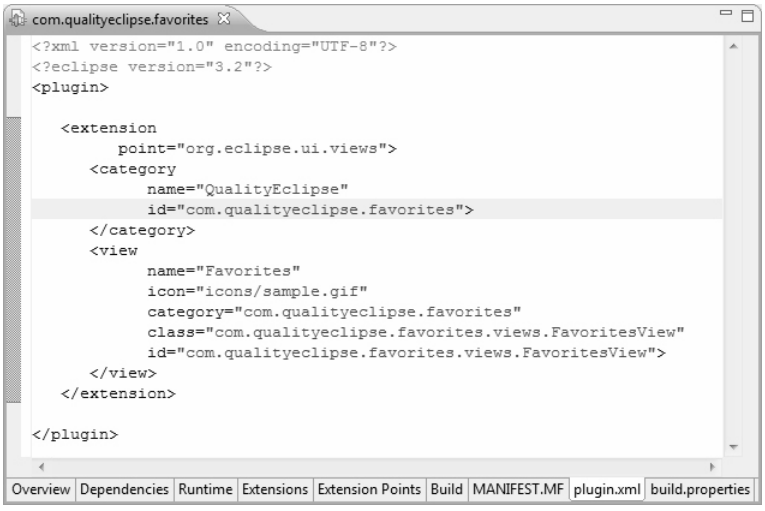

Figure 2–8 Plug-in manifest editor plugin.xml page.

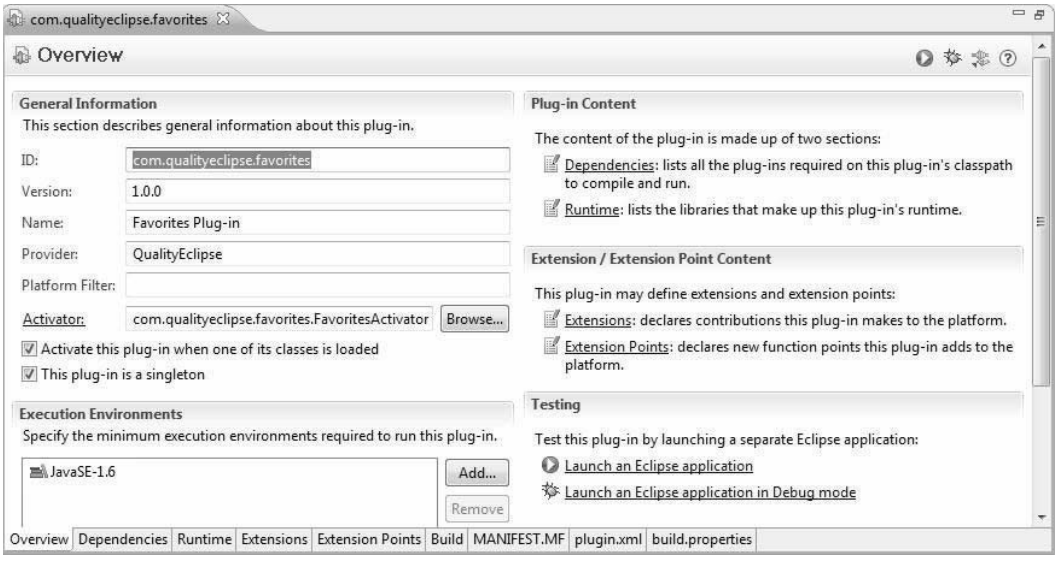

Figure 2–9 Plug-in manifest editor Overview page.
The **This plug-in is a singleton** option was added in Eclipse 3.4.

The **Overview** page of the manifest editor shows a summary of the plug-in manifest (see Figure 2–9). The section on this page describing general information, such as the plug-in identifier (ID), version, name, class, and provider, corresponds to the first chunk of source in the `META-INF/MANIFEST.MF` file:

```
Bundle-Name: Favorites Plug-in
Bundle-SymbolicName: com.qualityeclipse.favorites; singleton:=true
Bundle-Version: 1.0.0
Bundle-Activator: com.qualityeclipse.favorites.FavoritesActivator
Bundle-Vendor: QualityEclipse
Bundle-RequiredExecutionEnvironment: JavaSE-1.5
```

You can edit the information on the **Overview** page or switch to the **MANIFEST.MF** page and edit the source directly.

Tip: Making changes to any page other than the **plugin.xml** and **MANIFEST.MF** pages may cause the manifest editor to reformat the source. If you are particular about the formatting of either manifest file, then either use only the **plugin.xml** and **MANIFEST.MF** pages to perform editing or use another editor.

Caution: The formatting rules of `META-INF/MANIFEST.MF` include some quite nonintuitive rules related to line length and line wrapping. Edit `plugin.xml` with care, and `META-INF/MANIFEST.MF` with caution!

The reliance of this plug-in on other plug-ins in the system appears on the **Dependencies** page of the plug-in manifest editor (see Figure 2–10).

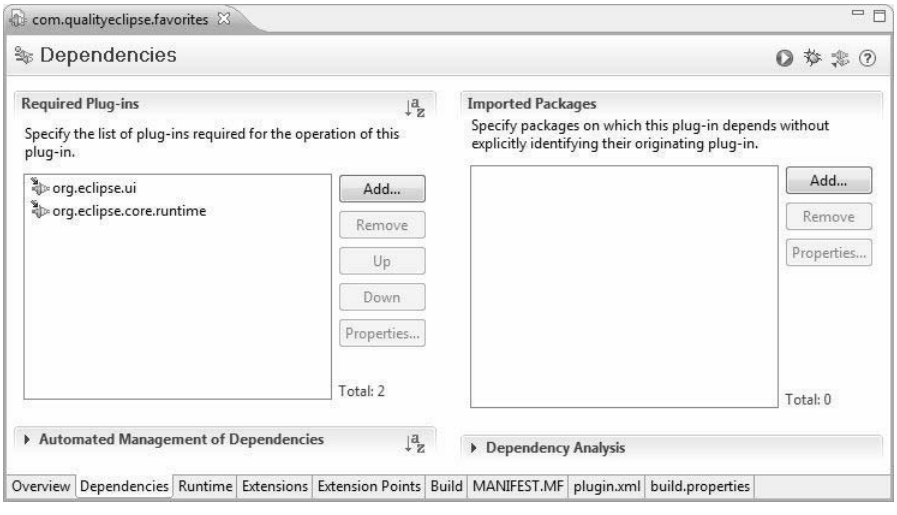

Figure 2–10 Plug-in manifest editor Dependencies page.

This corresponds to the `Require-Bundle` chunk of source in the `META-INF/MANIFEST.MF` file:

```
Require-Bundle: org.eclipse.ui,
 org.eclipse.core.runtime
```

For the **Favorites** plug-in, this section indicates a dependency on the `org.eclipse.core.runtime` and `org.eclipse.ui` plug-ins. This dependency declaration differs from the **Favorites** project's Java build path (also known as the compile-time classpath) because the Java build path is a compile-time artifact, while the plug-in dependency declaration comes into play during plug-in execution. Because the project was created as a plug-in project and has the `org.eclipse.pde.PluginNature` nature (see Section 14.3, Natures, on page 561 for more on project natures), any changes to this dependency list will automatically be reflected in the Java build path, but not the reverse. If these two aspects of your plug-in get out of sync, then you can have a plug-in that compiles and builds but does not execute properly.

Tip: Edit this dependency list rather than the Java build path so that the two are automatically always in sync.

Alternatively, the dependencies could have been expressed as **Imported Packages** on the **Dependencies** page of the manifest editor (see Figure 2–10 and the end of Section 3.3.3, Plug-in dependencies, on page 116). This would correspond to an `Import-Package` chunk of source in the `META-INF/MANIFEST.MF` file looking something like this:

```
Import-Package: org.eclipse.ui.views,
 org.eclipse.core.runtime.model
```

The **Runtime** page of the manifest editor (see Figure 2–11) corresponds to the `Bundle-ClassPath` chunk of source in the `META-INF/MANIFEST.MF` file, which defines what libraries are delivered with the plug-in and used by the plug-in during execution, what package prefixes are used within each library (used to speed up plug-in loading time), and whether other plug-ins can reference the code in the library (see Section 21.2.5, Related plug-ins, on page 783 for more on package visibility). For the **Favorites** plug-in, all the code is contained in the `com.qualityeclipse.favorites_1.0.0.jar` itself, so no `Bundle-ClassPath` declaration is necessary.

The **Favorites** plug-in does not export any packages for other plug-ins to use or extend.

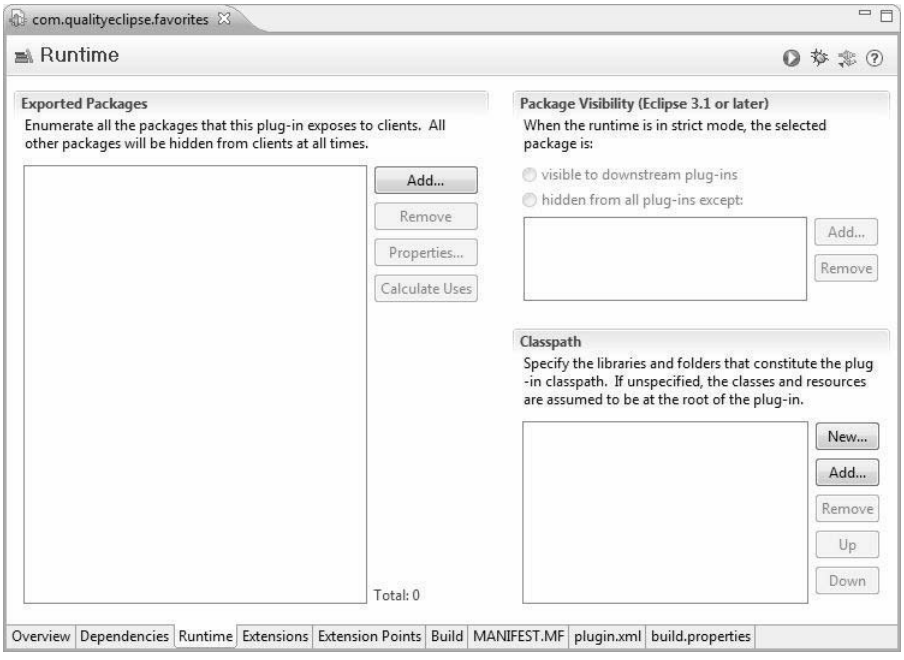

Figure 2–11 Plug-in manifest editor Runtime page.

The **Extensions** page (see Figure 2–12) displays how this plug-in augments the functionality already provided by other plug-ins in the system, and corresponds to the `<extension point="org.eclipse.ui.views">` chunk of XML in the `plugin.xml` file:

```
<extension
      point="org.eclipse.ui.views">
   <category
         name="QualityEclipse"
         id="com.qualityeclipse.favorites">
   </category>
   <view
         name="Favorites"
         icon="icons/sample.gif"
         category="com.qualityeclipse.favorites"
         class="com.qualityeclipse.favorites.views.FavoritesView"
         id="com.qualityeclipse.favorites.views.FavoritesView">
   </view>
</extension>
```

The **Favorites** plug-in declares an extension to the `org.eclipse.ui` plug-in using the `org.eclipse.ui.views` extension point by providing an additional category of views named **QualityEclipse** and a new view in that cat-

egory named **Favorites**. Selecting an item in the tree on the left in the **Extensions** page causes the properties for that item to appear on the right. In this case, selecting **Favorites** (**view**) on the **Extensions** page displays the name, identifier, class, and more information about the **Favorites** view that is being declared. This corresponds to the XML attributes defined in the `<view>` chunk of XML shown previously.

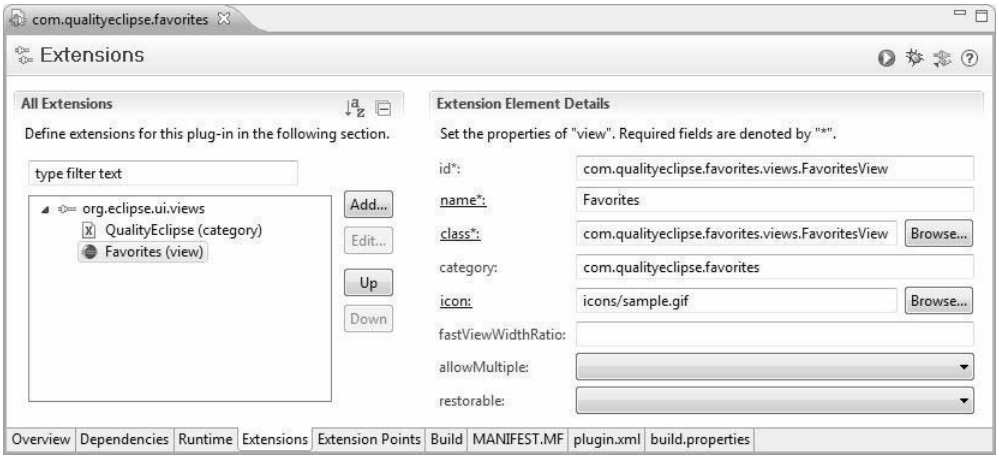

Figure 2–12 Plug-in manifest editor Extensions page.

Finally, the **Extension Points** page of the manifest editor (see Figure 2–13) facilitates the definition of new extension points so that other plug-ins can augment the functionality provided by this plug-in. At this time, the **Favorites** plug-in doesn't define any extension points and therefore cannot be augmented by other plug-ins (see Section 17.2, Defining an Extension Point, on page 639).

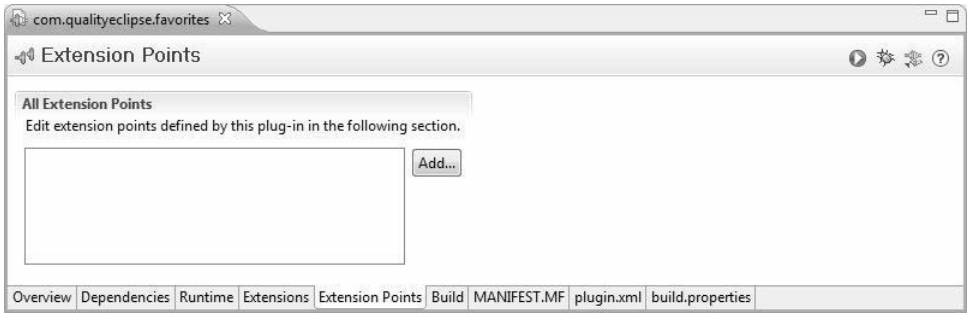

Figure 2–13 Plug-in manifest editor Extension Points page.

2.3.2 The Activator or Plug-in class

Every plug-in optionally can declare a class that represents the plug-in from a programmatic standpoint as displayed on the manifest editor's **Overview** page (see Figure 2–9). This class is referred to as an **Activator** (or, in earlier versions of Eclipse, a **Plug-in** class). In the **Favorites** plug-in, this class is named `com.qualityeclipse.favorites.FavoritesActivator`.

```
package com.qualityeclipse.favorites;

import org.eclipse.jface.resource.ImageDescriptor;
import org.eclipse.ui.plugin.AbstractUIPlugin;
import org.osgi.framework.BundleContext;

/**
 * The activator class controls the plug-in life cycle
 */
public class FavoritesActivator extends AbstractUIPlugin {

   // The plug-in ID
   public static final String PLUGIN_ID
      = "com.qualityeclipse.favorites";

   // The shared instance
   private static FavoritesActivator plugin;

   /**
    * The constructor
    */
   public FavoritesActivator() {
   }

   /**
    * This method is called upon plug-in activation.
    */
   public void start(BundleContext context) throws Exception {
      super.start(context);
      plugin = this;
   }

   /**
    * This method is called when the plug-in is stopped.
    */
   public void stop(BundleContext context) throws Exception {
      plugin = null;
      super.stop(context);
   }

   /**
    * Returns the shared instance
    */
   public static FavoritesActivator getDefault() {
      return plugin;
   }
```

```
    /**
     * Returns an image descriptor for the image file at the given
     * plug-in relative path
     *
     * @param path the path
     * @return the image descriptor
     */
    public static ImageDescriptor getImageDescriptor(String path) {
       return imageDescriptorFromPlugin(PLUGIN_ID, path);
    }
}
```

If the `Bundle-ActivationPolicy` in the `META-INF/MANIFEST.MF` file is `lazy`, then when the plug-in is activated, the Eclipse system instantiates the activator class before loading any other classes in it. This corresponds to the "Activate this plug-in when one of its classes is loaded." checkbox on the **Overview** page of the manifest editor (see Figure 2–9). This single activator class instance is used by the Eclipse system throughout the life of the plug-in and no other instance is created.

Tip: For more background on Bundle-ActivationPolicy, see http://wiki.eclipse.org/Lazy_Start_Bundles.

Typically, activator classes declare a static field to reference this singleton so that it can be easily shared throughout the plug-in as needed. In this case, the **Favorites** plug-in defines a field named `plugin` that is assigned in the start method and accessed using the `getDefault` method.

Tip: The Eclipse system always instantiates exactly one instance of an active plug-in's `Activator` class. Do not create instances of this class yourself.

2.3.3 The Favorites view

In addition to the plug-in manifest and plug-in class, the **New Plug-in Project** wizard generated code for a simple view (in the following sample) called **Favorites**. At this point, the view creates and displays information from a sample model; in subsequent chapters, however, this view will be hooked up to a favorites model and will display information from the favorites items contained within that model. Eclipse 3.4 generates additional code unnecessary for this exercise, so adjust the generated code to appear as shown below.

```
package com.qualityeclipse.favorites.views;

import org.eclipse.swt.widgets.Composite;
import org.eclipse.ui.part.*;
import org.eclipse.jface.viewers.*;
import org.eclipse.swt.graphics.Image;
import org.eclipse.jface.action.*;
import org.eclipse.jface.dialogs.MessageDialog;
import org.eclipse.ui.*;
import org.eclipse.swt.widgets.Menu;
import org.eclipse.swt.SWT;

/**
 * This sample class demonstrates how to plug-in a new workbench
 * view. The view shows data obtained from the model. The sample
 * creates a dummy model on the fly, but a real implementation
 * would connect to the model available either in this or another
 * plug-in (e.g., the workspace). The view is connected to the
 * model using a content provider.
 * <p>
 * The view uses a label provider to define how model objects
 * should be presented in the view. Each view can present the
 * same model objects using different labels and icons, if
 * needed. Alternatively, a single label provider can be shared
 * between views in order to ensure that objects of the same type
 * are presented in the same way everywhere.
 * <p>
 */
public class FavoritesView extends ViewPart {
   private TableViewer viewer;

   /*
    * The content provider class is responsible for providing
    * objects to the view. It can wrap existing objects in
    * adapters or simply return objects as-is. These objects may
    * be sensitive to the current input of the view, or ignore it
    * and always show the same content (Task List, for
    * example).
    */

   class ViewContentProvider
      implements IStructuredContentProvider
   {
      public void inputChanged(
         Viewer v, Object oldInput, Object newInput) {
      }

      public void dispose() {
      }

      public Object[] getElements(Object parent) {
         return new String[] { "One", "Two", "Three" };
      }
   }
```

```
   /*
    * The label provider class is responsible for translating
    * objects into text and images that are displayed
    * in the various cells of the table.
    */

   class ViewLabelProvider extends LabelProvider
      implements ITableLabelProvider
   {
      public String getColumnText(Object obj, int index) {
         return getText(obj);
      }

      public Image getColumnImage(Object obj, int index) {
         return getImage(obj);
      }

      public Image getImage(Object obj) {
         return PlatformUI.getWorkbench().getSharedImages()
               .getImage(ISharedImages.IMG_OBJ_ELEMENT);
      }
   }

   /**
    * The constructor.
    */
   public FavoritesView() {
   }

   /**
    * This is a callback that will allow us to create the viewer
    * and initialize it.
    */

   public void createPartControl(Composite parent) {
      viewer = new TableViewer(
         parent, SWT.MULTI | SWT.H_SCROLL | SWT.V_SCROLL);
      viewer.setContentProvider(new ViewContentProvider());
      viewer.setLabelProvider(new ViewLabelProvider());
      viewer.setInput(getViewSite());
   }

   /**
    * Passing the focus request to the viewer's control.
    */
   public void setFocus() {
      viewer.getControl().setFocus();
   }
}
```

2.4 Building a Product

Building a product involves packaging up only those elements to be delivered in a form that the customer can install into his or her environment. You can

build the product in several different ways, including manually or by using a Windows batch script, a UNIX shell script, or an Apache Ant script. You can deliver the end product as a single compressed file or as a stand-alone executable. For our purposes, the **Favorites** plug-in will be delivered with source code as a single compressed zip file.

2.4.1 Building manually

Building a product manually involves launching an Eclipse **Export** wizard, filling out a few fields, and clicking the **Finish** button. Select the **File > Export** command to launch the desired export wizard. On the first wizard page (see Figure 2–14), select **Deployable plug-ins and fragments** and then click the **Next** button.

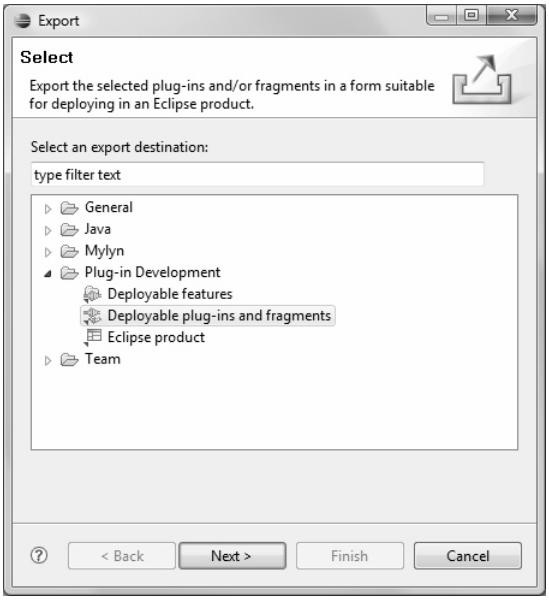

Figure 2–14 Export wizard page 1—choosing the type of export.

On the second page of the **Export** wizard (see Figure 2–15), select the plug-ins to be exported, enter the name of the zip file to contain the result, and select the options shown. In addition, specify that this export operation be saved as an Ant script in a file named `build-favorites.xml` in the `com.qualityeclipse.favorites` project and check the **Include source code** option, then click **Finish**.

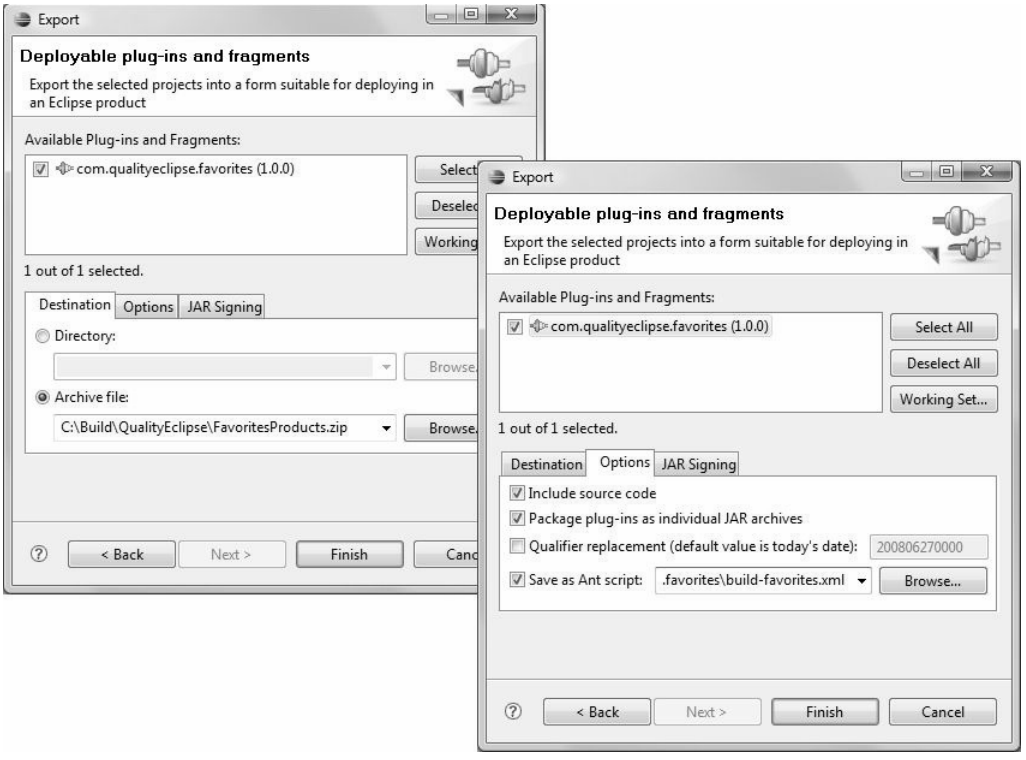

Figure 2–15 Export wizard page 2—specifying the zip file's contents.

The created zip file contains a single plug-in JAR file (a plug-in can be deployed as a single JAR file as of Eclipse 3.1):

```
plugins/com.qualityeclipse.favorites_1.0.0.jar
```

And that plug-in JAR file contains the plug-in as specified in the **Export** wizard:

```
com.qualityeclipse.favorites classes
com.qualityeclipse.favorites source files
plugin.xml
icons/sample.gif
META-INF/MANIFEST.MF
```

Unfortunately, this process is manual, and therefore prone to errors. Manually building a product is fine once or twice, but what if a different person in the company needs to build the product? What happens as the product grows and encompasses more plug-ins? A product needs a repeatable and reliable method for building it.

2.4.2 Building with Apache Ant

An Apache Ant script provides a reliable, flexible, and repeatable process for building a plug-in project. There is a little more up-front work to set up an Ant script, but it is much less error-prone over time than building a product manually. For more information about Ant and constructing more complex build scripts, see Chapter 19.

Eclipse can generate simple Ant scripts. The prior section specified that the **Export** wizard generates an Ant script file named `build-favorites.xml` in the `com.qualityeclipse.favorites` project:

```
<?xml version="1.0" encoding="UTF-8"?>
<project default="plugin_export" name="build">
   <target name="plugin_export">
      <pde.exportPlugins
         destination="C:\Build\QualityEclipse"
         exportSource="true"
         exportType="zip"
         filename="FavoritesProduct.zip"
         plugins="com.qualityeclipse.favorites"
         useJARFormat="true" />
   </target>
</project>
```

The preceding simple script works well from the Eclipse UI; however, unfortunately, the `pde.exportPlugins` and other `pde.export*` tasks are asynchronous and cannot be used in a headless environment (see Bugzilla entry 58413 at *bugs.eclipse.org/bugs/show_bug.cgi?id=58413*) making it difficult to build more than simple scripts.

If you want your build script to do more than just export plug-ins (see Section 3.2.1, Link files, on page 111), then you'll need a more complex Ant script similar to the following. For more on Ant and build scripts, see Chapter 19, Building a Product.

```
<?xml version="1.0" encoding="UTF-8"?>
<project default="plugin_export" name="build">
   <target name="plugin_export">

      <!-- Define build directories -->
      <property name="build.root"
         location="/Build/QualityEclipse" />
      <property name="build.temp"
         location="${build.root}/temp" />
      <property name="build.out"
         location="${build.root}/product" />

      <!-- Create build directories -->
      <delete dir="${build.temp}" />
      <mkdir dir="${build.temp}" />
      <mkdir dir="${build.out}" />

      <!-- Read the MANIFEST.MF -->
      <copy file="META-INF/MANIFEST.MF" todir="${build.temp}" />
      <replace file="${build.temp}/MANIFEST.MF">
         <replacefilter token=":=" value="=" />
         <replacefilter token=":" value="=" />
         <replacetoken>;</replacetoken>
         <replacevalue>
         </replacevalue>
      </replace>
      <property file="${build.temp}/MANIFEST.MF"/>

      <!-- Plugin locations -->
      <property name="plugin.jarname" value=
         "com.qualityeclipse.favorites_${Bundle-Version}" />
      <property name="plugin.jar" location=
         "${build.temp}/jars/plugins/${plugin.jarname}.jar" />
      <property name="product.zip" value=
         "${build.out}/Favorites_v${Bundle-Version}.zip" />

      <!-- Assemble plug-in JAR -->
      <mkdir dir="${build.temp}/jars/plugins" />
      <zip destfile="${plugin.jar}">
         <zipfileset dir="bin" />
         <zipfileset dir="." includes="META-INF/MANIFEST.MF" />
         <zipfileset dir="." includes="plugin.xml" />
         <zipfileset dir="." includes="icons/*.gif" />
         <zipfileset dir="." includes="src/**/*" />
      </zip>

      <!-- Assemble the product zip -->
      <zip destfile="${product.zip}">
         <fileset dir="${build.temp}/jars" />
      </zip>

   </target>
</project>
```

To execute this Ant script, right-click on the `build-favorites.xml` file and select **Run Ant…** (see Figure 2–16). When the Ant wizard appears, click on the **JRE** tab and select the **Run in the same JRE as the workspace** option (see Figure 2–17). Click the **Run** button to build the product.

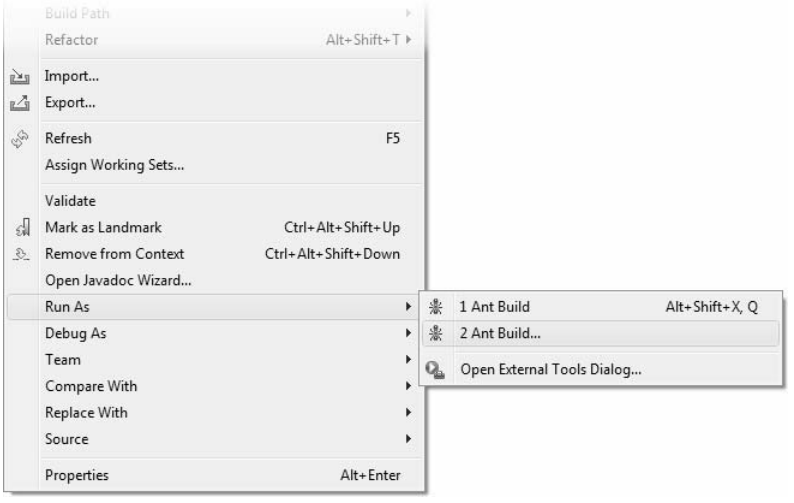

Figure 2–16 The build.xml popup context menu.

Tip: If your Ant script uses Eclipse-specific Ant tasks, such as `pde.exportPlugins`, then you must select the **Run in the same JRE as the workspace** option for your Ant script to execute properly.

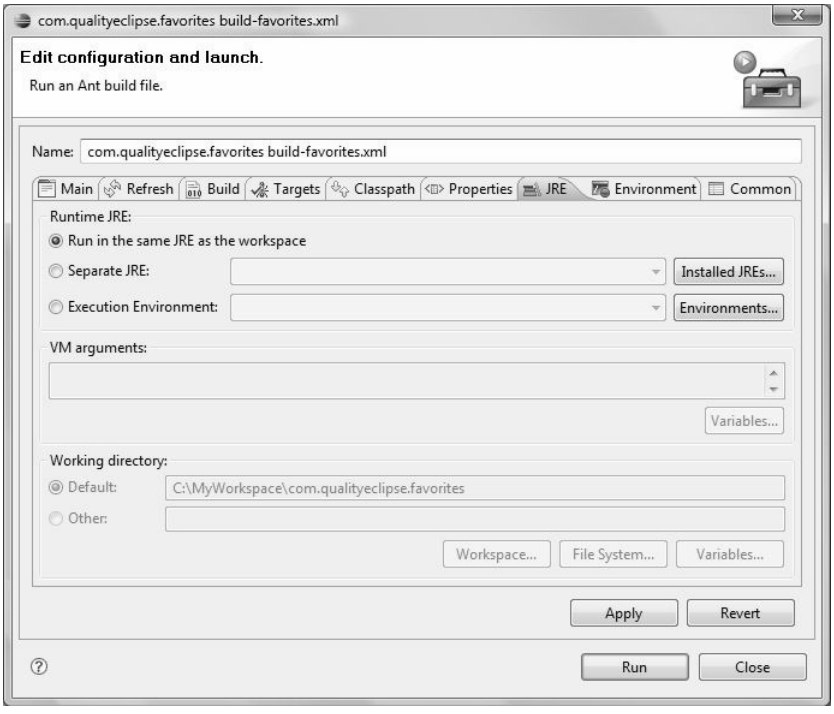

Figure 2–17 The Ant wizard.

2.5 Installing and Running the Product

To install the **Favorites** plug-in, do the following.

- Shut down Eclipse
- Unzip the `Favorites_v1.0.0.zip` file into your Eclipse directory (e.g., `C:/eclipse`)
- Verify that the favorites plug-in is in the `/plugins` directory (e.g., `C:/eclipse/plugins/com.qualityeclipse.favorites_1.0.0.jar`)
- Restart Eclipse

Tip: Eclipse caches plug-in information in a configuration directory (see Section 3.4.5, Plug-in configuration files, on page 123). If you are installing a new version of your plug-in over an already installed one without incrementing the version number, then use the `-clean` command-line option when launching Eclipse so that it will rebuild its cached plug-in information.

After Eclipse has restarted, from the **Window** menu, select **Show View > Other...** (see Figure 2–18) to open the **Show View** dialog (see Figure 2–19). In the dialog, expand the **Quality Eclipse** category, select **Favorites**, and then click the **OK** button. This causes the **Favorites** view to open (see Figure 2–20).

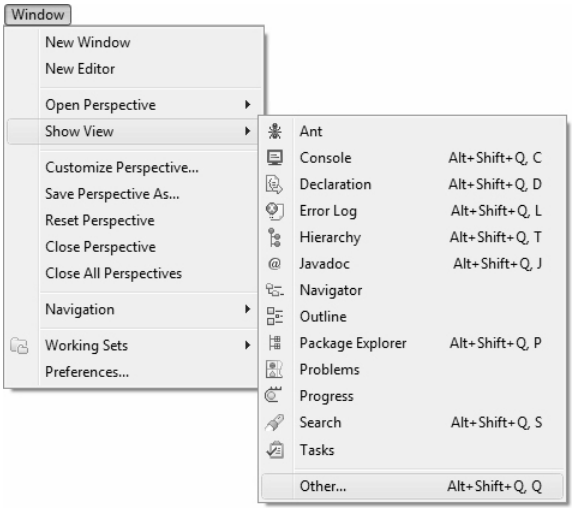

Figure 2–18 Show View > Other… from the Window menu.

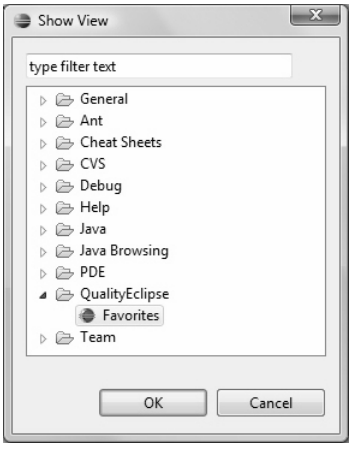

Figure 2–19 Show View dialog.

Figure 2–20 The Favorites view in its initial and simplest form.

2.6 Debugging the Product

Inevitably, during the course of producing a product, you'll need to debug a problem or you'll simply have to gain a better understanding of the code through a means more enlightening than just reviewing the source code. You can use the **Runtime Workbench** to determine exactly what happens during product execution so that you can solve problems.

2.6.1 Creating a configuration

The first step in this process is to create a configuration in which the product can be debugged. Start by selecting **Debug Configurations…** in the **Debug** toolbar menu (see Figure 2–21).

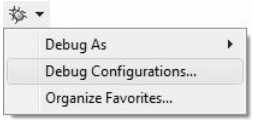

Figure 2–21 Debug menu.

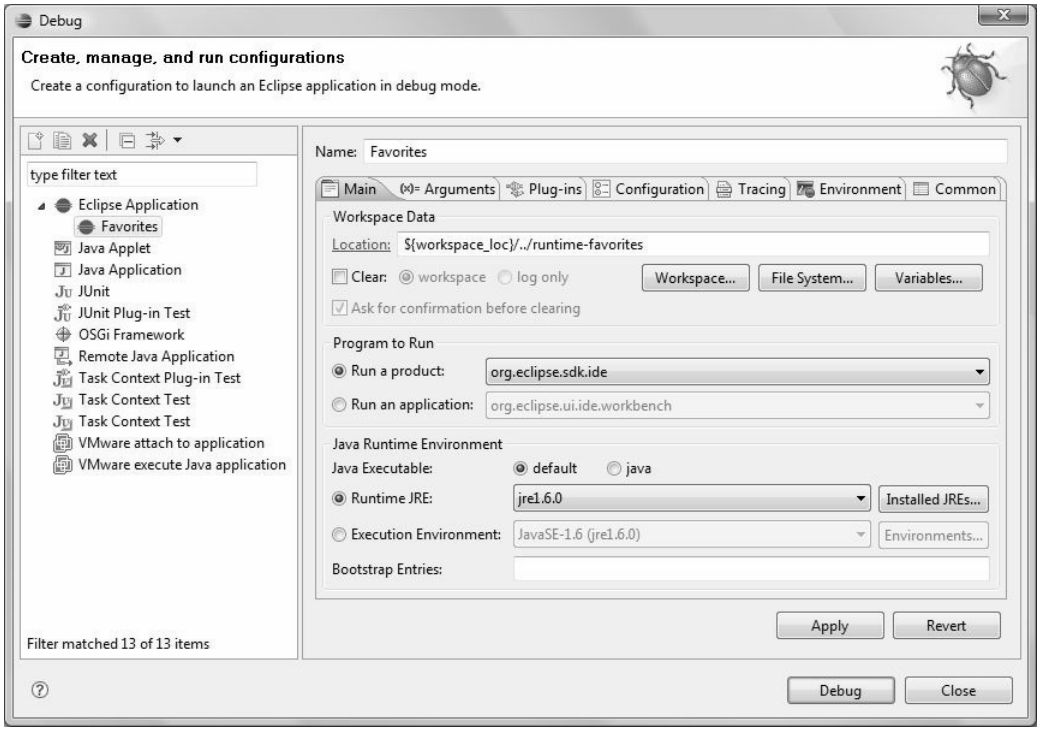

Figure 2–22 Defining a new configuration.

In the dialog that appears (see Figure 2–22), select **Eclipse Application** and then click the **New** button. Next, enter "Favorites" as the name of the configuration.

2.6.2 Selecting plug-ins and fragments

After the preceding, select the **Plug-ins** tab and **plug-ins selected below only** in the **Launch with** combo box (see Figure 2–23). In the list of plug-ins, make sure that the **Favorites** plug-in is selected in the **Workspace Plug-ins** category but not in the **External Plug-ins** category.

Tip: Plug-in projects specified in the configuration take precedence over plug-ins installed in Eclipse itself. If you have a plug-in project with the same identifier as a plug-in installed in Eclipse and want to use the installed plug-in in the **Runtime Workbench** rather than the plug-in project, uncheck the plug-in project in the **Workspace Plug-ins** category and check the installed plug-in in the **External Plug-ins** category.

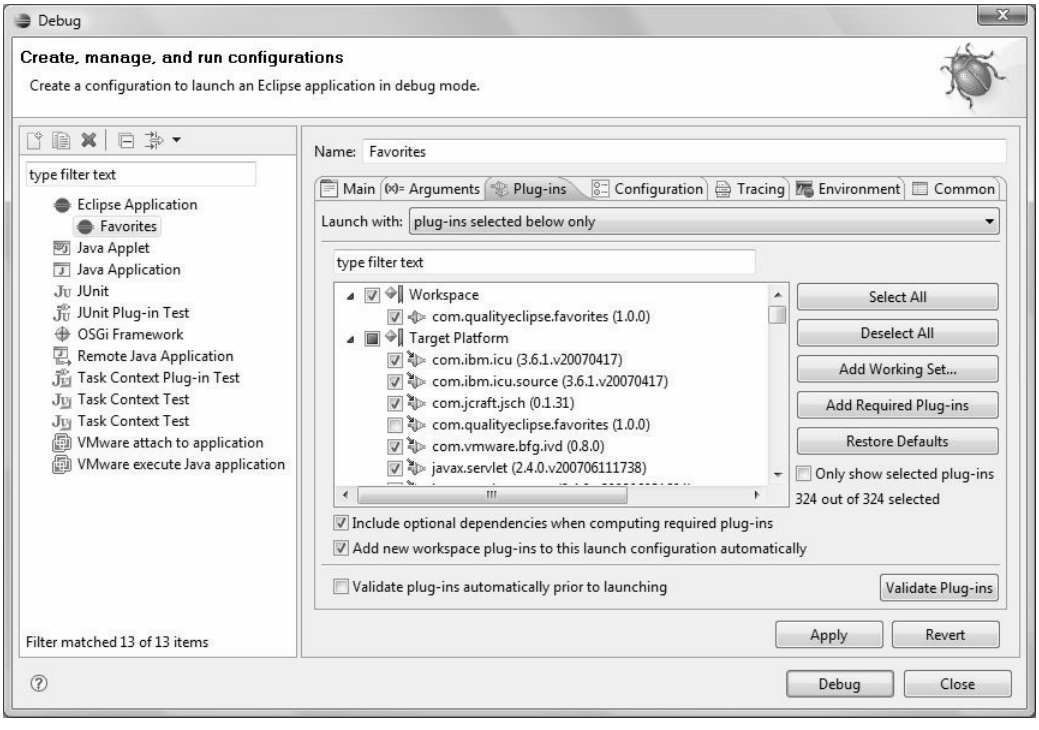

Figure 2–23 Selecting plug-ins in the configuration.

2.6.3 Launching the Runtime Workbench

Click the **Debug** button to launch the **Eclipse Application** in the **Runtime Workbench** to debug the product. Now that you've defined the configuration and used it once, it appears in the **Debug** toolbar menu (see Figure 2–21). Selecting it from that menu launches the **Runtime Workbench** without opening the **Configuration** wizard.

After clicking the **Debug** button in the **Configuration** wizard or selecting **Favorites** from the **Debug** toolbar menu, Eclipse opens a second workbench window (the **Runtime Workbench**, as opposed to the **Development Workbench**). This **Runtime Workbench** window executes the code in the projects contained in the **Development Workbench**. Making changes and setting breakpoints in the **Development Workbench** affects the execution of the **Runtime Workbench** (see Section 1.10, Introduction to Debugging, on page 59 for more about this).

2.7 PDE Views

The Plug-in Development Environment (PDE) provides several views for inspecting various aspects of plug-ins. To open the various PDE views, select **Window > Show View > Other…**; in the **Show View** dialog, expand both the **PDE** and **PDE Runtime** categories.

2.7.1 The Plug-in Registry view

The **Plug-in Registry** view displays a tree view of all plug-ins discovered in the current workspace (see Figure 2–24). Expanding the plug-in in the tree shows its components such as extension points, extensions, prerequisites, and runtime libraries.

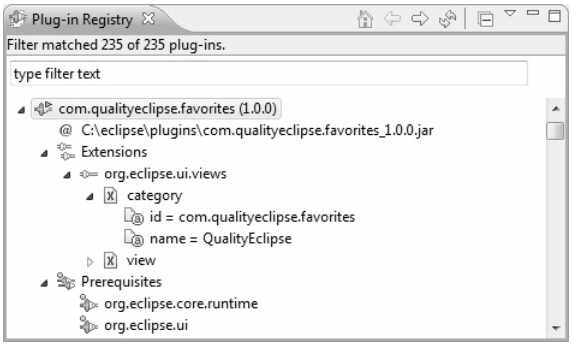

Figure 2–24 The Plug-in Registry view.

2.7.2 The Plug-ins view

The **Plug-ins** view shows a tree list of external plug-ins and plug-in projects in the current workspace and provides a quick way to review plug-ins that already exist (see Figure 2–25). In the tree, you can expand each external plug-in to browse the files located in the plug-in directory. Unfortunately, if that plug-in is contained in a JAR file rather than a directory (new in Eclipse 3.1), the files are not displayed in this view (see Bugzilla entry 89143 at *bugs.eclipse.org/bugs/show_bug.cgi?id=89143*). Double-clicking on a file element opens that file in an editor for viewing, and there are several useful actions in the context menu such as **Add to Java Search** (see Section 1.6.2.1, Java Plug-in Search, on page 32), Find References and Open Dependencies.

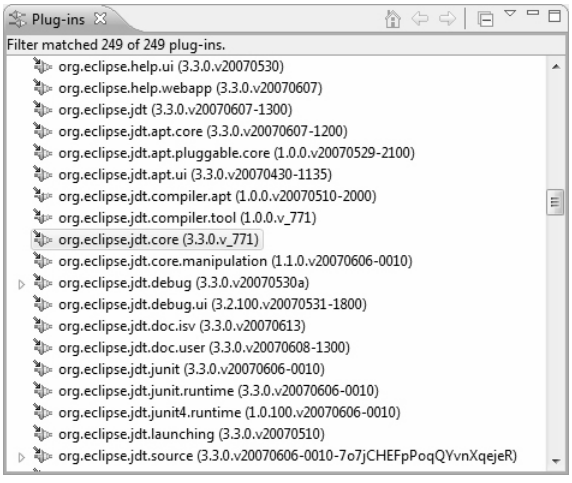

Figure 2–25 The Plug-ins view.

2.7.3 The Plug-in Dependencies view

The **Plug-in Dependencies** view shows a hierarchy of which plug-ins are dependent on which other plug-ins, which in turn are dependent on other plug-ins, and so on (see Figure 2–26). When the view opens, first right-click on the `com.qualityeclipse.favorites` plug-in and select **Focus On**. Double-clicking on an element in the tree opens the plug-in manifest editor for the corresponding plug-in.

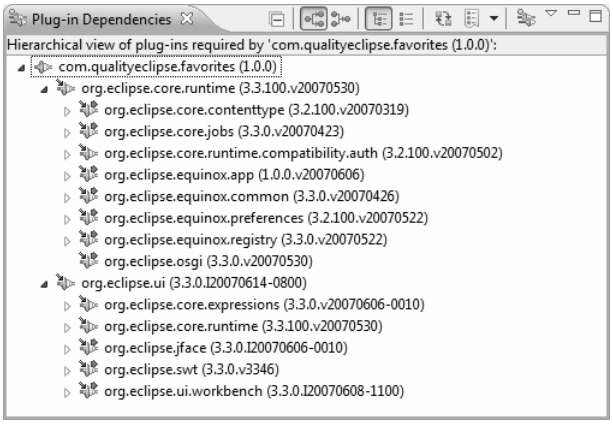

Figure 2–26 The Plug-in Dependencies view.

2.7.4 Plug-in Artifact Search

In addition to the views described above, PDE provides the ability to search for extension references, extension point declarations, and plug-ins all in one place. Type Ctrl+Shift+A to open the PDE search dialog (see Figure 2–27), then enter the plug-in ID to filter the list. The dialog also includes filters for extensions and extension points to help you quickly and easily find what you are looking for.

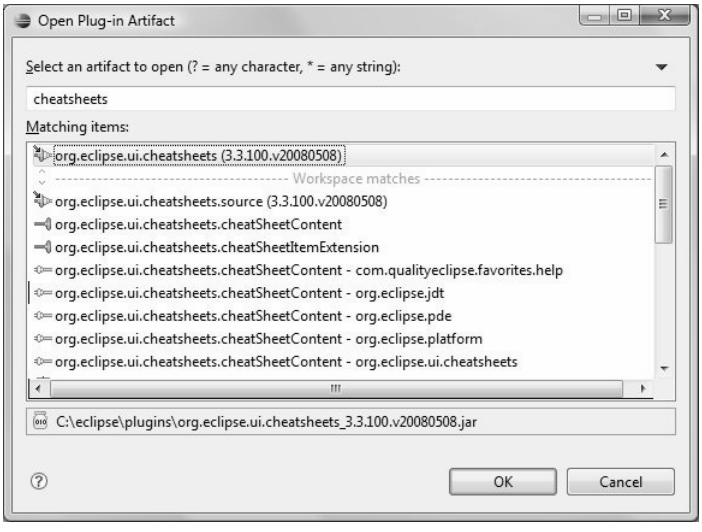

Figure 2–27 PDE Plug-in Artifact dialog.

2.7.5 Plug-in Spy

To find more information about the currently selected user interface element, open the Plug-in Spy (see Figure 2–28) by pressing Alt+Shift+F1. The Plug-in Spy (also known as the PDE Spy) currently provides information about selections, editors, views, dialogs, preference pages, and wizards. When reviewing the information provided by the Plug-in Spy, clicking on the various hyperlinks opens the Plug-in Manifest editor on that plug-in.

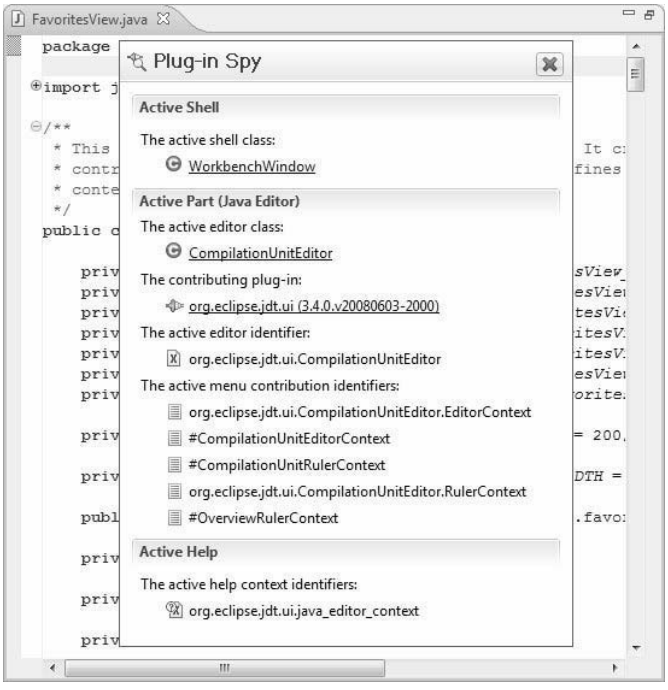

Figure 2–28 Plug-in Spy popup.

2.8 Writing Plug-in Tests

Eclipse is a continually moving target, and when building plug-ins, tests are necessary to ensure that the product continues to function properly over multiple releases. If the goal was to develop and release a plug-in once, then manual testing would suffice; however, a combination of automated and manual tests are better at preventing regressions from creeping into the product over time.

2.8.1 Test preparation

Before a test for the **Favorites** view can be created, you must modify the Favorites plug-in manifest so that the appropriate classes are visible to the test plug-in. Open the plug-in manifest editor by double-clicking on the `plugin.xml` file, then switch to the **Runtime** page (see Figure 2–11). In the **Exported Packages** section, click **Add...**, select the `com.qualityeclipse.favorites.views` package and save the changes by selecting **File > Save**.

> **Tip:** You can limit the visibility of your exported packages by specifying which plug-ins can access a package in the **Package Visibility** section of the **Runtime** page in the plug-in manifest editor (see Section 21.2.5, Related plug-ins, on page 783). Alternatively, you can place tests into a fragment so that no packages need to be exported (for more on fragments, see Section 16.3, Using Fragments, on page 629).

Next, add the appropriate accessor so that the test can validate the view content. In the `FavoritesView` class, add the following method:

```
/**
 * For testing purposes only.
 * @return the table viewer in the Favorites view
 */
public TableViewer getFavoritesViewer() {
   return viewer;
}
```

2.8.2 Creating a Plug-in test project

Use the same procedure as outlined in Section 2.2, Creating a Plug-in Project, on page 72, to create a new plug-in project with the following exceptions:

- Name the project **com.qualityeclipse.favorites.test**
- Uncheck the **Create a plug-in using one of these templates** checkbox

After the project has been created, use the **Dependencies** page of the plug-in manifest editor (see Figure 2–10 on page 79) to add the following required plug-ins and then save the changes:

- **com.qualityeclipse.favorites**
- **org.junit4**

2.8.3 Creating a Plug-in test

When a project has been created and the plug-in manifest modified, it's time to create a simple test for the **Favorites** plug-in (see the following code example). The goal of the test is to show the **Favorites** view, validate its content, and then hide the view.

```
package com.qualityeclipse.favorites.test;

import static org.junit.Assert.assertArrayEquals;
import static org.junit.Assert.assertEquals;
import org.eclipse.core.runtime.Platform;
import org.eclipse.jface.viewers.IStructuredContentProvider;
import org.eclipse.jface.viewers.ITableLabelProvider;
import org.eclipse.jface.viewers.TableViewer;
import org.eclipse.swt.widgets.Display;
import org.eclipse.ui.PlatformUI;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import com.qualityeclipse.favorites.views.FavoritesView;
/**
 * The class <code>FavoritesViewTest</code> contains tests
 * for the class {@link
 *    com.qualityeclipse.favorites.views.FavoritesView}.
 * @pattern JUnit Test Case
 * @generatedBy CodePro Studio
 */

public class FavoritesViewTest
{
   private static final String VIEW_ID =
      "com.qualityeclipse.favorites.views.FavoritesView";

   /**
    * The object that is being tested.
    *
    * @see com.qualityeclipse.favorites.views.FavoritesView
    */
   private FavoritesView testView;

   /**
    * Perform pre-test initialization.
    */
   @Before
   public void setUp() throws Exception {
      // Initialize the test fixture for each test
      // that is run.
      waitForJobs();
      testView = (FavoritesView)
         PlatformUI
            .getWorkbench()
            .getActiveWorkbenchWindow()
            .getActivePage()
            .showView(VIEW_ID);

      // Delay for 3 seconds so that
      // the Favorites view can be seen.
      waitForJobs();
      delay(3000);
      // Add additional setup code here.
   }
```

```
/**
 * Run the view test.
 */
@Test
public void testView() {
   TableViewer viewer = testView.getFavoritesViewer();
   Object[] expectedContent =
      new Object[] { "One", "Two", "Three" };
   Object[] expectedLabels =
      new String[] { "One", "Two", "Three" };

   // Assert valid content.
   IStructuredContentProvider contentProvider =
      (IStructuredContentProvider)
         viewer.getContentProvider();
   assertArrayEquals(expectedContent,
      contentProvider.getElements(viewer.getInput()));

   // Assert valid labels.
   ITableLabelProvider labelProvider =
      (ITableLabelProvider) viewer.getLabelProvider();
   for (int i = 0; i < expectedLabels.length; i++)
      assertEquals(expectedLabels[i],
         labelProvider.getColumnText(expectedContent[i], 1));
}

/**
 * Perform post-test cleanup.
 */
@After
public void tearDown() throws Exception {
   // Dispose of test fixture.

   waitForJobs();
   PlatformUI
      .getWorkbench()
      .getActiveWorkbenchWindow()
      .getActivePage()
      .hideView(testView);

   // Add additional teardown code here.
}

/**
 * Process UI input but do not return for the
 * specified time interval.
 *
 * @param waitTimeMillis the number of milliseconds
 */
private void delay(long waitTimeMillis) {
   Display display = Display.getCurrent();

   // If this is the UI thread,
   // then process input.
```

```
            if (display != null) {
               long endTimeMillis =
                  System.currentTimeMillis() + waitTimeMillis;
               while (System.currentTimeMillis() < endTimeMillis)
               {
                  if (!display.readAndDispatch())
                     display.sleep();
               }
               display.update();
            }
            // Otherwise, perform a simple sleep.

            else {
               try {
                  Thread.sleep(waitTimeMillis);
               }
               catch (InterruptedException e) {
                  // Ignored.
               }
            }
         }

         /**
          * Wait until all background tasks are complete.
          */
         public void waitForJobs() {
            while (!Job.getJobManager().isIdle())
               delay(1000);
         }
      }
```

2.8.4 Running a Plug-in test

The next step after creating a test class is to configure and execute the test. Similar to creating a runtime configuration (see Section 2.6.1, Creating a configuration, on page 94), creating a test configuration involves right-clicking on the `FavoritesViewTest` in the **Package Explorer** and selecting the **Run As > JUnit Plug-in Test** command. This automatically builds a test configuration and executes the test. You should then see the Runtime Workbench appear, the **Favorites** view open, and the Runtime Workbench close. The **JUnit** view indicates that your test executed successfully and the **Favorites** view content has been validated (see Figure 2–29).

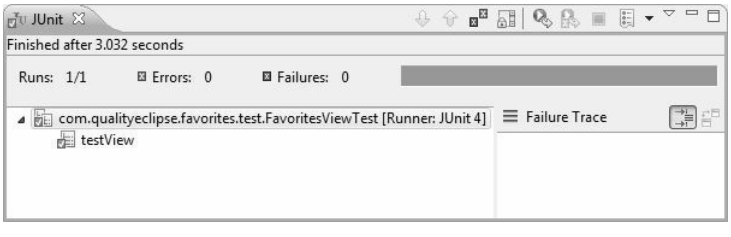

Figure 2–29 The JUnit view.

Right clicking on the `FavoritesViewTest` once again and selecting **Run As > Run Configurations...** opens the **Configuration** wizard (see Figure 2–30). Here you can specify whether a single test should be executed by itself or whether all tests in a project should be executed simultaneously. Eclipse defaults to launching a product, which opens the Welcome view (see Figure 1–2 on page 4) rather than an application. To change this, click on the **Main** tab and select the **Run an application** radio button.

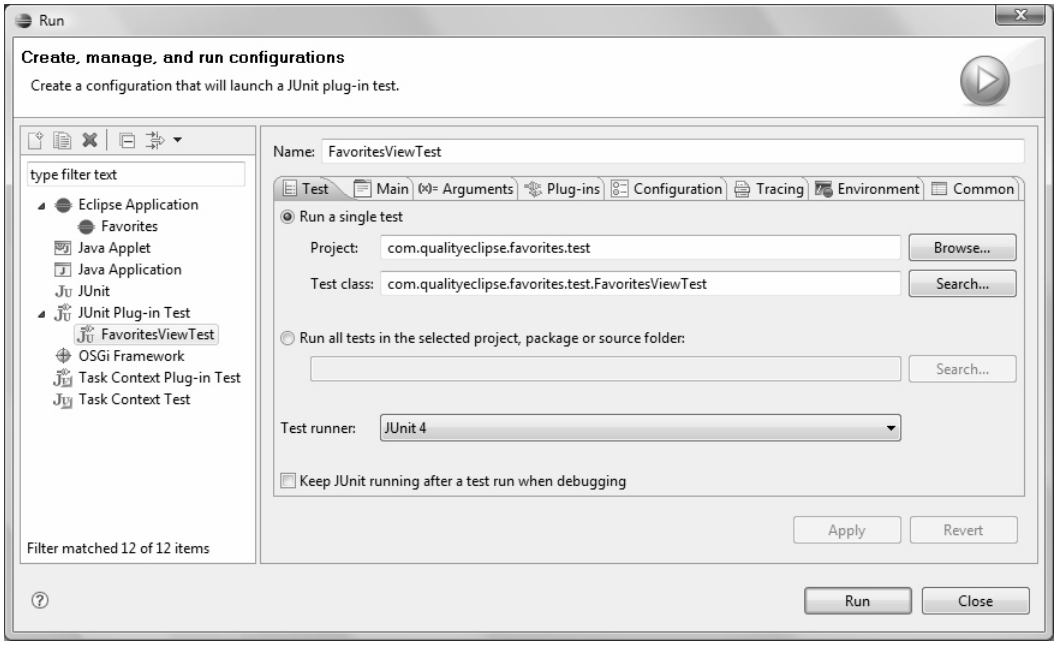

Figure 2–30 The test Configuration wizard.

2.8.5 Uninstalling the Favorites plug-in

Use the following steps to delete the **Favorites** plug-in from the **Development Workspace**:

1. Close the **Favorites** view.
2. Shut down Eclipse.
3. Delete the `com.quality.favorites_1.0.0.jar` file in the Eclipse plug-ins directory.
4. Restart Eclipse. If you get an error message (see Figure 2–31) when restarting, at least one of the **Favorites** views was not closed when Eclipse was shut down in Step 2.

5. Verify that the **Favorites** view is no longer available by opening the **Show View** dialog (see Figure 2–18) and verifying that the **Quality Eclipse** category is no longer present (see Figure 2–19).

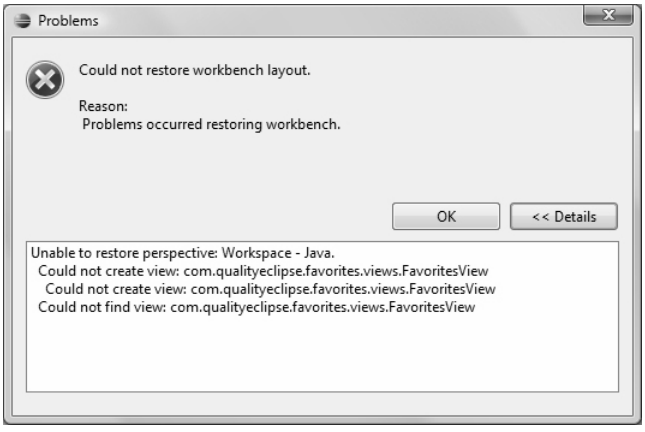

Figure 2–31 Problems dialog when restarting Eclipse.

2.9 Book Samples

The sample code for each chapter in this book can be downloaded and installed into Eclipse for you to review. Download and install the book samples from *http://www.qualityeclipse.com* or using the update manager (see Section 18.3.5, Accessing the update site, on page 685) by entering "*http://www.qualityeclipse.com/update*" (see Figure 18–29 on page 687). Once installed, open the view by selecting **Window > QualityEclipse Book Samples** (see Figure 2–32).

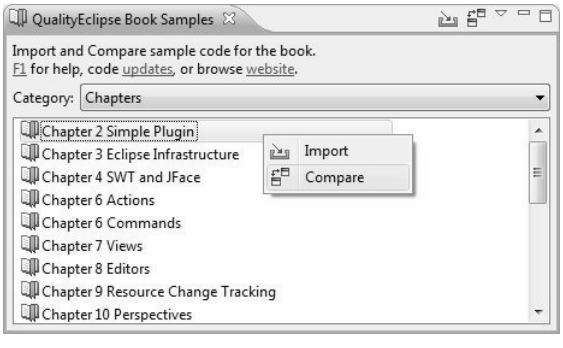

Figure 2–32 QualityEclipse Book Samples View

Using the book samples view, you can compare your work to the sample code and load the code for particular chapters for review.

2.10 Summary

This chapter covered the process of creating, running, debugging, inspecting, and testing a simple plug-in from start to finish. Subsequent chapters will cover every aspect of this process, plus more in much greater detail.

References

Gamma, Eric, and Kent Beck, *Contributing to Eclipse*. Addison-Wesley, Boston, 2003.

McAffer, Jeff, and Jean-Michel Lemieux, *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java Applications*. Addison-Wesley, Boston, 2005.

FAQ How do I find a particular class from an Eclipse plug-in? (*http://wiki.eclipse.org/FAQ_How_do_I_find_a_particular_class_from_an_Eclipse_plug-in%3F*)

Helpful tools in PDE Incubator (*http://www.eclipse.org/pde/incubator/*)

INDEX

G

M

N

Q

R

S

U

X

Z