the eclipse series

Eclipse Rich Client Platform

Second Edition

Jeff McAffer • Jean-Michel Lemieux • Chris Aniszczyk

Series Editors Jeff McAffer • Erich Gamma • John Wiegand

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

McAffer, Jeff.
Eclipse Rich Client Platform / Jeff McAffer, Jean-Michel Lemieux, Chris Aniszczyk.—2nd ed.
p. cm.
Includes index.
ISBN 0-321-60378-8 (pbk. : alk. paper)
1. Computer software—Development. 2. Java (Computer program language)
3.Application software—Development. I. Lemieux, Jean-Michel. II. Aniszczyk,
Chris. III. Title.
QA76.76.D47M383 2010
005.13'3—dc22 2010006689

ISBN-13: 978-0-321-60378-4
ISBN-10: 0-321-60378-8
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, May 2010

Foreword

In my foreword to the first edition of this book, I wrote that the Eclipse Rich Client Platform (RCP) is a lot like the enormous rockets that carry NASA's robots into space: powerful, sophisticated, essential, but ultimately just the launch vehicle that propels our creations safely to their destinations. Four years later, the RCP continues to serve as the launch vehicle for the tools that my team develops to control a broad variety of spacecraft and robots that drive, fly, float, and move in ways that defy categorization. It provides us with a firm foundation for component-based development, a flexible framework for rich user interfaces, and countless other capabilities that surround and support the small nugget of software that my team actually develops.

My team is extremely proud of our small nugget of code and what it accomplishes at NASA, but when our missions succeed, I think we all celebrate with an acute awareness that a space exploration project of interplanetary scale demands the combined success of hundreds and sometimes thousands of experts inside and outside NASA who specialize in everything from designing cruise trajectories to the art of neatly routing cables through the limbs of a robot (and trust me, it is an art—I've seen those people work). Sure, it probably took only a few people to strap a gunpowder-filled tube to the side of an arrow to make the first rocket more than a thousand years ago, but it's only through an enormous feat of simultaneous specialization and cooperation that we could hope to achieve something as complex as landing and operating a rover on Mars.

This remarkable combination of specialization and cooperation can be found in many other fields. A few handymen can get together and build a shed, but a skyscraper requires the combined effort of architects, carpenters, plumbers, electricians, masons, and hundreds of other specialists with only basic knowledge of each other's disciplines. Modern construction, just like modern space exploration, is simply too ambitious and complex to accomplish any other way.

I think modern software applications are more like deep space robotic explorers than rocket-propelled arrows and more like skyscrapers than sheds. Complex application development demands specialization and cooperation, and I think that is the fundamental reason for the existence and sustained success of the Eclipse RCP. Behind the platform described in this book is a worldwide community of experts—specialists in everything from provisioning to user assistance who have cooperated for years in their own mission to create a free, extensible framework that can be used to build almost any kind of application. If you decide to use this framework, you'll soon discover that you've joined an even larger community of people who are also building applications on the Eclipse RCP—an entirely different breed of specialists. You might be surprised to discover valuable vendors, customers, and collaborators among them. We certainly did.

My team has attended every EclipseCon (the largest yearly gathering of Eclipse developers and users) since 2005, and I've consistently been amazed by the diversity of applications that people are building on top of the RCP. I've seen RCP applications for controlling nuclear reactors, scheduling trains, trading stocks, designing data centers, managing inventory, fighting terrorism, analyzing proteins, monitoring fishing boats, sharing files, and editing every programming language that I've ever heard of. After a couple of EclipseCon conferences, we even came across another space agency building mission control applications on top of the RCP. (You can imagine we had plenty to talk about!) But what's more surprising than the diversity of RCP applications is everything that our applications have in common. For example, APC uses the same graphical editor framework to arrange server racks in their data center design program that my team uses to manipulate Mars images in our rover operations program. My team built our spacecraft command editor with the same basic components used in most of the Eclipse programming tools. These commonalities allow us to combine our resources, learn from each other, and ultimately deliver better products to our customers.

Some of your colleagues may think it's risky to base your application on software developed by such a far-flung group. It might be tempting to think that it would be easier and safer to just build it all yourselves. But would it be safe to have NASA's programmers build rocket engines or to ask a skyscraper's plumber to pour the foundation? Not only is it impossible for your team to specialize in every aspect of rich application development, but merely trying to do so is a distraction that could endanger your whole project. For example, let's say you have a team of three people who need to build an application during the next year, and one of the features it needs is a way to perform long-running tasks and keep the user aware of progress. Sure, your team could develop that from scratch, but I asked members of the Eclipse platform team and they estimated that they spent nearly three work years building the Jobs API, a robust and flexible framework for this purpose. The Eclipse RCP can save you from spending your project's budget on things that aren't even your specialty.

My specialty is developing tools that operate robots and spacecraft. Your specialty might be developing tools for anything from health care to clean energy. The authors of this book, however, are specialists in making it easier for you and me to write our tools and, in the end, spend more time focusing on our specialties. If you decide to join the community building on top of the RCP, I look forward to learning how you've used these tools to support your work at a future EclipseCon. You might even discover ways to contribute your specialty to the improvement of the RCP itself.

—Jeff Norris
Supervisor, Planning Software Systems Group
Jet Propulsion Laboratory
California Institute of Technology

This page intentionally left blank

Preface

In many ways this book is one of the design documents for the Eclipse Rich Client Platform (RCP). The first edition was originally written during the Eclipse 3.1 development cycle by members of the development team. Its chapters were sometimes written before the related function was even implemented. The second edition was written during the Eclipse 3.5 development cycle.

The exercise of explaining how things work forced upon us the realities of using the mechanisms and concepts that make up the Eclipse RCP. This was not always pleasant. It did, however, give us a unique opportunity to correct the course of the Eclipse RCP.

Whenever we came across something that was hard to explain or complicated to use, we were able to step back and consider changing Eclipse to make things easier. Often we could, and often we (or, more accurately, the Eclipse Platform team as a whole) did. It is somewhat hard to convey the joyful feeling of deleting a complicated, detailed ten-page set of instructions or explanation and replacing it with just a paragraph detailing a new wizard or facility.

On other occasions we gained key insights that helped us produce a clearer, simpler description of a function. Fixing bugs discovered during this process provided welcome distractions as we were writing, coding, learning, and trying to have real lives all at the same time.

We learned an incredible amount about Eclipse as an RCP and trust that you will, too.

About This Book

This book guides you, the would-be RCP developer, through all stages of developing and delivering an example RCP application called Hyperbola, an instant messaging chat client.

We develop Hyperbola from a blank workspace into a full-featured, branded RCP application. The choice of the instant messaging domain allowed us to plausibly touch a wide range of RCP issues, from building pluggable and dynamically extensible systems to using third-party code libraries to packaging applications for a variety of environments. We cover scenarios ranging from PDAs to kiosks, to stand-alone desktops, to full integration with the Eclipse IDE. This book enables you to do the same with your applications.

Roughly speaking, the book is split in two. The first half, Parts I and II, sets the scene for RCP and presents a tutorial-style guide to building an RCP application. The tutorial incrementally builds Hyperbola into a functioning, branded chat client complete with Help, Update, and other advanced capabilities. The tutorial is written somewhat informally to evoke the feeling that we are there with you, working through the examples and problems. We share some of the pitfalls and mishaps that we experienced while developing the application and writing the tutorial.

The second half of the book looks at what it takes to "make it real." It's one thing to write a prototype and quite another to ship a product. We don't leave you hanging at the prototype stage; Parts III and IV are composed of chapters that dive into the details required to finish the job—namely, the refining and refactoring of the first prototype, customizing the user interface, and building and delivering products to your customers. This part is written as more of a reference, but it still includes a liberal sprinkling of step-by-step examples and code samples. The goal is to cover most of the major stumbling blocks reported in the community and seen in our own development of professional products.

A final part, Part V, is pure reference. It covers the essential aspects of OSGi, the base execution framework for Eclipse, and touches on various functions available in the Eclipse Platform but not covered earlier in the book.

Since one book could not possibly cover everything about Eclipse, and there are many existing books that cover Eclipse and plug-in development, we focus on the areas directly related to RCP functionality, API, and development.

Audience

This book is targeted at several groups of Java developers. Some Java programming experience is assumed and no attempt is made to introduce Java concepts or syntax.

For developers new to the Eclipse RCP, there is information about the origins of the platform, how to get started with the Eclipse IDE, and how to write your first RCP application. Prior experience with Eclipse is helpful but not necessary.

For developers experienced with creating Eclipse plug-ins, the book covers aspects of plug-in development that are unique to RCP development. For example, not only are there special hooks for RCP applications, but RCP applications have additional characteristics such as branding, plug-in building as part of a release engineering process, deployment, and installation, to name a few.

For experienced Eclipse RCP developers, this book covers new RCP features and functions in Eclipse 3.5 as well as the new tooling that makes designing, coding, and packaging RCP applications easier than ever before.

Sample Code

Reading this book can be a very hands-on experience. There are ample opportunities for following along and doing the steps yourself as well as writing your own code. The samples that accompany the book include code for each chapter and can be obtained from the book's Web site: *http://eclipsercp.org*. Instructions for managing these samples are given in Chapter 3, "Tutorial Introduction," and as needed in the text. In particular, the following resources are included:

- A README.HTML file with installation and use instructions
- Eclipse 3.5.2 SDK
- Eclipse 3.5.2 RCP SDK
- Eclipse 3.5.2 RCP delta pack
- Code samples for each chapter as needed
- A prebuilt, complete version of Hyperbola

Conventions

The following formatting conventions are used throughout the book:

Bold—used for UI elements such as menu paths (e.g., **File > New > Project**) and wizard and editor elements

Italics—used for emphasis and to highlight terminology

`Lucida`—Used for Java code, property names, file paths, bundle IDs, and the like that are embedded in the text

`Lucida Bold`—Used to highlight important lines in code samples

Notes and sidebars are used often to highlight information that readers may find interesting or helpful in using or understanding the function being described in the main text. We tried to achieve an effect similar to that of an informal pair-programming experience where you sit down with somebody and get impromptu tips and tricks here and there.

CHAPTER 2

Eclipse RCP Concepts

The Eclipse environment is very rich, but there are just a few concepts and mechanisms that are essential to *Eclipse-ness*. In this chapter we introduce these concepts, define some terminology, and ground these concepts and terms in technical detail. The ultimate goal is to show you how Eclipse fits together, both physically and conceptually.

Even if you are familiar with Eclipse, you might want to flip through this chapter to ensure that we have a common base of understanding and terminology. Writing RCP applications is subtly different from just writing plug-ins. You have the opportunity to define more of the look and feel, the branding, and other fundamental elements of Eclipse. Understanding these fundamentals enables you to get the most out of the platform. With this understanding you can read the rest of the book and see how Eclipse fits into your world.

2.1 A Community of Plug-ins

In Chapter 1, "Eclipse as a Rich Client Platform," we described the essence of Eclipse and its role as a component framework. The basic unit of functionality in this framework is called a *plug-in* (or a *bundle* in OSGi terms), the unit of modularity in Eclipse. Everything in Eclipse is a plug-in. An RCP application is a collection of plug-ins and a *framework* on which they run. An RCP developer assembles a collection of plug-ins from the Eclipse base and elsewhere and adds in the plug-ins he or she has written. These new plug-ins include an *application* and a *product* definition along with their domain logic. In addition to understanding how Eclipse manages plug-ins, it is important to know which existing plug-ins to use and how to use them, and which plug-ins to build yourself and how to build them.

Small sets of plug-ins are easy to manage and talk about. As the pool of plug-ins in your application grows, however, grouping abstractions are needed to help hide some of the detail. The Eclipse teams define a few coarse sets of plug-ins, as shown in Figure 2-1.

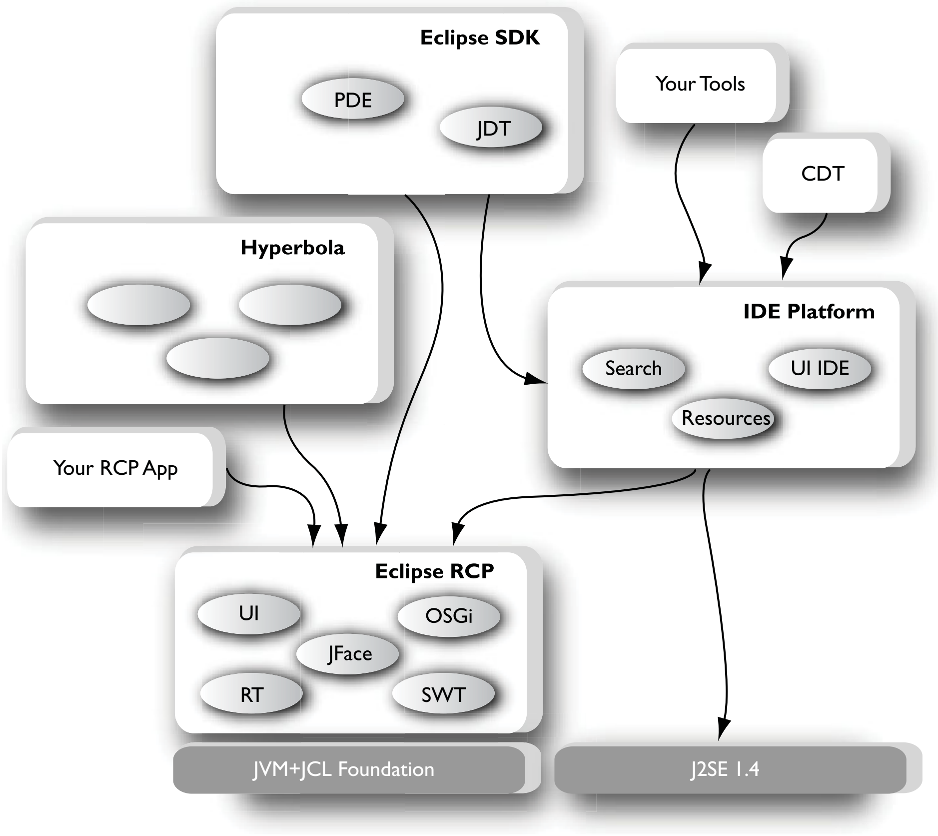

Figure 2–1 Ten-thousand-foot system architecture view

At the bottom of the figure is the Eclipse RCP as a small set of plug-ins on top of a Java Runtime Environment (JRE). The RCP on its own is much like a basic OS or the Java JRE itself—it is waiting for applications to be added.

NOTE

Don't take the boxes in Figure 2-1 too seriously. They are a guess, by the producers of the plug-ins, at groupings that are coherent to consumers of the plug-ins. The groupings are useful abstractions; but remember, for every person who wants some plug-in inside a box, there is someone else who wants it outside. That's OK. You can build your own abstractions.

Fanning upward in the figure is a collection of RCP applications—some written by you, some by others, and some by Eclipse teams. The Eclipse IDE Platform, the traditional Eclipse used as a development environment, is itself just a highly functional RCP application. As shown in Figure 2-1, the IDE Platform requires some of the plug-ins in the Eclipse RCP. Plugged into the IDE Platform is the Eclipse Software Development Kit (SDK) with its Java and plug-in tooling and hundreds of other tools written by companies and the open-source community.

This pattern continues. The general shape of the Eclipse RCP and of your products is the same—both are just sets of plug-ins that make up a coherent whole. These themes of consistency and uniformity recur throughout Eclipse.

NOTE

Notice in Figure 2-1 that the Eclipse RCP requires only Foundation Java class libraries. Foundation is a J2ME standard class set typically meant for embedded or smaller environments. See *http://java.sun.com/products/foundation* for more details. If you are careful to use only a Foundation-supported API, you can ship Eclipse-based applications on a Java Runtime that is only about 6MB rather than the 40MB J2SE 1.4 JRE.

The internal detail for the Eclipse RCP plug-in set is shown in Figure 2-2. These plug-ins form the base of your RCP applications. Here we see a set of interdependent plug-ins that provide various capabilities as noted in the callout boxes. We could have zoomed in on any of the plug-in sets in Figure 2-1 and seen the same basic uniform structure. You are in fact free to slice and dice the RCP itself or any other plug-in set to suit your needs as long as the relevant plug-in interdependencies are satisfied. In this book we focus on *RCP applications* as applications that use the full RCP plug-in set.

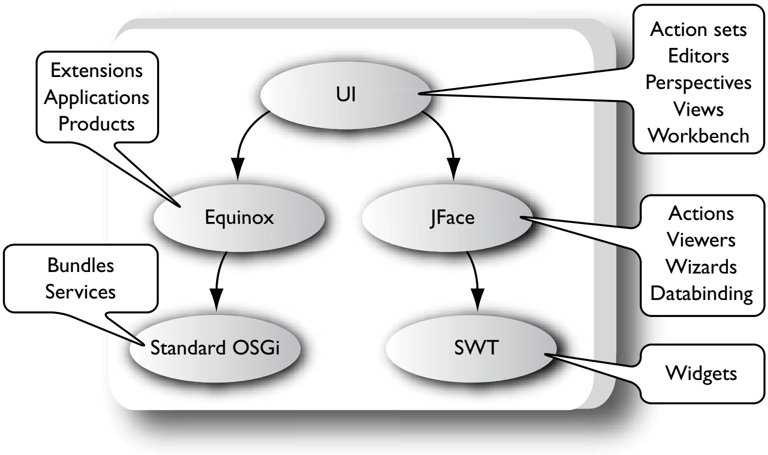

Figure 2–2 Thousand-foot RCP view

Managing the dependencies is a large part of building an Eclipse application. Plug-ins are self-describing and explicitly list the other plug-ins or functions that must be present for them to operate. The OSGi's job is to resolve these dependencies and knit the plug-ins together. It's interesting to note that these interdependencies are not there because of Eclipse but because they are implicit in the code and structure of the plug-ins. Eclipse allows you to make the dependencies explicit and thus manage them effectively.

2.2 Inside Plug-ins

Now that you've seen the 10,000- and 1,000-foot views of Eclipse, let's drop down to 100 feet and look at plug-ins, the basic building blocks of Eclipse. A plug-in is a collection of files and a manifest that describe the plug-in and its relationships to other plug-ins.

Figure 2-3 shows the layout of the `org.eclipse.ui` plug-in. The first thing to notice is that the plug-in is a Java Archive (JAR). As a JAR, it has a `MANIFEST.MF`. The manifest includes a description of the plug-in and its relationship to the rest of the world.

Plug-ins can contain code as well as read-only content such as images, Web pages, translated message files, documentation, and so on. For instance, the UI plug-in in Figure 2-3 has code in the `org/eclipse/ui/…` directory structure and other content in `icons/` and `about.html`.

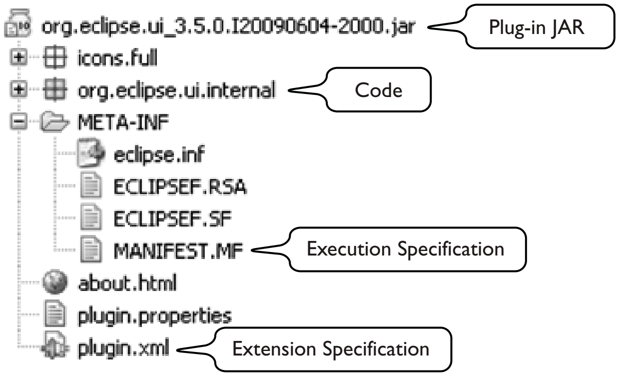

Figure 2–3 Plug-in disk layout

Notice that the plug-in also has a `plugin.xml` file. Historically, that was the home of the execution-related information now stored in the `MANIFEST.MF`. The `plugin.xml` continues to be the home of any extension and extension point declarations contributed by the plug-in.

2.3 Putting a System Together

With all these plug-ins floating around, what does an Eclipse system look like on disk? Figure 2-4 shows a typical RCP SDK install. The topmost directory is the *install location*. It includes a plug-in store, some bootstrap code, and a launcher, `eclipse.exe`, which is used to start Eclipse.

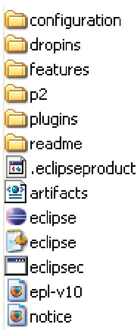

Figure 2–4 The anatomy of an Eclipse installation

The *plug-in store* (`plugins` directory) contains a directory or JAR file for each plug-in. By convention, the name in the file system matches the identifier of the plug-in and is followed by its version number. Each plug-in contains its files and folders as described earlier.

The *configuration location* (`configuration` directory) contains the configuration definition. This definition describes which plug-ins are to be installed and run. The configuration location is also available to plug-ins for storing settings and other data such as preferences and cached indexes. By default, the configuration location is part of the install location. This is convenient for standard single-user installs on machines where users have full control. Products and shared, or multiconfiguration, installs on UNIX systems may, however, put the configuration location elsewhere, such as the current user's home directory.

2.4 OSGi Framework

The Eclipse plug-in component model is based on the Equinox implementation of the OSGi framework R4.2 specification (*http://osgi.org*). You can see it at the bottom of Figure 2-5. In a nutshell, the OSGi specification forms a framework for defining, composing, and executing components or *bundles*. Think of bundles as the implementation of plug-ins. The term *plug-in* is used historically to refer to components in Eclipse and is used throughout the documentation and tooling.

There are **no fundamental or functional differences between plug-ins and bundles** in Eclipse. Both are mechanisms for grouping, delivering, and managing code. In fact, the traditional Eclipse `Plugin` API class is just a thin, optional layer of convenience functioning on top of OSGi bundles. To Eclipse, everything is a bundle. As such, we use the terms interchangeably and walk around chanting, "A plug-in is a bundle. A bundle is a plug-in. They are the same thing."

It is convenient to think of the OSGi framework as supplying a component model to Java; that is, think of it as a facility at the same level as the base JRE. OSGi frameworks manage bundles and their code by managing and segmenting their class loading—every bundle gets its own class loader. The classpath of a bundle is dynamically constructed based on the dependencies stated in its *manifest*. The manifest defines what a plug-in is and on what it depends. All plug-ins are self-describing.

The `MANIFEST.MF` shown in Figure 2-5 gives the `org.eclipse.ui` plug-in a *plug-in ID*, or *bundle symbolic name*, and a version. Common practice is to use Java package name conventions such as `org.eclipse.ui` for the identifier and [`major.minor.service.qualifier`] tuples for the version number. The ID and version are paired to uniquely identify the plug-in. The pairs are then used to express dependency relationships. You can see this in the `Require-Bundle` header of the manifest—the UI plug-in requires the Runtime, JFace, and SWT plug-ins.

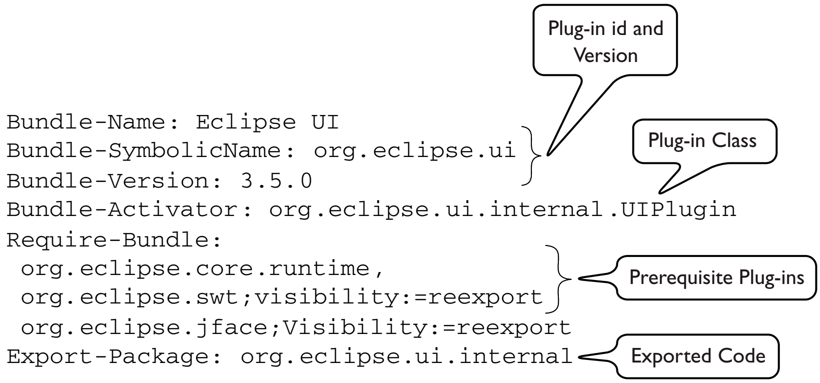

Figure 2–5 Plug-in manifest

In the context of Eclipse, OSGi's main role is to knit together the installed plug-ins, allowing them to interact and collaborate. The rigorous management of dependencies and classpaths enables tight and explicit control over bundle interactions and thus the creation of systems that are more flexible and more easily composed.

> ### OSGi and Eclipse
>
> The OSGi Alliance (*http://osgi.org*) was formed independently about the same time the Eclipse project started. Its original mission was to provide a Java component and service model for building embedded devices such as residential gateways, set-top boxes, car dashboard computers, and so on.
>
> The RCP focus during the Eclipse 3.0 development cycle spun off the Equinox technology project (*http://eclipse.org/equinox*), which explored ways of making the Eclipse runtime more dynamic and support plug-in install and uninstall without restarting. Various existing alternatives were considered, and OSGi emerged as a standard, dynamic framework, quite similar to Eclipse. As a result, Eclipse is based on the Equinox implementation of the OSGi framework specification. Eclipse 3.5 includes a stand-alone OSGi implementation in `org.eclipse.osgi_3.5.0.jar`.

2.5 Equinox

Historically, the Eclipse Runtime included the plug-in model and various functional elements. As you have seen, the plug-in or bundle model has moved down to the OSGi layer. This is implemented by the Equinox project. Most of the other functionality previously supplied by the Eclipse Runtime is now also part of the Equinox project. So we distinguish between the base standard OSGi implementation and the value-added function elements of Equinox discussed in this section.

2.5.1 Applications

Like JVMs and standard Java programs, OSGi systems have to be told what to do. To run Eclipse, someone has to define an *application*. An application is very much like the `main()` method in normal Java programs. After Equinox starts, it finds and runs the specified application. Applications are defined using *extensions*. Application extensions identify a class to use as the main entry point. When you run Eclipse, you can specify an application to run. Once invoked, the application is in full control of Eclipse. When the application exits, Eclipse shuts down.

Stand-alone versus Extension Offerings

Offerings are the things that you ship to customers. We distinguish between *stand-alone* and *extension* offerings. A stand-alone offering is one that comes as a complete set of plug-ins, with its own branding and its own application entry point—end users run stand-alone offerings.

Some stand-alone offerings are closed—they are not intended to be extended. The true power of Eclipse comes from offerings that are designed to be extended by others and thus create *platforms*. The Eclipse SDK is a platform, as are the offerings described in Chapter 1.

Extension offerings are sets of plug-ins that are incomplete and destined to be added to some platform. For example, sets of tooling plug-ins such as the Eclipse Modeling Framework (EMF), Graphical Editing Framework (GEF), and C Development Tooling (CDT), which are added to the Eclipse SDK tooling platform, are extension offerings. They do not have an entry point of their own, nor do they have substantial branding.

For most of this book these distinctions are academic. When it comes to discussions of packaging, branding, and updating, the differences become apparent.

2.5.2 Products

A *product* is a level above an application. You can run Eclipse by just specifying an application, but the product branding context (e.g., splash screen and window icons) and various bits of customization (e.g., preferences and configuration files) would be missing. The notion of a product captures this diffuse information into one concept—something that users understand and run.

NOTE

Any given Eclipse installation may include many applications and many products, but only one product and application pair can be running at a time.

2.5.3 Extension Registry

OSGi provides a mechanism for defining and running separate components and a services mechanism to support inter-bundle collaboration. Equinox adds to that a mechanism for declaring relationships between plug-ins—the *extension registry.* Plug-ins can open themselves for extension or configuration by declaring an *extension point*. Such a plug-in is essentially saying, "If you give me the following information, I will do...." Other plug-ins then *contribute* the required information to the extension point in the form of *extensions*.

The canonical example of this is the UI plug-in and its `actionSets` extension point. Simplifying somewhat, action sets are how the UI talks about menu and toolbar entries. The Eclipse UI exposes the extension point `org.eclipse.ui.actionSets` and says, "Plug-ins can contribute `actionSets` extensions that define actions with an ID, a label, an icon, and a class that implements the interface `IActionDelegate`. The UI will present that label and icon to the user, and when the user clicks on the item, the UI will instantiate the given action class, cast it to `IActionDelegate`, and call its `run()` method."

Figure 2-6 shows this relationship graphically.

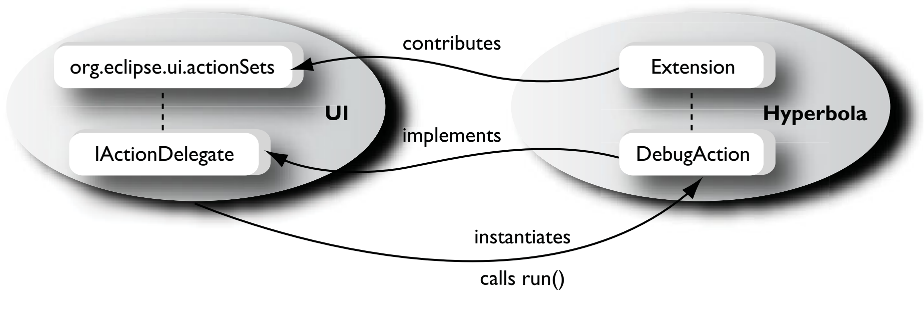

Figure 2–6 Extension contribution and use

Extension-to-extension-point relationships are defined using XML in a file called `plugin.xml`. Each participating plug-in has one of these files. In this scenario, `org.eclipse.ui`'s `plugin.xml` includes the following:

org.eclipse.ui/plugin.xml
```
<extension-point id="actionSets" name="Action Sets"/>
```

The Hyperbola plug-in, `org.eclipsercp.hyperbola`, developed later in the book, similarly contributes an extension using the markup shown in the following `plugin.xml` snippet:

org.eclipsercp.hyperbola/plugin.xml
```
<extension point="org.eclipse.ui.actionSets">
  <actionSet id="org.eclipsercp.hyperbola.debugActionSet">
    <action
        id="org.eclipsercp.hyperbola.debug"
        class="org.eclipsercp.hyperbola.DebugAction"
        icon="icons/debug.gif"
        label="Debug Chats"/>
  </actionSet>
</extension>
```

The `actionSets` extension point contract plays out as follows: The UI presents the label "Debug Chats" along with the `debug.gif` icon. When the user clicks on the action, the class `DebugAction` is instantiated and its `run()` method is called.

This seemingly simple relationship is extremely powerful. The UI has effectively opened up its implementation of the menu system, allowing other plug-ins to contribute menu items. Furthermore, the UI plug-in does not need to know about the contributions ahead of time, and no code is run to make the contributions—everything is declarative and lazy. These turn out to be key characteristics of the registry mechanism and Eclipse as a whole. Some other characteristics worth noting here are these:

- Extensions and extension points are used extensively throughout Eclipse for everything from contributing views and menu items to connecting Help documents and discovering builders that process resource changes.
- The mechanism can be used to contribute code or data.
- The mechanism is declarative—plug-ins are connected without loading any of their code.
- The mechanism is lazy in that no code is loaded until it is needed. In our example the `DebugAction` class was loaded only when the user clicked on the action. If the user does not use the action, the class is not loaded.
- This approach scales well and enables various approaches for presenting, scoping, and filtering contributions.

2.6 Standard Widget Toolkit (SWT)

Sitting beside the OSGi and Equinox is the SWT. SWT is a low-level graphics library that provides standard UI controls such as lists, menus, fonts, and colors, that is, a library that exposes what the underlying window system has to offer. As the SWT team puts it, “SWT provides efficient, portable access to the UI facilities of the OSs on which it is implemented.”

This amounts to SWT being a thin layer on top of existing windowing system facilities. SWT does not dumb down or sugarcoat the underlying window system but rather exposes it through a consistent, portable Java API. SWT is available on a wide variety of window systems and OSs. Applications that use SWT are portable among all supported platforms.

The real trick of SWT is to use native widgets as much as possible. This makes the look and feel of SWT-based applications match that of the host window system. As a result, SWT-based systems are both portable and native.

Notice that SWT does not depend on Equinox or OSGi. It is a stand-alone library that can be used outside of Eclipse or RCP.

2.7 JFace

Whereas SWT provides access to the widgets as defined by the window system, JFace adds structure and facilities for common UI notions. The UI team describes JFace as follows: “JFace is a UI toolkit with classes for handling many common UI programming tasks. JFace is window system-independent in both its API and implementation, and is designed to work with SWT without hiding it.”

It includes a whole range of UI toolkit components, from image and font registries, text support, dialogs, databinding, and frameworks for preferences and wizards to progress reporting for long-running operations. These and other JFace UI structures, such as actions and viewers, form the basis of the Eclipse UI.

2.8 UI Workbench

Just as JFace adds structure to SWT, the Workbench adds presentation and coordination to JFace. To the user, the Workbench consists of some *views* and *editors* arranged in a particular layout. In particular, the Workbench

- Provides contribution-based UI extensibility
- Defines a powerful UI paradigm with *windows*, *perspectives*, *views*, *editors*, and *actions*

2.8.1 Contribution-Based Extensibility

Whereas JFace introduces actions, preferences, wizards, windows, and so on, the Workbench provides extension points that allow plug-ins to define such UI elements *declaratively.* For example, the wizard and preference page extension points are just thin veneers over the related JFace constructs.

More than this, however, the use of extensions to build a UI has a fundamental impact on the scalability of the UI in terms of both complexity and performance. Declarative extensions enable the description and manipulation of sets of contributions such as the action sets we discussed earlier. For example, the Workbench's *capabilities* mechanism supports progressive disclosure of functionality by filtering actions until their defining action sets are triggered. Your application may have a huge number of actions, but users see only the ones in which they are interested—the UI grows with users' needs.

Since all of these extensions are handled lazily, applications also scale better. As your UI gets richer, it includes more views, editors, and actions. Without declarative extensibility, such growth requires additional loading and execution of code. This increases code bulk and startup time, and the application does not scale. With extensions, no code is loaded before its time.

2.8.2 Perspectives, Views, and Editors

The Workbench appears to the user as a collection of windows. Within each window the Workbench allows users to organize their work in much the same way as you would organize your desk—you put similar documents in folders and stack them in piles on a desk. A *perspective* is a visual container for a set of *views* and content *editors*—everything shown to the user is in a view or an editor and is laid out by a perspective.

Users organize content in perspectives in the following ways:

- Stack editors with other editors.
- Stack views with other views.
- Detach views from the main Workbench window.
- Resize views and editors and minimize/maximize editor and view stacks.
- Create fast views that are docked on the side of the window.

A perspective supports a particular set of tasks by providing a restricted set of views and supporting action sets as well as shortcuts to relevant content creation wizards, other related views, and other related perspectives. Users can switch between perspectives, for example, to change between developing code, trading stocks, working on documents, and instant messaging. Each of these tasks may have unique layouts and content.

2.9 Summary

In Eclipse, everything is a plug-in. Even the OSGi framework and the Equinox functionality show up as plug-ins. All plug-ins interact via the extension registry and public API classes. These facilities are available to all plug-ins. There are no secret back doors or exclusive interfaces—if it can be done in the Eclipse IDE, you can do it in your application.

SWT, JFace, and the UI Workbench plug-ins combine to form a powerful UI framework that you can use to build portable, highly scalable, and customizable UIs that have the look and feel of the platform on which you are running.

In short, Eclipse is an ideal technology for building modular RCPs based on OSGi.

2.10 Pointers

- The SWT page (*http://eclipse.org/swt*) has snippets and examples perfect for beginners.
- The Eclipse FAQs (*http://wiki.eclipse.org/Eclipse_FAQs*) is a great resource for some common Eclipse development questions.

Index

D

E

J

K

L

P

V

W

X

Z