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
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.

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.
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`. 
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.
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.
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.

**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](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](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.
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:

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

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

```xml
org.eclipsrcp.hyperbola/plugin.xml
<extension point="org.eclipse.ui.actionSets">
  <actionSet id="org.eclipsrcp.hyperbola.debugActionSet">
    <action id="org.eclipsrcp.hyperbola.debug" class="org.eclipsrcp.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](http://eclipse.org/swt)) has snippets and examples perfect for beginners.
- The Eclipse FAQs ([http://wiki.eclipse.org/Eclipse_FAQs](http://wiki.eclipse.org/Eclipse_FAQs)) is a great resource for some common Eclipse development questions.
Index

*, 400
., 400, 438, 453–454
/, 165, 400, 412, 416
{ } braces, 500–501
+ plus sign, 36
+ character (keystrokes), 180
$ dollar sign, 500
= sign, 296

A
About dialog text and images, 125
About image format, 125
About information, 87, 124–126, 209
AbstractHandler, 300–301
AbstractPresentationFactory, 322–323
AbstractTextEditor, 498–499
Accelerators, 179, 386–387
ActionBarAdvisor, 55, 83–86, 97–99, 220–223, 267
ActionFactory, 91, 172, 183, 274–276, 278
ActionFactory.PREFERENCES, 172
Action(s)
declarative, 265–273, 277–278
extension points, 234–235
global action handler, 277
Hyperbola, 83–87
progress reporting, 282–289
registering, 86
responsibilities, 264–265
retargetable actions, 275–277
standard actions, 274–275
and the status line, 281–282
toolbar actions, 278–281
actionSetPartAssociation, 234
actionSets, 23–24, 234
Activators, 462–468
Active (bundle), 461
activeWhen, 301
activities extension point, 235
Adapter factory, 75–77, 80–81, 149
Adapter mechanism, 75–78
add() and remove() methods, 365
addActionSet(), 246, 270
AddContactAction, 88–93, 181, 200, 300–301
AddContactAction.ID, 179
addExtension(), 361
addFastView(), 245
addNewWizardShortcut(), 246, 278
addPart(), 323, 331
addPerspectiveShortcut(), 246, 248–249
addPlaceholder(), 245, 252
addRepository, 349
AddSelectionListener(), 255, 265
addShowInPart(), 246
addStandaloneView(), 69, 245
addView(), 69–70, 242, 245, 251–252, 501
Adobe Flash/AIR, 4
Advisors. See Workbench advisors
Agent, 338
alloc(), 365
all-permissions, 436–437
Alt key, 180
Alt+Shift+F1, 43, 296
Alt+Shift+F2, 296
Anchor, 195–196
Anonymous extensions, 393–394
Ant pattern syntax, 400
Ant properties, 406, 409
Ant script generator, 398
Ant scripts, 409
Antrunner, 408
Apache, 138
Application (Eclipse), 52
ApplicationActionBarAdvisor, 91, 182, 190
Applications (Equinox), 22
Applications extension point, 52
ApplicationWorkbenchAdvisor, 52–53, 65, 144, 147, 219, 221, 244
Architecture, 338–341, 474–475
archivePrefix, 402–403
Archives, 431–432
Archiving Hyperbola, 132
Arguments, 296
Arrows, in plug-ins list, 50
Artifacts, 340
ASCII characters (key sequences), 180
Auto-login preferences, 170–175
Automatic updates, 214–215
Auto-substitution, version numbers, 414

B
Back button, 276
base (property), 404
baseLocation (property), 404
baseos, ws, arch (properties), 402, 404
BasicLoginDialog, 156
BeanProperties, 481
BeansObservables, 477
Binary build specification, 130
Binding, 264–265, 475, 484–487
Bin.excludes, 399–400
Bin.includes, 399–400, 415, 436
bmp images, 122–123
bootclasspath (property), 406
Branding Hyperbola
  About dialog (text and images), 124–126
  product configuration, 115–120
  program launcher, 121–122
  splash screen, 122–124
  window images, 120–121
Breakpoint, 57–60, 409
Browser plug-in, 502
Bug warning, 37
Build scripts, 399–400, 410
buildID, 402–404
buildLabel, 402–404
Build.properties, 399–404, 407, 417–418
buildType, 402–404
build.xml, 417, 419
Bundle cleanup, 465
Bundle lifecycle, 460–465
Bundle listeners, 363–364
Bundle pooling, 443–444
Bundle singleton, 460
BundleActivator, 451–452, 462–463
BundleContext, 451, 503
BundleContext.getService*, 459
Bundle.getEntry(String), 80
Bundles and plug-ins, 20, 27, 450
Bundle-symbolic name, 20, 47, 140, 206, 455–458, 460
Bundling Smack, 138–141

C
C Development Tooling (CDT), 22
Cache management, 132
Caching, 356–359
Callbacks, 410
Cancel button, 283
Capability mechanism, 26, 339
Categories (commands), 178–179
Chat Editor, 103–114, 147
Chat Model, 70–71
ChatAction class, 110–112
ChatEditorInput, 259
chmod, 348
-clean command-line argument, 132
cleanupcopy, 348
cleanupzip, 347
Clear workspace data before launching, 70
Closable/nonclosable, 103, 242, 246, 252–253, 323
closePart(), 325
Code reuse, 138
Code structure
  Hyperbola layering, 383–384
  icons and images, 389
  key bindings, 386–387
  optional dependencies, 388–389
  preferences, 388
  property pages, 388
  views and editors, 387–389
  wizards, 388
  workbench contributions, 384–386
Colors, 317
Command key (Mac OS X), 180
commandId, 179–180, 184, 296–299, 301
Command-line arguments, 60, 132, 435
Commands
  and actions, 386
    command category, 293
    defining, 177–182
    extension points, 235
    framework, 291–292, 297
    handlers, 299–301
    Workbench, 291–301
Common Navigator Framework (CNF), 502
Compare editor, 35–36
Compile errors, 146
compilelogs directory, 408
compilerArg (property), 403, 406
Composites, 328, 331, 350–351
configs (property), 402, 405, 418, 422
Configuration area (data area), 469–471
Configuration elements, 391
Configuration location, 20
Configuration methods, 228–229
Configuration scope, 166, 171, 173–175
ConfigureShell(Shell), 160
Configuring the Help plug-ins, 190
ConnectionDetails, 146–147, 152, 162–164, 424
ConnectionDetailsTestCase, 425
connectWithProgress(Session), 158
-console, 457, 461
-consoleLog, 60, 408
Console plug-in, 499–500
Console view, 242–244
ContactsDecorator, 256–257
ContactsGroup, 71, 73–78, 90–91, 145–146
ContactsList, 65, 74
ContactsView, 65–70, 72–79, 81, 113, 252, 256, 272
ContactsView icons, 67, 79
Content categorization, 350
Content provider, 73–77, 150
contentArea, 329–331
Context menus, 272–273
contextID, 184, 197
Context-sensitive Help (Hyperbola), 196–197
contexts.xml, 197–198
Contribution extension points, 236–237
Contributions, 294–299
Contributions area (status line), 96
Control properties, 399–401
Converters, 484–485
Coolbar, 55, 85, 92, 280
copy, 348
create(), 365
createActionBarAdvisor(), 55
createCoolBarControl(), 306
createDocument(), 499
createExecutableExtension(), 361, 392, 468
createFieldEditors(), 172
createFolder(), 245, 252
createFromClassPath(), 80
createInitialLayout(), 242
createMenuBar(), 305–306
createMonitor(), 288
createPageComposite(), 306
createPartControl(Composite), 72
createPlaceholderFolder(), 245
createStatusLine(), 306
createWindowContents(), 85, 305–306
createWorkbenchWindowAdvisor(), 222, 224
Cross-platform building, 134, 187, 398, 405, 422
Ctrl+, 146
Ctrl-3, 43
Ctrl key, 180
Custom build.xml, 400–401, 410
customAssembly, 410
Customizable toolbars, 93
Customizing
 the build scripts, 410
the Launcher, 121–122
metadata, 346
Workbench windows, 303–318
See also Presentations (Workbench)
CustomPresentationFactory, 323
customTargets.xml, 412–413
CVS, 276, 403, 411, 413, 422

D
Data areas, 469–471
Data location, 470
Databinding
 architecture, 474–475
bindings, 484–487
observables, 475–480
properties, 480–483
Debug perspective, 241–243
DebugConsole, 243–244
Debugger stepping functions, 59–60
Debugging, 55–62, 409
Declarative actions, 101, 265–273, 277–278
Declarative Services (DS), 213, 503
decorators extension point, 236
Decoupling pattern, 387, 394–395
Default scope, 166, 174
Defining commands, 177–182
Defining target platforms, 38–42
Delta pack, 37, 40, 133, 402
Dependencies, 50, 204, 383
Dependency analysis, 50
Deployed (bundle), 460
Deprecated actions, 100, 292
Descriptive extensions, 356
Descriptors (images), 79–81
Development environment installation, 33–34
Dialog.buttonPressed(), 167
DialogPageSupport, 487
DIP (Dependency Inversion Principle), 387π
Director (p2), 341, 350
Display, 156
Display.asyncExec(), 148, 283, 480
Display.syncExec(), 148, 283
dispose(), 85, 91, 231, 274, 325, 329, 365
doSaveDocument(), 499
Drag and drop, 259–262
Drop adapters, 259–260
dropAction extension points, 235
Dynamic classpaths, 141
Dynamic plug-ins
dynamic awareness, 355–364
dynamic enablement, 364–366
dynamic extension scenarios, 355–362
object handling, 362–363
Dynamic variable, 500–501

E
Early activation extensions, 465–466
earlyStartup(), 465–466
Eclipse
configuration location, 20
databinding, 474–487
Equinox, 21–24
FAQs, 27
install location, 19–20
JFace, 25
and OSGi, 20–22
platform, 491
plug-ins, 15–20, 491
runtime, 21, 32, 166, 450–452
SWT, 6, 25, 27, 460
tools, 490
touchpoint actions, 348–349
UI Workbench, 25–27
Eclipse Classic (SDK), 16–17, 19, 33–34, 56, 351, 453, 489–491
Eclipse Community Forums, 44, 490
Eclipse Java Integrated Development Environment (IDE), 5
Eclipse Modeling Framework (EMF), 22, 388, 474, 478, 490
Eclipse Rich Ajax Platform (RAP), 4
Eclipse Technology, 490
Eclipse Tutorial videos, 62
Eclipse Update Manager, 337
Eclipse User Interface Guidelines, 127
eclipse.exe, 19
Editor ID, 108
editorAction extension points, 235
editorInputTransfer, 260–261
EDITOR_MINIMUM_CHARACTERS, 226
Editors, 26–27, 65, 69, 147, 251–255, 387–388
eilders extension point, 237
elementFactories extension point, 235
eergencyClose(), 225–228
eergencyClosing(), 224–225
enabledWhen, 301
enableMultiple, 251
Encrypting passwords, 168–170
Engine, 341
Equinox
applications, 22
extension registry, 23–24
offerings, 22
products, 22–23
Event loops, 57, 222, 224–225, 227–228
Exception, 57, 227
Executable extensions, 356
Executables feature, 401–402
execute(), 300
Export Contacts, 269–272
Export wizards, 345
Exporting Hyperbola, 129–134
Exporting/reexporting, 21, 50
Export-Package, 21, 141, 456
exportWizards extension point, 236
Extensibility, 32
Extensible HyperText Markup Language (XHTML), 32, 202
Extensible Messaging and Presence Protocol (XMPP), 32, 137, 144, 243
Extension(s)
caching, 357–358
dynamically, 465–466
factory, 393–394
Hyperbola, 66, 68, 191–192
identifier, 51, 53, 393
named and anonymous, 393–394
offering, 22
registry, 23–24, 278, 357, 392, 394, 460
tracker, 360
Extension configuration (IDE), 371, 379–380
Extension delta (IExtensionDelta), 357
Extension Element Details, 51, 66, 68, 105, 184
Extension point(s)
action, 234–235
contribution, 236–237
New Extension wizard, 66, 192
perspective, 237
scalability, 235–236
startup, 238
Extension Point Reference, 234
Extension point schema, 390–391
Externalize strings, 89
Extra.library, 400

F
FAQs (Eclipse), 27
Fast Views (status line), 95
Feature builder, 417–419
Feature dependencies, 204
Feature IDs, 201, 205–206
Feature Name, 206
Feature properties, 205
Feature Provider, 206
Feature.xml, 201, 205, 346, 380–381, 414, 420
Fetching, 403–404, 411–413
Field editors, 171–172
FileLocator.resolve(URL), 441
Fill*(), 85
fillCoolBar(), 85, 92, 97, 113, 231
fillMenu(), 267, 294, 297
fillMenuBar(), 85–88, 92, 97, 113, 191, 231
fillStatusLine(), 85, 95, 97, 231, 281
Fixed layout, 246
Focus changes, 325
Fonts, 223, 236, 256–257, 282, 496–497
FormAttachment, 307–309
FormLayout, 307–308
Forms plug-in, 501
Forward button, 276
Foundation Java class libraries, 16–17
Fragments, 123, 133–134, 202, 424, 454–457
free(), 365
Free Moving perspective, 240–242, 252
“Friendly” plug-ins, 394–395

G
Galileo, 7, 188
Galileo SR1, 37
Galileo SR2, 34
getCache(), 358
getChildren(), 74–76
ggetData(), 225, 230
getDefaultPageInput(), 229
ggetExtension(), 358, 394
ggetExtensionDeltas(), 358, 360
ggetExtensionRegistry(), 392
gFactoryld(), 261
getInitialWindowPerspectiveId(), 229
getMainPreferencePageId(), 229
getMapFiles, 413
getNames(), 261
ggetObjects(), 362
ggetProgressMonitor(), 95
ggetProvider(), 361
ggetTransparencyMask(), 315
ggetViewRegistry(), 254
ggetWorkbenchConfigurer(), 225
ggetWorkbenchErrorHandler(), 229
GIF images, 78, 120, 125
Global action handler, 277
Globally unique identifiers, 47
Google Web Toolkit (GWT), 4, 474
Graphical Editing Framework (GEF), 22
GroupMarkers, 268–270

H
handleException(), 228
Handlers, 277, 299–301
HandlerUtil, 300
headless-build/customTargets, 410
“Hello, World” application, 45–55
Help (Hyperbola)
actions, 190–191
adding to the target platform, 187–189
configuring the plug-ins, 189
content, 191–196
callsensitive, 196–197
exporting plug-ins, 197–198
infopops (F1 help), 196–197
hookMinimize(), 98–100
Hospital IM scenario, 370–371
Host plug-in, 454–456
HTML, 126, 191–196
html/, 197
Hyperbola, 31–33
About information, 87, 124–126, 209
ActionBarAdvisor, 55, 85–86
Actions, 83–87
AddContact action, 88–93, 181, 200, 300–301
auto-login preferences, 170–175
automatic updates, 214–215
branding, 115–127
Chat editor, 103–114
chat model, 70–71
chatting with Eliza, 152–153
ContactsView, 65–70, 72–79, 81, 113, 197, 252, 256, 272
customizable toolbars, 93
Hyperbola (continued)
  debugging, 55–60
  development environment installation, 33–34
  examples to browse, 42–43
  exporting, 129–134
  extensions, 51, 66, 68, 191–192, 393
  "Hello, World" application, 45–55
  Help. See Help (Hyperbola)
  Help menu, 85–87
  images, 78–81, 389
  IWorkbenchAdapters, 75–77
  key bindings, 177–186
  label provider, 77–78
  launch configuration, 59–61
  launcher, 121–122
  login dialog, 155–161
  login settings, 161–170
  menus, 85–88
  messaging support, 137–154
  packaging, 129–135
  perspective, 53–54, 67
  project names, 47
  prototype, 63–65
  refactoring the model, 143–148
  running and debugging, 55–62
  sample code, 34–36
  software management, 199–215
  splash screen, 122–124
  status line, 93–96
  system tray integration, 96–100
  target platform setup, 36–42
  testing, 424–428
  third-party library, 138–143
  top-level menu, 85–87
  updating the UI, 149–151
  using commands, 294–299
  using views, 251–258
  WorkbenchAdvisor, 53
  WorkbenchWindowAdvisor, 54
  Hyperbola kiosk, 381–383
  Hyperbola layering, 383–384
  Hyperbola plug-in structure, 354
  Hyperbola product configurations
    Extension configuration (IDE), 379–380
    Hyperbola kiosk, 381–383
    JFace configuration, 377–379
    PDA configuration, 379
    Workbench configuration, 380–381
  Hyperbola projects, 374–375
  hyperbola.builder, 401–402, 407, 415, 422, 439
  HyperbolaProviderManager, 360
  Hyperbola.target, 37

I
  IAction, 292
  IActionConfigurer, 232
  IAction.setDefinitionId(String), 186
  IApplication, 52
  IArtifactRepositoryManager, 342–344
  IBM Lotus and Eclipse RCP, 7–10
  IBM Lotus Expeditor client for desktop, 9
  IBM Lotus Notes, 10
  ICommandService, 294, 300
  Icon/Message (status line), 96
  Icons, 67, 79, 120, 389
  IDE, 374, 379
  IDE platform, 16–17
  Idleness, 227
  IDocument, 497–498
  ID/version pairing, 20
  IEditorInput, 107, 109, 259–261, 499
  IEngine, 342–344
  IExtensionChangeHandler, 360
  IExtensionDelta, 357
  IExtensionRegistry, 394
  IFontProvider, 257
  IHandlerService, 300
  IImageKeys, 79
  InstallableUnit, 342–344
  Image path, 161
  ImageDescriptorFromPlugin(), 80–81, 160–161
  ImageDescriptors, 79–81
  ImageRegistry, 389, 465
  Images
    bmp, 122–123
    GIF, 78, 120, 125
    Hyperbola, 78–81, 389
    login dialog, 160–161
  IMetadataRepositoryManager, 342–343
  Import Contacts, 269–272
  importWizards extension point, 236
  Include required software (checkbox), 40, 189
  Included Features, 202–203, 208, 420, 436
  Infopops (F1 help), 196–197
  -initialize, 440
  initialize(), 224, 226
  initializeSession(), 72
  initializeTracker(), 360–361
  initializeUsers(), 163
  initTaskItem(), 98
  Install area (data area), 469–471
  Install location, Eclipse system, 19–20
  Installable units (IUs), 339–340, 343–350
  Installation and updates. See p2
  Installation Details list, 86–87, 126, 211
Installation management. See p2
installBundle, 348
Installed (bundle), 460–461
Installers, 432–433
installFeature, 348
installShield, 432
Instance area, 441, 469–471
Instance data, 60, 471
Instance scope, 166
Instant messaging client. See Hyperbola
Internationalize strings, 89
Internet Engineering Task Force (IETF), 32, 144
intro extension point, 238
intro parts, 238, 491–492
Inversion of Control (IoC), 387n
IPageLayout, 244–247
IPageLayout.addView(), 69, 242, 501
IPartService, 255
IPersistableElement, 259, 261
IPerspectiveFactory, 53, 70, 242
IPerspectiveListeners, 250
IPerspectiveRegistry, 250
IPlanner, 342–343
IProfileRegistry, 342
iqProviders, 391
IRegistryChangeEvent, 357
IRegistryChangeListener, 357
ISafeRunnable, 228, 363
ISelectionService, 255
isModifiable(), 499
isReadOnly(), 499
IUViewQueryContext, 212
IViewDescriptor, 254
IWorkbenchAdapters, 75–77, 150
IWorkbench.close(), 225, 227
IWorkbenchConfigurer, 225–229
IWorkbenchConfigurer:emergencyClose(), 226
IWorkbenchPage, 109, 229, 250
IWorkbench.restart(), 225
IWorkbenchSite, 255
IWorkbenchWindow, 67, 109, 182–183, 229–231, 252
IWorkbenchWindow.close(), 222
IWorkbenchWindowConfigurer, 54, 230, 305

Java development tooling (JDT), 16, 33, 141, 490
Java Network Launch Protocol. See JNLP (Java Network Launch Protocol)
Java Runtime Environment (JRE), 16–17, 34, 60, 409, 431
Java search, 43
Java Virtual Machine (JVM), 6, 16, 49
Java Web Start. See JNLP (Java Network Launch Protocol)
JavaBeans, 477, 481
JDK (Java SDK), 34, 437
Jetty, 190
JFace, 25, 73, 374, 377–379, 388
Jive Software, 138–141, 146, 206
JNLP (Java Network Launch Protocol), 397, 433–439
Jobs, 284–285
Jobs Progress area (status line), 96
JUnit, 424–428
junit.jar, 452

K
Key bindings, 386–387
categories, 178–179
defining commands, 178–182
extension point description, 178–179
key schemes, 184–185
key sequences, 180
keys preference page, 185
for Workbench items, 182–184
Key configuration, 226
KEY_CONFIGURATION_ID, 226
Keys without ASCII representation, 180
Keystore, 436–437
Keystroke, 180
Keytool, 437–438
Kiosk, 381–383
Kiosk (RCP) product configuration, 372–374

L
Label provider, 73, 77–78, 151
Launch configurations, 59–61
Launcher, 121–122
layoutNormal(), 308
Lazy activation, 467–468
LDAP (Lightweight Directory Access Protocol), 455
Legal Info, 126
License agreement, 204
LicenseManager, 212
Lifecycle, bundle, 460–465
Lifecycle, Workbench, 221–225
link, 347
Locale-specific files, 123–125, 454

J
Jabber Enhancement Proposals (JEPs), 32, 144, 391
JAR (Java Archive), 18–19
files, 126, 140, 203, 452–454
signing, 436–437
jarsigner, 34, 143, 437
Java class libraries (JCL), 16–17, 403, 406
locationURI, 294–299
Locking error, 38
login(), 156
Login dialog, 155
  auto-login, 170–175
  connectWithProgress(Session), 157–158
  icons, 159–160
  images, 160–161
  preferences, 164–170
  progress reports, 157
  ProgressMonitorDialog, 158
  settings, 161–170
  splash screen, 158–159
  timing the login prompt, 155–156
  user ID combo box, 161–164
  window images, 160–161
login(session), 150, 156–159
Mirroring repositories, 349–350
mkdir, 347
Modeling and reporting (BIRT), 490
Modifier keys, 180
mouseListener, 332–333
Multiple configurations, 420–422
Multiple views, 251–252
Multiple windows, 258–259
Multiuser chat (MUC), 32, 354
Multiuser installs, 441–444
MultiValidator, 486

N
Named and anonymous extensions, 393–394
Namespace, 339, 375, 460
Naming convention, 375
Naming, project/plug-in ID, 139–140
NASA, 10–11
Native installers, 432
Native touchpoint actions, 347–348
Native user experience, 5–6
Navigate >, 42–43
Navigator Content Extensions (NCE), 502
Nested JARs, 454
Nest menu managers, 88
New Extension wizard, 66, 192
New Product Configuration wizard, 115–116
New Product Definition dialog, 118
New Project wizard, 48
newWizards extension point, 236
nl directory, 123, 455
nl/en, 123
NL (National Language) fragment, 455
Node structure, 165
@nodefault, 470
@none, 470
Nonmodal progress, 284
Nonrectangular windows, 304, 312–318
NSIS, 432
NullPointerException, 57

O
Object caching, 359
Object handling, 362–363
Observables, 475–480
Observer pattern, 475
Offerings, 22
Online Help, 233
openChatEditor, 148
OpenInNewWindow, 258
Open-source code, 138
OpenViewAction, 252
openWindows(), 224, 258
openWorkbenchWindow(), 258, 274
Operations, 386
Option key (Mac OS X), 180
Optional dependencies, 388–389
org.apache.lucene, 190
org.eclipse.core.expressions, 499
org.eclipse.core.resources, 492
org.eclipse.core.runtime, 50, 491, 492, 495, 499–502
org.eclipse.core.variables, 500
org.eclipse.equinox.ds, 503
org.eclipse.equinox.internal.provisional.p2.ui.policy
   .Policy, 212
org.eclipse.equinox.preferences, 164
org.eclipse.equinox.p2.user.ui, 208, 211
org.eclipse.equinox.util, 503
org.eclipse.help, 190, 208, 491, 501
org.eclipse.help.appserver, 190
org.eclipse.help.base, 190
org.eclipse.jface, 495
org.eclipse.jface/ITreeContentProvider, 74
org.eclipse.jface.text, 243, 495–496, 498–499
org.eclipse.osgi, 503
org.eclipse.rcp, 208
org.eclipse.rp.feature.xml, 201
org.eclipse.rp/hyperbola.Application, 52
org.eclipse.rp/hyperbola/ContactsView, 72
org.eclipse.rp/hyperbola/DebugConsole, 243
org.eclipse.rp/hyperbola.extensionProviders, 356, 358
org.eclipse.rp/hyperbola.iqProviders, 356
org.eclipse.rp/hyperbola.model, 71
org.eclipse.rp/hyperbola/Perspective, 70
org.eclipse.rp/hyperbola/target, 37
org.eclipse.rp/hyperbola.ui, 384
org.eclipse.rp/hyperbola.ui.workbench, 380–381, 384
org.eclipse.swt, 495
org.eclipse.text, 243, 495–496
org.eclipse.ui, 20, 50
org.eclipse.ui, 491, 495, 499, 501–502
org.eclipse.ui/actionSets, 23, 246, 266, 292
org.eclipse.ui/bindings, 179, 184, 293
org.eclipse.ui/browser, 502
org.eclipse.ui/console, 243–244, 499–500
org.eclipse.ui/editorActions, 266
org.eclipse.ui/forms, 206, 491, 501
org.eclipse.ui/handlers, 299, 301
org.eclipse.ui/intro, 491–492
org.eclipse.ui/main.menu, 294–297
org.eclipse.ui/main.toolbar, 295–297
org.eclipse.ui/navigator, 502
org.eclipse.ui/popup.any, 295
org.eclipse.ui/popupMenus, 266, 273, 292
org.eclipse.ui/presentationMenus, 320, 322, 327
org.eclipse.ui/presentations, 323
org.eclipse.ui/presentations.r21, 321–322
org.eclipse.ui/viewActions, 266, 292
org.eclipse.ui/views, 501
org.eclipse.ui/views.ContentOutline, 295
org.eclipse.ui/views.ProblemView, 295
org.eclipse.ui/workbench.textOutline, 243, 495, 496, 499
org.jivesoftware.smack, 206
OSGi
   bundle lifecycle, 460–465
   console, 457
   data areas, 469–471
   early activation extensions, 465–466
   and the Eclipse runtime, 450–452
   fragments, 454–457
   framework, 20–21
   lazy activation, 467–468
   and lazy activation, 468
   plugins, 452–454
   services, 459–460
   start level, 467
   version numbering, 457–459
OSGi Alliance, 21
OSGi and Equinox, 215
OSGi Framework Specification Release 4, 450
osgi.bundles, 466
Outline view, 501

P
p2
   agent, 339, 341, 443
   API, 342–344
   artifacts, 340
   director, 341, 350
   engine, 341
   installable units (IUs), 339–340, 343–350
   installation phases, 346–347
   installer, 433
   metadata management, 345–349
   profiles, 341
   repositories, 340, 349–350, 405–406
   roles, 337–338
   touchpoints, 341, 347–349
p2.artifact.repo, 406, 418
p2.compress, 406
p2.gathering, 403, 405–406, 418–419, 422
p2.metadata.repo, 406, 418
Packaging Hyperbola
archive file, 132
binary build specification, 130
cache management, 132
delta pack, 133
exporting, 129–134
platform-specific code, 132–134
Product Export wizard, 131, 134
synchronization, 131
Packet listener, 145
Paint listener, 330
Part list menu, 333
Part listeners, 255–256
PartTab, 331–333
PDA configuration, 374, 379
PDE Build, 345, 397
benefits, 398
debugging, 409
plug-in build.properties, 399–401
root files, 415–416
templates, 410
Persistence (stack), 325
Perspective
factory, 68–70, 237, 242, 251–252
Hyperbola, 53–54, 65, 67, 69
IPerspectiveFactory, 53, 70, 242
IPerspectiveListeners, 250
menu, 248–249
registry, 250
Workbench, 26–27
PerspectiveDebug, 242, 244, 249
perspectiveExtensions extension point, 237
PerspectiveFreeMoving, 242, 249, 252
Perspectives (Workbench)
adding, 240–242
Console view, 242–244
debugging, 250
IPageLayout, 244–247
IPerspectiveListeners, 250
IPerspectiveRegistry, 250
IWorkbenchPage, 250
perspective bar, 226–227, 247–250, 304, 312
perspective extension points, 237
perspective menu, 248–249
SwitchPerspectiveAction, 249–250
Placeholder, 231, 245, 252–253, 267–272
Platform design, 390–394
Platform Developer Guide, 233
Platform.getAdapterManager(), 77, 150–151
Platform.getExtensionRegistry(), 392
Platform.getProduct(), 160
Platform.getStateLocation(Bundle), 164
Platform-specific code, 132–134
Plug-in(s)
arrows, 50
build.properties, 399–401
and bundles, 20, 27, 450
content, 49
content definitions, 46–47
Eclipse, 15–19, 491
editor, 48–50
fragments, 454–457
Help, 188–190, 197–198
ID, 20, 47, 140, 206, 455–458, 460
ID and project name, 139
ID and version, 20–21
RCP-friendly, 394
reexported, 50
sources, 490
store, 20
Plugin class, 462–463
Plug-in Development Environment (PDE), 33, 36–37
Plug-in Spy, 43, 296
pluginPath (property), 402, 405, 410–411
plugins/customBuildCallbacks, 410
plug-in.xml, 19, 320
/plugin-id/preference-name, 194
POJO, 424, 475–476
PojoObservables, 475–476
PojoProperties, 481
popupMenu extension point, 235
Portability of clients, 6
postSetup, 412
postShutdown(), 222, 224
postStartup(), 224
postWindowCreate(), 316
postWindowOpen(), 230
preferenceCustomization property, 194
preferencePages extension point, 236
PreferencePageSupport, 487
Preferences, 388
auto-login, 170–175
help system, 193
initializer, 174
login dialog, 164–170
node structure, 165–166, 168
Workbench, 226–227, 274–275
Preferences.flush(), 167
preferences.ini, 194, 198, 227, 247, 321
Preinitialized configurations, 440
presenceToKey(), 80–81
Presentable parts, 319
presentationFactories extension point, 237, 320–321
presentationFactoryID, 322
PRESENTATION_FACTORY_ID, 226, 321
Presentations (Workbench)
  adding, selecting, removing parts, 331–333
classes, 323
default features, 324–325, 325–326
menus, 333
plug-in, 322
presentation factory, 327
samples, 320–322
size and position, 330–331
StackPresentation, 324–330
widgets, 324
preShutdown(), 222, 224
preStartup(), 222, 224, 287
preWindowOpen(), 54, 94, 230, 305
Product configuration, 115–120, 205, 371–383
Product definitions, 376
Product Export wizard, 131, 134
Product (property), 403
productBuild.xml, 401, 408–409, 419
ProfileChangeRequest, 344
Profiles, 341
Program launcher, 121–122
Progress area (status line), 96
Progress reports, 157, 282–289
Progress view, 285–287
ProgressMonitorDialog, 158, 288
ProgressProvider, 287–288
Project name and plug-in ID, 139
Project names, 47, 375
Properties, databinding, 480–483
Property listener, 332
Property pages, 388
Property view, 501
propertyPages extension point, 236
Prototype, 63–65
Prototype classes, 146–147
Providers (content and label), 73–78
ProvisioningContext, 343–344
Publisher, 345–346
Publishing metadata, 345–346
Quick fix, 191
Quick search panel, 307, 311–312
Quit action, 177, 183, 275
Realm.asyncExec(), 480
Realms, 480
Rectangular windows, 304, 312–318
Reexported plug-ins, 50
Refactoring, 143–148
Region (shell), 313
Register adapters, 77
RegisterContextMenu(), 273
Registering Actions, 86
Registering listeners, 464
Registering services, 464
registerObject(), 361–362
Registry change events, 357
Registry change listeners, 357
Release engineering builds, 350, 397–399, 417
remove(), 365
remove (touchpoint action), 348
removeExtension(), 361–362
removePart(), 331
Rendering, 264–265
repoBaseLocation, 411, 418
Repositories, 212, 340, 342–344, 411
repository directory, 408
Repository management, 349–350
RepositoryManipulator, 212
Resolved (bundle), 461
Resources plug-in, 492–493
ResourcesPlugin.getWorkspace(), 451, 494
Restore, 162, 221–222, 224–225, 323, 325
restoreState(), 224, 323, 325
Restructuring Hyperbola, 372–374
Retargetable actions, 275–277
Reverse domain name convention, 47
revertPerspective(), 250
Rich client, 3–5
Rich client platform (RCP), 5–10
Rich client platform (RCP) SDK, 37
Rich Internet applications (RIAs), 4
rmdir, 347
root.os.ws.arch, 416
Root files, 415–416
Roster classes, 144–150
rowLayout, 328–330
RPM, 432
R21 presentation, 320–322
Q
qualifier, 414–415, 458
$qualifier$, 346
Qualifying version numbers, 414
run(), 52, 89, 112, 228
Run Configurations (dialog), 61
Run in background, 284–285
Run menu, 61
Runtime widget parenting, 324

S
Sample code, 34–36, 369
Samples Manager, 35–36
SAT4J, 341
saveState(), 224–225, 261, 323, 325
Scalability extension points, 235–236
schemeId, 180, 184
ScopedPreferenceStore(), 171–172
Scopes, 165–168, 172–175
scriptsproductBuild productBuild.xml, 401
SDK features, 420–422, 453
search >, 43
Secondary ID, 251–253
SecurePreferencesFactory, 168–169
Selection listener, 89
selectionChanged(), 89–90, 271
selectPart(), 323, 325
Separator, 93, 268
Services, 459–460
ServiceTracker, 459
Session, 71, 146–147
setActionDefinitionId(), 181, 183
setActive(), 323, 325
setAfterConvertValidator(), 486
setAfterGetValidator(), 486
setBounds(), 317, 323, 324, 329–331
setCloseable(), 242, 246, 252
setConnection(), 152
setConnectionDetails(), 152, 156, 173
setContentProvider(), 73–74
setControl(), 280
setConverter, 484
setData(), 225, 230
setEditorAreaVisible(), 69, 113, 246
setMessage(), 95
setExitOnLastWindowClose(), 225
setFixed, 246
setInitialSize(), 229
setMessage(), 95
setProgramProperty, 349
setProperty(), 286
setQueryContext(), 212
setRegion(), 313–314, 317
setRepositoryManipulator(), 212
setSaveAndRestore(), 65, 221, 224
setSelectionProvider(), 255
setShowMenuBar(), 229
setShowStatusLine(), 94
setShowToolbar(), 229
setStartLevel, 348
setTitle(), 54, 119, 229
setVisible(), 323–324
Shared installs, 442–444
Shift key, 180
Short status (ss) command, 456–457
Shortlist, 246, 248, 254, 278
Show In prompter, 246
showMenu(), 325
showPart(), 331
SHOW_PROGRESS_ON_STARTUP, 226
SHOW_TRADITIONAL_STYLE_TABS, 226–227
Simple clients, 3
Singleton, 456, 460
skipBase (property), 402, 404, 407
skipFetch, 405, 422
skipMaps, 403, 405, 412–413
Smack, 32, 206
APIs, 144–145
bundling, 138–141
chat editors, 147–148
Jive Software, 138
library, 138, 154
message types, 144
naming conventions, 139–140
refactoring, 146–147
Roster classes, 145
stream management, 144
testing, 141–143
Software Configuration Management (SCM) access
control, 405, 411–414
Software management
automatic updates, 214–215
branding Features, 209–210
categories, 213–214
customizing the p2 UI, 211–213
Equinox p2, 199–200
features, 200–210
Hyperbola, 199–215
updating, 210–211, 214–215
Software sites, 38
source.library, 400
Splash screen, 122–124, 158–159
Stack persistence, 325
StackPresentation, 324–330
Stand-alone offering, 22
<table>
<thead>
<tr>
<th><strong>Index</strong></th>
<th>517</th>
</tr>
</thead>
</table>

Stand-alone product configuration, 371–374
Standalone views, 69, 245
Standard actions, 274–275
Start level, 467
Start(BundleContext), 450–452, 461–464, 467
startChat(), 147–148
Starting (bundle), 461
startup extension point, 238
Startup time, 464
State location, 470
Status line, 93–96, 281–282
statusHandlers extension point, 237
Step functions, debugging, 59–60
Sticky views, 253–254
Stop(BundleContext), 450, 452, 461–464
stopMethod(), 365
Stopping (bundle), 461
Stream management, 144
Strings, externalize, 89
StyledText, 497
Support classes, 487
SVN, 411, 413
SwitchPerspectiveAction, 249–250
SWT (Standard Widget Toolkit), 6, 25, 27, 460
display, 156
fragments, 202
SWTBot, 426–428
SWTObservables, 478–479
SWTWorkbenchBot, 427
Synchronization, 118, 131
Synchronization error, 38
System menu, 333
System tray integration, 96–100
systemSummarySections extension point, 237

T

Tab styles, 226
TAR, 431–432
Target content, 42
Target Definition wizard, 39
Target editor warning, 37
Target Export wizard, 402
.target files, 37
Target platform setup, 36–42
Task tray, 64, 84, 97–100
Templates, 410
templatesheadless-build, 401
Test and performance tools platform (TPTP), 490
Test case, 424–426
Testing Hyperbola, 424–428
Text control, 497
Text editing, 495–499
Text plug-ins, 495–496
TextEditors, 498
TextViewer, 498
themes extension point, 236
Thin clients, 4–6
Third-party library, 31–32, 138–143
titleArea, 328–333
TitleAreaDialogSupport, 487
TOC files (Help), 191, 195–197
togettingstarted.xml, 195, 197
toctasks.xml, 195–197
toc.xml, 195–197
Toggle actions, 309
Toolbars
customization, 93
Hyperbola, 93
show/hide, 309
text, 279
Workbench, 278–281
See also Coolbar
Top-level feature, 376
Top-level menu, 84–87
Top-level toolbar, 92
topLevelElementID, 417–418
topLevelElementType, 417–418
Touchpoints, 341–342, 347–349
Transfer types, 259
transformedRepoLocation, 411, 418
Translations, 89, 123, 125, 210
TrayItem, 97–99
TreeViewer, 72–73, 90

U

UI Workbench, 25–27
Unhandled event loop exception, 57
uninstallBundle, 348
Uninstalled (bundle), 461
uninstallFeature, 348
Unit testing, 424–426
Unregister adapters, 77
unzip, 347
updateEnablements(), 310–311
UpdateValueStrategy, 483–484, 486
Updating Hyperbola, 210–211
Updating software, 210, 214
Updating the UI, 149–151
User area (data area), 469–471
User interface testing, 426–429
@user.dir, 471
@user.home, 470
validateBeforeSet(), 486
Validation, 485–486
Validity testing, 362–363
Value variable, 500–501
Variable manager, 500–501
Variables, 500–501
$version$, 346
Version numbers, 414, 457–459
Version substitution, 346
View icons, 67
View ID, 66, 251–252, 254, 275, 501
View pane menu, 333
View registry, 254
View shortlist, 254, 278
viewActions extension point, 235
Viewers, compared with Views, 73
ViewersObservables, 479
ViewerSupport, 487
ViewPart, 65–66
Views, 65
 compared with editors, 104
 compared with viewers, 73
 and editors, 251–258, 387–389
 Workbench, 26–27
views extension point, 237
visibleWhen, 294, 299

Weak references, 361–363
Web tools platform (WTP), 490
WidgetProperties, 482
Widgets, 25, 85, 156, 170
Wildcards, 42–43, 253
Window images, 120–121
Windows, customizing, 303–318
WizardPageSupport, 487
Wizards, 388
Workbench
 action responsibilities, 265
 actions, 263–289
 advisor types, 220–221
 closing, 225–226
 commands, 291–301
 configuration, 228–229, 374, 380–381
 contributions, 294–299, 384–386
 declarative actions, 265–273
drag and drop, 259–262
extension factory, 185
extension points, 232–238
handlers, 299–301
Keys preference page, 185
lifecycle, 221–225
multiple windows, 258–259
perspectives, 240–250
preferences, 226–227, 274–275
presentations, 319–334
progress reports (feedback), 282–289
retargetable actions, 275–277
standard actions, 274–275
status line, 281–282
toolbars, 278–281
Views and Editors, 251–258
windows, customizing, 303–318

Workbench advisors
 ActionBarAdvisor, 55, 85–86, 97–99, 220–223, 267
IActionBarConfigurer, 232
IWorkbenchConfigurer, 225–229
IWorkbenchWindowConfigurer, 230, 305
WorkbenchAdvisor, 53–54, 220–229, 260
WorkbenchAdvisor.eventLoopException(), 227
WorkbenchAdvisor.initialize(), 65, 222
WorkbenchObservables, 480
WorkbenchProperties, 482
WorkbenchWindow, 85, 97, 222
workingSets extension point, 236
Workspace, 494
Workspace Data, 60, 70

XHTML (Extensible Hypertext Markup Language), 32, 202
XML, 24, 144, 149, 243, 392, 433, 503
XMPP (Extensible Messaging and Presence Protocol), 32, 137, 144, 243
XMPPConnection, 142, 144, 147
XULRunner, 4

Zero-argument constructor, 392
ZIP, 431–432