



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 we have a common base of understanding and terminology. Writing RCP applications is subtly different than 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 as its role as a component framework. The basic unit of function in this framework is called a *plug-in*—the unit of modularity in Eclipse. Everything in Eclipse is a plug-in. An RCP application is a collection of plug-ins and a *Runtime* 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 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.

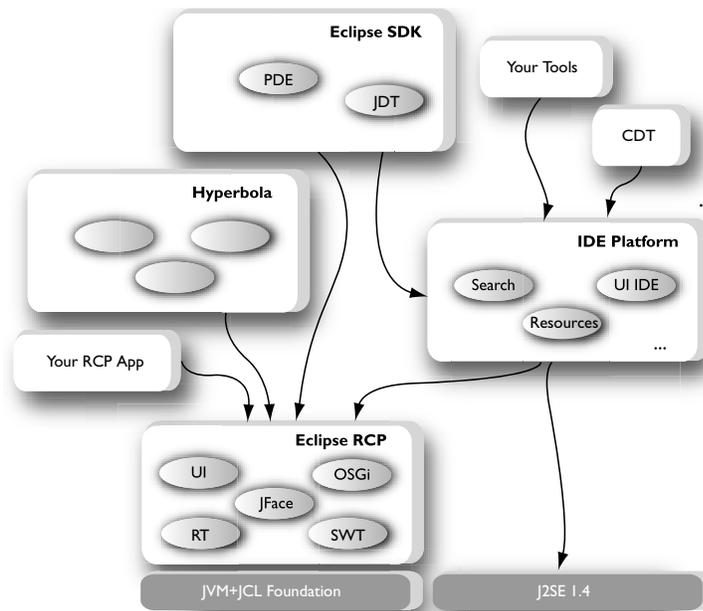
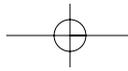


Figure 2–1 10,000-foot system architecture view

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 that wants some plug-in inside a box, there is someone who wants it outside. That's OK. You can build your own abstractions.

Fanning upwards 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 your products is the same—they are both 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, then 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 structure—an example of uniformity. 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.

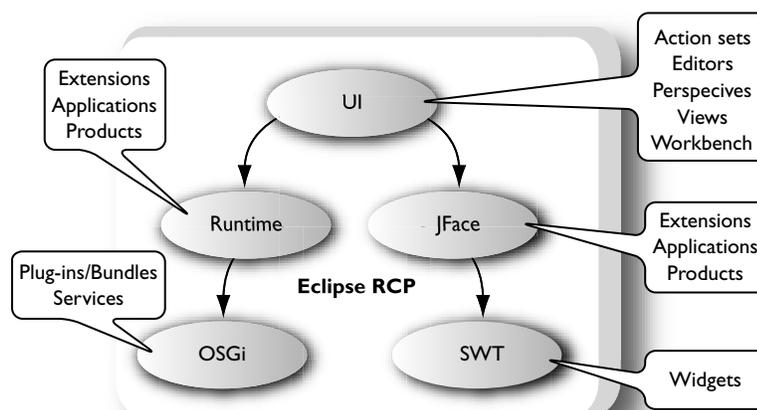
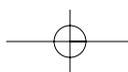
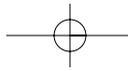


Figure 2–2 1,000-foot RCP view





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 Runtime'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.

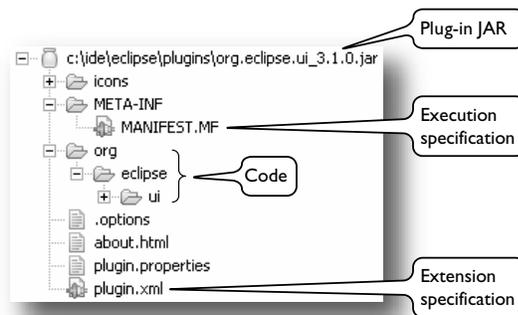
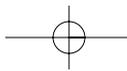


Figure 2-3 Plug-in disk layout

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), `org.eclipse.ui_3.1.0.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 and/or 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.



Eclipse 3.1 vs. 3.0

Readers familiar with Eclipse 3.0 may be surprised by some of the characterizations here. This book is written to reflect the best practices for Eclipse 3.1. In particular, Eclipse 3.1 plug-ins are delivered as JARs rather than as directories. The execution-related information formerly kept in the `plugin.xml` file is now in the `MANIFEST.MF` file. Of course, Eclipse 3.1 is fully backward-compatible with both the Eclipse 3.0 approach to plug-in definition and delivery and legacy plug-ins.

Unless otherwise stated, the remainder of the book details Eclipse 3.1.

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 top-most 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.

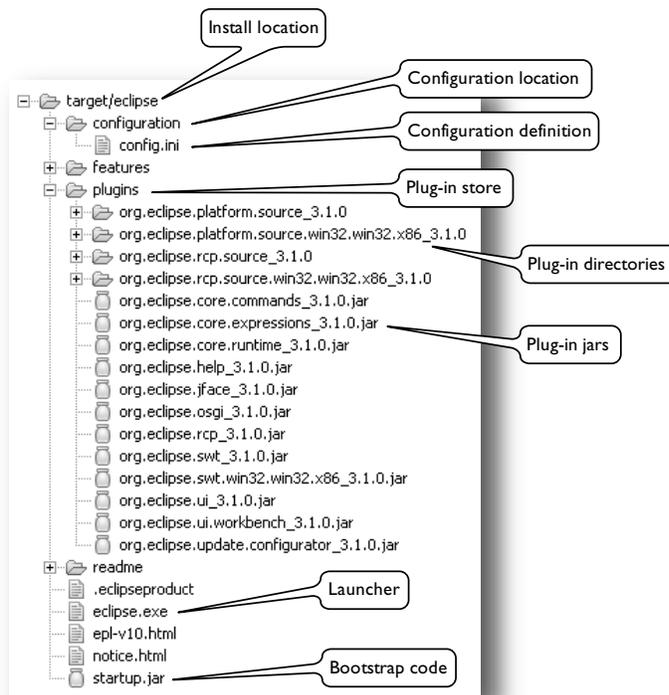
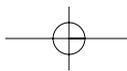
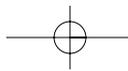


Figure 2-4 The anatomy of an Eclipse installation





The *plug-in store* contains a directory or JAR file for each plug-in. By convention, the name in the filesystem 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* contains the configuration definition. This definition describes which plug-ins are to be installed into the Runtime and run by Eclipse. 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 multi-configuration, 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 an implementation of the OSGi framework R4.0 specification (<http://osgi.org>). You can see it at the bottom of Figure 2-4. 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 content. 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 classloading—every bundle gets its own classloader. The classpath of a bundle is dynamically constructed based on the dependencies stated in its *manifest*. The manifest defines who a bundle is and on whom it depends. All bundles 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 together 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.

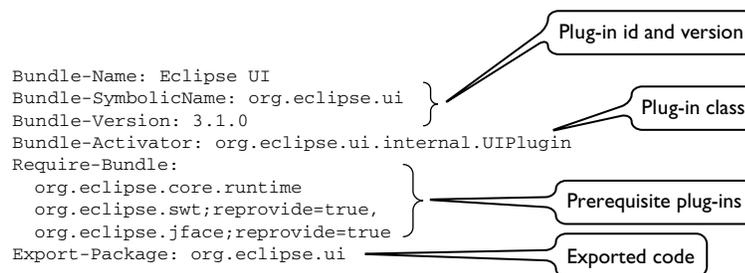
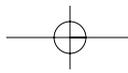


Figure 2-5 Plug-in manifest

In the context of Eclipse, the OSGi framework'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.

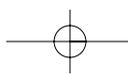
OSGi and Eclipse

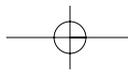
The OSGi Alliance (<http://osgi.org>) was formed independently about the same time the Eclipse project started. Its original mission was to provide a Java component and service model for building embedded devices such as residential gateways, set-top boxes, car dashboard computers, and so on.

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 Runtime more dynamic to 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 3.x is based on an implementation of the OSGi framework specification and Eclipse 3.1 includes a standalone OSGi implementation in `org.eclipse.osgi_3.1.0.jar`. See <http://eclipse.org/osgi> for more details.

2.5 The Runtime

Historically, the Eclipse Runtime also included the plug-in model. As you have seen, the plug-in model has moved down to the OSGi layer. This leaves the remainder of the Runtime on top. The Runtime is home to several key mechanisms, in particular, the application model and extension registry.





2.5.1 Applications

Like the OSGi framework and JVMs, the Eclipse Runtime has 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 the Runtime 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.

Standalone vs. extension offerings

Offerings are the things that you ship to customers. We distinguish between *standalone* and *extension* offerings. A standalone 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 standalone offerings.

Some standalone 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.

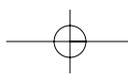
Extension offerings are sets of plug-ins that are incomplete and destined to be added to some platform. For example, sets of tooling plug-ins such as the Eclipse Modeling Framework (EMF), Graphical Editor Framework (GEF), and C Development Tooling (CDT), which are added to the Eclipse SDK tooling platform,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

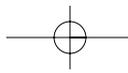
For most of this book, these distinctions are academic. When it comes to discussions of packaging, branding, and updating, the differences become apparent.

2.5.2 Products

The notion of a *product* is a level above applications. 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 set of 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

The OSGi specification provides a mechanism for defining and running separate components. The Eclipse Runtime 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.

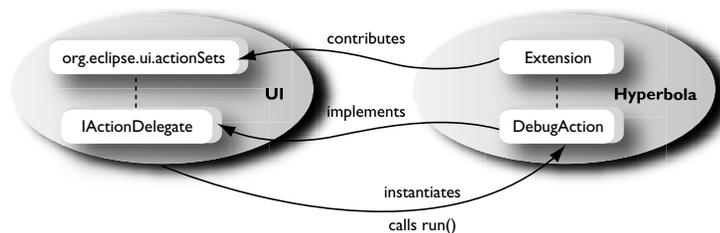


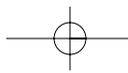
Figure 2–6 Extension contribution and use

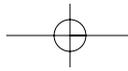
Extension-to-extension point relationships are defined using XML in a file called `plugin.xml`. Each participating plug-in has one of these files. In this scenario, `org.eclipse.ui`'s `plugin.xml` includes the following:

```

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

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





```
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>
```

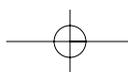
The `actionSets` extension point contract plays out as follows: The UI presents the label “Debug Chats” along with the `debug.gif` icon. When the user clicks on the action, the class `DebugAction` is instantiated and its `run()` method is called.

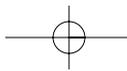
This seemingly simple relationship is extremely powerful. The UI has effectively opened up its implementation of the menu system, allowing other plug-ins to contribute menu items. Further, the UI plug-in does not need to know about the contributions ahead of time and no code is run to make the contributions—everything 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:

- 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 only loaded 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 SWT

Sitting beside the Runtime is the Standard Widget Toolkit (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 the Runtime or OSGi framework. It is a standalone 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, 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.

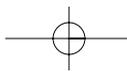
2.8 UI Workbench

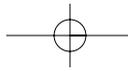
As JFace adds structure to SWT, the Workbench adds presentation and coordination to JFace. To the user, the Workbench consists of *views* and *editors* arranged in a particular layout. In particular, the Workbench:

- 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 both in terms of 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 function by filtering actions until their defining action sets are triggered. Your application may have a huge number of actions, but the user sees only the ones in which she is interested—the UI grows with the user's 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.

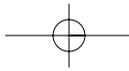
2.8.2 Perspectives, Views, and Editors

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 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 Runtime 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 rich client applications.

