Chapter 5. All about Plug-Ins

Part I discussed the Eclipse ecosystem: how to run it, how to use it, and how to extend it. In this chapter, we revisit the topic of plug-ins and lay the groundwork for all plug-in development topics to be discussed in later chapters. This chapter answers questions about the core concepts of the Eclipse kernel, including plug-ins, extension points, fragments, and more. All APIs mentioned in this chapter are found in the `org.eclipse.core.runtime` plug-in.

What is a plug-in?

FAQ 94

In retrospect, *plug-in*, perhaps wasn't the most appropriate term for the components that build up an Eclipse application. The term implies the existence of a socket, a monolithic machine or grid that is being plugged into. In Eclipse, this isn't the case. A plug-in connects with a universe of other plug-ins to form a running application. The best software analogy compares a plug-in to an object in object-oriented programming. A plug-in, like an object, is an encapsulation of behavior and/or data that interacts with other plug-ins to form a running program.

A better question in the context of Eclipse is, What isn't a plug-in? A single Java source file, `Main.java`, is not part of a plug-in. This class is used only to find and invoke the plug-in responsible for starting up the Eclipse Platform. This class will typically in turn be invoked by a native executable, such as `eclipse.exe` on Windows, although this is just icing to hide the incantations required to find and launch a Java virtual machine. In short, just about everything in Eclipse is a plug-in.

More concretely, a plug-in minimally consists of a *plug-in manifest file*, `plugin.xml`. This manifest provides important details about the plug-in, such as its name, ID, and version number. The manifest may also tell the platform what Java code it supplies and what other plug-ins it requires, if any. Note that everything except the basic plug-in description is optional. A plug-in may provide code, or it may provide only documentation, resource bundles, or other data to be used by other plug-ins.

A plug-in that provides Java code may specify in the manifest a concrete subclass of `org.eclipse.core.runtime.Plugin`. This class consists mostly of convenience methods for accessing various platform utilities, and it may also

implement `startup` and `shutdown` methods that define the lifecycle of the plug-in within the platform.

☞ **FAQ 96** *What is the plug-in manifest file (`plugin.xml`)?*
FAQ 98 *What are extensions and extension points?*

FAQ 95

Do I use *plugin* or *plug-in*?

That depends. Those with a slavish devotion to the dictates of the English language—copy editors, English teachers, and automatic spellcheckers—will insist that because "plugin" is not a recognized word, the hyphenated plug-in is required. Others—writers, hackers, and the general public—recognize that language is an organic structure that must adapt and evolve to remain relevant. This latter group welcomes the introduction of new words into the lexicon and, thus, will happily adopt the new word *plugin*. While we use *plug-in* in this book, here is a more practical answer: If you are searching in the documentation, use plug-in. If you are grepping or writing code, use plugin. If you do not think *grepping* is a word, you are a lost cause.

FAQ 96

What is the plug-in manifest file (`plugin.xml`)?

The plug-in manifest file, `plugin.xml`, describes how the plug-in extends the platform, what extensions it publishes itself, and how it implements its functionality. The manifest file is written in XML and is parsed by the platform when the plug-in is loaded into the platform. All the information needed to display the plug-in in the UI, such as icons, menu items, and so on, is contained in the manifest file. The implementation code, found in a separate Java JAR file, is loaded when, and only when, the plug-in has to be run. This concept is referred to as *lazy loading*. Here is the manifest file for a simple plug-in:

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin id="com.xyz.myplugin" name="My Plugin"
   class="com.xyz.MyPlugin" version="1.0">
   <runtime>
      <library name="MyPlugin.jar"/>
   </runtime>
   <requires>
```

```
        <import plugin="org.eclipse.core.runtime"/>
    </requires>
</plugin>
```

The processing instructions at the beginning specify the XML version and character encoding and that this plug-in was built for version 3.0 of the Eclipse Platform. The `plugin` element specifies the basic information about the plug-in, including, in this case, the optional `class` attribute to specify an instance of the `Plugin` class associated with this plug-in. Because it contains a subclass of `Plugin`, this plug-in must include a `runtime` element that specifies the JAR file that contains the code and a `requires` element to import the `org.eclipse.core.runtime` plug-in where the superclass resides. The manifest may also specify *extensions* and *extension points* associated with the plug-in. Of all this, only the `plugin` element with the `id`, `name`, and `version` attributes are required.

FAQ 94 *What is a plug-in?*
FAQ 98 *What are extensions and extension points?*
FAQ 101 *When does a plug-in get started?*

How do I make my plug-in connect to other plug-ins?

Like members of a community, plug-ins do not generally live in isolation. Most plug-ins make use of services provided by other plug-ins, and many, in turn, offer services that other plug-ins can consume. Some groups of plug-ins are tightly related, such as the group of plug-ins providing Java development tools—the JDT plug-ins—and other plug-ins, such as SWT, stand-alone without any awareness of the plug-ins around them. Plug-ins can also expose a means for other plug-ins to customize the functionality they offer, just as a handheld drill has an opening that allows you to insert other attachments such as screwdrivers and sanders. When designing a plug-in, you need to think about what specific plug-ins or services it will need, what it will expose to others, and in what ways it wants to allow itself to be customized by others.

To rephrase all this in Eclipse terminology, plug-ins define their interactions with other plug-ins in a number of ways. First, a plug-in can specify what other plug-ins it *requires*, those that it absolutely cannot live without. A UI plug-in will probably require the SWT plug-in, and a Java development tool will usually

require one or more of the JDT plug-ins. Plug-in requirements are specified in the *plug-in manifest file* (`plugin.xml`). The following example shows a plug-in that requires only the JFace and SWT plug-ins:

```
<requires>
   <import plugin="org.eclipse.jface"/>
   <import plugin="org.eclipse.swt"/>
</requires>
```

Your plug-in can reference *only* the classes and interfaces of plug-ins it requires. Attempts to reference classes in other plug-ins will fail.

Conversely, a plug-in can choose which classes and interfaces it wants to expose to other plug-ins. Your plug-in manifest must declare what libraries (JARs) it provides and, optionally, what classes it wants other plug-ins to be able to reference. This example declares a single JAR file and exposes classes only in packages starting with the prefix `com.xyz.*`:

```
<runtime>
  <library name="sample.jar">
    <export name="com.xyz.*"/>
  </library>
</runtime>
```

Finally, a plug-in manifest can specify ways that it can be customized (*extension points*) and ways that it customizes the behavior of other plug-ins (*extensions*).

☞ **FAQ 96** *What is the plug-in manifest file (`plugin.xml`)?*
FAQ 98 *What are extensions and extension points?*
FAQ 104 *What is the classpath of a plug-in?*

FAQ 98

What are extensions and extension points?

A basic rule for building modular software systems is to avoid tight coupling between components. If components are tightly integrated, it becomes difficult to assemble the pieces into different configurations or to replace a component with a different implementation without causing a ripple of changes across the system.

Loose coupling in Eclipse is achieved partially through the mechanism of extensions and extension points. The simplest metaphor for describing extensions

and extension points is electrical outlets. The outlet, or socket, is the extension point; the plug, or light bulb that connects to it, the extension. As with electric outlets, extension points come in a wide variety of shapes and sizes, and only the extensions that are designed for that particular extension point will fit.

When a plug-in wants to allow other plug-ins to extend or customize portions of its functionality, it will declare an extension point. The extension point declares a contract, typically a combination of XML markup and Java interfaces, that extensions must conform to. Plug-ins that want to connect to that extension point must implement that contract in their extension. The key attribute is that the plug-in being extended knows nothing about the plug-in that is connecting to it beyond the scope of that extension point contract. This allows plug-ins built by different individuals or companies to interact seamlessly, even without their knowing much about one another.

The Eclipse Platform has many applications of the extension and extension point concept. Some extensions are entirely *declarative*; that is, they contribute no code at all. For example, one extension point provides customized key bindings, and another defines custom file annotations, called *markers*; neither of these extension points requires any code on behalf of the extension.

Another category of extension points is for overriding the default behavior of a component. For example, the Java development tools include a code formatter but also supply an extension point for third-party code formatters to be plugged in. The resources plug-in has an extension point that allows certain plug-ins to replace the implementation of basic file operations, such as moving and deletion.

Yet another category of extension points is used to group related elements in the user interface. For example, extension points for providing views, editors, and wizards to the UI allow the base UI plug-in to group common features, such as putting all import wizards into a single dialog, and to define a consistent way of presenting UI contributions from a wide variety of other plug-ins.

FAQ 85 *How do I declare my own extension point?*
FAQ 94 *What is a plug-in?*
FAQ 96 *What is the plug-in manifest file (`plugin.xml`)?*
FAQ 97 *How do I make my plug-in connect to other plug-ins?*
FAQ 99 *What is an extension point schema?*
Go to **Platform Plug-in Developer Guide > Programmer's Guide > Platform architecture**

FAQ 99

What is an extension point schema?

Each extension point has a *schema* file that declares the elements and attributes that extensions to that point must declare. The schema is used during plug-in development to detect invalid extensions in the `plugin.xml` files in your workspace and is used by the schema-based extension wizard in the plug-in Manifest Editor to help guide you through the steps to creating an extension. Perhaps most important, the schema is used to store and generate documentation for your extension point. The schema is *not* used to perform any runtime validation checks on plug-ins that connect to that extension point. In fact, extension point schema files don't even need to exist in a deployed plug-in.

The exact format of the schema file is an implementation detail that you probably don't want to become familiar with. Instead, you should use the graphical schema editor provided by the Plug-in Development Environment.

☞ **FAQ 85** *How do I declare my own extension point?*
FAQ 88 *Can my extension point schema contain nested elements?*
FAQ 98 *What are extensions and extension points?*

FAQ 100

How do I find out more about a certain extension point?

To find out more about a given extension point, try the following.

- Consult **Platform Plug-in Developer Guide** > **Reference** > **Extension Points Reference**. Here you will find the official documentation for all extension points, including extension point schema descriptions and examples.

- While adding a plug-in in the PDE Manifest Editor, click the **Details** button (Figure 5.1). It will take you to the page discussed above.

- Perform **Search** > **Plug-in Search** (the keyboard shortcut is Ctrl+H). You can search for a given extension point and also find all contributors to a given extension point. This gives you valuable access to examples of how the rest of the platform uses that extension point.

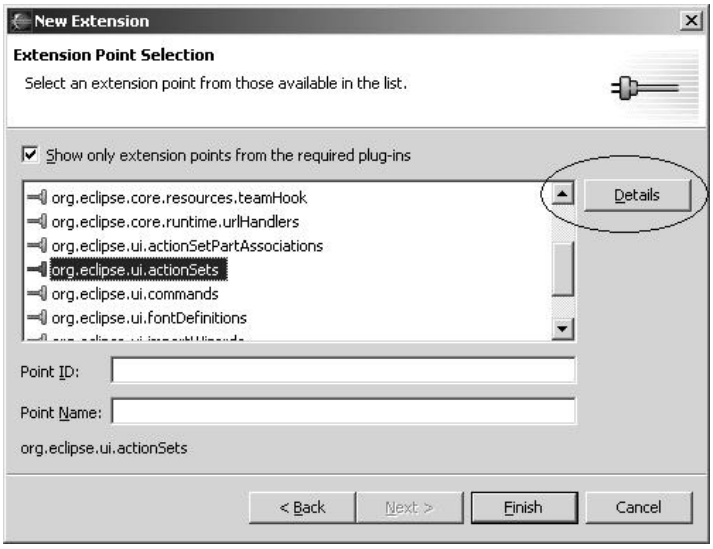

Figure 5.1 The **Details** button with more extension point information

When does a plug-in get started?

FAQ 101

A plug-in gets started when the user needs it. In UI design, an often cited rule is that the screen belongs to the user. A program should not make changes on the screen that the user didn't somehow initiate. Making users feel that they are in control of what is happening builds their confidence in the UI and results in a much pleasanter user experience. This rule is followed by the Eclipse UI, but the underlying principle has been applied to a much broader scope. In Eclipse, one of the goals is to have the screen, the CPU, and the memory footprint belong to the user; that is, the CPU should not be doing things the user didn't ask it to do, and memory should not be bloated with functions that the user may never need.

This principle is enforced in the Eclipse Platform through lazy plug-in activation. Plug-ins are activated only when their functionality has been explicitly invoked by the user. In theory, this results in a relatively small start-up time and a memory footprint that starts small and grows only as the user begins to invoke more and more functionality.

The extension point mechanism plays an important role in lazy activation. Each plug-in can be viewed as having a *declarative* section and a code section. The declarative part is contained in the `plugin.xml` file. This file is loaded into a registry when the platform starts up and so is always available, regardless of

whether a plug-in has started. This allows the platform to present a plug-in's functionality to the user without going through the expense of loading and activating the code segment. Thus, a plug-in can contribute menus, actions, icons, editors, and so on, without ever being loaded. If the user tries to run an action or open a UI element associated with that plug-in, only then will the code for that plug-in be loaded.

To get down to specifics, a plug-in can be activated in three ways.

1. If a plug-in contributes an *executable extension*, another plug-in may run it, causing the plug-in to be automatically loaded. For more details, read the API javadoc for `IExecutableExtension` in the `org.eclipse.core.runtime` package.

2. If a plug-in exports one of its libraries (JAR files), another plug-in can reference and instantiate its classes directly. Loading a class belonging to a plug-in causes it to be started automatically.

3. Finally, a plug-in can be activated explicitly, using the API method `Platform.getPlugin()`. This method returns a fully initialized plug-in instance.

In all these cases, if the plug-in contributes a runtime plug-in object (subclassing `org.eclipse.core.runtime.Plugin`), its class initializer, constructor, and `startup` method will be run before any other class in the plug-in gets loaded. Of course, if the plug-in's constructor or `startup` method references any of its own classes, they will be loaded and, possibly, instantiated before the plug-in is fully initialized.

It is a common misconception that adding a plug-in to your `requires` list will cause it to be activated before your plug-in. This is *not true*. Your plug-in may very well be loaded and used without plug-ins in your `requires` list ever being started. Never assume that another plug-in has been started unless you know you have referenced one of its classes or executed one of its extensions.

To play along with the rule of lazy activation, plug-in writers should follow some general rules.

- Do an absolute minimum of work in your `Plugin.startup` method. Does the code in your `startup` method need to be run immediately? Do you need to load those large in-memory structures right away? Consider deferring as much work as possible until it is needed.
- Avoid referencing other plug-ins during your `Plugin.startup`. This can result in a sequence of cascading plug-in activations that ends up loading large amounts of unneeded code. Load other plug-ins—either through executable extensions or by referencing classes—only when you need them.
- When defining extension points, make the extension *declarative* as much as possible. Keep in mind that extensions can contribute text strings, icons, and simple logic statements via `plugin.xml`, allowing you to defer or possibly completely avoid plug-in activation.

FAQ 96 *What is the plug-in manifest file (`plugin.xml`)?*
FAQ 182 *Can I activate my plug-in when the workbench starts?*

Where do plug-ins store their state?

FAQ 102

Plug-ins store data in two standard locations. First, each plug-in has its own install directory that can contain any number of files and folders. The install directory must be treated as read-only, as a multi-user installation of Eclipse will typically use a single install location to serve many users. However, your plug-in can still store read-only information there, such as images, templates, default settings, and documentation.

The second place to store data is the plug-in state location. Each plug-in has within the user's workspace directory a dedicated subdirectory for storing arbitrary settings and data files. This location is obtained by calling the method `getStateLocation` on your `Plugin` instance. Generally, this location should be used only for cached information that can be recomputed when discarded, such as histories and search indexes. Although the platform will never delete files in the plug-in state location, users will often export their projects and preferences into a different workspace and expect to be able to continue working with them.

If you are storing information that the user may want to keep or share, you should either store it in a location of the user's choosing or put it in the preference store. If you allow the user to choose the location of data, you can always store the location information in a file in the plug-in state location.

Plug-ins can store data that may be shared among several workspaces in two locations. The *configuration location* is the same for all workspaces launched on a particular configuration of Eclipse plug-ins. You can access the root of this location by using `getConfigurationLocation` on `Platform`. The *user location* is shared by all workspaces launched by a particular user and is accessed by using `getUserLocation` on `Platform`.

Here is an example of obtaining a lock on the user location:

```
Location user = Platform.getUserLocation();
if (user.lock()) {
   // read and write files
} else {
   // wait until lock is available or fail
}
```

Note that these locations are accessible to all plug-ins, so make sure that any data stored here is in a unique subdirectory based on your plug-in's unique ID. Even then, keep in mind that a single user may open multiple workspaces simultaneously that have access to these areas. If you are writing files in these shared locations, you must make sure that you protect read-and-write access by locking the location.

☞ **FAQ 103** *How do I find out the install location of a plug-in?*
FAQ 111 *What is a configuration?*
FAQ 123 *How do I load and save plug-in preferences?*

FAQ 103

How do I find out the install location of a plug-in?

You should generally avoid making assumptions about the location of a plug-in at runtime. To find resources, such as images, that are stored in your plug-in's install directory, you can use URLs provided by the `Platform` class. These URLs use a special Eclipse Platform protocol, but if you are using them only to read files, it does not matter.

The following snippet opens an input stream on a file called `sample.gif` located in a subdirectory, called `icons`, of a plug-in's install directory:

```
Bundle bundle = Platform.getBundle(yourPluginId);
Path path = new Path("icons/sample.gif");
URL fileURL = Platform.find(bundle, path);
InputStream in = fileURL.openStream();
```

If you need to know the file system location of a plug-in, you need to use `Platform.resolve(URL)`. This method converts a platform URL to a standard URL protocol, such as HyperText Transfer Protocol (HTTP), or file. Note that the Eclipse Platform does not specify that plug-ins must exist in the local file system, so you cannot rely on this method's returning a file system URL under all circumstances in the future.

FAQ 102 *Where do plug-ins store their state?*

What is the classpath of a plug-in?

FAQ 104

Developers coming from a more traditional Java programming environment are often confused by classpath issues in Eclipse. A typical Java application has a global namespace made up of the contents of the JARs on a single universal classpath. This classpath is typically specified either with a command-line argument to the VM or by an operating system environment variable. In Eclipse, each plug-in has its own unique classpath. This classpath contains the following, in lookup order:

- *The OSGi parent class loader*. All class loaders in OSGi have a common parent class loader. By default, this is set to be the Java boot class loader. The boot loader typically only knows about `rt.jar`, but the boot classpath can be augmented with a command-line argument to the VM.

- *The exported libraries of all imported plug-ins*. If imported plug-ins export their imports, you get access to their exported libraries, too. Plug-in libraries, imports, and exports are all specified in the `plugin.xml` file.

- *The declared libraries of the plug-in and all its fragments*. Libraries are searched in the order they are specified in the manifest. Fragment libraries are added to the end of the classpath in an unspecified order.

In Eclipse 2.1, the libraries from the `org.eclipse.core.boot` and `org.eclipse.core.runtime` were also automatically added to every plug-in's classpath. This is not true in 3.0; you now need to declare the runtime plug-in in your manifest's `requires` section, as with any other plug-in.

FAQ 96 *What is the plug-in manifest file (`plugin.xml`)?*
FAQ 97 *How do I make my plug-in connect to other plug-ins?*
FAQ 105 *How do I add a library to the classpath of a plug-in?*
FAQ 106 *How can I share a JAR among various plug-ins?*
FAQ 107 *How do I use the context class loader in Eclipse?*

FAQ 105

How do I add a library to the classpath of a plug-in?

In **FAQ 104**, we explained how the classpath for a plug-in is computed. To access a given library from a plug-in, the library needs to be added to the classpath of the plug-in.

A JAR can be added to the classpath of a plug-in in four ways.

1. The JAR can be added to the boot classpath. This is generally a bad idea, however, as it requires an extra VM argument, and it also affects the classpath of all other installed plug-ins. If the JAR adds types—classes or interfaces—that mask types declared in other plug-ins, you will probably break those other plug-ins. Nonetheless, if you are looking for a quick and dirty hack, this is the easiest approach.

2. The JAR can be added to the declared libraries for a plug-in. This is fine if you don't anticipate a need for other plug-ins also to use that JAR.

3. A new plug-in can be created that is a wrapper for the library; then the new plug-in is added to the list of required plug-ins for all plug-ins that want access to the library.

4. The OSGi parent loader can be changed by setting the `osgi.parentClassloader` system property on startup. This is also generally a bad idea, for the same reasons listed for changing the boot classpath. Valid values for the parent loader property are:

    - `boot`. The Java boot class loader. This is the default OSGi parent loader, and has access to all JARs on the VM's boot classpath.
    - `ext`. The Java extension class loader. This class loader has access to the JARs placed in the `ext` directory in the JVM's install directory. The parent of the extension loader is typically the boot class loader.
    - `app`. The Java application class loader. This class loader has access to the traditional classpath entries specified by the `-classpath` command-line argument. In Eclipse this typically includes only the bootstrap classes in `startup.jar`. The parent of the application class loader is the extension class loader.
    - `fwk`. The OSGi framework class loader. This is the class loader that is responsible for starting the OSGi framework. Typically you will not want to use the class loader, as its classpath is not strictly specified.

Using a separate plug-in to contain a library is the most powerful approach because it means that other plug-ins can make use of that library without having to load your plug-in or add the library to their own classpath explicitly. This approach is used throughout the Eclipse Project to add third-party libraries, such as Xerces, Ant, and JUnit.

Of course, because this is Java, there is always a way to load classes outside the scope of your classpath. You can instantiate your own class loader that knows how to find the code you need and use that to load other classes. This is a very powerful mechanism because it can change dynamically at runtime, and it can even load classes that aren't in your file system, such as classes in a database or even classes generated on the fly. Manipulating class loaders is a bit outside the scope of this book, but plenty of information is available in Java programming books or at the Java Web site (http://java.sun.com).

FAQ 106 *How can I share a JAR among various plug-ins?*
FAQ 109 *What is a plug-in fragment?*

FAQ 106

How can I share a JAR among various plug-ins?

Suppose that plug-in *A* and plug-in *B* both use `xmlparser.jar`. In your workspace are two projects (for *A* and *B*), each containing a copy of the `xmlparser.jar` library. This is clearly not ideal: Two copies of the JAR are loaded at runtime, and classes from those JARs will not be compatible with each other, as they are loaded by different class loaders. (You will get a `ClassCastException` if you try to cast a type from one library into a type from the other library.)

Declaring `xmlparser.jar` as an external JAR does not work, as there is no easy way during deployment of your plug-ins to manipulate your plug-in's classpath so that they can see the library. The best way to share libraries is to create a new plug-in that wraps the library you want to share.

Declare a new plug-in, *C*, to contain the library JAR, and make both plug-in *A* and plug-in *B* dependent on plug-in *C*. Make sure that plug-in *C* exports its library so other plug-ins can see it:

```
<runtime>
   <library name="xmlParserAPIs.jar">
      <export name="*"/>
   </library>
</runtime>
```

When you deploy these three plug-ins, they will all share the same library. Note that in some situations, sharing libraries between plug-ins is not possible. If two plug-ins require different or incompatible versions of the same library, they have no choice but to each have a copy of the library.

☞ **FAQ 104** *What is the classpath of a plug-in?*
FAQ 105 *How do I add a library to the classpath of a plug-in?*

FAQ 107

How do I use the context class loader in Eclipse?

In Java, each thread can optionally reference a *context class loader*. This loader can be set at any time by an application and is used for loading classes only when it is explicitly requested to do so. Many code libraries, in particular Java Database Connectivity (JDBC) and Xerces, use the context class loader in factory methods to allow clients of the library to specify what class loader to use. Although the context loader is not used by Eclipse itself, you may need to be aware of it when referencing third-party libraries from within Eclipse.

By default, the context loader is set to be the application class loader, which is not used in Eclipse. Because Eclipse has a separate class loader for each installed plug-in, a default class loader generally does not make sense as the context loader for a given thread. If you are calling third-party libraries that rely on the context loader, you will need to set it yourself. The following code snippet sets the context class loader before calling a library. Note that the code politely cleans up afterward by resetting the context loader to its original value:

```
Thread current = Thread.currentThread();
ClassLoader oldLoader = current.getContextClassLoader();
try {
   current.setContextClassLoader(getClass().getClassLoader());
   //call library code here
} finally {
   current.setContextClassLoader(oldLoader);
}
```

FAQ 104 *What is the classpath of a plug-in?*

FAQ 108

Why doesn't Eclipse play well with Xerces?

Many plug-ins in the Eclipse Platform require an XML parser for reading and storing various data in XML format. In particular, the platform uses an XML parser to read each plug-in's `plugin.xml` file during start-up. Because the JDK before 1.4 did not provide an XML parser, Eclipse used to ship with a plug-in containing a Xerces parser. Xerces is one of the two XML parser implementations maintained by the Apache Project.

The first problems began to appear when users tried to start Eclipse using a JDK that also contained an implementation of Xerces. Prior to JDK 1.4, it was common practice to throw a copy of Xerces or Xalan into the JDK's `ext` directory so that it could be used by all applications. Thus, two copies of Xerces were available when Eclipse was starting up, one in the `ext` directory and one in the `org.apache.xerces` plug-in. Because libraries provided by the JDK always appear at the beginning of the runtime classpath, the one in the JDK is always found first. If this copy of Xerces was slightly different from the one the platform expected, various linkage errors or `ClassCastException`s occurred on start-up, often preventing Eclipse from starting up at all. The workaround in this case was pretty straightforward: Omit the `ext` directory from the classpath when starting Eclipse:

```
eclipse -vmargs -Djava.ext.dirs=
```

The situation became worse with JDK 1.4, which added a specification for XML called Java API for XML Processing (JAXP). The exact parser implementation was left unspecified, so each JDK was free to choose its own, as long as it was compliant with the interfaces defined by JAXP. Sun decided to use the Apache Crimson parser, and IBM went with the Apache Xerces parser. Worst of all, because these packages were now part of the standard JDK libraries, there was no easy workaround as there has been for the `ext` problem. The JDK people, realizing that they had messed up by bundling such widely used packages in their JDKs, thereby breaking any application that used slightly different versions of those packages, plan to prefix the implementation package names with a unique prefix to prevent these collisions in the future. However, it was too late to fix this problem for JDK 1.4. The bottom line: You cannot use Eclipse 2.1 or earlier with an IBM 1.4 VM.

For Eclipse 3.0, the problem was solved by tossing out the Xerces plug-in and simply using JAXP for any XML processing. Now at most one Xerces is on the classpath, so there cannot be any collisions. Of course, this means that you need at least JDK 1.4 to run Eclipse 3.0.

FAQ 109

What is a plug-in fragment?

Sometimes it is useful to make part of a plug-in optional, allowing it to be installed, uninstalled, or updated independently from the rest of the plug-in. For example, a plug-in may have a library that is specific to a particular operating system or windowing system or a language pack that adds translations for the plug-in's messages. In these situations, you can create a fragment that is associated with a particular host plug-in. On disk, a fragment looks almost exactly the same as a plug-in, except for a few cosmetic differences.

- The manifest is stored in a file called `fragment.xml` instead of `plugin.xml`.
- The top-level element in the manifest is called `fragment` and has two extra attributes—`plugin-id` and `plugin-version`—for specifying the ID and version number of the host plug-in.
- The fragment manifest does not need its own `requires` element. The fragment will automatically inherit the `requires` element of its host plug-in. It can add `requires` elements if it needs access to plug-ins that are not required by the host plug-in.

Apart from these differences, a fragment appears much the same as a normal plug-in. A fragment can specify libraries, extensions, and other files. When it is loaded by the platform loader, a fragment is logically, but not physically, merged into the host plug-in. The end result is exactly the same as if the fragment's manifest were copied into the plug-in manifest, and all the files in the fragment directory appear as if they were located in the plug-in's install directory. Thus, a runtime library supplied by a fragment appears on the classpath of its host plug-in. In fact, a Java class in a fragment can be in the same package as a class in the host and will even have access to package-visible methods on the host's classes. The methods `find` and `openStream` on `Plugin`, which take as a parameter a path relative to the plug-in's install directory, can be used to locate and read resources stored in the fragment install directory.

FAQ 94 *What is a plug-in?*

FAQ 110

Can fragments be used to patch a plug-in?

A common misconception is that a fragment can be used to patch or replace functionality in its host plug-in. Although this is possible to a certain extent, this is not what fragments were designed for. A plug-in and its fragments each contribute a manifest, and each may also contribute native libraries, Java code libraries, and other resources. At runtime, these contributions are all merged into a single manifest and a single namespace of libraries and resources. If a fragment defines the same library as its host, whether the fragment's library will be found over the host's library is undefined. This makes it impractical to use fragments as a way of replacing libraries or other resources defined by a plug-in.

Nonetheless, it is possible to design a plug-in so that it allows a portion of its functionality to be implemented or replaced by a fragment. Let's look at a notable example of how this is applied in the `org.eclipse.swt` plug-in. The SWT plug-in manifest declares a runtime library by using a special path-substitution variable:

```
<library name="$ws$/swt.jar">
```

When the plug-in manifest is loaded, the platform will substitute the `$ws$` variable with a string describing the windowing system of the currently running operating system. Each windowing system has a separate SWT plug-in fragment that will provide this library. For example, when running on windows, `$ws$` will resolve to `ws/win32`. You can make use of this path-substitution facility in your own plug-in code by using the `Plugin.find` methods. The fragment `org.eclipse.swt.win32` supplies the `swt.jar` library at the path `org.eclipse.swt.win32/ws/win32/swt.jar`. Thus, in this case the fragment will supply a library that was specified by its host plug-in.

The same principle can be used to allow a fragment to provide a patch to a host plug-in. The host plug-in can specify both its own library and a patch library in its plug-in manifest:

```
<runtime>
   <library name="patch.jar">
      <export name="*"/>
   </library>
   <library name="main.jar">
      <export name="*"/>
   </library>
</runtime>
```

The host plug-in puts all its code in `main.jar` and does not specify a `patch.jar` at all. When no patch is needed, the `patch.jar` library is simply missing from the classpath. This allows a fragment to be added later that contributes the `patch.jar` library. Because the host plug-in has defined `patch.jar` at the front of its runtime classpath, classes in the patch library will be found before classes in the original library.

This technique is used in Eclipse 3.0 to provide backward-compatibility support for plug-ins based on Eclipse 2.1 or earlier. The plug-in `org.eclipse.ui.workbench` defines a library called `compatibility.jar` at the start of its classpath. When the platform detects a plug-in written prior to Eclipse 3.0, a fragment called `org.eclipse.ui.workbench.compatibility` containing `compatibility.jar` is automatically added to the plug-in's classpath. This library adds back some old API that was moved in Eclipse 3.0. The beauty of this mechanism is that it allows the backward-compatibility support to be added or removed with no impact on the host plug-in.

☞ **FAQ 104** *What is the classpath of a plug-in?*

What is a configuration?

FAQ 111

A configuration is the set of plug-ins available in a particular instance of the Eclipse Platform. A given installation of Eclipse may contain hundreds or even thousands of plug-ins. More than one Eclipse-based application can share this same install location, but they don't always want to use all the same plug-ins. When Eclipse is started, a *configurator* determines what subset of the installed pool of plug-ins will be used for that particular instance of the platform. By default, all installed plug-ins will be in the configuration, but a configuration can be customized to contain different groups of plug-ins. Go to **Help > Software Updates > Manage Configuration** to see and modify what plug-ins are in your configuration.

☞ **FAQ 243** *What is the minimal Eclipse configuration?*

FAQ 112

How do I find out whether the Eclipse Platform is running?

If you have a library of code that can be used within both the Eclipse Platform and a stand-alone application, you may need to find out programmatically whether the Eclipse Platform is running. In Eclipse 3.0, this is accomplished by calling `Platform.isRunning`. In 2.1, call `BootLoader.isRunning`. You will need to set up the classpath of your stand-alone application to make sure that the boot or runtime plug-in's library is reachable. Alternatively, you can reference the necessary class via reflection.

You can find out whether an Ant script is running from within Eclipse by querying the state of the variable `eclipse.running`. You can use this information to specify targets that are built only when Ant is invoked from within Eclipse:

```
<target name="properties" if="eclipse.running"/>
```

FAQ 315 *What is Ant?*

FAQ 113

Where does `System.out` and `System.err` output go?

Most of the time, the answer is *nowhere*. Eclipse is simply a Java program, and it acts like any other Java program with respect to its output streams. When launched from a shell or command line, the output will generally go back to that shell. In Windows, the output will disappear completely if Eclipse is launched using the `javaw.exe` VM. When Eclipse is launched using `java.exe`, a shell window will be created for the output.

Because the output is usually lost, you should avoid using standard output or standard error in your plug-in. Instead, you can log error information by using the platform logging facility. Other forms of output should be written to a file, database, socket, or other persistent store. The only common use of standard output is for writing debugging information, when the application is in debug mode. Read up on the platform tracing facility for more information.

FAQ 121 *How do I use the platform logging facility?*
FAQ 122 *How do I use the platform debug tracing facility?*

How do I locate the owner plug-in from a given class?

You can't. Some known hacks were used prior to Eclipse 3.0 to obtain this information, but they relied on implementation details that were not strictly specified. For example, you could obtain the class's class loader, cast it to `PluginClassLoader`, and then ask the class loader for its plug-in descriptor. This relied on an assumption about the class loading system that is subject to change and, in fact, has changed in Eclipse 3.0. The correct answer to this question is that there is no way to reliably determine this information. If you are exploiting knowledge of the Eclipse runtime's implementation to obtain this information, expect to be foiled when the runtime implementation changes.

How does OSGi and the new runtime affect me?

Just when you thought you were beginning to understand how the Eclipse kernel worked, those pesky Eclipse developers replaced it all for Eclipse 3.0. The kernel is now built on another Java component framework, the Open Services Gateway initiative (OSGi). The reasons for this convergence between Eclipse and OSGi are manifold. The two frameworks had many similarities to begin with, and each framework had many features that the other lacked.

By bringing the two together, Eclipse gained the infrastructure for many new features, especially dynamic addition and removal of plug-ins and a more robust security model. Eclipse in turn has a powerful declarative model—extensions and extension points—that OSGi lacked, in addition to more advanced support for multiple versions, fragments, a commercial-quality open source implementation, and great tooling support. Rather than creating a derivative OSGi++, the Eclipse community is contributing a number of important Eclipse features back into the OSGi specification, paving the way for better interoperability between the two frameworks. All in all, it's what the marketing types like to call a win-win situation.

Now, to the question of how plug-ins are affected: The new runtime is 100 percent backward compatible with the runtime that existed in all versions before Eclipse 3.0. Plug-ins written prior to 3.0 will continue to run without requiring any modification. When you port a plug-in to 3.0, you can still make use of the old runtime API by explicitly importing the backward-compatibility layer, which is found in a separate plug-in. Although the `boot` and `runtime` plug-ins were imported automatically prior to Eclipse 3.0, the runtime must now be imported explicitly. The new runtime compatibility plug-in contains the deprecated portions of the API from the `boot` and `runtime` plug-ins and also exports the new runtime plug-in. In short, all you now have to import is the new runtime compatibility plug-in, and you will get access to both the new runtime API and the old. The following example from a `plugin.xml` file imports both the old and new runtimes:

```
<requires>
   <import plugin="org.eclipse.core.runtime.compatibility"/>
</requires>
```

Apart from that one change, you can continue using runtime facilities as you did prior to Eclipse 3.0. Over time, more elements of the old runtime will likely become deprecated, and plug-ins will begin to use the equivalent OSGi APIs instead. However, for release 3.0, the focus is on getting the technology in place with minimal disruption to the rest of the platform. For now, the fact that Eclipse is running on OSGi is an implementation detail that will not significantly affect you.

FAQ 116 *What is a dynamic plug-in?*
The OSGi Web site (http://www.osgi.org)

FAQ 116

What is a dynamic plug-in?

Prior to Eclipse 3.0 the platform had to be restarted in order for added, removed, or changed plug-ins to be recognized. This was largely owing to the fact that the plug-in registry was computed statically at start-up, and no infrastructure was available for changing the registry on the fly. In Eclipse 3.0, plug-ins can be added or removed dynamically, without restarting. Dynamicity, however, does not come for free. Roughly speaking, plug-ins fall into four categories of dynamicity.

1. *Nondynamic* plug-ins do not support dynamicity at all. Plug-ins written prior to Eclipse 3.0 are commonly in this category. Although they can still often be dynamically added or removed, there may be unknown side effects. Some of these plug-ins' classes may still be referenced, preventing the plug-ins from being completely unloaded. A nondynamic plug-in with extension points will typically not be able to handle extensions that are added or removed after the plug-in has started.

2. *Dynamic-aware* plug-ins support other plug-ins being dynamic but do not necessarily support dynamic addition or removal of themselves. The generic workbench plug-in falls into this category. It supports other plug-ins that supply views, editors, or other workbench extensions being added or removed, but it cannot be dynamically added or removed itself. It is generally most important for plug-ins near the bottom of a dependency chain to be dynamic aware.

3. *Dynamic-enabled* plug-ins support dynamic addition or removal of themselves, but do not necessarily support addition or removal of plug-ins they interact with. A well-behaved plug-in written prior to Eclipse 3.0 should already be dynamically enabled. Dynamic enablement involves following good programming practices. If your plug-in registers for services, adds itself as a listener, or allocates operating system resources, it should always clean up after itself in its inherited `Plugin.shutdown` method. A plug-in that does this consistently is already dynamic enabled. It is most important for plug-ins near the top of a dependency chain to be dynamic enabled.

4. *Fully dynamic* plug-ins are both dynamic aware and dynamic enabled. A system in which all plug-ins are fully dynamic is very powerful, as any individual plug-in can be added, removed, or upgraded in place without taking the system down. In fact, such a system would never have any reason to shut down as it could heal itself of any damage or bug by doing a live update of the faulty plug-in. OSGi, the Java component architecture that is used to implement the Eclipse kernel, is designed around this goal of full dynamicity.

The dynamicity of a plug-in depends on the dynamic capabilities of the plug-ins it interacts with. Even if a plug-in is fully dynamic, it may not be possible to cleanly remove it if a plug-in that interacts with it is not dynamic aware. As long as someone maintains a reference to a class defined in a plug-in, that plug-in cannot be completely removed from memory. When a request is made to

dynamically add or remove a plug-in, the platform will always make a best effort to do so, regardless of the dynamic capabilities of that plug-in. However, there may be unexpected errors or side effects if all plug-ins in the system that reference it are not well-behaved, dynamic citizens.

FAQ 117 *How do I make my plug-in dynamic enabled?*
FAQ 118 *How do I make my plug-in dynamic aware?*

FAQ 117

How do I make my plug-in dynamic enabled?

Most of the effort required to make a plug-in dynamic enabled can be summed up as doing what you should be doing anyway as part of good programming practice. Most importantly, to be dynamic enabled, your plug-in has to properly clean up after itself in the `Plugin` `shutdown` method. You need to keep in mind the following checklist for your plug-in's `shutdown` method.

- If you have added listeners to notification services in other plug-ins, you need to remove them. This generally excludes any listeners on SWT controls created by your plug-in. When those controls are disposed of, your listeners are garbage collected anyway.
- If you have allocated SWT resources, such as images, fonts, or colors, they need to be disposed of.
- Any open file handles, sockets, or pipes must be closed.
- Any metadata stored by other plug-ins that may contain references to your classes needs to be removed. For example, session properties stored on resources in the workspace need to be removed.
- Other services that require explicit uninstall need to be cleaned up. For example, the runtime plug-in's adapter manager requires you to unregister any adapter factories that you have manually registered.
- If your plug-in has forked background threads or jobs, they must be canceled and joined to make sure that they finish before your plug-in shuts down.

Prior to Eclipse 3.0, the consequences of failing to clean up properly were not as apparent as plug-ins were shut down only when the VM was about to exit. In a potentially dynamic world, the consequence of not being tidy is that your plug-in cannot be dynamic enabled.

☞ **FAQ 116** *What is a dynamic plug-in?*
FAQ 118 *How do I make my plug-in dynamic aware?*

How do I make my plug-in dynamic aware?

Dynamic awareness requires extra steps that were not required prior to the introduction of dynamic plug-ins. Dynamic awareness requires that you remove all references to classes defined in other plug-ins when those plug-ins are removed from the system. In particular, if your plug-in defines extension points that load classes from other plug-ins—executable extensions—you need to discard those references when other plug-ins are dynamically removed. The extension registry allows you to add a listener that notifies you when extensions are being added or removed from the system. If your plug-in maintains its own cache of extensions that are installed on your extension point, your listener should update this cache for each added or removed extension.

The following is an example of a simple class that maintains its own cache of the set of extensions installed for a given extension point. This example is a bit contrived as simply caching the extension objects has no value. Typically, your plug-in will process the extensions to extract useful information and possibly load one or more classes associated with that extension. The basic structure of this cache example is as follows:

```
public class ExtCache implements IRegistryChangeListener {
   private static final String PID = "my.plugin";
   private static final String PT_ID =
      PID + "." + "extension.point";
   private final HashSet extensions = new HashSet();
   ...
}
```

The `extensions` field stores the set of installed extensions for a particular extension point.

The cache has a `startup` method that loads the initial set of extensions and then adds an extension registry listener in order to be notified of future changes:

```
public void startup() {
   IExtensionRegistry reg = Platform.getExtensionRegistry();
   IExtensionPoint pt = reg.getExtensionPoint(PT_ID);
   IExtension[] ext = pt.getExtensions();
   for (int i = 0; i < ext.length; i++)
      extensions.add(ext[i]);
   reg.addRegistryChangeListener(this);
}
```

The class implements the `IRegistryChangeListener` interface, which has a single method that is called whenever the registry changes:

```
public void registryChanged(IRegistryChangeEvent event) {
   IExtensionDelta[] deltas =
                        event.getExtensionDeltas(PID, PT_ID);
   for (int i = 0; i < deltas.length; i++) {
      if (deltas[i].getKind() == IExtensionDelta.ADDED)
         extensions.add(deltas[i].getExtension());
      else
         extensions.remove(deltas[i].getExtension());
   }
}
```

This class is now dynamic aware but is not yet dynamic enabled; that is, the class does not yet support itself being dynamically removed. The final step is to implement a `shutdown` method that clears all values from the cache and removes the listener from the extension registry:

```
public void shutdown() {
   extensions.clear();
   IExtensionRegistry reg = Platform.getExtensionRegistry();
   reg.removeRegistryChangeListener(this);
}
```

This `shutdown` method must be called from the `shutdown` method of the plug-in that defines the cache. For the complete source code of this example, see the `ExtCache` class in the FAQ Examples plug-in.

Note that not only extensions points acquire and maintain references to classes defined in other plug-ins. You need to be especially aware of static fields and caches that contain references to objects whose class is defined in other plug-ins.

If you hold onto classes defined in other plug-ins through different mechanisms, you also need to discard those references when those other plug-ins are removed.

FAQ 116 *What is a dynamic plug-in?*
FAQ 117 *How do I make my plug-in dynamic enabled?*

This page intentionally blank