CHAPTER 3

Eclipse Infrastructure

This chapter discusses the architecture behind the code generated in the previous chapter. Before diving deeper into every aspect of the program, it's time to step back and look at Eclipse as a whole.

The simple example plug-in that was started and described in Chapter 2—the **Favorites** plug-in—provides a concrete basis on which to discuss the Eclipse architecture.

3.1 Structural Overview

Eclipse isn't a single monolithic program, but rather a small kernel called a plug-in loader surrounded by hundreds (and potentially thousands) of plug-ins (see Figure 3–1) of which the **Favorites** example plug-in is one. Each plug-in may rely on services provided by another plug-in, and each may in turn provide services on which yet other plug-ins may rely.

This modular design lends itself to discrete chunks of functionality that can be more readily reused to build applications not envisioned by Eclipse's original developers.

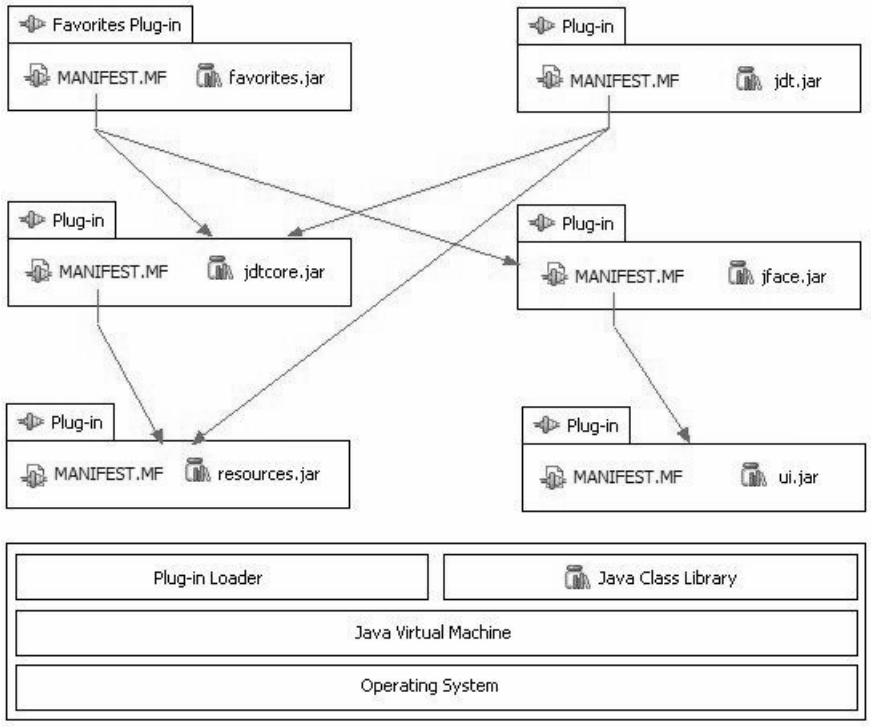

Figure 3–1 Eclipse plug-in structure.
An example of how plug-ins depend on one another.

3.1.1 Plug-in structure

The behavior of every plug-in is in code, yet the dependencies and services of a plug-in (see Section 2.3.1, The Plug-in manifests, on page 71) are declared in the `MANIFEST.MF` and `plugin.xml` files (see Figure 3–2). This structure facilitates lazy-loading of plug-in code on an as-needed basis, thus reducing both the startup time and the memory footprint of Eclipse.

On startup, the plug-in loader scans the `MANIFEST.MF` and `plugin.xml` files for each plug-in and then builds a structure containing this information. This structure takes up some memory, but it allows the loader to find a required plug-in much more quickly, and it takes up a lot less space than loading all the code from all the plug-ins all the time.

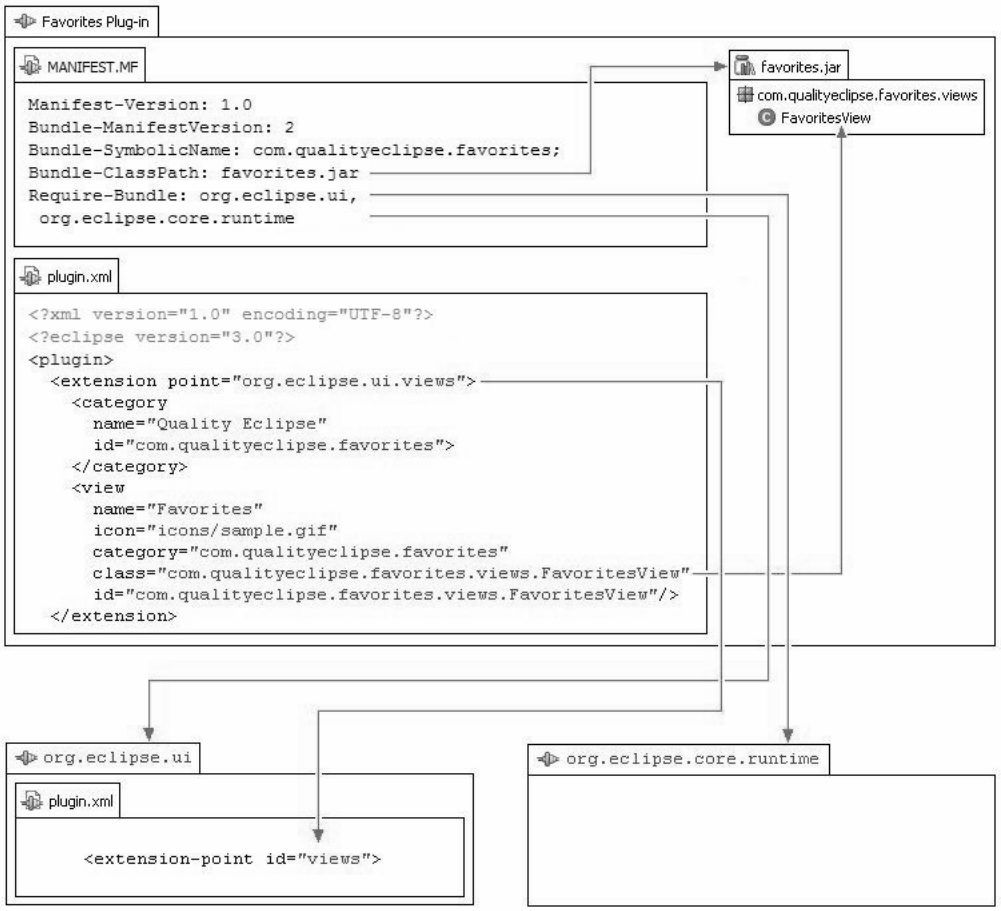

Figure 3–2 Declaring a new extension.
This is an example of how a new extension is declared in the plug-in manifest with lines highlighting how the plug-in manifest references various plug-in artifacts.

Plug-ins Are Loaded But Not Unloaded In Eclipse 3.1, plug-ins are loaded lazily during a session but not unloaded, causing the memory footprint to grow as the user requests more functionality. In future versions of Eclipse, this issue may be addressed by unloading plug-ins when they are no longer required (see *eclipse.org/equinox*; and for more specifics on deactivating plug-ins see *dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/equinoxhome/dynamicPlugins/deactivatingPlugins.html*).

3.1.2 Workspace

The Eclipse IDE displays and modifies files located in a ***workspace***. The workspace is a directory hierarchy containing both user files such as projects, source code, and so on, and plug-in state information such as preferences (see Section 3.4.4, Plug-in preferences, on page 116). The plug-in state information located in the workspace directory hierarchy is associated only with that workspace, yet the Eclipse IDE, its plug-ins, the plug-in static resources (see Section 3.4.3, Static plug-in resources, on page 115) and plug-in configuration files (see Section 3.4.5, Plug-in configuration files, on page 116) are shared by multiple workspaces.

3.2 Plug-in Directory or JAR file

The **Favorites** plug-in directory (or JAR file, see the first entry below) contains files similar to a typical plug-in, including `*.jar` files containing code, various images used by the plug-in, and the plug-in manifest.

favorites.jar—A file containing the actual Java classes comprising the plug-in. Typically, the JAR file is named for the last segment in the plug-in's identifier, but it could have any name, as long as that name is declared in the `META-INF/MANIFEST.MF` file. In this case, since the **Favorites** plug-in identifier is `com.qualityeclipse.favorites`, the JAR file is named `favorites.jar`.

icons—Image files are typically placed in an `icons` or `images` subdirectory and referenced in the `plugin.xml` and by the plug-in's various classes. Image files and other static resource files that are shipped as part of the plug-in can be accessed using methods in the plug-in class (see Section 3.4.3, Static plug-in resources, on page 115).

META-INF/MANIFEST.MF—A file describing the runtime aspects of the plug-in such as identifier, version, and plug-in dependencies (see Section 2.3.1, The Plug-in manifests, on page 71 and see Section 3.3.2, Plug-in runtime, on page 110).

plugin.xml—A file in XML format describing extensions and extension points (see Section 3.3.4, Extensions and extension points, on page 112).

The plug-in directory must have a specific name and be placed inside a specific directory so that Eclipse can find and load it. The directory name must be a concatenation of the plug-in identifier, an underscore, and the plug-in version in dot-separated form, as in:

```
com.qualityeclipse.favorites_1.0.0
```

The plug-in directory must be located in the `plugins` directory as a sibling to all the other Eclipse plug-ins, as is the case for the **Favorites** plug-in.

As of Eclipse 3.1, the plug-in can be delivered as a single JAR file containing the same files as a plug-in directory (see Section 2.4.1, Building manually, on page 81). If you wish to deliver the plug-in as a single JAR file rather than a directory of files, then it must be named in exactly the same way with a ".`jar`" suffix, as in

```
com.qualityeclipse.favorites_1.0.0.jar
```

Whenever we refer to a "plug-in directory," we are also referring to this alternate JAR file format.

3.2.1 Link files

Alternatively, plug-in directories comprising a product may be placed in a separate product-specific directory, and then a link file can be provided for Eclipse so that the program can find and load these plug-ins. Not only does this approach satisfy Ready for Rational Software (RFRS) requirements, but it also allows for multiple installations of Eclipse to be linked to the same set of plug-ins. You must make several modifications to the **Favorites** example so that it can use this alternate approach.

To begin, remove the existing **Favorites** plug-in in its current form from the **Development Workbench** using the steps outlined in Section 2.8.5, Uninstalling the Favorites plug-in, on page 98. Next, modify the Ant-based `build-favorites.xml` file so that the **Favorites** plug-in conforms to the new structure by inserting `QualityEclipse/Favorites/eclipse` in two places; then replace the following

```
<property name="plugin.jar" location=
   "${build.temp}/jars/plugins/${plugin.dir}.jar" />
```

with this (location must be on a single line)

```
<property name="plugin.jar" location=
   "${build.temp}/jars/QualityEclipse/Favorites/
      eclipse/plugins/${plugin.dir}.jar" />
```

Next, replace this:

```
<mkdir dir="${build.temp}/jars/plugins" />
```

with this (all on a single line):

```
<mkdir dir="${build.temp}/jars/QualityEclipse/Favorites/
   eclipse/plugins" />
```

When making these modifications, be sure that the location string is all on a single line; Ant does not handle paths that span multiple lines. When the modified `build-favorites.xml` is executed, the resulting zip file contains a new structure:

```
QualityEclipse/Favorites/eclipse/plugins/
   com.qualityeclipse.favorites_1.0.0.jar
```

The zip file can be unzipped to any location, but for this example, assume that the file is unzipped into the root directory of the C drive so that the plug-in directory is:

```
C:\QualityEclipse\Favorites\eclipse\plugins\
   com.qualityeclipse.favorites_1.0.0.jar
```

The locations for the Eclipse product directory and the Quality-Eclipse product directory are determined by the user and thus are not known at build time. Because of this, the link file that points to the Quality-Eclipse product directory must be manually created for now. Create the `links` subdirectory in the Eclipse product directory (e.g., `C:\eclipse\links`) and create a new file named `com.qualityeclipse.favorites.link` that contains this single line:

```
path=C:/QualityEclipse/Favorites
```

To do this in Windows, you can use Notepad to create and save the file as a `txt` file, which you can then rename appropriately. Note that the path in the `*.link` file must use forward slashes rather than backslashes. The new `*.link` file will be used by Eclipse once Eclipse has been restarted.

No Relative Paths in Link Files Eclipse 3.1 does not allow link files to contain relative paths. This restriction may be changed in future versions (see *bugs.eclipse.org/bugs/show_bug.cgi?id=35037* for Bugzilla entry 35037).

3.2.2 Hybrid approach

Some products use a hybrid approach, delivering the product in multiple forms. When installing, the installer places product plug-ins directly in the Eclipse plug-ins directory, whereas when installing into Rational Application

Developer or any of the other Rational IDE family of products, the product plug-ins are placed in a separate product directory and a link file is created. In addition, these products are available in various zip file formats, each targeted at a specific type and version of an Eclipse or WebSphere product. This hybrid approach facilitates a simpler and smaller zip-based installation for Eclipse where Ready for Rational Software (RFRS) certification is not required, and a cleaner and easier installer-based installation for the Rational IDE family of products.

After you install the QualityEclipse product and create the link file as just described, the QualityEclipse product is ready for use. Verify that you have correctly installed the QualityEclipse product in its new location by restarting Eclipse and opening the **Favorites** view. After you have installed and verified the product, be sure to uninstall it by deleting the link file so that the JUnit tests described in Section 2.8, Writing Plug-in Tests, on page 92 will still run successfully.

3.3 Plug-in Manifest

As stated earlier, there are two files—`MANIFEST.MF` and `plugin.xml`—per plug-in defining various high-level aspects so that the plug-in does not have to load until you need its functionality. The format and content for these files can be found in the Eclipse help facility accessed by **Help > Help Contents**; look under **Platform Plug-in Developer Guide > Reference > Other Reference Information > OSGi Bundle Manifest** and **Plug-in Manifest**.

What is OSGi? Eclipse originally used a home-grown runtime model/mechanism that was designed and implemented specifically for Eclipse. This was good because it was highly optimized and tailored to Eclipse, but less than optimal because there are many complicated issues, and having a unique runtime mechanism prevented reusing the work done in other areas (e.g., OSGi, Avalon, JMX, etc.). As of Eclipse 3.0, a new runtime layer was introduced based upon technology from the OSGi Alliance (www.osgi.org) that has a strong specification, a good component model, supports dynamic behaviour, and is reasonably similar to the original Eclipse runtime. With each new release of Eclipse, the Eclipse runtime API and implementation (e.g. "plug-ins") continues to align itself more and more closely with the OSGi runtime model (e.g. "bundles").

3.3.1 Plug-in declaration

Within each bundle manifest, there are entries for name, identifier, version, plug-in class, and provider.

```
Bundle-Name: Favorites Plug-in
Bundle-SymbolicName: com.qualityeclipse.favorites; singleton:=true
Bundle-Version: 1.0.0
Bundle-Activator: com.qualityeclipse.favorites.FavoritesPlugin
Bundle-Vendor: Quality Eclipse
```

Strings in the plug-in manifest, such as the plug-in name, can be moved into a separate `plugin.properties` file. This process facilitates internationalization as discussed in Chapter 16, Internationalization.

3.3.1.1 Plug-in identifier

The plug-in identifier (`Bundle-SymbolicName`) is designed to uniquely identify the plug-in and is typically constructed using Java package naming conventions (e.g., `com.<companyName>.<productName>`, or in our case, `com.qualityeclipse.favorites`). If several plug-ins are all part of the same product, then each plug-in name can have four or even five parts to it as in `com.qualityeclipse.favorites.core` and `com.qualityeclipse.favorites.ui`.

3.3.1.2 Plug-in version

Every plug-in specifies its version (`Bundle-Version`) using three numbers separated by periods. The first number indicates the major version number, the second indicates the minor version number, and the third indicates the service level, as in `1.0.0`. You can specify an optional qualifier that can include alphanumeric characters as in `1.0.0.beta_1` or `1.0.0.2006-03-20` (no whitespace). Eclipse does not use or interpret this optional qualifier in any way, so the product builder can use it to encode the build type, build date, or other useful information.

Tip: For an outline of the current use of version numbers and a proposed guideline for using plug-in version numbering to better indicate levels of compatibility, see *eclipse.org/equinox/documents/plugin-versioning.html*.

3.3.1.3 Plug-in name and provider

Both the name and the provider are human-readable text, so they can be anything and are not required to be unique. To see the names, versions, and providers of the currently installed plug-ins, select **Help > About Eclipse SDK** to

open the **About** dialog (see Figure 3–3), and then click the **Plug-in Details** button to open the **Plug-ins** dialog (see Figure 3–4).

Figure 3–3 The About Eclipse SDK dialog, showing information about the Eclipse platform.

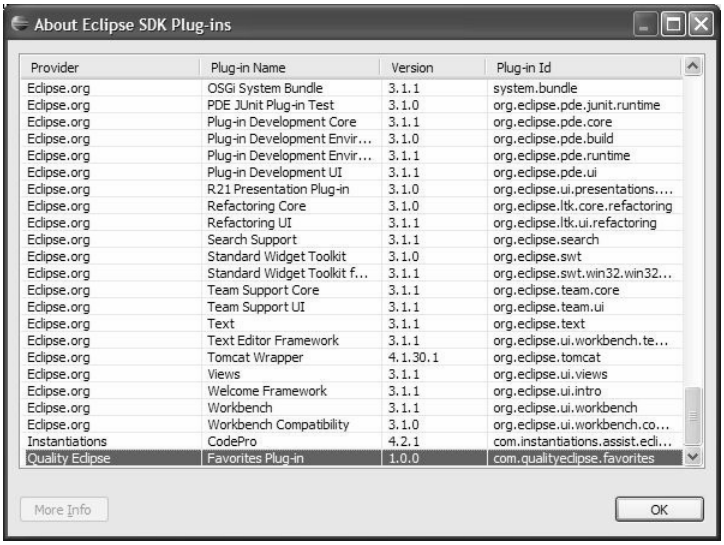

Figure 3–4 The About Eclipse SDK Plug-ins dialog, showing all the installed plug-ins with the Favorites plug-in highlighted at the bottom.

3.3.1.4 Plug-in class declaration

Optionally, every plug-in can specify a plug-in class (`Bundle-Activator`) as the **Favorites** plug-in does (see Section 3.4, Plug-in Class, on page 114).

3.3.2 Plug-in runtime

The `Bundle-ClassPath` declaration in the `MANIFEST.MF` file is a comma separated list describing which libraries (`*.jar` files) contain the plug-in code. The `Export-Package` declaration is a comma-separated list indicating which packages within those libraries are accessible to other plug-ins (see Section 20.2.4, How Eclipse is different, on page 713 and Section 20.2.5, Related plug-ins, on page 713).

```
Bundle-ClassPath: favorites.jar
Export-Package: com.qualityeclipse.favorites.views
```

Tip: When delivering your plug-in as a single JAR, the `Bundle-ClassPath` declaration should be empty so that Eclipse looks for classes in the plug-in JAR and not in a JAR inside your plug-in.

3.3.3 Plug-in dependencies

The plug-in loader instantiates a separate class loader for each loaded plug-in, and uses the `Require-Bundle` declaration of the manifest to determine which other plug-ins—thus which classes—will be visible to that plug-in during execution (see Section 20.9, Plug-in ClassLoaders, on page 742 for information about loading classes not specified in the `Require-Bundle` declaration).

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

If a plug-in has been successfully compiled and built but, during execution, throws a `NoClassDefFoundError`, it may indicate that the plug-in project's Java classpath is out of sync with the `Require-Bundle` declaration in the `MANIFEST.MF` file. As discussed in Section 2.3.1, The Plug-in manifests, on page 71, it is important to keep the classpath and the `Require-Bundle` declaration in sync.

When the plug-in loader is about to load a plug-in, it scans the `Require-Bundle` declaration of a dependent plug-in and locates all the required plug-ins. If a required plug-in is not available, then the plug-in loader throws an exception, generating an entry in the log file (see Section 3.6, Logging, on page 122) and does not load the dependent plug-in. When a plug-in gathers the list of plug-ins that extend an extension point it defines, it will not see any disabled plug-ins. In this circumstance, no exception or log entry will be generated for the disabled plug-ins.

If a plug-in can successfully execute without a required plug-in, then that required plug-in can be marked as optional in the plug-in manifest. To do so, open the plug-in manifest editor and then switch to the **Dependencies** tab (see Figure 2–10 on page 73). Select the required plug-in, click the **Properties** button and then check the **Optional** checkbox in the Properties dialog (see Figure 3–5).

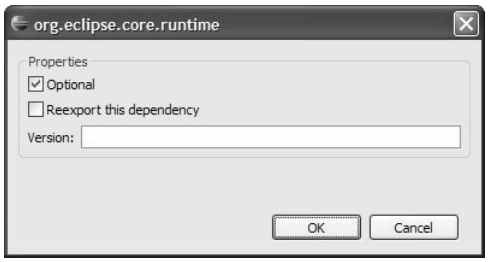

Figure 3–5 The required plug-in properties dialog.

Making this change in the plug-in manifest editor appends `;resolution:=optional` to the required plug-in in the `Require-Bundle` declaration so that it now looks something like this:

```
Require-Bundle: org.eclipse.ui,
 org.eclipse.core.runtime;resolution:=optional
```

If your plug-in requires not just any version of another plug-in, you can specify an exact version or a range of versions using the required plug-in properties dialog (see Figure 3–5). The following are some examples:

- [**3.0.0.test,3.0.0.test**]—requires a specific version
- [**3.0.0,3.0.1**)—requires version 3.0.0.x
- [**3.0.0,3.1.0**)—requires version 3.0.x
- [**3.0.0,3.2.0**)—requires version 3.0.x or 3.1.x
- [**3.0.0,4.0.0**)—requires version 3.x
- **3.0.0**—requires version 3.0.0 or greater

The general syntax for a range is

```
[ floor , ceiling )
```

where `floor` is the minimum version and `ceiling` is the maximum version. The first character can be `[` or `(` and the last character may be `]` or `)` where these characters indicate the following:

- `[` = floor is included in the range
- `(` = floor is **not** included in the range
- `]` = ceiling is included in the range
- `)` = ceiling is **not** included in the range

You can specify a floor or minimum version with no extra characters indicating that your plug-in needs any version greater than or equal to the specified version. Entering one of the preceding in the required plug-in properties dialog (see Figure 3–5) modifies the `Require-Bundle` declaration so that it now looks something like this:

```
Require-Bundle: org.eclipse.ui,
 org.eclipse.core.runtime;bundle-version="[3.0.0,3.1.0)"
```

Finally, check the **Reexport this dependency** checkbox (see Figure 3–5) to specify that the dependent plug-in classes are made visible (are (re)exported) to users of this plug-in. By default, dependent classes are not exported (i.e., they are not made visible).

`Import-Package` is similar to `Require-Bundle` except that `Import-Package` specifies names of packages that are required for execution rather than names of bundles. Using `Import-Package` can be thought of as specifying the service required whereas using `Require-Bundle` is like specifying the service provider. `Import-Package` makes it easier to swap out one bundle for another that provides the same service, but harder to know who is providing that service.

3.3.4 Extensions and extension points

A plug-in declares extension points so that other plug-ins can extend the functionality of the original plug-in in a controlled manner (see Section 17.1, The Extension Point Mechanism, on page 595). This mechanism provides a layer of separation so that the original plug-in does not need to know about the existence of the extending plug-ins at the time you build the original plug-in. Plug-ins declare extension points as part of their plug-in manifest, as in the views extension point declared in the `org.eclipse.ui` plug-in:

```
<extension-point
    id="views"
    name="%ExtPoint.views"
    schema="schema/views.exsd"/>
```

You can find documentation for this extension point in the Eclipse help (select **Help > Help Contents**, then in the **Help** dialog, select **Platform Plug-in Developer Guide > Reference > Extension Points Reference > Workbench > org.eclipse.ui.views**). It indicates that any plug-in using this extension point must provide the name of a class that implements the interface `org.eclipse.ui.IViewPart` (see Section 20.5, Types Specified in an Extension Point, on page 723).

Other plug-ins declare extensions to the original plug-in's functionality similar to the **Favorites** plug-in's view extensions. In this case, the **Favorites** plug-in declares a new category of views with the name **Quality Eclipse** and the class, `com.qualityeclipse.favorites.views.FavoritesView`, as a new type of view as follows:

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

Each type of extension point may require different attributes to define the extension. Typically, ID attributes take a form similar to the plug-in identifier. The category ID provides a way for the **Favorites** view to uniquely identify the category that contains it. The `name` attribute of both the category and view is human-readable text, while the `icon` attribute specifies a relative path from the plug-in directory to the image file associated with the view.

This approach allows Eclipse to load information about the extensions declared in various plug-ins without loading the plug-ins themselves, thus reducing the amount of time and memory required for an operation. For example, selecting the **Windows > Show View > Other…** menu opens a dialog showing all the views provided by all the plug-ins known to Eclipse (see Section 2.5, Installing and Running the Product, on page 86). Because each type of view is declared in its plug-in's manifest, the Eclipse runtime can present a list of views to the user without actually loading each plug-in that contains the view.

3.4 Plug-in Class

By default, the plug-in class or `Bundle-Activator` provides methods for accessing static resources associated with the plug-in, and for accessing and initializing plug-in-specific preferences and other state information. A plug-in class is not required, but if specified in the plug-in manifest, the plug-in class is the first class notified after the plug-in loads and the last class notified when the plug-in is about to shut down (see Section 3.5.2, Plug-ins and Bundles, on page 120 and the source code listing in Section 2.3.2, The Plug-in class, on page 77).

Tip: Historically, plug-ins have exposed their `Plugin` subclass as an entry point. To better control access to your plug-in's initialization, consider either a `Bundle-Activator` other than your `Plugin` subclass or moving public access methods to a new class and hiding your `Plugin` subclass.

3.4.1 Startup and shutdown

The plug-in loader notifies the plug-in class when the plug-in is loaded via the `start()` method and when the plug-in shuts down via the `stop()` method. These methods allow the plug-in to save and restore any state information between Eclipse sessions.

Be Careful When Overriding start() and stop() When overriding these methods, be careful; always call the superclass implementation, and only take the minimum action necessary so that you do not impact the speed or memory requirements during Eclipse startup or shutdown.

3.4.2 Early plug-in startup

Eclipse loads plug-ins lazily, so it may not call the `start()` method when it launches. Eclipse can provide resource change information indicating the changes that occurred while the plug-in was inactive (see Section 9.5, Delayed Changed Events, on page 387). If this is not enough and the plug-in *must* load

and start when Eclipse launches, the plug-in can use the `org.eclipse.ui.startup` extension point by inserting the following into its plug-in manifest:

```
<extension point="org.eclipse.ui.startup">
   <startup class="myPackage.myClass"/>
</extension>
```

Doing this requires that the `myPackage.myClass` class implement the `org.eclipse.ui.IStartup` interface so that the workbench can call the `earlyStartup()` method immediately after the UI completes its startup. For more on early startup and the issues involved, see Section 20.10, Early Startup, on page 747.

Like most plug-ins, the **Favorites** plug-in does not need to load and start when Eclipse launches, so it does not use this extension point. If there is a need for early startup, then place only what is necessary for it into a separate plug-in and use the early startup extension point there so that the additional overhead of early startup has only a small impact on startup time and memory footprint.

3.4.3 Static plug-in resources

Plug-ins can include images and other file-based resources that are installed into the plug-in directory along with the plug-in manifest and library file. These files are static in nature and shared between multiple workbench incarnations. Declarations, such as actions, views, and editors, in the plug-in manifest can reference resources such as icons stored in the plug-in installation directory. Additionally, the plug-in class provides methods for locating and loading these resources:

`find (IPath path)`—Returns a uniform resource locator (URL) for the given path or null if the URL could not be computed or created.

`openStream (IPath file)`—Returns an input stream for the specified file. The file path must be specified relative to the plug-in's installation location (the plug-in directory).

3.4.4 Plug-in preferences

Plug-in preferences and other workspace-specific state information are stored in the workspace metadata directory hierarchy. For example, if Eclipse is installed at `C:\eclipse` and the default workspace location is being used, then the **Favorites** preferences would be stored in:

```
C:/eclipse/workspace/.metadata/.plugins
        /com.qualityeclipse.favorites/pref_store.ini
```

The plug-in class provides methods for accessing plug-in preferences and other state-related files as follows:

`getPluginPreferences()`—Returns the preference store for this plug-in (see Section 12.3, Preference APIs, on page 467).

`getStateLocation()`—Returns the location in the local filesystem of the plug-in state area for this plug-in (see Section 7.5.2, Saving global view information, on page 311). If the plug-in state area did not exist prior to this call, it is created.

`savePluginPreferences()`—Saves the preference settings for this plug-in; it does nothing if the preference store does not need saving.

You can supply default preferences to a plug-in in several ways. In order to programmatically define default preference values, override the method `initializeDefaultPluginPreferences()`. Alternatively, you can specify default preferences in a `preferences.ini` file located in the plug-in directory (see Section 12.3.4, Specifying default values in a file, on page 472). Using this second approach also lets you easily internationalize the plug-in using a `preferences.properties` file (see Section 16.1, Externalizing the Plug-in Manifest, on page 576).

3.4.5 Plug-in configuration files

If you need to store plug-in information that needs to be shared among all workspaces associated with a particular Eclipse installation, then use the method `Platform.getConfigurationLocation()` and create a plug-in specific subdirectory. If Eclipse is installed in a read-only location, then `Platform.getConfigurationLocation()` will return `null`. You could add the following field and method to the `FavoritesPlugin` class to return a configuration directory for this plug-in. If Eclipse is installed in a read-only location, then this method would gracefully degrade by returning the workspace-specific state location rather than the configuration directory so that plug-in state information could still be stored and retrieved.

```
public static final String ID = "com.qualityeclipse.favorites";

public File getConfigDir() {
   Location location = Platform.getConfigurationLocation();
   if (location != null) {
      URL configURL = location.getURL();
      if (configURL != null
            && configURL.getProtocol().startsWith("file")) {
         return new File(configURL.getFile(), ID);
      }
   }
   // If the configuration directory is read-only,
   // then return an alternate location
   // rather than null or throwing an Exception.
   return getStateLocation().toFile();
}
```

Preferences can also be stored in the configuration directory by adding the following field and method to the `FavoritesPlugin` class.

Read-Only Installation Be warned that if Eclipse is installed in a read-only location, then this method will return `null`. In addition, neither the following code nor the Preferences object returned by the method below is thread safe.

```
private IEclipsePreferences configPrefs;

public Preferences getConfigPrefs() {
   if (configPrefs == null)
      configPrefs = new ConfigurationScope().getNode(ID);
   return configPrefs;
}
```

If you add the preceding method to your plug-in class, then you should also modify the `stop()` method to flush the configuration preferences to disk when Eclipse shuts down.

```
public void stop(BundleContext context) throws Exception {
   if (configPrefs != null) {
      configPrefs.flush();
      configPrefs = null;
   }
   plugin = null;
   super.stop(context);
}
```

When you launch a Runtime Workbench (see Section 2.6, Debugging the Product, on page 88), you can specify the configuration directory using the **Configration** page of the **Run** dialog (see Figure 3–6).

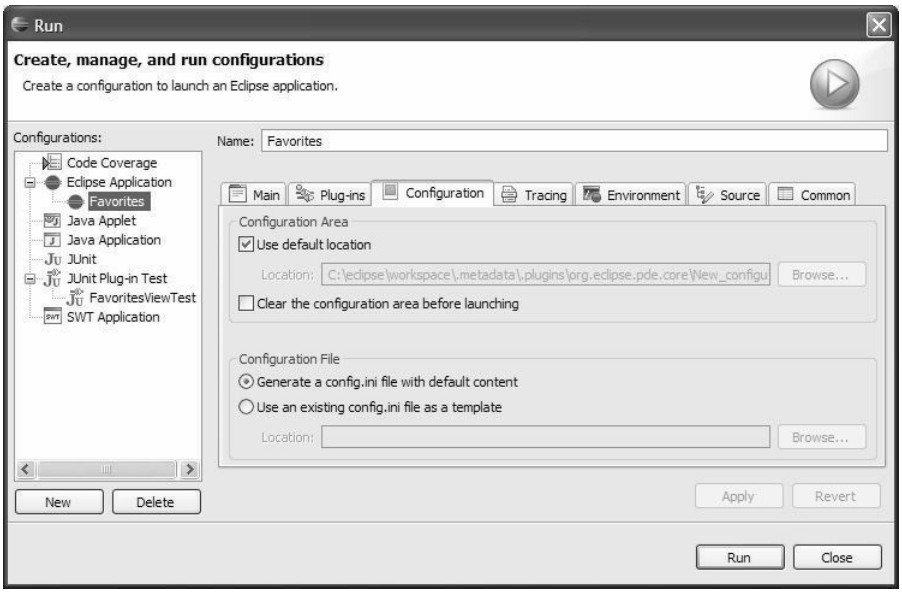

Figure 3–6 The Launch Configuration page for specifying the configuration directory.

3.4.6 Plugin and AbstractUIPlugin

All plug-in classes must implement the `BundleActivator` interface. Typically, UI-based plug-ins (plug-ins requiring the `org.eclipse.ui` plug-in) have a plug-in class that subclasses `AbstractUIPlugin`, while non-UI plug-ins subclass `Plugin`. Both classes provide basic plug-in services for the plug-in programmer, but there are important differences.

`AbstractUIPlugin` automatically saves plug-in preferences when the plug-in shuts down. When subclassing the `Plugin` class directly, modify the `stop()` method to always call `savePluginPreferences()` and `saveDialogSettings()` so that preferences will persist across sessions.

Older Preference Storage Methods `AbstractUIPlugin` provides alternate preferences storage methods and classes that you should not use. These methods, such as `getPreferenceStore()` and the associated `IPreferenceStore` interface, predate Eclipse 3.1 and the `Plugin` class preference methods, such as `getPluginPreferences()` and the associated class `Preferences`. They exist only for backward compatibility. These older preference storage methods do not provide any advantages when used in `AbstractUIPlugin`, so use the `Preferences` interface and associated methods unless the Eclipse API specifically requires the older interface (see Chapter 12, Preference Pages, for more on preferences).

Other methods provided by `AbstractUIPlugin` include:

`createImageRegistry()`—Returns a new image registry for this plug-in. You can use the registry to manage images that are used frequently by the plug-in. The default implementation of this method creates an empty registry. Subclasses can override this method if necessary.

`getDialogSettings()`—Returns the dialog settings for this UI plug-in (see Section 11.2.7, Dialog settings, on page 441). The dialog settings hold persistent state data for the various wizards and dialogs of this plug-in in the context of a workbench.

`getImageRegistry()`—Returns the image registry for this UI plug-in (see Section 4.4.3, Images, on page 181 and Section 7.7, Image Caching, on page 315).

`initializeImageRegistry(ImageRegistry reg)`—Initializes an image registry with images that are used frequently by the plug-in.

`loadDialogSettings()`—Loads the dialog settings for this plug-in by looking first for a `dialog_settings.xml` file in the plug-in's metadata directory, then for a file with the same name in the plug-in directory; failing both of these, it creates an empty settings object. This method can be overridden, although this is typically unnecessary.

3.5 Plug-in Model

When Eclipse first launches, it scans each of the plug-in directories and builds an internal model representing every plug-in it finds. This occurs by scanning each plug-in manifest without loading the plug-ins. The methods in the next two subsections are useful if you want to display information about plug-ins or perform operations based on specific plug-in characteristics without taking the time and memory usage hit associated with loading plug-ins.

3.5.1 Platform

The `org.eclipse.core.runtime.Platform` class provides information about the currently executing Eclipse environment. Using this class, you can obtain information about installed plug-ins (also known as Bundles), extensions, extension points, command line arguments, job manager (see Section 20.8, Background Tasks—Jobs API, on page 739), installation location, and more.

The following are some methods of note.

`asLocalURL(URL)`—Translates a plug-in-relative URL to a locally accessible URL.

`find(Bundle bundle, IPath path)`—Returns a URL to the resource in the specified bundle.

`getBundle(String)`—Returns the bundle with the specified unique identifier.

`getBundleGroupProviders()`—Returns an array of bundle providers that contain bundle groups that contain currently installed bundles.

`getExtensionRegistry()`—Returns extension and extension point information.

`getJobManager()`—Returns the platform job manager (see Section 20.8, Background Tasks—Jobs API, on page 739).

`getLog(Bundle)`—Returns the log for the specified bundle.

`getProduct()`—Returns the Eclipse product information.

`inDebugMode()`—Returns `true` if Eclipse is in debug mode, as it is when the user specifies the `-debug` command line argument.

`resolve(URL)`—Resolves a plug-in-relative URL to a URL native to the Java class library (e.g., file, http, etc.).

`run(ISafeRunnable)`—Runs the given runnable in a protected mode. Exceptions thrown in the runnable are logged and passed to the runnable's exception handler.

3.5.2 Plug-ins and Bundles

Information about the currently installed plug-ins, also known as Bundles, can be obtained using `Platform.getBundleGroupProviders()` or `Platform.getBundle(String)`. Accessing a plug-in class, also known as a bundle activator, requires the containing plug-in to be loaded whereas interacting with the `Bundle` interface does not carry such a penalty. If you already have a plug-in class, such as the **Favorites** plug-in, then you can obtain the `Bundle` interface for that plug-in by using something like this:

```
FavoritesPlugin.getDefault().getBundle()
```

After you obtain the `Bundle` object, several methods are of interest.

`getBundleId()`—Returns the bundle's unique identifier (a `long`), assigned by Eclipse when the bundle was installed.

`getEntry(String)`—Returns a URL for the specified `'/'`-separated bundle relative path name where `getEntry("/")` returns the bundle root. This provides access to resoures supplied with the plug-in that are typically read-only. Relative plug-in information should be written to the location provided by `Plugin.getStateLocation()`.

`getHeaders()`—Returns a dictionary of headers and values defined in the bundle's `MANIFEST.MF` file (see Section 3.3.1, Plug-in declaration, on page 108).

`getState()`—Returns the current state of a plug-in, such as `Bundle.UNINSTALLED`, `Bundle.INSTALLED`, `Bundle.RESOLVED`, `Bundle.STARTING`, `Bundle.STOPPING`, `Bundle.ACTIVE`.

`getSymbolicName()`—Returns the unique plug-in identifier (a `java.lang.String`), which is the same as the `Bundle-SymbolicName` declaration in the `MANIFEST.MF`.

The plug-in version number can be obtained using the `getHeaders()` method.

```
new PluginVersionIdentifier(
    bundle.getHeaders().get("Bundle-Version"))
```

3.5.3 Plug-in extension registry

You can access the plug-in extension registry using the `Plaform.getExtensionRegistry()` method. It contains plug-in descriptors, each representing a plug-in. The registry provides the following methods for extracting information about the various plug-ins without loading them (see Section 17.1, The Extension Point Mechanism, on page 595 for information on creating extension points).

`getConfigurationElementsFor(String extensionPointId)`—Returns all configuration elements from all extensions configured into the identified extension point.

`getExtensionPoint(String extensionPointId)`—Returns the extension point with the given extension point identifier in this plug-in registry.

Previously, extensions and extension-points did not change during execution, but that is slowly changing as the Eclipse plug-in model continues to align itself with OSGi. If you are interested in changes during execution, use `addRegistryChangeListener(IRegistryChangeListener)`.

Tip: For more on the plug-in registry, activation, and lifecycle, check out the Equinox project at *www.eclipse.org/equinox*.

3.6 Logging

The RFRS requirements indicate that exceptions and other service-related information should be appended to a log file. To facilitate this, the plug-in class provides a method for accessing the plug-in logging mechanism via the `getLog()` method. For convenience, the `FavoritesLog` wraps the `ILog` interface returned by the `getLog()` method with several utility methods:

```
package com.qualityeclipse.favorites;

import org.eclipse.core.runtime.IStatus;
import org.eclipse.core.runtime.Status;

public class FavoritesLog {
```

The first group of methods that follow are for convenience, appending information, error messages, and exceptions to the log for the Favorites plug-in.

```
public static void logInfo(String message) {
   log(IStatus.INFO, IStatus.OK, message, null);
}

public static void logError(Throwable exception) {
   logError("Unexpected Exception", exception);
}

public static void logError(String message, Throwable exception) {
   log(IStatus.ERROR, IStatus.OK, message, exception);
}
```

Each of the preceding methods ultimately calls the following methods which create a status object (see Section 3.6.1, Status objects, on page 123) and then append that status object to the log.

```
public static void log(int severity, int code, String message,
      Throwable exception) {
   log(createStatus(severity, code, message, exception));
}

public static IStatus createStatus(int severity, int code,
      String message, Throwable exception) {
   return new Status(severity, FavoritesPlugin.ID, code,
         message, exception);
}

public static void log(IStatus status) {
   FavoritesPlugin.getDefault().getLog().log(status);
}
```

The `log()` and `createStatus()` methods take the following parameters.

`severity`—the severity; one of these:
`IStatus.OK`, `IStatus.WARNING`, `IStatus.ERROR`,
`IStatus.INFO`, or `IStatus.CANCEL`

`code`—the plug-in-specific status code or `IStatus.OK`

`message`—a human-readable message, localized to the current locale

`exception`—a low-level exception, or `null` if not applicable

3.6.1 Status objects

The `IStatus` type hierarchy in the `org.eclipse.core.runtime` package provides a mechanism for wrapping, forwarding, and logging the result of an operation, including an exception if there is one. A single error is represented using an instance of `Status` (see method `createStatus` in the previous source code), while a `MultiStatus` object that contains zero or more child status objects represents multiple errors.

When creating a framework plug-in that will be used by many other plug-ins, it is helpful to create status subtypes similar to `IResourceStatus` and `ResourceStatus`; however, for the **Favorites** plug-in, the existing status types that follow will do:

`IStatus`—A status object that represents the outcome of an operation. All `CoreExceptions` carry a status object to indicate what went wrong. Status objects are also returned by methods needing to provide details of failures (e.g., validation methods).

`IJavaModelStatus`—Represents the outcome of a Java model operation. Status objects are used inside `JavaModelException` objects to indicate what went wrong.

`IResourceStatus`—Represents a status related to resources in the **Resources** plug-in and defines the relevant status code constants. Status objects created by the **Resources** plug-in bear its unique identifier, `ResourcesPlugin.PI_RESOURCES`, and one of these status codes.

`MultiStatus`—A concrete multistatus implementation, suitable either for instantiating or subclassing.

`OperationStatus`—Describes the status of a request to execute, undo, or redo an operation (see Section 6.6.2, Commands, on page 252).

`Status`—A concrete status implementation, suitable either for instantiating or subclassing.

`TeamStatus`—Returned from some Team operations or is the payload of some exceptions of type `TeamException`.

3.6.2 The Error Log view

The PDE provides an **Error Log** view for inspecting the Eclipse log file. To open the **Error Log** view, select **Window > Show View > Other…**, and in the **Show View** dialog, expand the **PDE Runtime** category to find the **Error Log** view (see Figure 3–7). Double-clicking on an entry opens a dialog showing details for the error log entry. If Eclipse is installed in `C:\Eclipse` and the default workspace location is being used, you can find the Eclipse log file at `C:\Eclipse\workspace\.metadata\.log`.

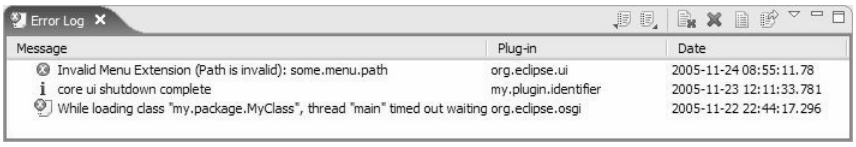

Figure 3–7 The Error Log view is provided by the Eclipse platform and displays information and exceptions generated while Eclipse is running.

3.7 Eclipse Plug-ins

Commercial plug-ins are built on one or more base plug-ins that are shipped as part of Eclipse. They are broken down into several groups, further separated into UI and Core, as follows. UI plug-ins contain aspects of a user interface or rely on other plug-ins that do, while you can use Core plug-ins in a headless environment (an environment without a user interface).

Core—A general low-level group of non-UI plug-ins comprising basic services such as extension processing (see Chapter 9, Resource Change Tracking, on page 375), resource tracking (see Chapter 17, Creating New Extension Points, on page 595), and so on.

SWT—The Standard Widget Toolkit, a general library of UI widgets tightly integrated with the underlying operating system (OS), but with an OS-independent API (see Chapter 4, The Standard Widget Toolkit, on page 127).

JFace—A general library of additional UI functionality built on top of SWT (see Chapter 5, JFace Viewers, on page 185).

Workbench core—Plug-ins providing non-UI behavior specific to the Eclipse IDE itself, such as projects, project natures, and builders (see Chapter 14, Builders, Markers, and Natures, on page 497).

Workbench UI—Plug-ins providing UI behavior specific to the Eclipse IDE itself, such as editors, views, perspectives, actions, and preferences (see Chapters 6, 7, 8, 10, and 12).

Team—A group of plug-ins providing services for integrating different types of source code control management systems (e.g., CVS) into the Eclipse IDE.

Help—Plug-ins that provide documentation for the Eclipse IDE as part of the Eclipse IDE (see Chapter 15, Implementing Help, on page 539).

JDT core—Non-UI-based Java Development Tooling (JDT) plug-ins for the Eclipse IDE.

JDT UI—JDT UI plug-ins for the Eclipse IDE.

3.8 Summary

This chapter tried to give you a more in-depth understanding of Eclipse and its structure in relation to creating plug-ins. The next two chapters explore the user-interface elements that should be used to create your own plug-ins.

References

Chapter source (*www.qualityeclipse.com/projects/source-ch-03.zip*).

"Eclipse Platform Technical Overview," Object Technology International, Inc., February 2003, (*www.eclipse.org/whitepapers/eclipse-overview.pdf*).

Melhem, Wassim, et al., "PDE Does Plug-ins," IBM, September 8, 2003 (*www.eclipse.org/articles/Article-PDE-does-plugins/PDE-intro.html*).

Xenos, Stefan, "Inside the Workbench: A Guide to the Workbench Internals," IBM, October 20, 2005 (*www.eclipse.org/articles/Article-UI-Workbench/workbench.html*).

Bolour, Azad, "Notes on the Eclipse Plug-in Architecture," Bolour Computing, July 3, 2003 (*www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html*).

Rufer, Russ, "Sample Code for Testing a Plug-in into Existence," Yahoo Groups Message 1571, Silicon Valley Patterns Group (*groups.yahoo.com/group/siliconvalleypatterns/message/1571*).

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*. Addition-Wesley, Boston, 1995.

Buschmann, Frank, et al., *Pattern-Oriented Software Architecture*. John Wiley & Sons, Hoboken, NJ, 1996.

Estberg, Don, "How the Minimum Set of Platform Plug-ins Are Related," Eclipse Wiki (*eclipsewiki.editme.com/MinimumSetOfPlatformPlugins*).

Watson, Thomas, "Deprecation of Version-Match Attribute," equinox-dev email, April 30, 2004.