Understanding how Eclipse plug-ins work with OSGi

Eclipse and OSGi from A to Z, or in this case, plugin.xml to manifest.mf

Scott Delap June 06, 2006

The core of the Eclipse integrated development environment (IDE) and Eclipse Rich Client Platform (RCP) applications is driven by an implementation of the Open Services Gateway Initiative (OSGi) specification. This article illustrates Eclipse's relationship with OSGi by describing what a plug-in is in terms of the Eclipse platform and traces the evolution of plug-ins from Eclipse V2.1 through today's OSGi-based implementation. It also explains the OSGi manifest.mf file options, along with the additions provided through Eclipse.

Most Java™ programming language developers are introduced to Eclipse through its function as an IDE. The Eclipse IDE actually consists of a collection of interacting components called *plug-ins*. These plug-ins, which make up the base of the IDE, can also be used to create other desktop applications. The minimal set of plug-ins needed to create an Eclipse-based application is referred to as the *Eclipse Rich Client Platform* (RCP). Plug-ins don't just start themselves, however. They require an environment in which to start and operate. Eclipse provides this environment with an implementation of the OSGi R4 specification.

Because Eclipse is at its core driven by OSGi, it's important to understand how the concept of an Eclipse plug-in relates to the OSGi framework. In this article, I will explain this relationship in detail by describing what a plug-in is in terms of the Eclipse platform. Then I'll describe the evolution of plug-ins in the Eclipse V2.1 platform through today's OSGi-based implementation. Finally, the OSGi-provided manifest.mf options that apply to Eclipse plug-ins will be covered in detail.

What is a plug-in?

The Eclipse online help defines a plug-in as follows:

> "Plug-ins are structured bundles of code and/or data that contribute function to the system. Function can be contributed in the form of code libraries (Java classes with public [application program interfaces] APIs), platform extensions, or even documentation. Plug-ins can define extension points, well-defined places where other plug-ins can add functionality."

The key point to focus on is that plug-ins contribute function in a structured manner. They may provide a service such as logging or piece functionality available in the user interface (UI), such as an editor. Regardless of their function, all plug-ins are defined in the same structured way.

The evolution to OSGi

As mentioned, Eclipse uses OSGi as the basis for its plug-in system. This wasn't always the case, however. Early versions of Eclipse were also designed as a collection of plug-ins, and Eclipse included its own propriety plug-in system to manage the interaction. However, as the requirements of the Eclipse IDE grew, it became apparent that a more robust solution was required. Basic requirements of this new system included the ability to dynamically handle the addition of new plug-ins and stopping existing plug-ins. After considerable research, the Eclipse creators decided to replace the proprietary plug-in framework by implementing the OSGi framework specification.

OSGi is a specification of a service platform. Eclipse provides one of many available implementations of this specification and serves as the reference implementation of the latest OSGi R4 specification. OSGi is a Java-based framework targeted for use by systems that require long running times, dynamic updates, and minimal disruptions to the running environment. Initially, OSGi was targeted at home automation and residential gateway devices. More recently, it has found use in everything from mobile phones to cars.

At its core, OSGi is a component and service model. The OSGi specification defines a unit of modularization called a *bundle*. (Unless explicitly noted during the rest of this article, the Eclipse term *plug-in* and the OSGi term *bundle* are used interchangeably because all Eclipse plug-ins are now OSGi bundles.) OSGi also provides a Java Virtual Machine (JVM)-level service registry that bundles can use to publish, discover, and bind to services.

The OSGi specification defines an infrastructure for a bundle's life cycle and how bundles interact. These rules are enforced through the use of special Java class loaders. In an average Java application, all classes in the CLASSPATH are visible to all other classes. In contrast, the OSGi class loaders restrict class interaction among bundles based on the OSGi specification and the options specified (covered in detail later in this article) in the manifest.mf file for each bundle.

The Eclipse IDE uses a subset of OSGi centered around modularization and bundle life cycle. However, it makes minimal use of the service support OSGi provides. Instead, Eclipse provides its own extension point system to enable bundle interaction. Bundles expose functionality as contributions to other extensions. Bundles also define their own extension points, to which other bundles may contribute. An example of using extension points in Eclipse is the Preferences window. A core Eclipse plug-in provides the central window and exposes an extension point to allow the contribution of additional preference pages. As new plug-ins are added to Eclipse, they can contribute their own pages. The model of extension points in Eclipse is different from basic OSGi services. Bundle extension points are owned by the defining bundle; other bundles simply contribute to them. In comparison, any bundle can implement and use an OSGi service.

Implementing Eclipse using OSGi

In versions of Eclipse before 3.1, you defined plug-in dependencies, as well as extensions and extension points, in each plug-in's plugin.xml file. In the newer versions of Eclipse that use OSGi, dependency information has been broken out into the manifest.mf file, leaving the plugin.xml file containing only XML definitions of extensions and extension points. It is useful to look at a live, working example of this evolution. Listing 1 shows an excerpt from the org.eclipse.pde.ui plug-in from Eclipse V3.0.

Listing 1. Excerpt from the org.eclipse.pde plug-in

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin
   id="org.eclipse.pde.ui"
   name="%name"
   version="3.0.2"
   provider-name="%provider-name"
   class="org.eclipse.pde.internal.ui.PDEPlugin">

   <runtime>
      <library name="pdeui.jar">
         <export name="*"/>
      </library>
   </runtime>
   <requires>
      <import plugin="org.eclipse.core.runtime.compatibility"/>
      <import plugin="org.eclipse.ui.ide"/>
      <import plugin="org.eclipse.ui.views"/>
      <import plugin="org.eclipse.jface.text"/>
      <import plugin="org.eclipse.ui.workbench.texteditor"/>
      <import plugin="org.eclipse.ui.editors"/>
      <import plugin="org.eclipse.ant.core"/>
      <import plugin="org.eclipse.core.resources"/>
      <import plugin="org.eclipse.debug.core"/>
      <import plugin="org.eclipse.debug.ui"/>
      <import plugin="org.eclipse.help.base"/>
      <import plugin="org.eclipse.jdt.core"/>
      <import plugin="org.eclipse.jdt.debug.ui"/>
      <import plugin="org.eclipse.jdt.launching"/>
      <import plugin="org.eclipse.jdt.ui"/>
      <import plugin="org.eclipse.pde"/>
      <import plugin="org.eclipse.pde.build"/>
      <import plugin="org.eclipse.search"/>
      <import plugin="org.eclipse.team.core"/>
      <import plugin="org.eclipse.ui"/>
      <import plugin="org.eclipse.update.core"/>
      <import plugin="org.eclipse.ui.forms"/>
      <import plugin="org.eclipse.ant.ui"/>
      <import plugin="org.eclipse.jdt.junit"/>
      <import plugin="org.eclipse.ui.intro"/>
      <import plugin="org.eclipse.ui.cheatsheets"/>
   </requires>


<!-- Extension points -->
   <extension-point id="pluginContent"
   name="%expoint.pluginContent.name"
   schema="schema/pluginContent.exsd"/>
   <extension-point id="newExtension"
   name="%expoint.newExtension.name"
   schema="schema/newExtension.exsd"/>
<extension-point id="templates"
name="%expoint.templates.name"
```

```
schema="schema/templates.exsd"/>
<extension-point id="samples"
name="%expoint.samples.name"
schema="schema/samples.exsd"/>

<!-- Extensions -->
   <extension
         point="org.eclipse.ui.perspectives">
      <perspective
            name="%perspective.name"
            icon="icons/eview16/plugins.gif"
            class="org.eclipse.pde.internal.ui.PDEPerspective"
            id="org.eclipse.pde.ui.PDEPerspective">
      </perspective>
   </extension>
```

The `<export name="*"/>` declaration exposes all packages in the plug-in for use in other plug-ins. The plug-in dependency imports section lists the prerequisite plug-ins the org.eclipse.pde.ui plug-in requires.

The next two sections define the extension points org.eclipse.pde.ui makes available to other plug-ins, as well as the contributions it makes to them. In this case, you can see the definition of a custom Eclipse Plug-in Development Environment (PDE) perspective.

Let's look at the same plug-in definition in Eclipse V3.1. Listing 2 shows the plugin.xml file.

Listing 2. Plugin.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin>


<!-- Extension points -->
   <extension-point id="pluginContent"
   name="%expoint.pluginContent.name"
   schema="schema/pluginContent.exsd"/>
   <extension-point id="newExtension"
   name="%expoint.newExtension.name"
   schema="schema/newExtension.exsd"/>
   <extension-point id="templates"
   name="%expoint.templates.name"
   schema="schema/templates.exsd"/>
   <extension-point id="samples"
   name="%expoint.samples.name"
   schema="schema/samples.exsd"/>

<!-- Extensions -->
   <extension
         point="org.eclipse.ui.perspectives">
      <perspective
            name="%perspective.name"
            icon="icons/eview16/plugins.gif"
            class="org.eclipse.pde.internal.ui.PDEPerspective"
            id="org.eclipse.pde.ui.PDEPerspective">
      </perspective>
```

Notice that the export and import information is gone. This information now resides in the manifest.mf file shown in Listing 3.

Listing 3. Manifest.mf

```
Manifest-Version: 1.0
Bundle-Name: %name
Bundle-SymbolicName: org.eclipse.pde.ui; singleton:=true
Bundle-Version: 3.1.0
Bundle-ClassPath: org.eclipse.pde.ui_3.1.0.jar
Bundle-Activator: org.eclipse.pde.internal.ui.PDEPlugin
Bundle-Vendor: %provider-name
Bundle-Localization: plugin
Require-Bundle: org.eclipse.core.runtime,
 org.eclipse.ui.ide,
 org.eclipse.ui.views,
 org.eclipse.jface.text,
 org.eclipse.ui.workbench.texteditor,
 org.eclipse.ui.editors,
 org.eclipse.ant.core,
 org.eclipse.core.resources,
 org.eclipse.debug.core,
 org.eclipse.debug.ui,
 org.eclipse.jdt.core,
 org.eclipse.jdt.debug.ui,
 org.eclipse.jdt.launching,
 org.eclipse.jdt.ui,
 org.eclipse.pde,
 org.eclipse.pde.build,
 org.eclipse.search,
 org.eclipse.team.core,
 org.eclipse.ui,
 org.eclipse.update.core,
 org.eclipse.ui.forms,
 org.eclipse.ant.ui,
 org.eclipse.jdt.junit,
 org.eclipse.ui.intro,
 org.eclipse.ui.cheatsheets,
 org.eclipse.update.configurator,
 org.eclipse.help.base
Bundle-ManifestVersion: 2
Eclipse-AutoStart: true
Export-Package: org.eclipse.pde.internal.ui;x-internal:=true,
 org.eclipse.pde.internal.ui.build;x-internal:=true,

 . . .

 org.eclipse.pde.ui,
 org.eclipse.pde.ui.internal.samples;x-internal:=true,
 org.eclipse.pde.ui.templates
```

The various plug-in imports are now specified as required bundles, and the * package export has been replaced with a list of explicitly exported packages.

The move away from plug-in-level dependencies to dependencies that required the explicit exporting and importing of packages generated a lot of commotion when Eclipse announced the news. The primary complaint was the lack of an equivalent of `<export name="*"/>`, which had existed in earlier versions of Eclipse. There are many reasons for this omission, however. The most important reason is the speed gains from having explicit import and exports. Previous versions of Eclipse had to open and scan each plug-in jar file to determine which classes it contained. Not including a * export also provides a level of protection against plug-ins exposing unwanted classes. Plug-in developers must make a conscious choice to make functionality in a plug-in available for outside use. This restriction allows internal packages to stay internal.

OSGi manifest options

The current draft specification for the OSGi R4 framework core is almost 300 pages in PDF form. Covering every portion of this specification is outside the scope of this article, but I do discuss the OSGi manifest.mf options of particular interest to Eclipse plug-in developers:

`Bundle-Activator`
: This class is used to start and stop the bundle. In the example plug-in above, the `org.eclipse.pde.internal.ui.PDEPlugin` class is specified. This class extends `org.eclipse.core.runtime.Plugin`, which implements the `BundleActivator` interface.

`Bundle-ClassPath`
: This property specifies the CLASSPATH to use for the bundle. The property may contain references to directories or jar files inside the bundle jar file. You can use the period to indicate the bundle's root. In the case of the example Eclipse PDE bundle, org.eclipse.pde.ui_3.1.0.jar in the bundle jar file is specified. If you import the source version of the plug-in into your workspace, the import process changes the bundle CLASSPATH to appear as `Bundle-ClassPath:`, which allows the development version of the plug-in to pick up the compiled bundle classes.

`Bundle-Version`
: This property specifies the version number of the bundle. Package imports and required bundle specifications may include a bundle version number.

`Export-Package`
: This property specifies all the packages to publicly expose to other plug-ins.

`Import-Package`
: This property specifies all the packages to explicitly import from required plug-ins. By default, all packages must be resolved for a bundle to start. You can also specify package imports as optional to support cases in which a package may not exist. Explicitly imported classes are resolved before packages from Require-Bundle plug-ins.

`Require-Bundle`
: This property specifies which bundles and their exported packages to import for use in the given bundle. Specified bundles are evaluated after explicit package imports.

Additional manifest options provided by Eclipse

Buddy class loader options

Begin by creating a plug-in for Hibernate. Then create a plug-in containing the domain-specific classes with a dependency on Hibernate. Add the following line to the Hibernate plug-in manifest: `Eclipse-BuddyPolicy: registered`.

Add the following manifest property to the manifest of the plug-in containing the domain-specific classes or resources: `Eclipse-RegisterBuddy: hibernate`.

This line allows plug-ins to declaratively expose themselves to the Hibernate plug-in without it knowing about them beforehand. Now the Hibernate plug-in can see the needed classes even though it did not specifically import them.

The manifest.mf configuration options that the OSGi specification includes don't provide all the functionality that the Eclipse platform requires. As a result, the Eclipse creators have

added several extensions (and also proposed them for inclusion in future versions of the OSGi specification):

`Export-Package` Header Extensions

Eclipse has two OSGi resolver methods -- `default` and `strict` -- that the `osgi.resolver` property can specify. Eclipse also includes two extensions to the `Export-Package` property -- `x-internal` and `x-friends` -- both of which are enforced when you enable Strict mode.

`x-internal`

The default value for this property is **false**. When internal packages are specified as **true** using this option, the Eclipse PDE discourages their use.

`x-friends`

This option is similar to `x-internal`, but allows certain bundles to use the exported packages that have this option. Others bundles are discouraged. The `x-internal` option takes precedence over `x-friends`.

`Eclipse-AutoStart`

By default, Eclipse loads bundles on demand. Therefore, bundles are loaded when the first class they contain is required by a bundle that imports it. Specifying this value as **??** causes Eclipse to load the bundle at startup. You can also specify a list of exceptions, which are classes, and resources, and which may be loaded without starting their containing bundle.

`Eclipse-PlatformFilter`

This property allows you to specify conditions that must evaluate to **true** for a bundle to start. You can include the following information in the expression you specify:

- `osgi.nl` for language
- `osgi.os` for operating system
- `osgi.arch` for architecture
- `osgi.ws` for windowing system

An example of how to use this property is to verify whether the operating system is not Mac OS X before starting a plug-in that uses the `SWT_AWT` bridge. (The Mac OS X implementation of the Standard Widget Toolkit (SWT) doesn't currently support this feature.)

`Eclipse-BuddyPolicy`

This option specifies the class loading policy of the bundle. Typically, bundles only have visibility in their internal classes and those imported from dependent bundles. The popular example that has been used in the Eclipse newsgroups to explain buddy class loading is Hibernate. The Hibernate framework must see classes and resources that are user-created and not part of Hibernate itself. One such case is when you use a project to dynamically fill a class from a Hibernate Query Language (HQL) query. By default, Hibernate won't be able to see classes outside the plug-in containing the Hibernate jar files, and requiring modification of the Hibernate plug-in for the creation of each plug-in containing classes that Hibernate maps is not acceptable. Fortunately, you can solve this using the buddy class loader options explained in the Buddy class loader options section.

Future directions of Eclipse and OSGi

Eclipse has benefited greatly in its use of OSGi, gaining a robust system managing the life cycle of components in a dynamic manner. New uses, such as dynamic Web Archive (WAR) files on the

server tier featuring servlets, JavaServer Pages, and other HTTP resources in Eclipse-style plug-ins are being explored every day.

The Eclipse Foundation has positioned itself to play a key role in driving the OSGi specification forward, both for its own uses and for those of other parties leveraging OSGi. During the transition from the propriety Eclipse plug-in framework to OSGi, many additions were made to the OSGi specification that became part of the OSGi R4 specification release. The Eclipse Equinox project has, as a result, become the OSGi reference implementation moving forward. This involvement, as well as the creation of Java Specification Request (JSR) 291 to manage the evolution OSGi, guarantees that the Eclipse/OSGi partnership will continue to be successful in the years ahead.

Related topics

- For more information about OSGi, visit the OSGi Alliance.
- Wikipedia provides a good definition of OSGi.
- Discover Oscar, an alternative OSGi implementation.
- Learn more about the Eclipse Equinox project.
- "Eclipse goes Server-side!" by Wolfgang Gehner, explains how to create Eclipse plug-ins you can deploy to servlet containers.
- Read JSR 291: Dynamic Component Support for Java SE, the JSR 291 specification.
- Expand your Eclipse skills by visiting IBM developerWorks Eclipse project resources.
- Browse all of the Eclipse content on developerWorks.
- Innovate your next open source development project with IBM trial software, available for download or on DVD.