DZone Refcardz

CONTENTS INCLUDE:

Eclipse Plug-in Development

By James Sugrue

ABOUT ECLIPSE PLUG-INS

The Eclipse platform consists of many plug-ins, which are bundles of code that provide some functionality to the entire system. Plug-ins contribute functionality to the system by implementing pre-defined extension points. You can provide extension points in your own plug-in to allow other plug-ins to extend your functionality.

Hot Tip

Eclipse has a dedicated perspective for development of plug-ins, the PDE (Plug-in Development Environment). You can download Eclipse for RCP/ Plug-in Developers with all you need to get started from http://www.eclipse.org.

HOW PLUG-INS WORK

A **plug-in** describes itself to the system using an OSGi manifest (**MANIFEST.MF**) file and a plug-in manifest (**plugin.xml**) file. The Eclipse platform maintains a registry of installed plug-ins and the function they provide. As Equinox, the OSGi runtime, is at the core of Eclipse, you can think of a plug-in as an OSGi bundle. The main difference between plug-ins and bundles is that plug-ins use extension points for interaction between bundles.

Plug-ins take a lazy-loading approach, where they can be installed and available on the registry but will not be activated until the user requests some functionality residing in the plug-in.

THE OSGI MANIFEST

MANIFEST.MF, usually located in the META-INF directory, deals with the runtime details for your plug-in. Editing of the manifest can be done through the editor provided, or directly in the MANIFEST.MF tab. The following is an example of one such manifest for a simple plug-in:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Myplugin
Bundle-SymbolicName: com.dzone.tests.myplugin
Bundle-Version: 1.0.0.qualifier
Bundle-Activator: com.dzone.tests.myplugin.Activator
Require-Bundle: org.eclipse.ui, org.eclipse.core.runtime
Bundle-ActivationPolicy: lazy
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
```

The Eclipse OSGi Framework implements the complete OSGi R4.1 Framework specification and all of the Core Framework services. Here we list the most common manifest headers and directives.

Manifest Entry	Use	Example
`Manifest-Version`	Manifest versioning information for your own records	`1.0`
`Bundle-ManifestVersion`	A bundle manifest may express the version of the syntax in which it is written by specifying a bundle manifest version. If using syntax from OSGi Release 4 or later, you must specify a bundle manifest version. The bundle manifest version defined by OSGi Release 4 is "2".	`2`
`Bundle-Name`	Human readable name for the plug-in.	`MyPlugin`
`Bundle-SymbolicName`	A unique name for this plug-in, usually in package naming convention.	`com.dzone.tests.myplugin`
`Bundle-Version`	The version of this plug-in. This should follow the typical three number versioning format of <major version>.<minor version>.<revision> This can also be appended by an alphanumeric qualifier.	`1.0.1.alpha`
`Bundle-Activator`	The activator, or plug-in class, that controls this plug-in.	`com.dzone.tests.myplugin.Activator`
`Bundle-Vendor`	Human readable string for the plug-in provider.	`DZone`
`Bundle-Classpath`	A comma-separated list of directories and jar files used to extend this bundle's functionality.	`lib/junit.jar,lib/xerces.jar`

`Require-Bundle`	A comma-separated list of symbolic names of other bundles required by this plug-in.	`rg.eclipse.ui, org.eclipse.core.runtime`
`Bundle-ActivationPolicy`	Manifest header identifying the bundle's activation policy. This replaces the deprecated Eclipse-LazyStart directive.	`Lazy`
`Bundle-Required ExecutionEnvironment`	Manifest header identifying the required execution environment for the bundle. The platform may run this bundle if any of the execution environments named in this header match one of the execution environments it implements.	`JavaSE-1.6`
`Export-package`	A list of the packages that this bundle provides for export to other plug-ins.	`com.dzone.tests.api`

Plug-in Runtime

The `Require-bundle` manifest header has some extra functionality to help you manage your runtime dependencies. Bundles can be marked as optional dependencies by annotating the bundle with `;resolution:=optional`.

You can also manage which version of the bundle your dependent on needs to be present using the `;bundle-version="<values>"` annotation. Here, the `<values>` that we refer to are a range of versions where you can specify minimum and maximum version ranges. The syntax of this range value is illustrated through these examples:

Example	Meaning
`3.5`	Dependent only on version 3.5 of this bundle
`[3.5, 3.5.1]`	Must be either version 3.5 or 3.5.1
`[3.0, 4.0]`	Must be a version of 3.0 or over, but not 4.0

Additional Eclipse Bundle Headers

Eclipse provides a number of addition bundle headers and directives. These extra headers are not part of the OSGi R4.1 specification, but allow developers to use additional Eclipse OSGi Framework functionality.

Manifest Entry	Use	Example

`Export-Package`	Additional directives are available to manage the access restriction of exported packages.
	
	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. Other bundles are discouraged. The `x-internal` option takes precedence over `x-friends`.
		`Export-Package: org.eclipse.foo.internal; x-internal:=true`
		
		`Export-Package: org.eclipse.foo.formyfriends; x-friends:="org.eclipse.foo.friend1"`

`Eclipse-PlatformFilter`	This allows you to set particular rules for your bundle before it can start. `osgi.nl` for language `osgi.os` for operating system `osgi.arch` for architecture `osgi.ws` for windowing system	`Eclipse-PlatformFilter: (& (osgi.ws=win32) (osgi.os=win32) (osgi.arch=x86))`

All entries in the manifest can be internationalized by moving them to a separate plugin.properties file.

THE PLUG-IN MANIFEST

With the Manifest.MF file looking after the runtime dependencies, plugin.xml deals with the plug-in extensions and extension points.

An extension allows you to extend the functionality of another plug-in in your system. An extension can be added through the plug-in editor's Extensions tab, or to your plugin.xml.

```
<extension point="org.eclipse.ui.preferencePages">
   <page
        class="com.dzone.tests.myplugin.preferences.
               SamplePreferencePage"
        id="com.dzone.tests.myplugin.preferences.
            SamplePreferencePage"
        name="Sample Preferences">
    </page>
 </extension>
```

Each extension point has a XML schema which specifies the elements and attributes that make up the extension. As you can see in the listing above, each extension point has a unique identifier. The <page> element above is specified in the XML schema for the org.eclipse.ui.preferencesPages extension.

Hot Tip Plug-ins and extension points are expected to have the same unique identifiers following the Java package naming pattern.

You can also define your own extension points, and we will detail that process in a later section.

PLUG-IN MODEL

The plug-in class is a representation of your plug-in running in the Eclipse platform. A plug-in class in Eclipse must extend `org.eclipse.core.runtime.Plugin`, which is an abstract class that provides generic facilities for managing plug-ins. When using the project wizard in the PDE, this class typically gets assigned Activator as its default name. Whatever name you assign to this plug-in class, it must be the same as that mentioned in the Bundle-Activator directive of your MANIFEST.MF.

The class has start and stop methods that refer to the BundleContext and are provided by the BundleActivator interface. These methods allow you to deal with the plug-ins lifecycle, so that you can do both initialization and cleanup activities at the appropriate times. When overriding these methods be sure to always call the superclass Implementations.

Hot Tip Plug-ins that contribute to the UI will have activators that extend `AbstractUIPlugin`, while non-UI plug-ins will extend `Plugin`.

Bundle Context

A `BundleContext` is associated with your plug-in when it is started. As well as providing information about the plug-in, the `BundleContext` can provide information about other plug-ins in the system. By providing a listener to `BundleEvent`, you can monitor the lifecycle of any other plug-in.

Bundle

The terms *Bundle* and *Plug-in* may be used interchangeably when discussing Eclipse. The `Bundle` class provides us with the OSGi unit of modularity. There are six states associated with bundles:

State	Meaning
`UINSTALLED`	The bundle is uninstalled and not available.
`INSTALLED`	A bundle is in the INSTALLED state when it has been installed in the Framework but is not or cannot be resolved
`RESOLVED`	Before a plug-in can be started, it must first be in the RESOLVED state. A bundle is in the RESOLVED state when the Framework has successfully resolved the bundle's code dependencies.
`STARTING`	A bundle is in the STARTING state when its start method is active. If the bundle has a lazy activation policy, the bundle may remain in this state until the activation is triggered.
`STOPPING`	A bundle is in the STOPPING state when its stop method is active. When the BundleActivator.stop method completes the bundle is stopped and must move to the RESOLVED state.
`ACTIVE`	A bundle is in the ACTIVE state when it has been successfully started and activated.

LAZY LOADING

Plug-ins are normally set to load lazily, so that the code isn't loaded into memory until it is required. This is normally a good thing as you don't want to affect the startup time of Eclipse. If you do require your plug-in to start up and load when Eclipse launches, you can use the `org.eclipse.ui.startup` extension point.

```
<extension point="org.eclipse.ui.startup">
    <startup class="com.myplugin.StartupClass"></startup>
</extension>
```

The startup class listed above must implement the `org.eclipse.ui.IStartup` interface which provides an `earlyStartup()` method. The method is called in a separate thread after the workbench initializes.

EXTENSION POINTS

The Eclipse platform provides a number of extension points that you can hook into, to provide additional functionality. The concept behind an extension point is that a class provides some extendable behavior, and publishes this behavior as an extension point. In order to run this code, the plug-in requires a host – in this case your own plug-in.

In your plugin.xml you take this extension point and provide extra information to help it run. You will usually need to provide some class that implements a particular interface in order to do this.

Here we will run through some useful extension points in the Eclipse platform. Note, that to make some of these available for your plug-in, you will usually need to add dependencies.

Example	Meaning
`org.eclipse.core.runtime .preferences`	Allows plug-ins to use the Eclipse preferences mechanism, including the setting of default preference values.
`org.eclipse.core.runtime .applications`	A plug-in that wishes to use the platform but control all aspects of its execution is an application.
`org.eclipse.core.resources .builders`	Useful for IDE builders who wish to provide an incremental project builder, processing a set of resource changes.
`org.eclipse.core.resources .markers`	Markers are used to tag resources with use information – this marker can then be utilized in the problems view.
`org.eclipse.ui.activities`	The activity extension point allows the filtering of plug-in contributions from users until they wish to use them.
`org.eclipse.ui.editors`	Allows the addition of new editors to the workbench, which can be tied to particular file extension types.
`org.eclipse.ui.intro`	When Eclipse is first started up the welcome page, or intro is displayed. This extension point allows contributions to the welcome page.
`org.eclipse.ui.menus`	Allows custom menus to be added to the workbench either in the main menu, toolbar or popup menus through the locationURI attribute.
`org.eclipse.ui.perspective`	Allows the addition of a perspective factory to the workbench, defining a particular layout of windows.
`org.eclipse. ui.propertyPages`	Adds a property page for objects of a given type.
`org.eclipse.ui.themes`	Allows the customization of the user interface, overriding the default colors and fonts.
`org.eclipse.ui.views`	Provides the ability to add views to the workbench.

CREATING YOUR OWN EXTENSION POINTS

As well as being a user of extension points, a plug-in can provide its own extensions for other plug-ins. Extension points allow loose coupling of functionality – your plug-in exposes a set of interfaces and an extension point definition for others to use.

Extension Point Definition

You can create your extension point through the **plugin.xml** file, or through the Add button in the **Extension Points** tab of the plug-in editor.

For identifying your extension point you need to provide a unique identifier and a human readable name. At this point you can also point to a schema file and edit it afterwards. An extension point schema must have **.exsd** as its suffix.

From here you can investigate the plug-in dependency hierarchy, starting with your plug-in as the root. You can also see which plug-ins are dependent on your own plug-in, as well as find any unused dependencies. This can be useful if you previously added a dependency to use an extension point, but have found that it is since no longer required. Finally, and most importantly, the tab provides a utility for investigating for cyclic dependencies.

Another useful tool for plug-in development is the Plug-in Registry view. This can be accessed from the **Window>Show View>Other..>Plug-in Development** category. This view will display all the plug-ins that are currently available in your Eclipse installation.

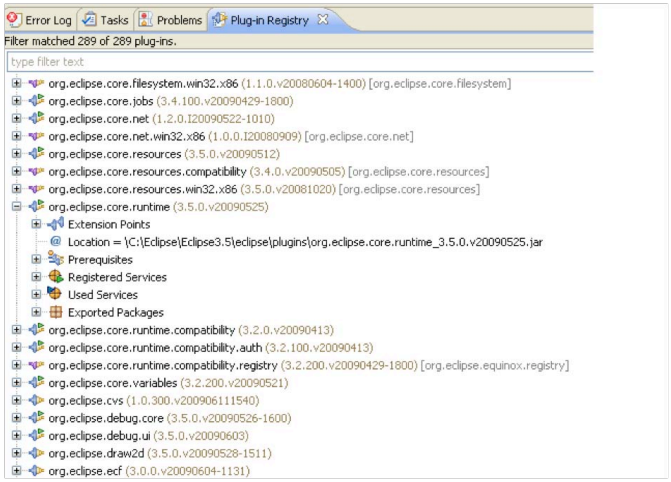

Figure 4: Plug-in Registry

Hot Tip When launching an application containing you plug-in, use the `-consoleLog` program argument from the `Run Configurations` dialog to see output to the system console.

Logging

It is recommended to log to a file, rather than using System. out. The Activator or plug-in class provides a facility to access the plug-in logging mechanism through the getLog() method, returning the org.eclipse.core.runtime.ILog interface.

Each log entry using this framework is of typ e IStatus. Any CoreExceptions thrown in Eclipse have an associated IStatus object. An implementation of this interface, Status, is available for use. There is also a MultiStatus class which allows multiple statuses to be logged at once.

DISTRIBUTING YOUR PLUG-IN

Since Eclipse 3.4, p2 has been used as the method to provision your application with new or updated plug-ins. For build managers who have used the Update Site mechanism before, there doesn't need to be any change.

To create an update site you can use the wizard provided to create a new site.xml file. Using the Software Updates menu, users can point to your update site on the web and download the plug-in.

By adding some extra functionality over this simple implementation, you can leverage p2 to add extra meta data to your update site, which will make the installation experience faster for end users.

p2 Update Site Publisher

The UpdateSite Publisher application is provided by p2 to generate an artifact.xml and content.xml files for your standard update site. You can run this application in headless mode using org.eclipse.equinox.p2.publisher.UpdateSitePublisher. The following shows an example of how to run this application, taken from the p2 wiki.

```
java -jar <targetProductFolder>/plugins/org.eclipse.equinox.
launcher_*.jar
 -application org.eclipse.equinox.p2.publisher.UpdateSitePublisher
 -metadataRepository file:/<some location>/repository
 -artifactRepository file:/<some location>/repository
 -source /<location with a site.xml>
 -configs gtk.linux.x86
 -compress
 -publishArtifacts
```

Read more about p2 at http://wiki.eclipse.org/Equinox/p2

ENHANCING YOUR PLUG-IN

When developing your plug-in, you should be aware of the wide variety of projects available in the Eclipse eco-system that help make your development easier and faster. This section gives an overview of just a few of the useful projects that exist, and explains how they can be used in your project.

Eclipse Modeling Project

http://eclipse.org/modeling/

The Eclipse Modelling Project provides a large set of tools for model driven development. The most popular part of this project is the Eclipse Modelling Framework (EMF). Using this technology, you can define a model in the ecore format, generate Java code to represent, serialise and de-serialise the model. Other tools within the modelling project utilise EMF to provide more specialised frameworks for developers.

The Connected Data Objects (CDO) project provides a three-tier architecture for distributed and shared models.

The Graphical Modelling Framework (GMF) allows you to generate graphical editors for your model based on EMF and the Graphical Editing Framework (GEF). For developers who want to provide textual editor for their own language or DSL, XText provides a EBNF grammar language and generates a parser, meta-model and Eclipse text editor from this input.

Eclipse Communication Framework

http://eclipse.org/ecf

If your plugin requires any communication functionality, the ECF project is the first place to look. ECF consists of a number of bundles that expose various communication APIs. These APIs range from instant messaging, dynamic service discovery, file transfer to remote and distributed OSGi. Real-time shared editing functionality is also available in the framework, allowing you to collaborate remotely on anything that you are editing within your plug-in's environment.

Business Intelligence and Reporting Tools

http://eclipse.org/birt

BIRT is an open source reporting system based on Eclipse.

BIRT provides both programmatic access to report creation, as well as functionality to create your own report template within the Eclipse IDE. While BIRT allows you to generate reports in file formats such as PDF, it is also possible to use BIRT on an application server to serve reports through a web browser.

Equinox
http://eclipse.org/equinox

As we have described in this card, Equinox is the Eclipse implementation of the OSGi R4 core framework specification, and provides the real runtime for all your plug-ins. However, as well as running your plug-ins on the desktop on an instance of Eclipse, you can take Equinox and run it on a server, allowing your plug-in to run on browsers as well as the desktop.

Rich Ajax Platform
http://eclipse.org/rap

With the emergence of the web as a real platform for rich applications, the Rich Ajax Platform allows you to take a standard RCP project, and with some minor modifications, make it deployable to the web. This idea of single-sourcing is key to the RAP project, and reduces the burden for developers to make an application ready for either the desktop or the web.

The same programming model is used, while qooxdoo is used for the client side presentation of your SWT and JFace widgets.

ABOUT THE AUTHOR

James Sugrue is a software architect at Pilz Ireland, a company using many Eclipse technologies. James is also editor at both EclipseZone and Javalobby. Currently he is working on TweetHub, a Twitter client based on RCP and ECF. James has also written previous Refcardz covering EMF and Eclipse RCP.

Zone Leader: EclipseZone, Javalobby

Twitter: @dzonejames

RECOMMENDED BOOKS

This book presents detailed, practical coverage of every aspect of plug-in development--with specific solutions for the challenges you're most likely to encounter.

BUY NOW
books.dzone.com/books/eclipse-plug-ins

In **Eclipse Rich Client Platform**, two leaders of the Eclipse RCP project show exactly how to leverage Eclipse for rapid, efficient, cross-platform desktop development.

BUY NOW
books.dzone.com/books/eclipse-rcp

DZone, Inc.
1251 NW Maynard
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-934238-62-2
ISBN-10: 1-934238-62-7
50795
9 781934 238622
$7.95