IBM

Plug-in development 101, Part 1: The fundamentals

Learn the basics of Eclipse plug-in development.

Chris Aniszczyk, Software Engineer, IBM, Software Group

Summary: Plug-in development in Eclipse is somewhat of an art form. If you're new to the concept of plug-ins, especially in the context of OSGi and Eclipse, it can be quite burdensome learning the myriad tools Eclipse has to help you write plug-ins. The purpose of this article is to help you learn some basic plug-in development skills with some best practices sprinkled in for good measure.

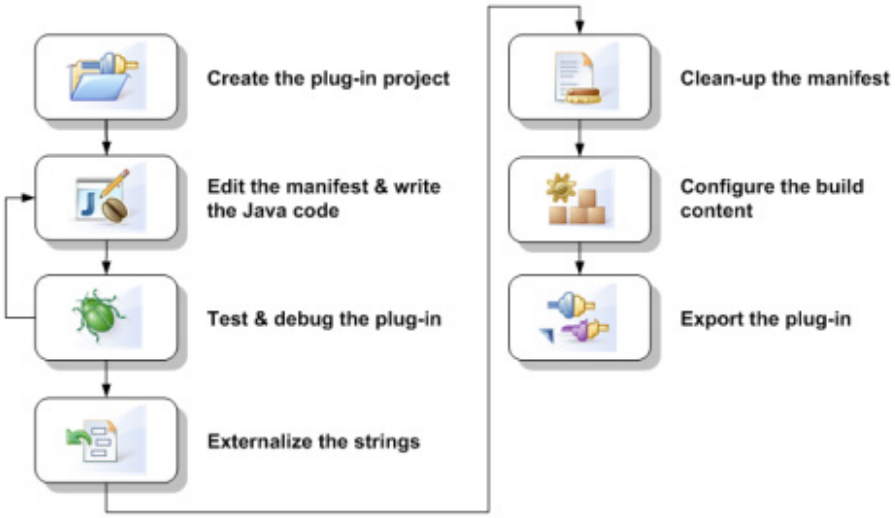
Date: 12 Feb 2008
Level: Intermediate
Activity: 6298 views
Comments: 0 (Add comments)

 Average rating (based on 54 votes)

This "Plug-in development 101" series of articles is all about developing plug-ins. But before we get started, we need to ensure that we have a proper environment in which to develop plug-ins. The first step is to download an Eclipse distribution that has the Plug-in Development Environment (PDE) in it from Eclipse.org. I recommend downloading the latest version of Eclipse Classic. In this series, we will use a milestone release of Eclipse V3.4 (M5). Once this done, you're ready to go. (See Resources to learn where to find Eclipse and additional background information if you are new to Eclipse.)

To make it easier to understand plug-in development, this article will follow a workflow detailed in Figure 1. In Part 1 of this series, we will focus on the first five steps of the workflow. We will leave the last two steps for Part 2, which will focus on rich-client applications.

Figure 1. Plug-in development workflow



What's this OSGi business about?

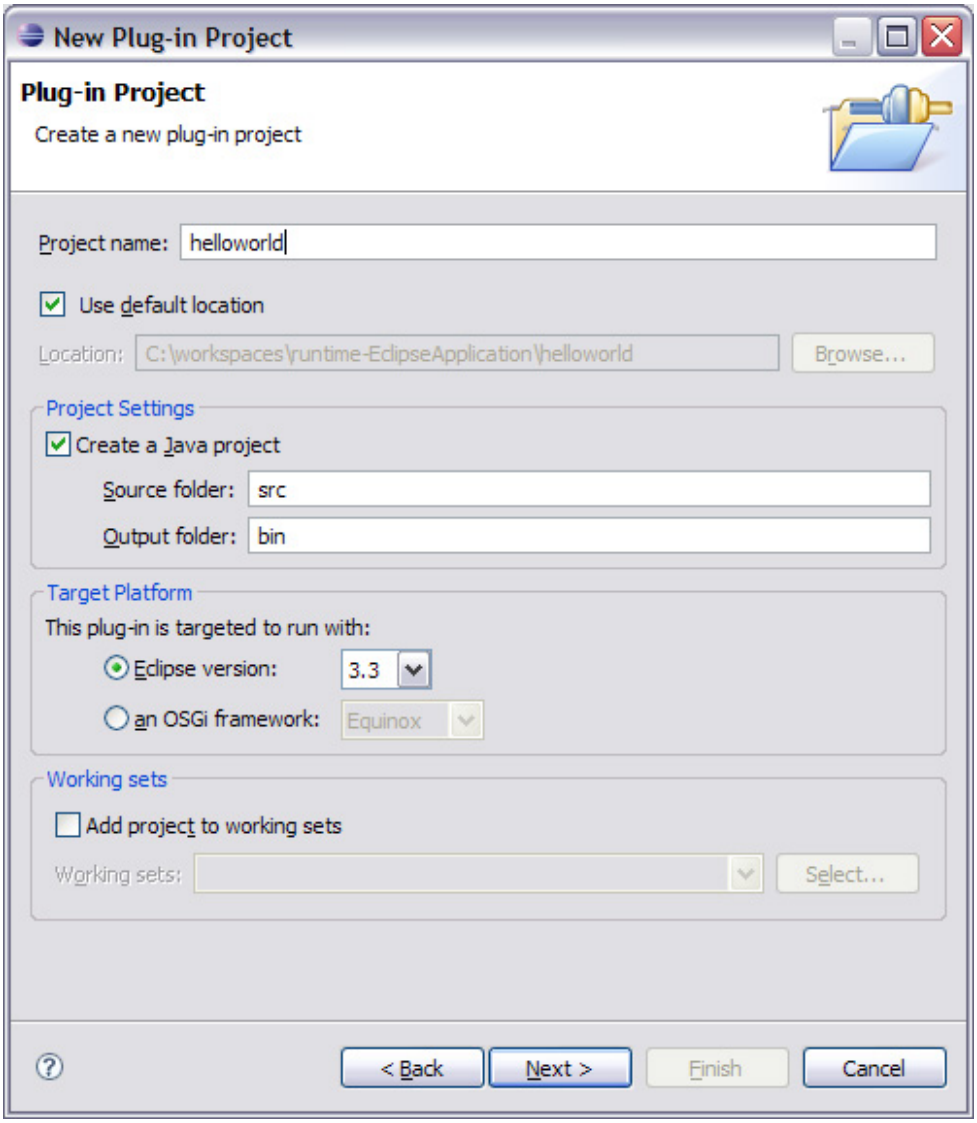
In V3.0, Eclipse made a big leap by choosing OSGi to replace the rickety Eclipse plug-in technology found in earlier versions. The OSGi Alliance is an independent, nonprofit corporation responsible for OSGi technology. It is akin to the Eclipse Foundation in function. The OSGi Alliance is responsible for producing specifications describing OSGi technology. In brief, OSGi technology provides a service-oriented plug-in-based platform for application development. Various implementations are based on these specifications. One of the most popular implementations is Equinox, which is Eclipse's implementation of the specification. (See Resources for background information.)

Before we dive into the details of plug-in creation, let's discuss what a plug-in exactly is. Technically, a plug-in is a Java™ Archive (JAR) that is self-contained and self-describing. It's *self-containing* because it houses the code and resources the plug-in needs to run. It's *self-describing* because it includes information stating what it is, what it requires from the world, and what it contributes to the world. In a plug-in, you will generally see two descriptor files: a MANIFEST.MF and plugin.xml.

In the beginning, there was creation

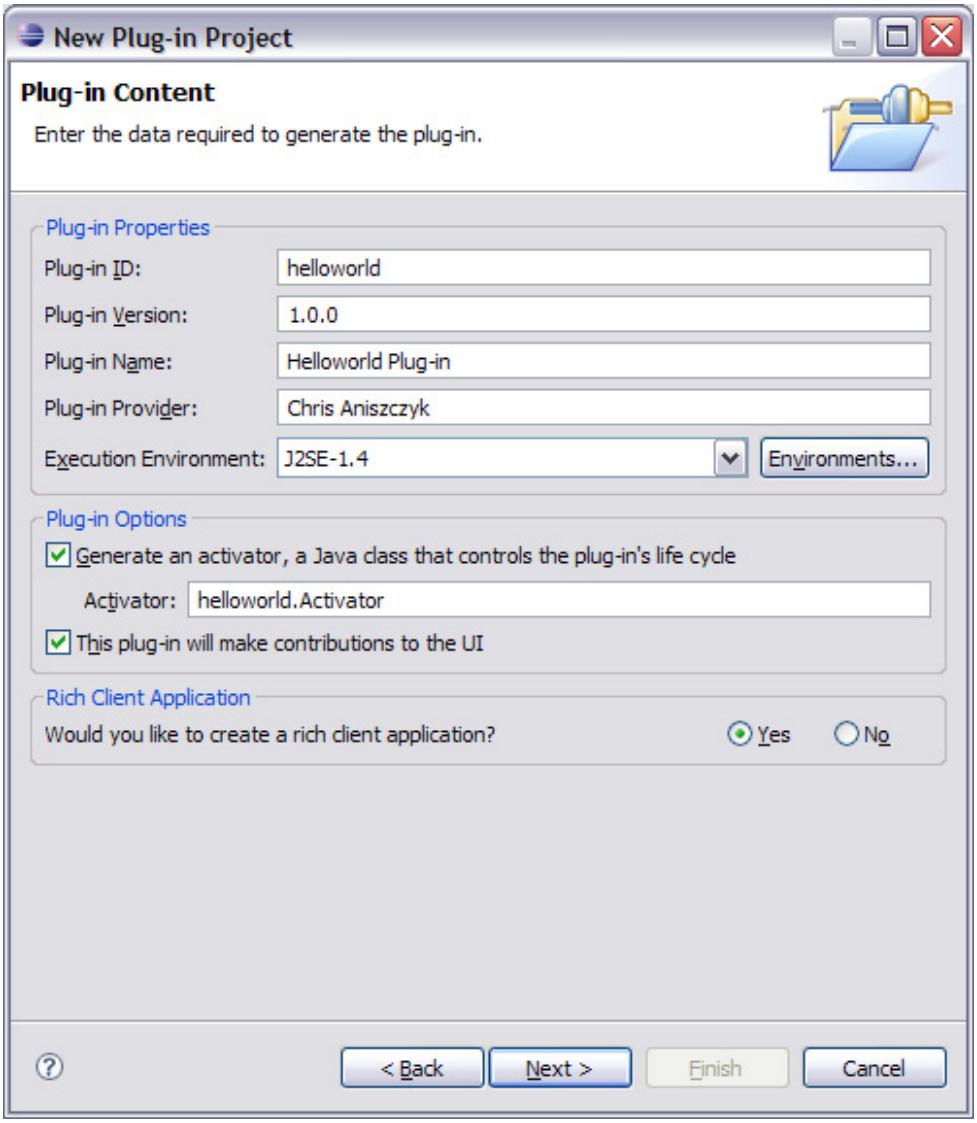
The first part of the plug-in development workflow involves creating a plug-in project. In Eclipse, this is easily done by selecting the **New >Project...** menu option. In the wizard presented, select **Plug-in Project** as the type of project you want to create.

Figure 2. New Plug-in Project wizard



Just like any other Eclipse project, the wizard asks you to choose a project name. I suggest `helloworld`. There is also an option to choose a target platform. A target platform in this context simply means whether you will target a version of Eclipse or an OSGi framework like Equinox. In this case, we'll target the 3.3 version of Eclipse to keep things straightforward. The next page of the New Plug-in Project wizard focuses on plug-in content.

Figure 3. Plug-in content

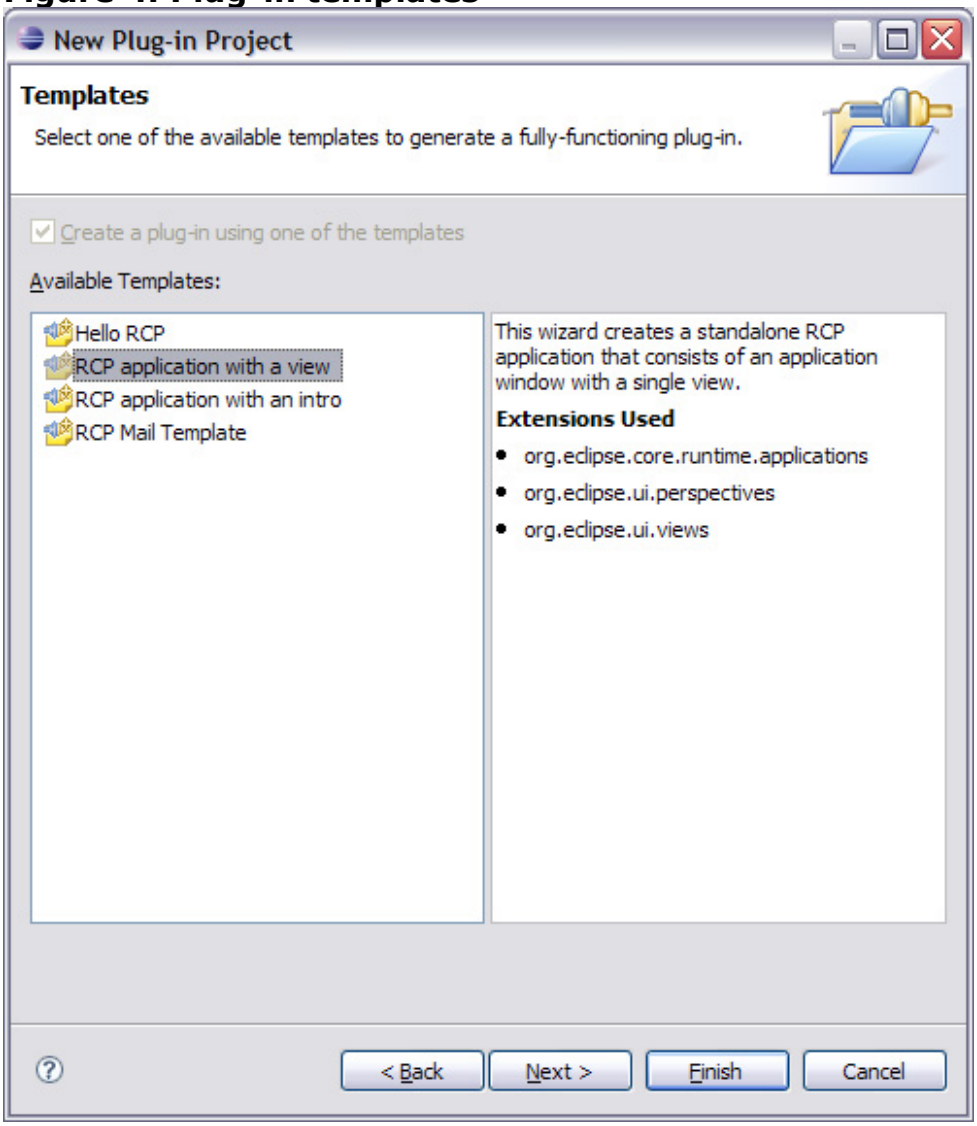


On the plug-in content wizard page, we must complete a form to make our plug-in compliant with Eclipse. The plug-in identifier is a field that represents a unique identifier for your plug-in. No other plug-in can share the same identifier. The plug-in version is composed of four segments (thee integers and a string) respectively named *major.minor.service.qualifier* (see Resources for the Eclipse Plug-in Versioning guide). The plug-in name simply represents a human-readable name. The plug-in provider field is a human-readable string signifying the author of the plug-in. In simple terms, the execution environment (EE) field represents what minimum JRE your bundle is capable of running on (see Resources).

The wizard gives us an option to generate a plug-in activator. A plug-in activator is simply a class that controls the life cycle of your plug-in (think of it as a start-and-stop method). Customarily, the activator is responsible for setting up things and properly disposing of resources when your plug-in isn't needed anymore. In this case, we desire an activator, a plug-in that makes contributions to the UI, and we're creating a Rich Client Platform (RCP) application (see Resources).

The final step in plug-in creation involves selecting a template (see Figure 4) to base your new plug-in on. When you get more advanced in the creation of plug-ins, this step is usually skipped. However, beginners need to start somewhere, and the Eclipse SDK ships many templates to help you get started. In this case, we select the basic Hello World with a view application.

Figure 4. Plug-in templates



Modification

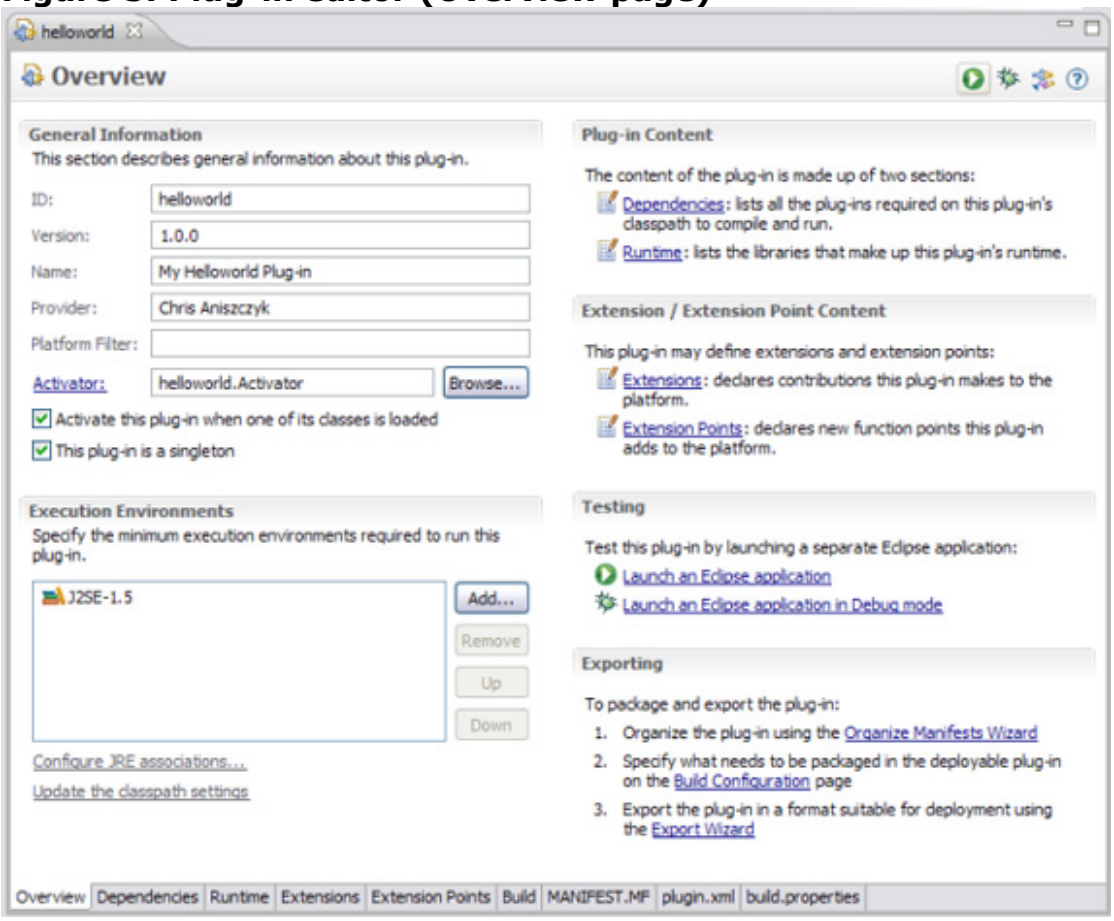
Inside the MANIFEST.MF

If you're interested in the various headers available to you inside a MANIFEST.MF file, read the OSGi specifications available from the OSGi Alliance (see Resources).

After completing the previous step, a new plug-in was created in the workspace. We're now presented with an editor (see Figure 5) to help us work with the fresh

plug-in. The first page in the editor we see is the Overview page, which contains modifiable plug-in identifier information with sections describing the various items you can edit within our plug-in. For example, we see the identifier (ID) and version fields we previously defined using the template shown in Figure 4.

Figure 5. Plug-in editor (Overview page)



Extensions and extension points

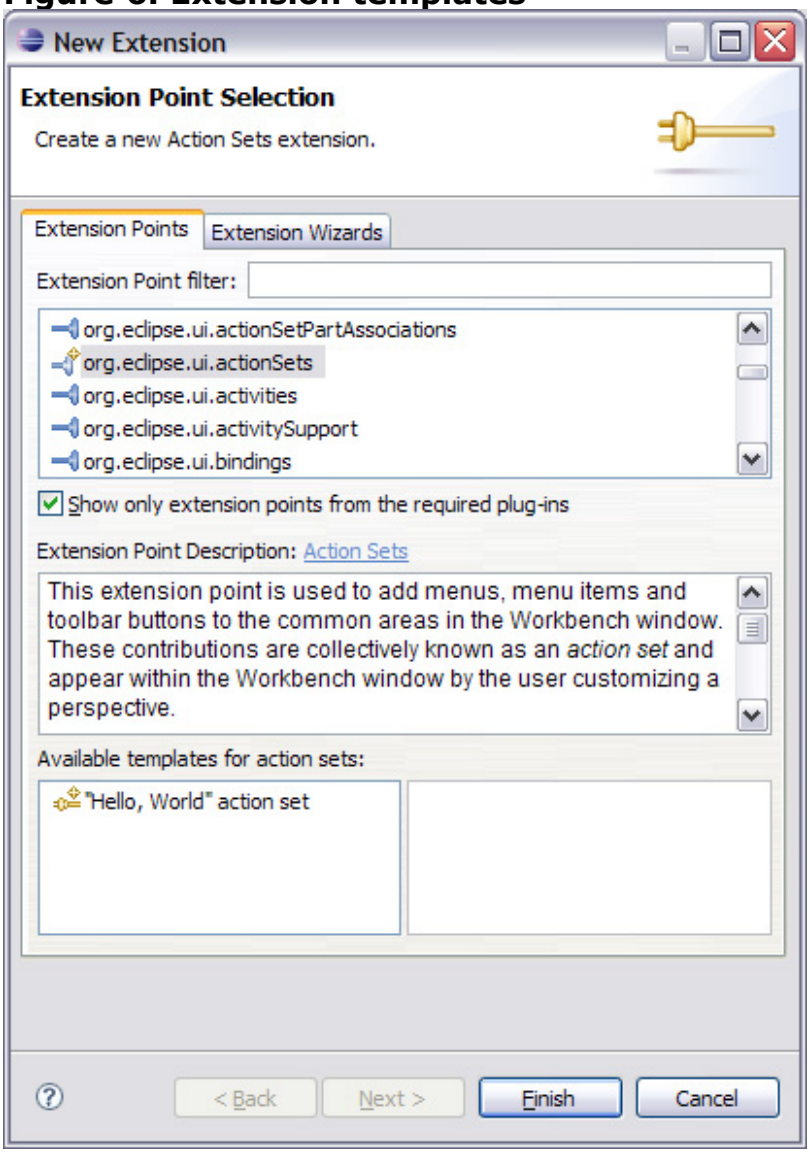
Extension points galore

As of Eclipse V3.3, there are more than 200 extension points found in the Eclipse SDK. To see a list of what's extensible (i.e., extension points) in the SDK, check out the Eclipse Foundation's document titled "Platform Extension Points" (see Resources).

The next step on our plug-in adventure involves adding an extension. However, before we dive into the details, let's first understand two important topics when developing extensible Eclipse plug-ins: extensions and extension points. Using a simple analogy, you can think of an *extension* as a plug and an *extension point* as a socket. Each extension point is unique and defines a contract to be adhered to. For example, Eclipse ships with an `org.eclipse.ui.editors` extension point (socket) that allows you to supply our own editor (the plug).

In our case, we'll create a new extension that contributes to the Eclipse toolbar. To accomplish this, we will use a template (similar in concept to the template we used to create our plug-in) for the `org.eclipse.ui.actionSets` extension point. From the Overview page in the plug-in editor, select the **Extensions** link in the **Extension/Extension Point Content** section. Click **Add...** and select the `org.eclipse.ui.actionSets` template (see Figure 6) with all the defaults filled in.

Figure 6. Extension templates

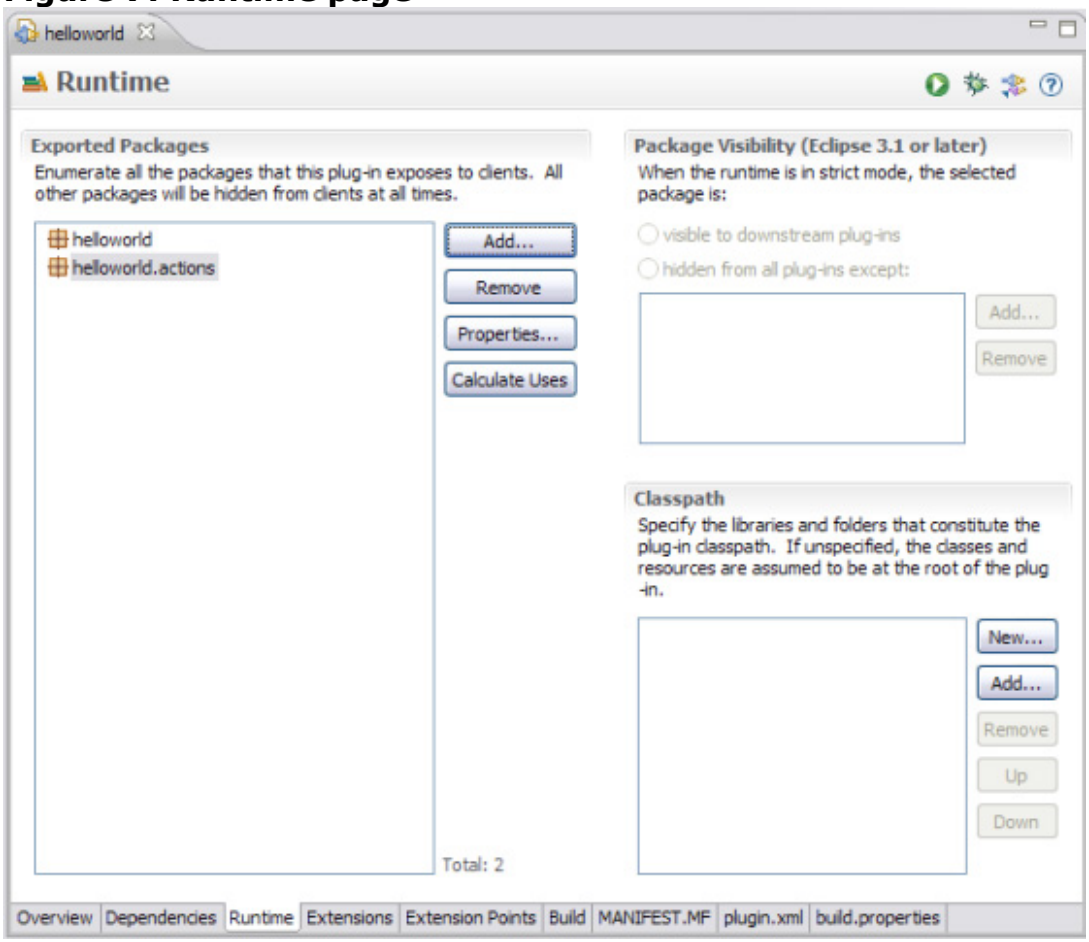


Runtime

An important part of our plug-in is that it's self-describing. One of the things our plug-in must describe is what it offers the world. In the context of Eclipse, these are called *exported packages*. In our plug-in, we get to decide what packages we export so other plug-ins can peek inside those packages to see they depend on our plug-in. It's also possible to mark exported packages as being internal, which tells plug-in developers that we don't consider the package in question as API. To specify

exported packages, we use the Runtime page within the manifest editor.

Figure 7. Runtime page



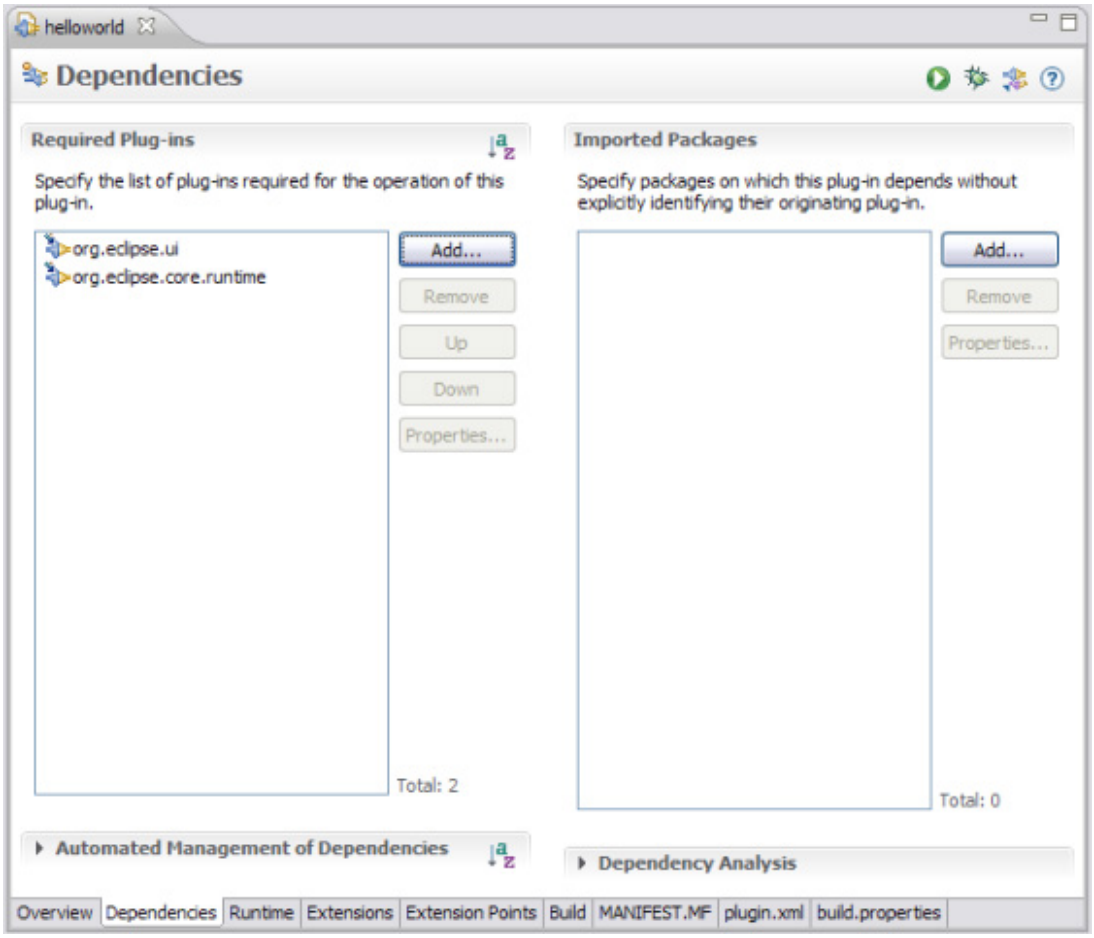
Dependencies

Plug-ins galore

There are so many plug-ins available in the Eclipse ecosystem to take advantage of that it's possible to get lost. I suggest starting with two sites when searching for plug-ins: The first stop is the Eclipse Foundation's project list, the second is Eclipse Plug-in Central (EPIC) (see Resources).

In the previous section, we described what a plug-in must do to expose its functionality to the world. Imagine you are a different plug-in and wanted to use the aforementioned functionality. How would you express your interest in doing so? In the context of Eclipse, these are called *dependencies*. In the simplest case, plug-ins can depend on other plug-ins. For example, if you want to build your own editor in Eclipse, you would need to depend on the `org.eclipse.ui.editors` plug-in. To specify dependencies, you have to be on the **Dependencies** page of the plug-in editor.

Figure 8. Dependencies page

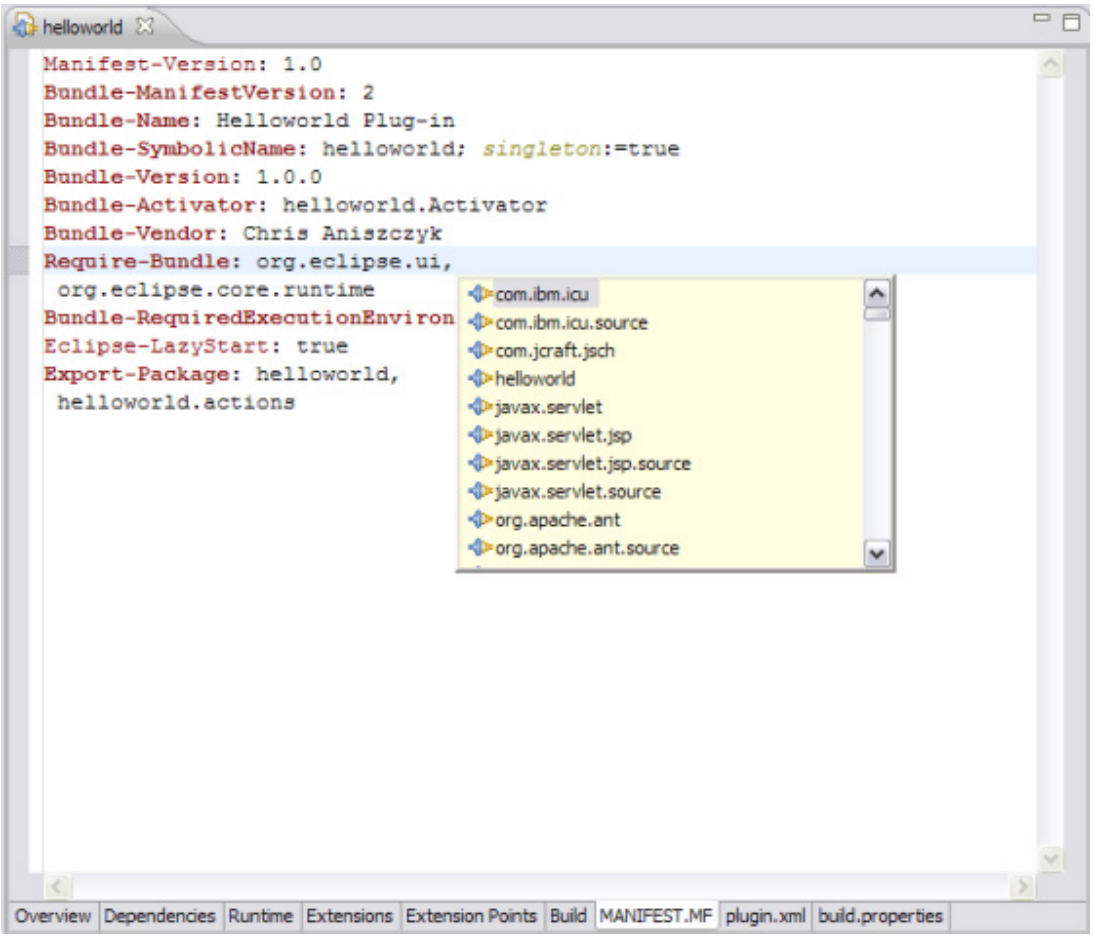


Note that on top of depending on individual plug-ins, you can choose to depend on packages that are exported from plug-ins (see the Imported Packages section on the Dependencies page). This is more of an advanced topic and useful when you don't want to tie your plug-in to a specific implementation. For example, imagine depending on a package `com.company.xml.parser` that supplied an XML parser. Now picture having two plug-ins like `com.company.xml.parser.mobile` and `com.company.xml.parser.desktop` that provided two different implementations of the same XML parser, but for different environments.

Source editors

The Runtime and Dependencies pages are just visual representations of what is specified mainly in MANIFEST.MF file. For advanced users wishing to edit the source of this file by hand, PDE offers a source editor with code completion for the various headers found in a plug-in manifest definition.

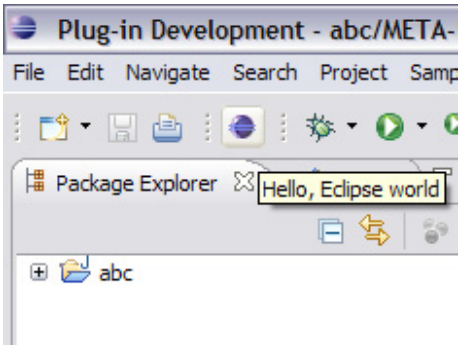
Figure 9. Source editing



Testing and debugging

The next step on our plug-in adventure involves testing and debugging. To test our plug-in, Eclipse and PDE have a notion of self-hosting. *Self-hosting* simply means that we're able to launch a new Eclipse with plug-ins we're currently working on in our workspace without actually having to export or deploy any plug-ins. To launch a new Eclipse, all you need to do is start another runtime workbench via the Overview page's **Testing** section (see the **Launch an Eclipse application** link within the Overview page). After a few seconds, you'll notice a new workbench pop-up with your plug-in's sample action visible in the toolbar.

Figure 10. Self-hosting in Eclipse



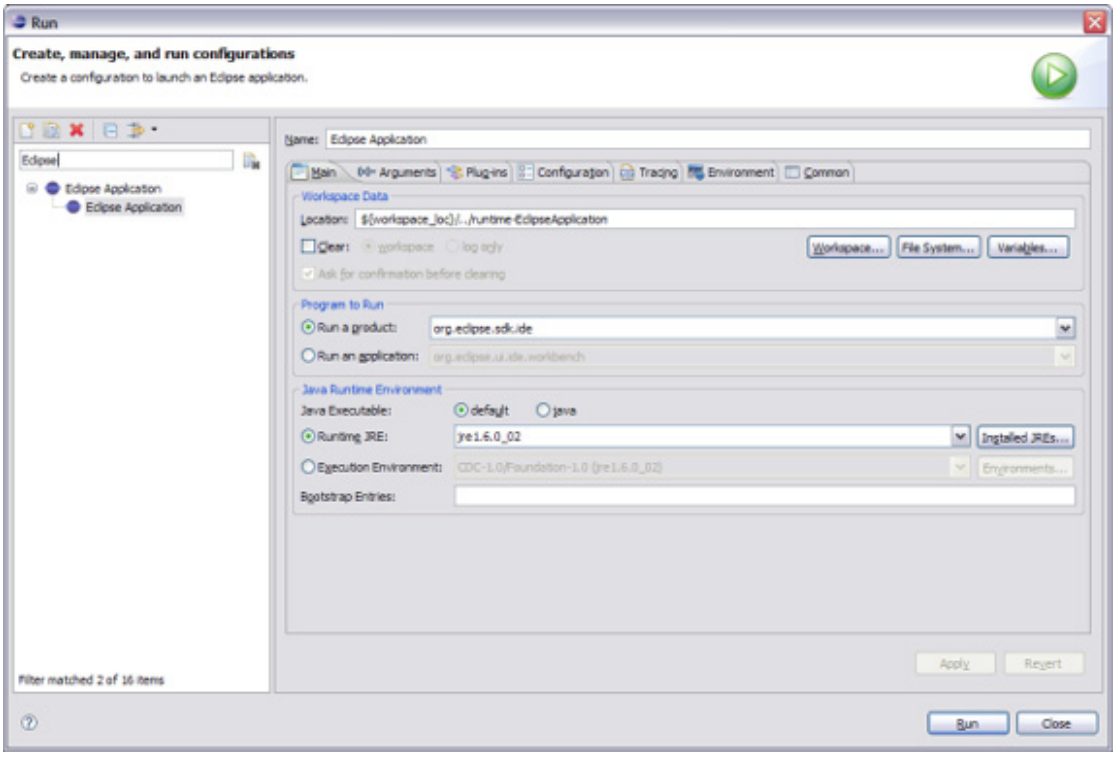
Test-driven plug-in development

If you're interested starting with nothing but tests before you actually develop your plug-in, it's possible to create a plug-in project that contains nothing but tests. There's a special launch configuration called Plug-in JUnit Test to run tests contained within plug-ins.

To launch your self-hosted plug-in in debug mode, simply click the **Launch an Eclipse application in debug mode** link in the **Testing** section of the **Overview** page. To actually debug your plug-in, you need to set relevant breakpoints. If you're new to debugging in Eclipse, I recommend checking out the developerWorks article "Debugging with the Eclipse Platform" to help get you started (see Resources).

To set more fine-grained options about launching or debugging your plug-in, go to the Launch Configurations dialog, which can be invoked using the **Run > Run Configurations...** menu option. The relevant launch configuration type for your plug-in is called an "Eclipse Application" since what we're launching is indeed an Eclipse application (see Figure 11). Within the launch configuration, you can set things like arguments and have control over which JRE is used to launch your application.

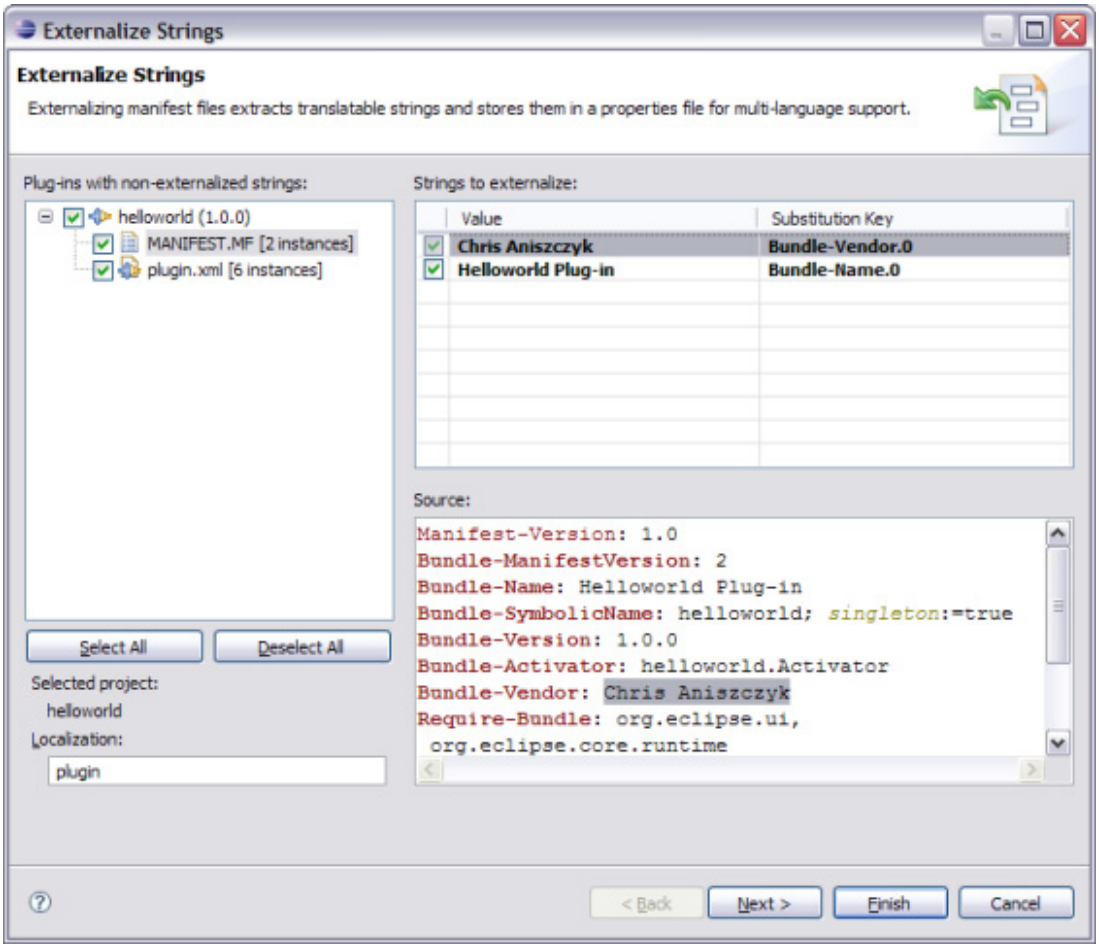
Figure 11. Launch configurations dialog



Externalizing strings

A common step in plug-in development is internationalization. Once you get to a point where your plug-in is useful and will be tested by multiple parties, you often get a request to make sure your plug-in is capable of running in a different language. Thankfully, the amount of work needed to externalize plug-in related strings is minimal. In PDE, there is a wizard that can be invoked via right-clicking on the **Overview** page of your plug-in and selecting the **Externalize Strings...** menu item. After the selection, a wizard will be presented that displays all the strings that are relevant to externalization.

Figure 12. The Externalize Strings wizard



In reality, this simply generates a plugin.properties file (that contains externalized strings) for your plug-in and adds the `Bundle-Localization` header to your MANIFEST.MF file. The `Bundle-Localization` header simply lets you define a name and optional location for your externalized strings. In this case, the `Bundle-Localization` is "plugin," which signifies that strings for various locales will be in files like plugin.properties (see Figure 13) plugin_fr.properties or plugin_de.properties. Anything after the underscore in the file name represents the locale.

Figure 13. The plugin.properties file

plugin.properties

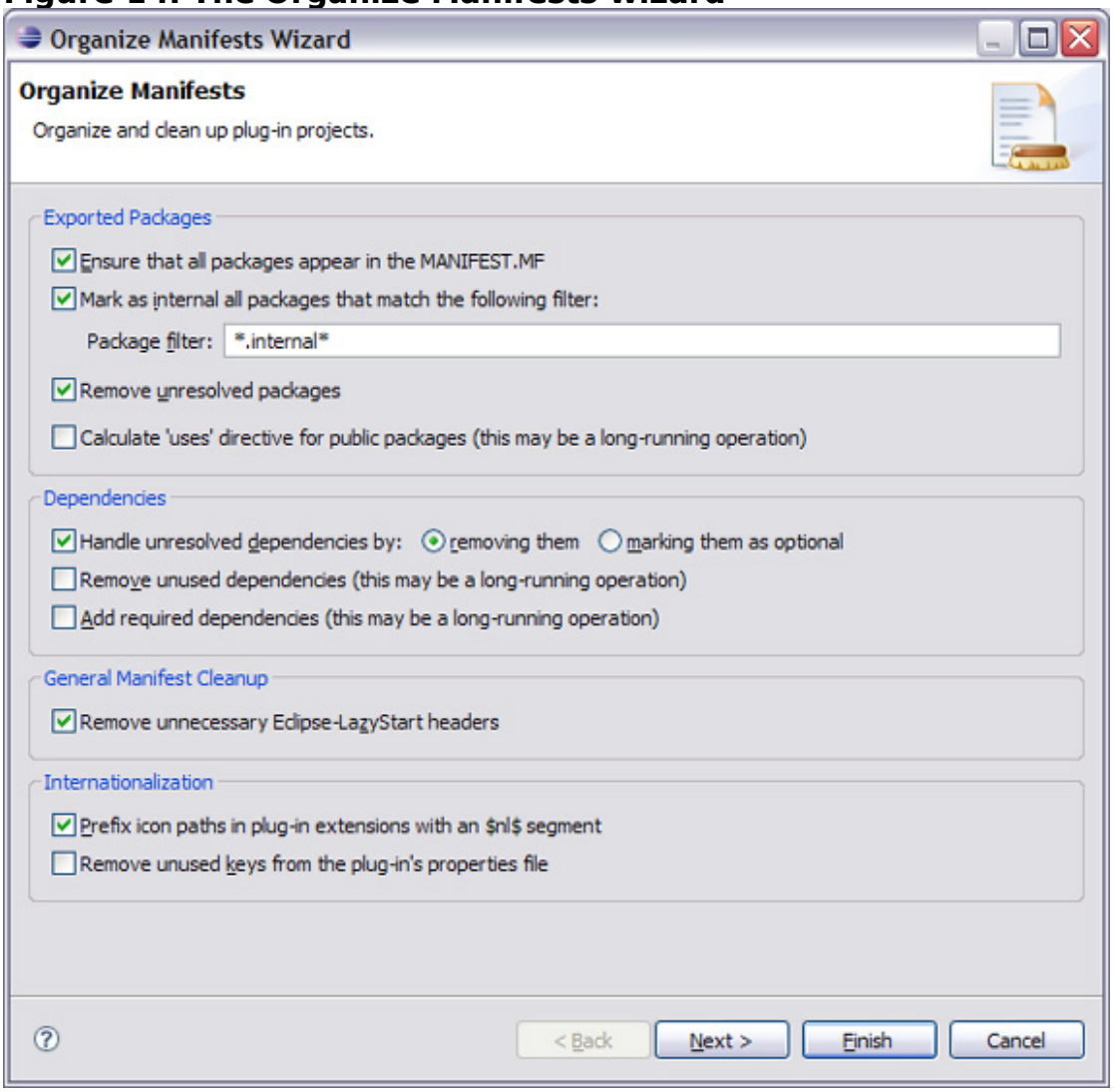
```
#Properties file for helloworld
Bundle-Vendor.0 = Chris Aniszczyk
Bundle-Name.0 = Helloworld Plug-in
perspective.name.0 = Perspective
view.name.0 = View
actionSet.label.0 = Sample Action Set
menu.label.0 = Sample &Menu
action.label.0 = &Sample Action
action.tooltip.0 = Hello, Eclipse world
```

That's it! That's all that's involved to get started in internationalizing our plug-in. For more information, I recommend reading these outdated yet still relevant articles from the Eclipse corner: "How to Internationalize your Eclipse Plug-In" and "How to Test Your Internationalized Eclipse Plug-In" (see Resources).

Organizing manifests

The next step on our journey is to organize our manifest files (i.e., MANIFEST.MF and plugin.xml) with some best practices. PDE provides a convenient wizard that can be invoked via the Overview page's **Exporting** section. Once the wizard is launched (see Figure 14), you're presented with a variety of options that can be tweaked. The defaults are very reasonable, but there are certain options like making sure there are no stray keys in the plugin.properties file, which is very useful.

Figure 14. The Organize Manifests wizard



Conclusion

On the whole, this article's mission was to give an introduction to the basics of plug-in development with some best practices sprinkled in. We accomplished that by creating a sample plug-in and going through a typical plug-in development workflow. Once the workflow is learned, it becomes much easier to develop plug-ins and even easier to maintain them with best practices like the Organize Manifests wizard. Part 2 focuses on using the tooling available for developing rich-client applications and finishing the rest of the plug-in development workflow presented in Figure 1.

Resources

Learn

- Learn more about OSGi Alliance.
- Learn more about Eclipse Plug-in Versioning at the Eclipse Foundation wiki.
- See the Eclipse EE guide at the Eclipse Foundation wiki.
- Learn more about the Eclipse Rich Client Platform.
- To learn more about the Eclipse SDK, see the Eclipse Foundation documentation titled "Platform Extension Points."
- Need help debugging in Eclipse? Read "Debugging with the Eclipse Platform."
- Good background information on internationalizing plug-ins can be found in the Eclipse Foundation's articles "How to Internationalize your Eclipse Plug-In" and "How to Test Your Internationalized Eclipse Plug-In ."
- Check out the "Recommended Eclipse reading list."
- Browse all the Eclipse content on developerWorks.
- New to Eclipse? Read the developerWorks article "Get started with Eclipse Platform" to learn its origin and architecture, and how to extend Eclipse with plug-ins.
- Expand your Eclipse skills by checking out IBM developerWorks' Eclipse project resources.
- To listen to interesting interviews and discussions for software developers, check out developerWorks podcasts.
- Stay current with developerWorks' Technical events and webcasts.
- Watch and learn about IBM and open source technologies and product functions with the no-cost developerWorks On demand demos.

- Check out upcoming conferences, trade shows, webcasts, and other Events around the world that are of interest to IBM open source developers.
- Visit the developerWorks Open source zone for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM's products.

Get products and technologies

- Download an Eclipse distribution containing the Plug-in Development Environment (PDE) from the Eclipse Foundation.
- The first stop to make when seeking plug-ins is the Eclipse Foundation's project list.
- The second stop to make when seeking plug-ins is Eclipse Plug-in Central (EPIC).
- Check out the latest Eclipse technology downloads at IBM alphaWorks.
- Download Eclipse Platform and other projects from the Eclipse Foundation.
- Download IBM product evaluation versions, and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.
- Innovate your next open source development project with IBM trial software, available for download or on DVD.

Discuss

- The Eclipse Platform newsgroups should be your first stop to discuss questions regarding Eclipse. (Selecting this will launch your default Usenet news reader application and open eclipse.platform.)
- The Eclipse newsgroups has many resources for people interested in using and extending Eclipse.
- Participate in developerWorks blogs and get involved in the developerWorks community.

About the author

Chris Aniszczyk is an Eclipse committer at IBM Lotus who works on OSGi-related development. His primary focus these days is improving Eclipse's Plug-in Development Environment (PDE) and spreading the Eclipse love inside of IBM's Lotus

organization. He is an open source enthusiast at heart, specializing in open source evangelism. He evangelizes about Eclipse in his blog, and he's honored to represent the Eclipse committers on the Eclipse Foundation's board of directors. He's always available to discuss open source and Eclipse over a frosty beverage.