Eclipse Corner Article

Your First Plug-in

Developing the Eclipse "Hello World" plug-in

Summary
The traditional Hello World program doesn't do that much, but it can be invaluable when exploring a new development environment. In this article we'll develop the Eclipse Hello World plug-in and show you how to integrate it with the Eclipse Workbench. After you read this article you should know how to use the Eclipse Java™ Development Tooling to create, run, and debug a simple plug-in that extends the Eclipse Platform. You'll see how to setup a project for your plug-in, edit the Java code, compile, and run or debug your plug-in in another launched copy of Eclipse. We'll be looking at plug-in manifest files, extensions, and extension points to see how plug-ins are described to Eclipse and how they are integrated with the Platform.

By Jim Amsden, OTI
Updated September 6, 2002 for Eclipse release 2.0 by **Andrew Irvine, OTI**
Last revised January 28, 2003

Editor's note: *Sept. 2003 - This article has been retired from service.* The newer article PDE Does Plug-ins shows how PDE greatly simplifies the task of creating a plug-in.

This article describes Eclipse release 2.0, which differs in minor ways from the previous Eclipse release. If you are still working with Eclipse release 1.0, you should consult the original version of this article.

The Eclipse Plug-in Development Environment (PDE) project provides a very nice environment for creating plug-ins and integrating them with the Eclipse Platform and/or other plug-ins. We're not proposing an alternative to PDE here. PDE is definitely the way to go. But sometimes its helpful to do things "by hand" in order to gain a more thorough understanding of how something works and how the pieces fit together. That's the approach we're going to take in this article. We'll develop a very simple plug-in implementing the Eclipse version of the classic Hello World sample. This will allow us to focus on the relationships between the various components of Eclipse and our plug-in without getting bogged down in the details of the example itself. We'll go raw as much as possible so we can see what's actually happening under the covers. Hopefully this will give you a better understanding of Eclipse, help you make better use of PDE, and give you some idea where to look when things don't go quite as planned.

The Problem

For the Eclipse Hello World program, let's start out with a simple design. We're going to add a button to the Workbench toolbar that when pressed, displays an information dialog containing the string "Hello World". Pressing OK dismisses the dialog. Nothing fancy, and certainly not anything that begins to exploit the full extensibility of Eclipse. But something we can easily use to get our feet wet. And it turns out that more complicated examples follow roughly the same pattern so we'll get a lot out of this very simple example.

So lets get started! I'll assume you've already downloaded the latest Eclipse code drop from eclipse.org, have it installed, and know how to start it up. If not, checkout the downloads page at the Eclipse site, and follow the instructions there.

Step 1: Getting ready to write Java code

Before we get started, we need to check some workbench preferences to make sure its configured properly for plug-in development. We'll be setting other properties later in the article, but for now, let's just set the default project layout and select the JRE we're going to use.

Setting the default project layout: Edit the Java preferences by selecting the `Window->Preferences` menu item. Expand the `Java` list item. Click on the `New Project item` and check the `Folders` radio button. Ensure "`Source folder name`" is set to 'src' and that "`Output folder name`" is set to 'bin'. If you do not alter this setting, all of the class files and source files will be placed in directories based at the root of the project. This is fine for small scale development, but is not the structure you want when constructing bigger programs.

Note: The Hello World plug-in assumes that the project layout has been altered to use 'src' and 'bin' folders. If you do not do this, you will get runtime errors when testing your plug-in.

Step 2: Creating the plug-in Project

Now we are ready to create a Java project for your plug-in. Go to the Navigator view (Resource Perspective), and select `File->New->Project`. The project creation wizard has a list of categories each with a list of projects. Select the `Java` category in the left hand list and `Java project` in the right. Creating a Java project vs. some other project creates a project which understands Java resources and how to manage changes in them. In the Java project wizard, enter a project name for your plug-in. Generally it is a good idea to use the plug-in name or id as the project name to make it easier to deploy your plug-in, and to make sure your plug-in name doesn't collide or get confused with some other plug-in. In our example, we'll use the plug-in id as the project name. Enter *org.eclipse.examples.helloworld*.

Step 3: Integrating with Eclipse

Before we can start writing our code, we need to determine how we're going to integrate with Eclipse. That's because all extensions to Eclipse are done through plug-ins, and plug-ins integrate with each other through extensions on extension points. Eclipse plug-ins typically provide extensions to the platform that support some additional capability or semantics. What is needed is a way for plug-ins to allow other plug-ins to change their behavior in a controlled manner.

Eclipse provides an extensibility mechanism that is scalable, avoids name collisions, doesn't require compilation of the whole product as a unit, and supports multiple versions of the same component at the same time. Eclipse does this by introducing the notion of a plug-in which encapsulates functional extensions to the Platform. Each plug-in has a name, id, provider name, version, a list of other required plug-ins, and a specification for its runtime. A plug-in can also have any number of extension points that provide a portal into which other plug-ins can add their functionality. This is how Eclipse enables other plug-ins to handle the variability supported by your plug-in. In order to integrate with other plug-ins, a plug-in provides extensions on these extension points (perhaps even its own extension points in order to provide some default behavior).

A plug-in is described in an XML file called the plug-in manifest file. This file is always called plugin.xml, and is always contained in the plug-in sub-directory. The Eclipse Platform reads these manifest files and uses the information to populate and/or update a registry of information that is used to configure the whole platform.

In our case, we want to use a button on the Workbench toolbar, so the extension point we will use is `org.eclipse.ui.actionSets`. An action set is a strategy for the addition and removal of menu and toolbar items. This strategy is executed if the user explicitly adds the action set to the workbench. A user can add an action set to the workbench by invoking `Window->Customize Perspective->Other...` to

display the available actions. To activate an action set, you would navigate the action set categories, select the desired action set, and press OK. A perspective can also add an action set to the initial page layout by invoking `IPageLayout.addActionSet(id)`. For details on the actionSets extension point, open the help reference material by invoking `Help->Help Contents...` select `Platform Plug-in Developer Guide->Reference->Extension Points Reference->Workbench`. Select this hyper-link and you will be presented with the Platform Extension Points. You are interested in `org.eclipse.ui.actionSets`.

Step 4: Creating the plug-in Manifest File

Now that we know what we've got to do, lets tell Eclipse about our plug-in by creating the plug-in manifest file. Use `File->New->Other...` and select `Simple->File` to create a file called plugin.xml in the org.eclipse.examples.helloworld project.

Edit the plugin.xml file so that it looks like this:

```
<?xml version="1.0"?>
<plugin
        name="Eclipse Hello World Example"
        id="org.eclipse.examples.helloworld"
        version="0.0.0"
        provider-name="OTI">

    <requires>
        <import plugin="org.eclipse.core.resources"/>
        <import plugin="org.eclipse.ui"/>
    </requires>

    <runtime>
        <library name="helloworld.jar"/>
    </runtime>

    <extension point = "org.eclipse.ui.actionSets">
        <actionSet
            id="org.eclipse.examples.helloworld.HelloWorldActionSet"
            label="Hello World"
            visible="true"
            description="The action set for the Eclipse Hello World example">
            <menu
                id="org.eclipse.examples.helloworld.HelloWorldMenu"
                label="Samples">
                <separator name="samples"/>
            </menu>
            <action id="org.eclipse.examples.helloworld.actions.HelloWorldAction"
                menubarPath="org.eclipse.examples.helloworld.HelloWorldMenu/samples
                toolbarPath="Normal"
                label="Hello World"
                tooltip="Press to see a message"
                icon="icons/helloworld.gif"
                class="org.eclipse.examples.helloworld.HelloWorldAction"/>
        </actionSet>
    </extension>
</plugin>
```

You can use Eclipse to edit this file using the default text editor. To do so, select `Window->Preferences`, expand the Workbench entry, and select `File Associations`. Add resource extension xml, and add the "Text Editor" to its associations. Now when you attempt to open an editor on a .xml file, you'll do so with the default text editor. This will do just fine for our simple example. PDE makes this a LOT easier for complex plug-ins.

Now lets take a look at the manifest file and see exactly what's there. First we see that we're declaring a `plugin`. That's always the root element of a plug-in manifest file. The attributes of the `plugin` element give the plug-in name, id, version, and provider name. The id is the really interesting attribute. It specifies the identifier the platform uses to reference this plug-in. Since this name has to be unique for all installed plug-ins, we use Java package naming conventions to create a unique id. Anything will do, but this is a reasonable convention that should be followed. You'll see how these ids are used a little later when we have to reference another plug-in's extension point.

The `requires` element is where you specify all the other plug-ins your plug-in depends on. In our case, we're using the actionSets extension of plug-in `org.eclipse.ui` , so we specify that plug-in here.

The plug-in `runtime` element is how you tell the platform where to find the classes in your plug-in. Essentially, the `requires` and the `runtime` elements go together to specify the "classpath" for the plug-in. This approach allows each plug-in to have its own classpath independent of any other plug-in. Further, each plug-in has its own classloader which is used to load all classes defined by that plug-in (i.e., the classes found in its library declarations).

Now we're finally getting to the real integration! The extension element (in our plugin.xml file) describes an extension on the org.eclipse.ui.actionSets extension point. In our example, we're extending the actionSets extension-point of plug-in `org.eclipse.ui`. The id of our extension, how the platform refers to it, is `org.eclipse.examples.helloworld.HelloWorldActionSet` while the display name is "Hello World". If you look at the documentation referenced above for the actionSets extension-point, you'll see that it can have a number of action elements which describe the actions in the set. Each action has a class attribute that specifies the class that implements the required interface, `org.eclipse.ui.IWorkbenchWindowActionDelegate`. This is what we have to implement for our extension. You'll also need to provide the helloworld.gif file to provide an icon for your action. Anything will due for testing. Create the icons folder in your plug-in folder. All plug-in file and folder pathnames are relative to the plug-in folder.

Don't get confused between an extension-point element and an extension element with a point attribute. The extension-point element defines a hook into the platform while the extension specifies and instance of using the hook. The point attribute is the id of the extension-point you're extending.

Step 5: Setting up the plug-in Project

Switch to the Java perspective if you're not already there with `Window->Open Perspective->Other` and select the Java perspective (or if Java is available on the menu, you may select it directly). Then select the packages explorer tab to view the packages in your project. You should only see the packages you've specified in your Java build path so far, probably just the JRE_LIB entry for your rt.jar. Since your going to be using the `IWorkbenchWindowActionDelegate` interface, and other parts of the platform to develop our example, you need to include the referenced plug-in runtimes in your Java build path.

From the plug-in manifest file above, we can see that we require plug-in `org.eclipse.ui`. Referencing the plug-in in the requires element of our plug-in allows the platform to find classes we reference at runtime, but it doesn't help the compiler at development time. We need to get this plug-in's runtime in your project's build path so we can compile against imported classes and interfaces. To do this, select the project and view its properties; right click on the project. Select the `Java Build Path` entry, and click on the `Libraries` tab. Then click on the `Add External JARs...` button and browse to <your eclipse install directory>/eclipse /plugins/org.eclipse.ui_2.0.0, the directory for the 2.0 version of the Eclipse UI plug-in.

See why we like to have the plug-in ids correspond to the plug-in directory names? It makes it a lot easier to find things. In directory org.eclipse.ui_2.0.0 you'll see the `workbench.jar` file. Select this file to add it to your build path. The Eclipse UI plug-in depends on the Eclipse SWT plug-in, so we will need to add that one too: `org.eclipse.swt.win32_2.0.0->ws->win32` contains `swt.jar`, assuming you are running on Windows®. There are similar directories for Linux users running either motif or gtk. The platform automatically provides a special runtime support plug-in: `org.eclipse.core.runtime_2.0.0` - you should also add the `runtime.jar` from this plug-in to your build path. If your plug-in depends on any other plug-ins, you'll need to add their jar files too. PDE takes care of all of these things for you, but we'll save that for another article.

Note: Eclipse release 2.1 re-architects the UI plug-in. The `workbench.jar` is located in the directory `org.eclipse.ui.workbench_2.1.0`. In addition the JFace plug-in is required. The `jface.jar` file can be found in the `org.eclipse.jface_2.1.0`. directory. The remaining plug-ins can be found in the corresponding release 2.1 directories.

Now we're ready to write the code and compile it against workbench.jar and swt.jar. The workbench has the JDK set, the project has all the jar files in its build path we need, and the plug-in manifest file tells us what classes we have to start with.

Step 6: Implementing the IWorkbenchWindowActionDelegate Interface

In this example, our implementation of interface `IWorkbenchWindowActionDelegate` is `org.eclipse.examples.helloworld.HelloWorldAction`. Go take a look at the JavaDoc for the `IWorkbenchWindowActionDelegate` interface to see what methods we need to implement. The JavaDoc is available from the `Help->Help Contents` menu item. Select `Platform Plug-in Developer Guide` expand to `Reference->API Reference->Workbench->org.eclipse.ui`. In addition to using standard Java package naming conventions, you should consider including the plug-in id as part of your package name. While not required, it will help you keep your code organized.

To create the implementation class, we'll first create a package for our plug-in. Select the src folder of the `org.eclipse.examples.helloworld` project in the packages view, right click, and select `New->Package`. Create a package named *org.eclipse.examples.helloworld*.

Next, select the newly created package, right click, and select `New->Class`. If you prefer using the tool bar, there are buttons to create a package and a class, among other Java-specific buttons. In the dialog that comes up, verify the package name is `org.eclipse.examples.helloworld` (it won't be if you didn't select it first), and fill in the class name *HelloWorldAction* as specified in the class attribute of the action element in our extension (see the manifest file plugin.xml). The superclass `java.lang.Object` is fine, but add the interface `IWorkbenchWindowActionDelegate`. Just click on the `Add...` button on the right and start typing *IWorkbenchWindowActionDelegate*. You'll see the selection list zero in as you type. Select `IWorkbenchWindowActionDelegate` and click OK. You'll see `org.eclipse.ui.IWorkbenchWindowActionDelegate` in the list of interfaces your class is going to implement. For convenience, click on the `Inherited abstract methods` check box so the new class wizard will create stubs for all the interface methods you need to implement. This will save you a lot of typing. If you don't see the class in the selection list that you expect, this is a good indication that your Java build path isn't including everything it needs. Check your build path in the project preferences and try again.

Now we have our `IWorkbenchWindowActionDelegate` implementation class. From the interface we have four methods to implement, `init` and `dispose` from `IWorkbenchWindowActionDelegate`, and `run` and `selectionChanged` from `IActionDelegate`. Edit the class, and provide the method implementations given below.

```
package org.eclipse.examples.helloworld;


import org.eclipse.ui.IWorkbenchWindow;
import org.eclipse.ui.IWorkbenchWindowActionDelegate;
import org.eclipse.jface.action.IAction;
import org.eclipse.jface.dialogs.MessageDialog;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.swt.widgets.Shell;


/** HelloWorldAction is a simple example of using an
 * action set to extend the Eclipse Workbench with a menu
 * and toolbar action that prints the "Hello World" message.
 */
```

```
public class HelloWorldAction implements IWorkbenchWindowActionDelegate {
        IWorkbenchWindow activeWindow = null;

        /** Run the action. Display the Hello World message
         */
        public void run(IAction proxyAction) {
                // proxyAction has UI information from manifest file (ignored)
                Shell shell = activeWindow.getShell();
                MessageDialog.openInformation(shell, "Hello World", "Hello World!");
        }

        // IActionDelegate method
        public void selectionChanged(IAction proxyAction, ISelection selection) {
                // do nothing, action is not dependent on the selection
        }

        // IWorkbenchWindowActionDelegate method
        public void init(IWorkbenchWindow window) {
                activeWindow = window;
        }

        // IWorkbenchWindowActionDelegate method
        public void dispose() {
                //  nothing to do
        }
}
```

The code isn't too complicated, so I won't cover it in any more detail. Other articles will be addressing the specific Eclipse features we're using, but we want to focus on plug-ins and the end-to-end integration story rather than the code specifics in this article.

Step 7: Testing your plug-in

We're ready to test our plug-in! The best way to do this is to use two Eclipse workbenches, one for plug-in development, a second for testing and debug. Let's call them the development workbench and the testing workbench to avoid confusion. We don't want to use the same instance the Eclipse workbench for both development and test for a number of reasons:

- You have to restart the workbench every time there's a change in your plug-in so it gets reloaded properly. If you use your development instance for this, you'll be restarting it all the time.
- If there's a problem in your plug-in, it could hang the development workbench making it difficult to locate and fix the problem.
- You can't debug a workbench with the same workbench that's running the debugger. That's because when the debugger hits a breakpoint, the whole workbench is stopped.

Prior to launching a testing workbench you must tell Eclipse where the required plugins are located. Edit the Plug-in Development preferences by selecting the `Window->Preferences` menu item. Expand the `Plug-In Development` list item. Select `Target Platform` and click on the `Not in Workspace` button on the right hand side. You will note the selected items in the list to the left of this button are all now checked. Finally click on the `OK` button to accept these changes.

To start Eclipse with Eclipse simply choose `Run->Run As->Run-time Workbench`. Eclipse automatically starts executing in a second workbench, the testing workbench. Be patient this operation may appear to take a long time; when a second workbench opens the operation is complete. To view the location of this second workbench browse the launch configuration via `Run->Run...`. Alternatively both these functions may be achieved via the Workbench toolbar Run icon.

Now we can test the Hello World actions. You'll see the Hello World icon that was specified in the icon attribute of the action element in our plug-in manifest file on the Workbench toolbar (if you don't provide an icon, it will

appear as a gray or red square). Move the cursor over it and you'll see the hover help you entered. Click and you see the message. Not quite as simple as `printf("Hello World");` but it looks a lot nicer. To turn off the Hello World action set (and remove the associated button), select `Window->Customize Perspective...` and expand the `Other` category. Now you can uncheck the Hello World item.

When you turn off the Hello World action set, you'll see that the icon is removed from the toolbar and you can no longer invoke the plug-in action.

Debugging your plug-in

Debugging your plug-in is just as easy. Just open your Java source and put a breakpoint where you would like to start debugging. Then debug your test workbench by selecting `Run->Debug As->Run-time Workbench` . The workbench will open a debug perspective page, and you'll see the Eclipse process running. Eclipse will come up, and be ready for you to invoke your plug-in. Again, enable the Hello World actions and click on the Hello World button. The test workbench will halt at your breakpoint and you'll see your code, breakpoint, and variables in the debug perspective in your development workbench. Go ahead and experiment with the debugger. Then press resume when ready, and up comes the message. Remember you can only have one active Run-time workbench, so you will need to close your testing workbench between trials.

Conclusion

We've seen the complete development of a plug-in extension to Eclipse from design to debug and test. We've done all the development, testing, and debugging using Eclipse itself and the Java Development Tooling. The Hello World example doesn't do much, and the complexity per function is pretty high. But more complex plug-ins follow the same pattern, and Eclipse provides a lot more than we've seen with this simple example. You should now have a pretty good idea how all the pieces fit together and what it means to develop and integrate a plug-in. Now its time to do something real. You'll want to use PDE for that, so look for that article coming soon.