CHAPTER 6

Java Project Configuration

Eclipse includes features such as Content Assist and code templates that enhance rapid development and others that accelerate your navigation and learning of unfamiliar code. Automatic compilation and building of complex projects provides additional acceleration by giving immediate feedback on code changes and project status. All of these features depend on correct configuration of the projects in your workspace.

We continue development of the product catalog and order processing application by configuring the project dependencies required to build and run that code. Part of the configuration consists of including a JAR file for the Apache log4j logging utility and a shared library of components from the Apache Axis Web Services toolkit.

This chapter does not describe configuration and use of a source code version control repository. Eclipse has excellent support for team repositories such as CVS, which is described in Chapter 13, "Team Ownership with CVS." If you are joining an existing development team, you can skip directly to that chapter after reading this one.

In this chapter, we'll see how to

- Configure your project's source and output folders
- Configure dependencies between Java projects
- Add libraries to your build path
- Create named user libraries composed of related JAR files
- Override workspace compiler preferences with project-specific settings

6.1 Java Build Path

Up to this point, our product catalog project had no dependencies other than the Java runtime libraries. It is now time to expand our development to include the other projects set up in Chapter 3, "Managing Your Projects." External dependencies are also added into the mix, such as Apache log4j and Axis Web Services toolkit. Correct configuration is essential for error-free compilation, execution, and full use of Eclipse features such as Content Assist.

Edit the configuration properties for a Java project by selecting the project and choosing **Project > Properties** from the main workbench menu bar. The dialog shown in Figure 6-1 is opened.

The same property editor dialog may be opened using another approach. Most Eclipse resources have property pages that describe or allow you to set that resource's properties. This is true for any resource, not only for projects. The last menu item in a resource's context menu is always named **Properties,** and it displays a shared dialog with one or more pages that are assigned based on the resource type. Projects with a Java nature have property pages as shown in Figure 6-1. Open this dialog by right-clicking on the `com.eclipsedistilled.orders` project in your workbench **Package Explorer** view and selecting **Properties**.

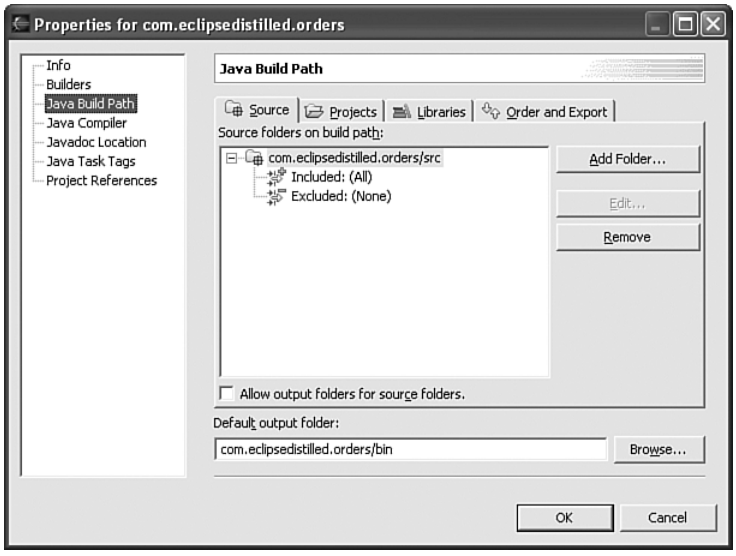

Figure 6-1 Configuring source folders in your Java project build path.

Each Java project has its own build path that specifies all dependencies required to compile the project. Those dependencies may come from other Java projects in the workspace, from Java archive `.jar` files, or from folders containing `.class` files.

The **Java Build Path** properties page contains four tabs:

- **Source.** The source and output folders. If you initially create a project without separate source and output folders, you can change it here. Multiple source folders can be used in one project; e.g., to separate application source code from unit tests.
- **Projects.** Check-off other projects in the workspace whose output folders should be added to this build path.
- **Libraries.** Location of other archive files required by this project.
- **Order and Export.** Order in which projects and libraries appear in the build path and the default runtime classpath; e.g., use classes from a workspace project before using the same classes from an archive library.

6.1.1 Source and Output Folders

Each Java project is configured with a *builder* that automatically compiles every `.java` file from the source folders and saves the `.class` files into an output folder. Your source folder must contain subfolders for the complete package hierarchy as used by your Java source files. As described in Chapter 2, "Hello Eclipse," you can create these folder hierarchies easily using the **New Java Package** wizard.

All non-Java files from the source folder are copied unchanged to the corresponding folder hierarchy in the output folder. These non-Java files are usually properties files and resource files. This sometimes creates confusion when Eclipse users store other configuration or documentation files within their source folder tree and then are surprised to see them copied into the output folder. These other non-source files should be saved in regular project folders that are not configured as source folders in the configuration. You can create regular folders outside the source tree by right-clicking on a project and selecting **New > Folder** instead of **New > Package**.

Figure 6-1 shows the source folder tab in the Java project properties. This project was created with separate source and output folders named `src` and `bin`, respectively. This setup will suffice for most new projects created within Eclipse, but you can change that configuration here.

If you want to keep your unit test code within the same project as the application code, then it's a good idea to create a separate source folder, named `test`

for example. Click the **Add Folder…** button on the source configuration tab and then click **Create New Folder…** on the resulting dialog. If you create a test folder without adding it as a configured source folder, then Java source files within it will not be compiled.

Although it's fairly common for developers to keep unit test code in the same project as the code being tested, it is preferable to create a separate project for JUnit test cases because they often need a different **Java Build Path**. This is especially true if you are building applications using Java runtime libraries other than the J2SE libraries. For example, if you're building a J2ME application that depends on the Mobile Information Device Profile (MIDP), you'll have to put your JUnit test cases in a separate project because JUnit requires J2ME's Foundation as a minimum class library. It's also common to use additional JUnit framework libraries when testing Web and database applications.

The most common reason for using multiple source folders is to accommodate preexisting source code that was created outside of Eclipse. Developers can be very creative when organizing their projects! A test folder is sometime embedded within the application source folder, or several logically separate source trees may be included in the same folder.

Eclipse provides other ways to split these sources into logically separate projects or source folders without changing the original structure, which might be required by other tools or Ant build files. You can add inclusion and exclusion filters on a source folder to explicitly select the files that are or are not used to build this project. For example, if documentation files are stored within the source, you could exclude `**/*.html` files so that they are not copied into the output folder.

There are many other possibilities for configuring preexisting code within an Eclipse project. Search for "project configuration tutorial" in the Eclipse help documentation where other detailed scenarios and suggestions are provided.

6.1.2 Project Dependencies

When we created the projects for our order management application in Chapter 3, the project dependencies were not yet specified in the configuration. These dependencies are shown as a UML package diagram in Figure 6-2. These package names are shortened versions of the fully qualified project names used in our workspace. They represent the import dependencies between top-level packages in our application, but not necessarily the dependencies of all sub-packages and external utility libraries.

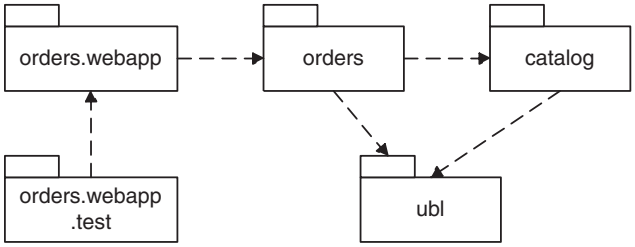

Figure 6-2 Order processing application package dependencies.

Click on the **Projects** tab in the build path configuration, as shown in Figure 6-3. All of the projects in your current workspace are listed except for the project we are now configuring, which is `com.eclipsedistilled.orders`.

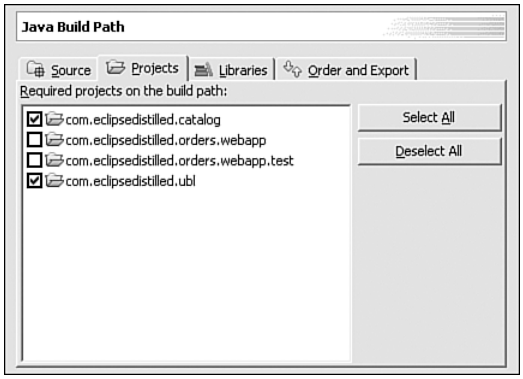

Figure 6-3 Configuring project dependencies for `com.eclipsedistilled.orders`.

Referring to the package diagram, we see that `orders` depends on `catalog` and `ubl`. Configure the dependencies in your Eclipse project by selecting the checkboxes for those two projects.

The end result is that the output folders from these other two projects are included in the build path of the current project, and their classes are available while compiling classes for `com.eclipsedistilled.orders`. Configuring these project references also causes their classes to be included in Quick Assist completion lists, so typing "`cat`" and then **Ctrl+Space** will now include the `Catalog` and `CatalogItem` classes in the pick list while writing the `Order` class.

6.1.3 Project Libraries

The **Libraries** tab of the **Java Build Path** dialog allows you to add other libraries into a project's classpath. A library is a JAR or ZIP archive containing Java class files or a project folder containing class files. An archive file may be in a project within your current workspace or elsewhere on your file system.

The library configuration for `com.eclipsedistilled.orders` is shown in Figure 6-4. The JRE System Library is included automatically in every Java project; it contains the standard Java APIs. We'll review two approaches for adding individual JAR libraries to this project and then create a named *user library* that bundles a collection of related JARs.

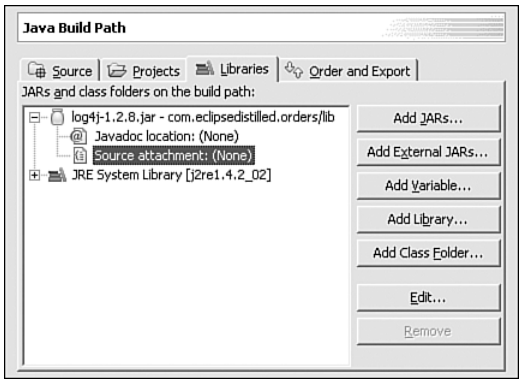

Figure 6-4 Configuring libraries for `com.eclipsedistilled.orders`.

The **Libraries** tab contains five buttons for specifying the location of required library files:

- **Add JARs.** Select archive files located within any project from the current workspace; projects are not required to be Java projects.
- **Add External JARs.** Select archive files anywhere on your file system external to the current workspace.
- **Add Variable.** Use a Java classpath variable as the base path for external archives.
- **Add Library.** Select from a list of user libraries that define collections of related archive files.
- **Add Class Folder.** Select any folder from a project in the current workspace that contains Java `.class` files.

An important consideration when planning your project configuration is portability between different developer workstations in a team environment, which might include portability across operating systems such as Windows, Linux, and Macintosh. The first library option, **Add JARs**, is usually the most portable but not always possible or desirable when using libraries from other vendor products. Using external libraries with absolute file paths is the least portable. We'll use the first approach to add the Apache log4j library to our project (see References).

It is common practice to create a subfolder named `lib` within a Java project that contains other JAR files required to build the project. Right-click on your project and select **New > Folder** to create this folder. Download the `log4j.jar` binary file and copy it into your project `lib` folder (the file name may include a version number).

If you copy the file into your project using the operating system command line or file explorer, then your Eclipse **Navigator** view or **Package Explorer** view is updated automatically if you have automatic refresh enabled; otherwise, you must manually refresh the `lib` folder (see Chapter 3).

Tip: If you are using Eclipse on Windows, you can copy/paste or drag-and-drop files between the Windows file explorer and your Eclipse workbench folders in the same way you would between folders in the Windows Explorer. You can also cut/copy/drag between two Eclipse folders within the workbench on any operating system.

Now click the **Add JARs…** button, where you'll see a list of all projects in your workspace. Expand the project and `lib` folder containing `log4j.jar` and add it to this project's build path. It should appear as in Figure 6-4.

If you expand the `log4j.jar` entry in the configuration dialog, there are two optional entries about this library.

- **Source attachment.** The folder or JAR file containing Java source code for classes in this library.
- **Javadoc location.** The URL or external location containing a folder or ZIP file of Javadoc HTML for classes in this library.

This source attachment location is the same kind of entry we configured in Chapter 5, "Rapid Development," to enable Javadoc hover and Content Assist for the Java runtime library. If you have source code for other libraries, such as log4j, then edit this library entry to get the same benefits when working with its classes.

You can open a Web browser with the full Javadoc HTML documentation for a library's entries by pressing **Shift+F2** while the cursor is positioned on a class or method name in the editor. However, for this to work, you must configure the URL or directory where the HTML files are located.

Shortcut: Shift+F2: Open the full Javadoc HTML for a Java class, method, or field at the current cursor position. This command is also accessible via the menu **Navigate > Open External Javadoc**.

You can also configure the Javadoc location for the Java runtime libraries by expanding the JRE System Library in this same configuration dialog. Expand the `rt.jar` archive and edit the Javadoc location. This location is preset with the value `http://java.sun.com/j2se/1.4.2/docs/api` when you install Eclipse (with version number appropriate to the JVM you used during installation). However, this will work only while you are connected to the Internet. You can change this URL to a local file path if you want to enable this feature while working offline.

This approach to project configuration is the easiest way to ensure that library locations are portable between different developer workstations and operating systems. All libraries are stored within the project folders, and locations (except for Javadoc files) are relative to the workspace home. If you zip your workspace and send it to another developer, he or she can simply unzip and open it in his or her Eclipse workbench. All project building and Content Assist will work without change.

Another way to configure library locations that also has benefits of machine and platform portability is to use *classpath variables*. Click the **Add Variable…** button in the **Java Build Path** dialog, which presents a new dialog, as shown in Figure 6-5.

In this example we'll add the standard J2EE Servlet API library to our `orders.webapp` project; a similar technique could be used for adding the log4j library. In Chapter 3, we reviewed the benefits of linked resource locations for gaining developer and platform portability of project files located outside of the workspace. Classpath variables are very similar to linked resource locations but require separate definitions.

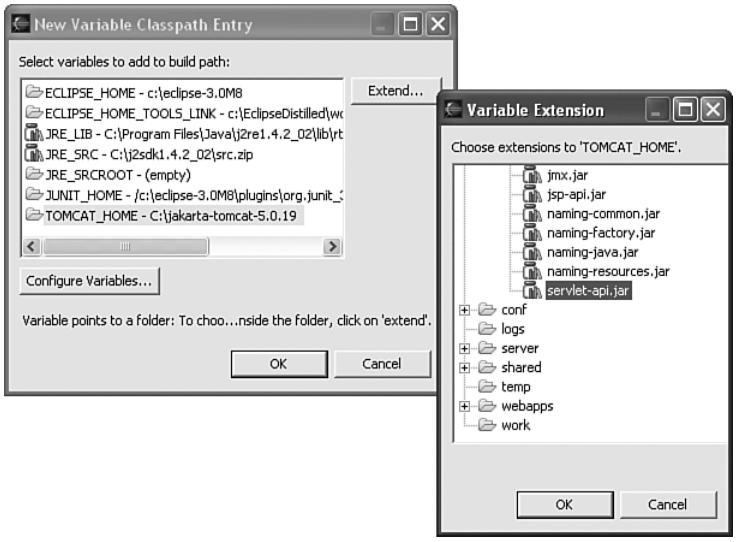

Figure 6-5 Extending a Java classpath variable in project build path.

Follow these steps to add a TOMCAT_HOME library location:

1. Click the **Configure Variables…** button in this dialog, where you can create a new variable or change a variable location value.
2. Add a new variable named TOMCAT_HOME with a location pointing to the root of your Tomcat application server installation, e.g., `C:/jakarta-tomcat-5.0.19`, and then click OK.
3. Back in the dialog shown in Figure 6-5, select this variable and click the **Extend…** button, which opens the second dialog also shown in the figure.
4. Expand the `common` and `lib` folders and then select `servlet-api.jar`. Click **OK**.

The Servlet library is now part of your project configuration. You can easily share this workspace or project with other developers who use a different path or different version of the Tomcat server. They only need to create a TOMCAT_HOME classpath variable with their location. All other aspects of this project configuration remain unchanged.

You can review and update any of your classpath variables in the Eclipse preferences category **Java > Build Path > Classpath Variables**.

6.1.4 Order and Export

After specifying project dependencies and library references, you may need to configure two other aspects of the build path. These are shown on the **Order and Export** tab of the **Java Build Path** properties in Figure 6-6.

- Change the order of class path entries in situations where the same class name exists in more than one project or JAR location.
- Choose which project or JAR entries are contributed to other projects that have this project as one of their dependencies.

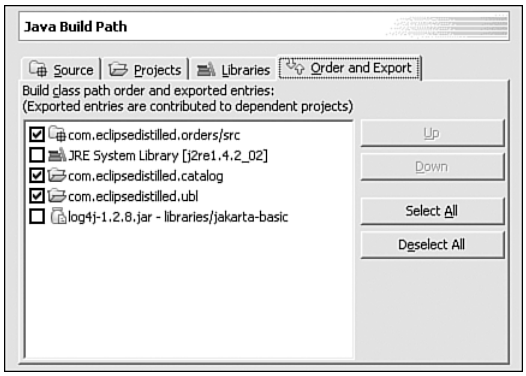

Figure 6-6 Configuring order and export of libraries for `com.eclipsedistilled.orders`.

The same class name may exist in more than one class path entry when you have a project in your workspace that includes an updated version of some classes in one of the referenced libraries. If you want to compile and run an application using the updated version, then you must place the project above the older library version in the build path order. We assume that the library JAR file contains other classes that you need; otherwise, just remove the old library from this project's build path.

A project's build path can also include the library entries defined within one or more of its required projects. For this to work, you must explicitly export a project's libraries that are shared with its clients. However, you need to be careful when exporting a project's libraries because doing so means that those libraries can be logically thought of as part of this project. Another approach is for the client project to import the library itself, in addition to importing the required project.

Refer to the package diagram in Figure 6-2 that shows dependencies between the projects in our order processing application. The `orders.webapp` project depends on only the `orders` project, but it will likely include references to classes from `catalog` and `ubl`. When configuring the build path for `orders.webapp`, we can include dependencies to these other two projects, or we can export these two projects from the `orders` project. We take the latter approach and select the export checkboxes for `catalog` and `ubl` when configuring the `orders` project in Figure 6-6.

The **Java Build Path** order also determines the order that source folders and libraries are displayed within a project in the **Package Explorer** view. In most situations, the order of libraries does not affect the way a project builds or runs, so you can reorder the source folders and libraries to appear in the **Package Explorer** view in a way that makes it easy to find references, such as ordering libraries alphabetically.

6.2 Create Shared User Libraries

When working with third-party commercial or open source libraries, or with standard APIs such as J2EE, it's common to require several JAR archives in combination. If these are used in only one project, then you can configure the build path as described in the previous section. However, you may need to include the libraries several times in a modular multi-project structure. It would be easier to define the combined library as a single entry.

This kind of configuration is called a *user library* in Eclipse. The JAR files contained within a user library are identified by an absolute file path external to the Eclipse workspace. It's helpful to have a consistent location for these libraries on your local or network file system. A library's files also might be located within a vendor product installation directory. We'll use the following file structure:

```
/eclipse-contrib/
    libraries/
        axis-1.2beta/
        j2ee/
        jarkata-basic/
        jakarta-j2ee/
```

Download the Apache Axis distribution (see References) and unzip its JAR files into the `axis-1.2beta` folder (or a similar folder name based on a newer version). Standard vendor-independent interface libraries are available for J2EE specifications such as Servlets, EJB, JNDI, JavaMail, and others; place these JARs into the `j2ee` folder. Many other useful utilities are available from the Apache

Jakarta project, including the log4j library. Place these JAR files in the `jakarta-basic` and `jakarta-j2ee` folders.

We could use a classpath variable to include J2EE library files from Tomcat or JBoss installations, just as we did with the Servlet library earlier in this chapter. However, because we may deploy to several different application servers, and because our project code is written to the standard J2EE APIs, we gain more flexibility by creating a vendor-independent J2EE user library. A user library allows us to add a single **Java Build Path** entry that includes all JAR files required for our J2EE development.

Open the Eclipse preferences page for user library configuration located in the category **Java > Build Path > User Libraries**. The configuration for the Apache Axis library is shown in Figure 6-7. Press the **Add JARs...** button, browse to the Axis library folder, and select the archive files.

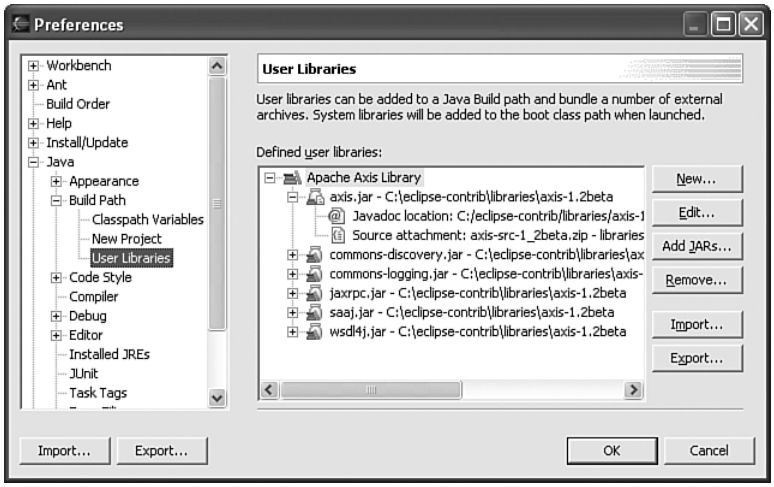

Figure 6-7 Create a new user library for the Apache Axis Web Services toolkit.

Download source and Javadoc ZIP files for each user library, if they are available. Doing so enables maximum benefit from Content Assist and Javadoc display when using these libraries in Eclipse. For convenience, place these files in the same directory as the binary JAR files. While adding each JAR file to the user library, also edit the associated Javadoc location and source attachment parameters. These values are shown for the `axis.jar` file in Figure 6-7.

The user library preferences page includes buttons for importing and exporting library definitions to a separate file that can be shared with your team

members—note that this import/export is separate from the more general import/export of all workbench preferences. Export your new libraries to a file named `EclipseDistilled.userlibraries` and then notify other team members that they should import this file into their user library settings. This file does not contain the library's files—it only contains the file path locations to JARs, source attachments, and Javadoc HTML.

If others who import this file use a different file structure for organizing their external library files, they must edit the library definitions to remove and add the JAR files with correct path locations. Unfortunately class path variables are not available to parameterize the library file locations.

6.2.1 Linked Library Project

A useful hybrid strategy is to configure a user library that is also available as a linked folder in your Eclipse workspace. Follow these steps:

1. Create a *simple project* in your workspace. Unlike Java projects, a simple project has no Java build path configuration in its properties. Use the command **File > New > Project > Simple Project**.
2. Uncheck the option to use a default project location within the workspace folder and enter the path for your `\eclipse-contrib\libraries` folder (see Figure 6-8).

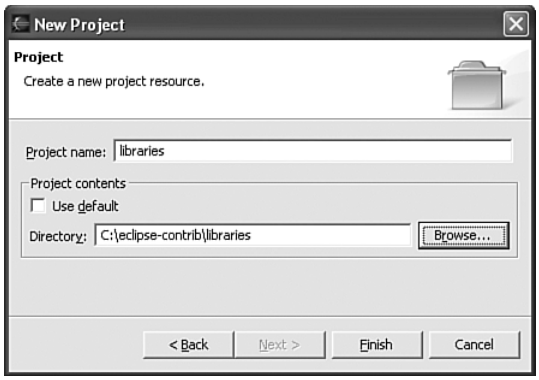

Figure 6-8 Create a new simple project with linked folder location.

3. Your new `libraries` project should look similar to the one in Figure 6-9.

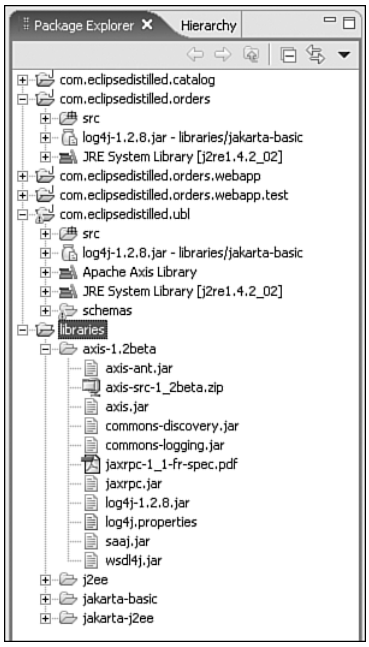

Figure 6-9 Java projects with sharing common libraries.

The Apache Axis distribution includes both the source code and Javadoc HTML files in one ZIP file, which is named `axis-src-1_2beta.zip` in this illustration. This file is used within the user library configuration to add source attachment and Javadoc to classes in this JAR file. Also notice the inclusion of a PDF specification file related to the JAX RPC library. You can double-click this file from within Eclipse to launch an external PDF reader.

In previous configuration of the `orders` project, we created a `lib` subfolder, copied the log4j JAR file into it, and then added this archive to the project build path. However, this approach can lead to a lot of duplication when we need the same JAR in several projects. Now we can use the shared `libraries` project to add log4j, or other `jakarta-basic` archives, into any of our projects.

In Figure 6-9, the shared log4j JAR has been added to both the `orders` and `ubl` projects. In addition, the Apache Axis Library is also included in the `ubl` project configuration as a user library; it includes a combination of six interdependent JAR files.

Eclipse has very flexible capabilities for configuring user libraries and leaves opportunity for creative arrangements. The hybrid approach described here has several benefits:

- Gather all of your open source libraries in a common folder named `/eclipse-contrib/libraries`.
- Download binary, source, Javadoc, and other related specifications into the same folder. Leave source and Javadoc files compressed in ZIP files.
- Create a simple project in each Eclipse workspace using a linked folder location for the project's contents. If you use multiple workspaces to separate your work as described in Chapter 3, then they can share the same reference libraries.
- Create user libraries when you often use several JAR files in combination. Export the user library definitions to share them between workspaces.
- Use the libraries project to add other individual JAR files to the build path of Java projects; use the **Add JARs…** button for portable location references.

6.3 Java Compiler Settings

The Java compiler settings enable you to control the problem messages produced by Eclipse while building your project. The problem severity level can be set to Error, Warning, or Ignore for more than 30 different conditions. These messages appear as markers within the generic **Problems** view, which also includes additional capabilities to sort and filter the messages (see Chapter 4, "Customizing Your Workbench").

Default compiler settings for all projects in the current workspace are assigned in the preference page category **Java > Compiler**. You have the option of overriding workspace preferences for the Java compiler with project-specific settings. The project properties dialog includes the **Java Compiler** page shown in Figure 6-10. Initially, the option to ***Use workspace settings*** is selected. Change this to ***Use project settings*** when you want to modify the problem messages for an individual project.

A common reason to modify these settings is when you are working on Java code from outside your team or from an open source repository. For example, the **Unused Code** tab is shown in Figure 6-10, where the **Unused imports** option is set to a default value of Warning. This option displays a warning message for each import statement in a Java source file that is not used within that class.

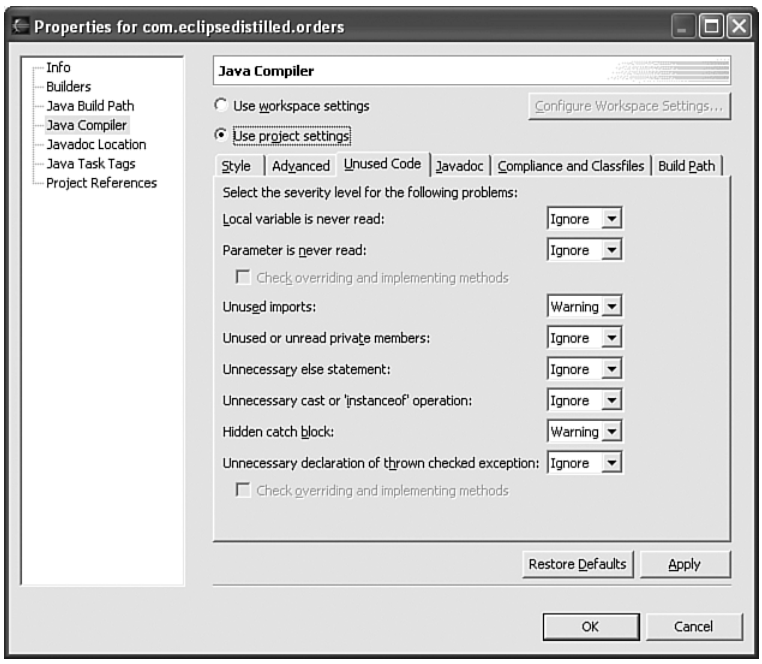

Figure 6-10 Use project-specific settings for compiler detection of unused code.

Other developers not working with Eclipse will often leave many unused imports in their code, sometimes leading to hundreds of warning messages when you create an Eclipse project containing that source code. It's best to leave this warning activated for your other projects, but you can eliminate the warnings in this single project by changing the option to ignore unused imports.

Your development team might also choose to establish very rigorous coding standards that rank unused imports as errors and issue a warning for unused local variables and unnecessary `else` statements. Set these coding standards as default Java compiler preferences and then export the preferences to a file that is imported by all team members.

6.4 Create Code Templates for Logging

Although not part of Java project configuration, the creation of code templates is naturally a part of our setup to improve development productivity. We will create several new templates that insert common statements used by the log4j logging facility.

Open the preferences page for **Java > Editor > Templates** and press the **New…** button. Fill in the template as shown in Figure 6-11. Press the **Insert Variable…** button while filling the template pattern to get a list of variables that automatically substitute values when the template is applied in your code.

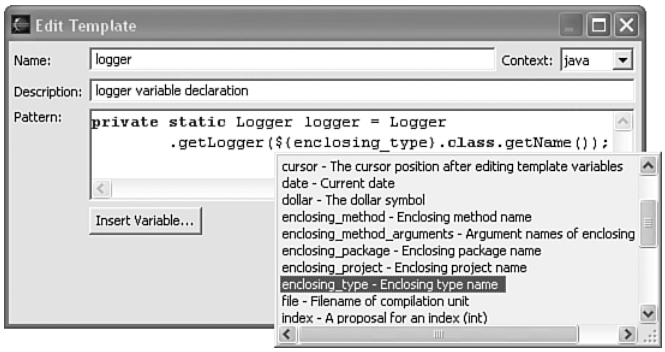

Figure 6-11 Create a new code template to insert log4j variable declaration.

Save the template and apply it within your `CatalogItem` class. Position the cursor below the class declaration, type "`logger`", and press **Ctrl+Space** to activate Content Assist. Select the `logger` template from the suggestion list and press **Enter**. Notice that the enclosing class name is automatically substituted when the following code is generated:

```
public class CatalogItem {
    private static Logger logger = Logger
            .getLogger(CatalogItem.class.getName());
```

However, you still receive an error that the `Logger` class cannot be resolved. Position the cursor over the `Logger` class name and press the shortcut **Ctrl+Shift+M**. Two different classes with this name are available:

```
java.util.logging.Logger
org.apache.log4j.Logger
```

A logging facility was added to Java version 1.4 that is similar in name and function to the Apache log4j library. But many developers still choose to use log4j instead of the `java.util.logging` package. You can prevent accidental choice of the wrong class and simplify use of Content Assist by filtering available types

within Eclipse to exclude the `java.util.logging` package. Open the preferences page for **Java > Type Filters** as shown in Figure 6-12 and add a new type filter.

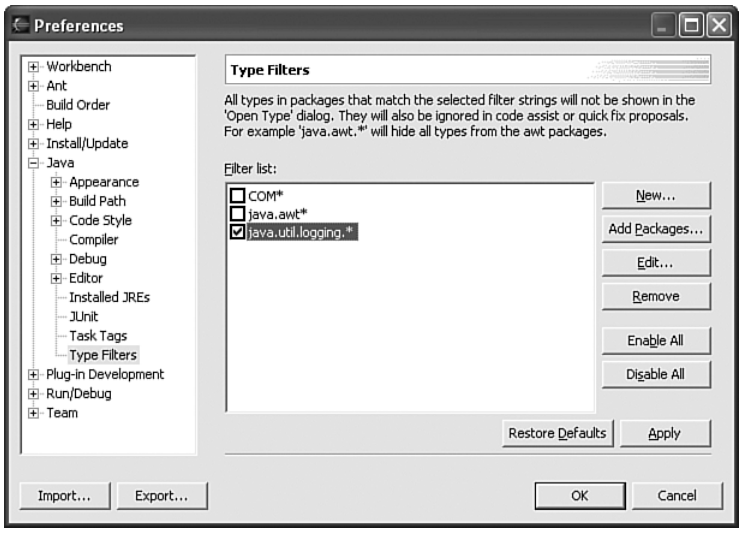

Figure 6-12 Filter the available types to exclude built-in Java logging, leaving the Apache log4j logging types as default.

Now when you press **Ctrl+Shift+M** to add an import statement for `Logger`, it is inserted immediately without prompting you because only one class by this name is available.

Now we'll create one more template for logging error messages. In Chapter 5, we used Quick Fix to automatically wrap a `try/catch` block around a `URL` class constructor to catch the `MalformedURLException`. We can now replace the default generated `catch` block with an error log message.

Create a new template named `logerr` and assign this pattern:

```
logger.error(${message}, ${e});
```

When you want to apply this pattern to log errors, simply type "logerr," press **Ctrl+Space** to activate Content Assist, select the template name, and press **Enter**. A single line is inserted, which is shown as bold here:

```
try {
    URL url = new URL("http://www.eclipse.org");
```

```
} catch (MalformedURLException e) {
    logger.error("message", e);
}
```

As with the other templates described in Chapter 5, you are prompted to replace the template variables. The two variables named `message` and `e` are highlighted in this pattern so that you can replace them with appropriate values, although the default name `e` is correct in this case.

6.5 Distilled

Eclipse includes a wide assortment of configuration options that control project compilation or enhance productivity. Don't get overwhelmed by the number of choices, but get started and gradually expand the customizations to suit your personal style and team development standards.

- Each Java project includes a *builder* that compiles its resources from source into output folders. A simple project has no builders, and other project types can add relevant builders that apply appropriate compilers or transformation utilities to the files.
- A project is built automatically; that is, the builder is applied automatically whenever a file is saved. For Java projects, the builder uses the *Java compiler settings* configured in the workspace preferences or overridden in a project.
- A Java project's *Java build path* defines which projects from the workspace and which JAR archive files are included in the class path when building or running the project.
- *User libraries* provide a convenient mechanism to define named collections of related JAR files that are used in combination. For example, J2EE standard APIs or the Apache Axis Web Services toolkit are good candidates for user libraries.
- A hybrid configuration is possible where third-party JAR files are collected in an external directory. Some of these archives are packaged in one or more named user libraries, and all are easily accessible from within Eclipse using a simple project and linked file location.

6.6 References

Apache Log4j is an open source logging facility available at *jakarta.apache.org/log4j/*.

Apache Axis is an open source Web Services development toolkit available at *ws.apache.org/axis/*.