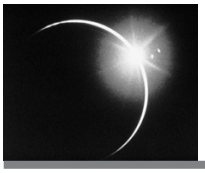

CHAPTER 20

Integrating Code Libraries

Even the most Eclipse-biased developer would concede that the majority of Java libraries out there are not shipped as plug-ins. This chapter discusses the integration of these libraries into Eclipse.

In Part II, you saw that the Smack messaging library proved to be very useful for Hyperbola. In Chapter 10, "Messaging Support," we used the term *bundling* to capture the idea of adapting or integrating one or more libraries (i.e., JARs) into Eclipse. We also showed you how to use PDE to convert Smack libraries into a bundle.

Bundling is typically a straightforward process, but there are choices to be made and issues to be resolved. In this chapter, we discuss the different bundling variants and common problems that arise when using existing code within Eclipse. In particular, we show you how to:

- Structure plug-ins differently.
- Bundle by injection—Add bundle metadata to existing JARs.
- Bundle by wrapping—Wrap JARs with bundle metadata.
- Bundle by reference—Add bundle metadata beside existing JARs without affecting the JARs, their original location, or their surrounding directory structure.
- Solve common classloading problems.

20.1 Plug-ins as JARs

Before diving into the different bundling techniques, it's worth looking at how plug-ins are typically structured. Traditional plug-ins are stored as a directory

structure, as shown on the left of Figure 20–1. The directory contains the code JARs, metadata such as a `plugin.xml` and `MANIFEST.MF`, and other non-code resources.

As of Eclipse 3.1, *plug-ins as JARs* is more the rule. The bottom right of Figure 20–1 shows this new structure. Note that it is more than a simple JAR'ing of the plug-in directory. The non-code contents of the original plug-in, that is, the metadata and files around the plug-in's code JAR, are injected *into* the root of the code JAR itself. This flips things around a bit and looks at plug-ins as code JARs with some supporting metadata and extra files inside. This view is subtle, but quite powerful.

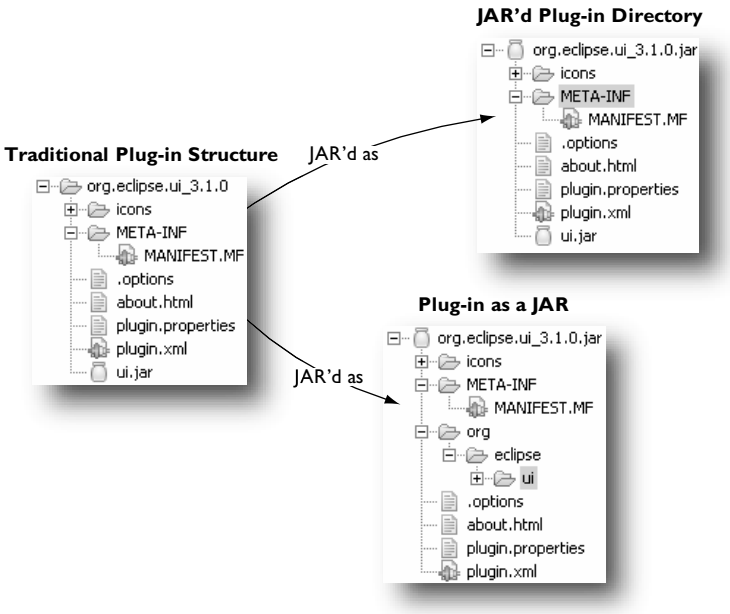

Figure 20–1 JAR'ing a traditional plug-in directory

For example, the basic structure of a standard, non-Eclipse code library and a JAR'd plug-in are the same—they are both just JARs. It is then easier and more natural for library producers to simply include the extra bundle metadata in their `MANIFEST.MF` and ship their library as both a standalone JAR and a bundle ready for integration into Eclipse. If all libraries were shipped this way, you could skip the rest of this chapter! That day may come, but in the meantime, this mindset should help you in the process of bundling the libraries you want to use in Eclipse.

The top right of Figure 20–1 shows a straightforward JAR'ing of a traditional plug-in directory. Note that this results in a plug-in JAR that contains nested code JARs. Chapter 26, "OSGi Essentials," and Section 20.3, "Bundling by Wrapping," outline a number of issues with this structure. Here, it suffices to say that plug-ins in this form can be used, but they are inefficient in both speed and space.

20.2 Bundling by Injection

As you saw in the previous section, JAR files can be used as plug-ins as long as they contain the metadata needed by Eclipse. Here we look at how to bundle existing code library JARs by injecting them with this Eclipse metadata. This approach retains all the benefits of JAR'd plug-ins and increases the chances that the library authors will include the injected metadata directly in their original releases—it directly illustrates the simplicity of the required changes.

Figure 20–2 shows the process of bundling the two Smack JARs on the left into one JAR that is a bundle. The operation merges the two input JARs and adds the required Eclipse bundle definition information in the `MANIFEST.MF` file. This is what we did with the Smack JARs in Chapter 10.

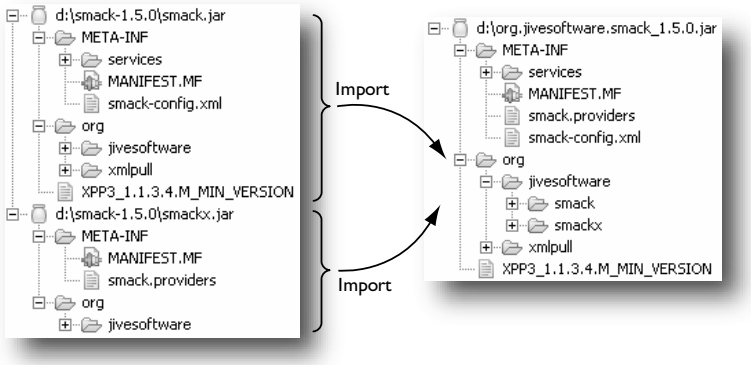

Figure 20–2 Injecting metadata into a code library

To bundle a set of library JARs, create a new plug-in project using **File > New > Project… > Plug-in from existing JARs**, as you did in Chapter 10. Ensure that the **Unzip the JAR archives into the project** box is checked so that the wizard unpacks all the JARs as they are imported into the new project. If more than one JAR is listed, as in Figure 20–2, they are merged as if they were on the classpath in the order specified in the wizard. That is, resources in subsequent JARs

do not overwrite resources in previous JARs. The wizard then generates a manifest that exports all the packages in the plug-in (i.e., those found in the original JARs).

There are two manual steps that you may need to take when bundling JARs:

Create a `plugin.xml`—Bundled libraries may be crafted to contribute extensions, but generally they do not contribute extension points. Extension points require Eclipse-specific processing and these libraries are, by definition, written without Eclipse in mind. If you want the library to contribute extensions, use the plug-in editor to define the extensions as outlined in Part II—the bundling wizard cannot generate these for you.

Define dependencies—In some cases, a library A uses code found in library B. Assuming you are bundling both A and B independently, you need to configure A to require B or import the needed packages. The wizard automatically generates exports for all the packages in both A and B, but does not analyze A's code to determine what packages it needs—you have to do that manually using the plug-in editor.

The resultant project is just like any other plug-in project. You can leave it in the workspace and code against it, you can run with it, and you can export it. A handy trick is to export it and add it to the target platform. You can then delete it from the workspace—it becomes just another plug-in that you are using. This keeps your workspace clean and allows the new plug-in to be shared between workspaces.

An alternative to combining JARs is to convert each to a bundle individually. This is certainly feasible, but is not always the best choice. For example, Ant comes as a set of about 28 JARs. Many of these are tiny (<10K). While the overhead of a bundle is small, this is too fine-grained—vast numbers of plug-ins are harder to manage.

Bundling JARs separately is also a problem when packages are split over two or more JARs. If code in these package fragments needs to see a package's private members in the other JARs, bundling separately does not work. Each bundle gets its own classloader. As such, the package `org.eclipsercp.hyperbola` loaded by bundle A is actually different from the one with the same name but loaded by bundle B. They do not share package visibility.

20.3 Bundling by Wrapping

Since injecting metadata as described in the previous section requires modification of the original JARs, it is not always feasible. The following list outlines the most common problems:

Licensing—Some licenses explicitly state that the licensed material cannot be modified or that modifications trigger further restrictions or obligations.

Signing—JARs are often signed to prevent otherwise undetected tampering of their contents. In this use of signing, it may be possible to inject the metadata and additional files—these files are either not signed or are signed by a different signer. In other situations, signing is used to imply permissions and rights. In these cases, it is less clear that metadata injection is feasible.

Multiple JARs—Some libraries come as multiple JARs. As discussed above, the JARs can be bundled separately or they can be combined. Both approaches are feasible, but may not be attractive in some cases.

The best approach may be to wrap the set of JARs in a plug-in directory and create a `MANIFEST.MF` that describes their dependencies and exports, as shown in Figure 20–3.

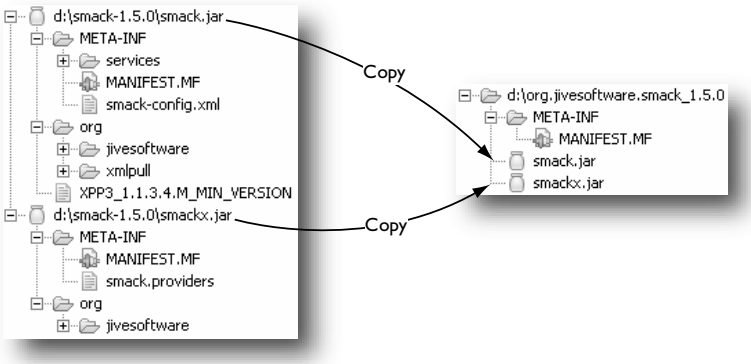

Figure 20–3 Wrapping a code library

The same bundling wizard used to inject metadata can be used to wrap JARs. Simply unchecking the **Unzip the JAR archives into the project** box tells the wizard to copy the JARs into the project without extracting their contents. The JARs are then listed on the bundle's classpath in the order you added them to the wizard.

Again, the resultant project is just like any other plug-in project. When you go to export it, however, you should package the plug-in as a directory. Use the **Package plug-ins and individual JAR archives** option on the **Export > Deployable plug-ins and fragments** wizard. Otherwise, you will end up with the

original library JARs nested inside the new plug-in JAR. Plug-ins in this layout are usable, but are inefficient since the nested JARs must be extracted before being used and standard Java compilers cannot compile against such JARs. Chapter 26, "OSGi Essentials," has a lengthy discussion of the pros and cons of various plug-in shapes.

20.4 Bundling by Reference

> **Warning:** This approach is experimental in Eclipse 3.1 and is subject to change. It breaks some fundamental notions of plug-in encapsulation and is not compatible with various parts of the Eclipse tooling. It does, however, address some real use cases. Use with caution and only when absolutely needed.

In some situations, installed JARs cannot be moved let alone modified. This typically happens when the libraries are delivered as part of another product and are laid down by an installer. The JARs are in a specific spot and are expected to be there to be found by other programs.

The bundling approaches outlined so far do not work because they modify either the JAR itself or its surroundings. For example, wrapping adds plug-in metadata beside the JAR being wrapped. If there is only one set of JARs to wrap in a directory, the generated metadata can be directly added to that directory. If there are multiple libraries in the same directory, the metadata files conflict with each other. Metadata injection can only be used if the issues mentioned earlier are not applicable. For example, the JAR has to be writable.

Even if injection or wrapping was used, there is still the problem of how to get the resultant plug-in installed into Eclipse. The natural Eclipse pattern is to have plug-ins either in the main `plugins` directory in the Eclipse install or in a `plugins` directory in an extension location. Both approaches require moving the newly bundled library. Plug-ins can be installed using markup in the `config.ini` file (see Chapter 26 for a discussion of the `osgi.bundles` property), but that is cumbersome and hard to manage. It would be better to simply create a plug-in in one of the normal spots (i.e., in a `plugins` directory that is known by Eclipse) and point at the required JARs wherever they happen to be on disk, that is, put the metadata *on the side* and leave the code JARs external to the plug-in.

To illustrate how this works, consider a mythical Java database connectivity (JDBC) driver JAR that comes with a database product. The product installer puts `jdbc.jar` in `c:\db\drivers\jdbc.jar` and you cannot modify it, move it, or add files to the `drivers` directory.

To set this up, proceed as though you are using the wrapping approach from the previous section. Run the **New Project** wizard and create a JDBC plug-in based on `jdbc.jar`. Don't worry about the libraries being copied into the project; you can use them to do your normal development.

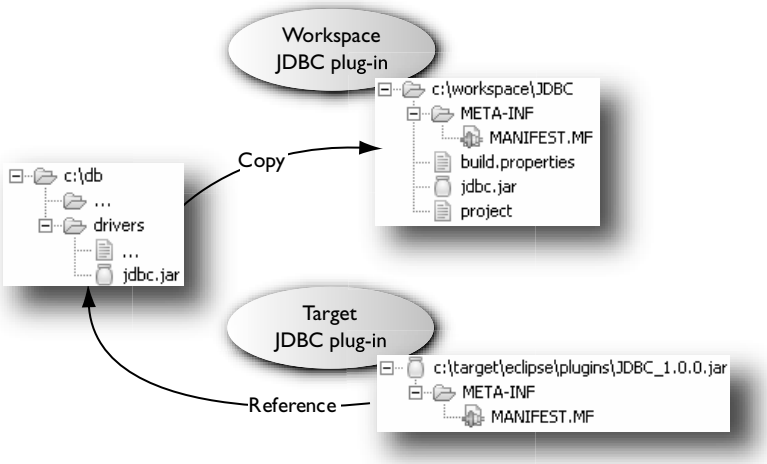

Figure 20–4 External plug-in JARs

When you go to run your application, you need to use the original JDBC libraries. Use the following steps to set up the structure shown in Figure 20–4:

1. Export the newly created JDBC plug-in from your workspace to your target's `plugins` directory.
2. Delete the exported JARs and extraneous files (e.g., `.project`) from the exported target plug-in.
3. Edit the exported target plug-in's `MANIFEST.MF` and change the classpath to point to the original JARs using absolute filesystem paths. For example, replace "jdbc.jar" with "external:$JDBC_HOME$/drivers/jdbc.jar". You can use environment or system properties, or full filesystem paths to identify the desired JAR.
4. In the IDE, use **Window > Preferences… > Plug-in Development > Target Platform > Reload** to refresh the target and add the new JDBC plug-in.
5. Set up an **Eclipse Application** launch configuration to run your product. On the **Plug-ins** page, select the third option, **Choose plug-ins and fragments to launch from the list …**. In the list of plug-ins, uncheck the JDBC

plug-in in the **Workspace Plug-ins** list and check the one in the **External Plug-ins** list.

6. Run the launch configuration. It is difficult to tell which JAR is being used, but it should be the original `c:\db\drivers\jdbc.jar`. You can confirm this by renaming the original JAR and running. The application should fail when trying to load JDBC classes.

When it comes time to deploy your application and the JDBC plug-in, you have to rely on a native installer or feature install handler to set up the plug-in's manifest and ensure that `jdbc.jar` is in fact installed. The task is quite a bit easier if the database product defines environment variables or Java system properties, as shown in the example, to describe the location of its install. For example, if the product defined `JDBC_HOME` as an environment variable, then you can set up the JDBC plug-in's manifest to include the line:

```
Bundle-Classpath: external:$JDBC_HOME$/drivers/jdbc.jar
```

This mechanism has the added benefit that the JDBC plug-in can be built and delivered using standard Eclipse mechanisms. Variables make this even easier.

The real danger in using this setup is the potential for mismatching the metadata and contents of the JARs. For example, you might generate the metadata based on version 3 of the JDBC drivers, but the actual installed drivers are version 2. Tracking down these kinds of bugs is challenging to say the least. Nonetheless, the mechanism is there and it solves some real problems. Use it with caution and care.

20.5 Troubleshooting Classloading Problems

Most code libraries are quite straightforward to bundle and then use in Eclipse-based systems. You've seen that the wizard to create plug-ins from existing JARs does most of the work for you. But what happens if there are problems after bundling? If you remember from Chapter 10, after bundling Smack, the next thing we did was write a test plug-in to see if Smack could be referenced as a plug-in. At this point, there are two main problems that could occur. The first is at compile time—classes in the bundled JAR may not be visible. This is easily fixed by ensuring that the plug-in containing the bundled JAR exports all the necessary packages from the JAR. But what if something goes wrong at runtime? The classic symptom is `ClassNotFoundExceptions` and `NoClassDefFoundErrors` showing up in the console or the log file.

This entire section is devoted to helping you understand and troubleshoot these runtime errors. Typically, these relate to the classloading structure inherent in Eclipse and OSGi. The OSGi classloading strategy and mechanism is discussed in Chapter 26, but here we detail some standard library coding patterns and how to handle them in Eclipse.

20.5.1 Issues with Class.forName()

Let's start with the classic example of `ClassNotFoundExceptions`, which occurs while using a bundled code library. Consider adding logging using log4j, a popular library for managing and logging events (*http://logging.apache.org/*), in Hyperbola. Using the techniques described earlier, you can bundle log4j and add it to either your workspace or target and continue development. At runtime, however, log4j throws a number of `ClassNotFoundExceptions` when trying to configure its *appenders*.

log4j is extensible in that it allows clients to supply log appenders—effectively log event handlers. Appenders are configured by naming their implementation classes in metadata files, much like Eclipse plug-ins define extensions. log4j then reads these files and loads the named classes using a code pattern similar to the snippet below:

```
public class AppenderHelper {
  private Appender createAppender(appenderName) {
     Class appenderClass = Class.forName(appenderName);
     return appenderClass.newInstance();
  }
}
```

Note: log4j actually uses a more advanced code pattern that is detailed in the next section. For the sake of this example, assume that log4j is running with the `log4j.ignoreTCL` property set to `true` and `Class.forName(String)` is its only classloading option.

`Class.forName(String)` is the classic Java mechanism for dynamic class discovery and loading. It uses the *current classloader* to look for and load the requested class, in this case, an appender. The current classloader is the classloader that loaded the class containing the method executing the `forName(String)` call. In the snippet above, the current classloader is the one that loaded `AppenderHelper`. The net result is the same as if a reference to the

appender class was compiled into `createAppender()`. This is exactly what using `Class.forName(String)` is trying to work around.

In Eclipse, this is problematic because the log4j plug-in typically does not depend on the plug-ins providing the appenders. This is actually the point—appenders are log4j's way of allowing its function to be extended, but the log4j plug-in cannot load these appenders because it does not have the proper visibility.

If log4j was written as a standard Eclipse plug-in, it could, for example, use the Eclipse extension registry and define an `appenders` extension point. Plug-ins wanting to provide extenders would then contribute executable extensions that name their appender classes and log4j would use `createExecutableExtension()` (see Chapter 23, "RCP Everywhere") rather than the code in `createAppender()` above. Unfortunately, log4j is not written as an Eclipse plug-in and this technique is not available.

Buddy classloading offers an alternative integration strategy that does not require code modification. The mechanism works as follows:

- Plug-ins declare that they need the help of other plug-ins to load classes.
- They also identify the kind of help they want by specifying a *buddy policy*. The policy defines what kinds of plug-ins are to be considered to be buddies as well as how (e.g., in what order) they are consulted.
- When a plug-in fails to find a desired class through all the normal routes as outlined in Section 26.10, "Classloading" (i.e., `Import-Package`, `Require-Bundle`, and local classes), its buddy policy is invoked.
- The invoked policy discovers a set of *buddies* and consults each one in turn until either the class is found or the list is exhausted.

Let's apply the built-in *registered* buddy policy to the log4j case and see how it helps. In the log4j scenario, there are a relatively large number of potential clients of the logging API and a small number of clients supplying appenders. For performance and simplicity, it makes sense to limit the buddy search scope to just those supplying appenders. The simplest approach is to make those plug-ins explicitly register as buddies of log4j.

To set this up, you first mark the log4j plug-in as needing classloading help and identify the "registered" policy as the policy to use. The following line added to log4j's `MANIFEST.MF` makes that declaration:

```
Eclipse-BuddyPolicy: registered
```

Then in each plug-in that supplies appenders, add the following line to the `MANIFEST.MF` to register the plug-in as a buddy of log4j (i.e., `org.apache.log4j`):

```
Eclipse-RegisterBuddy: org.apache.log4j
```

At runtime, when log4j goes to instantiate an appender using `Class.forName(String)`, it first tries all its normal prerequisites. Then, when it fails to find the appender class, each of its registered buddy plug-ins is asked to load the class. If all the appender plug-ins are registered, the appender class is sure to be found.

Note: Buddies are consulted as if they were originating the load class request using `Bundle.loadClass(String)`. That is, the buddy's imported packages, required bundles, and in fact, its own buddies are all invoked as necessary in the search for the desired class.

20.5.1.1 Built-in Buddy Policies

Eclipse supplies a number of built-in policies, as summarized in Table 20–1:

Table 20–1 Built-in Buddy Policies

Name	*Description*
`boot`	Indicates that the standard Java boot classloader is a buddy.
`ext`	Indicates that the standard Java extension classloader is a buddy. This policy is a superset of the `boot` policy.
`app`	Indicates that the standard Java application classloader is a buddy. This policy is a superset of the `ext` policy.
`parent`	Indicates that the plug-in's parent classloader is a buddy. By default, the parent classloader is the standard Java boot classloader. Plug-in classloader parentage is controlled on a global basis by setting the `osgi.parentClassloader` system property.
`dependent`	Consults all plug-ins that directly or indirectly depend on the current plug-in. Note that this casts a rather wide net and may introduce performance problems as the number of plug-ins increases.
`registered`	Is similar to the `dependent` policy, but only dependent plug-ins that have explicitly registered themselves as buddies of the current plug-in are consulted.

One plug-in can apply several policies simply by listing them on the `Eclipse-BuddyPolicy` line in the `MANIFEST.MF`. Eclipse invokes each policy in turn until either the class is found or all policies have been consulted.

20.5.1.2 Buddy Classloading Considerations

As powerful and useful as buddy classloading is, it is still a mechanism of last resort. There are a number of issues that you should consider carefully before using buddies in your system:

- Buddy classloading runs counter to the notions of component that Eclipse attempts to maintain and is not particularly well-suited to dynamic environments—particularly ones where buddies can be uninstalled.
- Buddy classloading also incurs various performance costs. For example, when a normal classload fails in a plug-in using buddy loading, the buddy policy is invoked. Typical Java resource bundle loading causes up to three classload failures and some number of resource load failures before finally getting the desired resource. Each of these failures repeats the buddy search.
- Buddy loading is relatively undirected. Normally, the OSGi classloading infrastructure knows exactly where to go to find any given package—this is the information gleaned from the `MANIFEST.MF` files. Typical buddy loading policies simply search successive buddies.
- It is possible that the buddy search will find the wrong class with the right name. If two buddies contain the same class, depending on the policy, it may be ambiguous as to which buddy ultimately supplies the class.

20.5.1.3 Dynamic-ImportPackage vs. Buddy Classloading

Readers familiar with OSGi may be scratching their heads asking, "What about `Dynamic-ImportPackage`?" For readers who are not familiar with OSGi, `Dynamic-ImportPackage` is a mechanism that allows a bundle to state its need to use a given set of packages but not force an early binding to the exporters of those packages. Rather, the binding to package exporters is done at runtime when the bundle tries to load from a dynamically imported package.

So, some `Class.forName()` problems can be alleviated simply by adding

```
DynamicImport-Package: <list of packages or *>
```

to the `MANIFEST.MF` for the plug-in using `Class.forName(String)`. This has the following drawbacks compared to the buddy loading described here:

- Dynamic importing is unscoped. That is, all bundles exporting packages are considered. As such, the search may include many irrelevant and unrelated bundles. By contrast, the buddy loading mechanism allows for poli-

cies that use dynamic information such as the plug-in dependency graph to drive the search for classes.

- Dynamic importing implies inter-bundle constraints. That is, when a bundle A loads a class from a bundle B using dynamic importing, A is then considered to be dependent on B. If B is refreshed or uninstalled, A is refreshed. This behavior is valuable for maintaining consistency when A actually uses and retains references to B's classes. However, several serialization scenarios have A simply using B's classes temporarily (e.g., to load some object stream)—there should be no lasting dependency.
- Dynamic import considers only packages explicitly exported by other bundles. Again this can be a desirable characteristic, but in various use cases such as serialization, the importing bundle potentially needs access to all classes in the system, for example, to load instances from an object stream.

This is not to say that `Dynamic-ImportPackage` should never be used, just that it should be used appropriately. For example, when the set of packages needed is well-known and the importing bundle has a lasting dependency on the imported packages.

20.5.2 Issues with Context Classloaders

Since Java 1.2, the `Class.forName(String)` mechanism has been largely superseded by *context classloading*. As such, most modern class libraries use a context classloader. In the discussion below, we show how Eclipse transparently converts the use of context classloaders into something equivalent to `Class.forName (String)`. Doing this allows the buddy loading and `Dynamic-Import` mechanisms described above to be used to eliminate `ClassNotFoundExceptions` and `NoClassDefFoundErrors`.

Each Java `Thread` has an associated context classloader field that contains a classloader. The classloader in this field is set, typically by the application container, to match the context of this current execution. That is, the field contains a classloader that has access to the classes related to the current execution (e.g., Web request being processed). Libraries such as log4j access and use the context classloader with the updated `AppenderHelper` code pattern below:

```
public class AppenderHelper {
  private Appender createAppender(String appenderName) {
    ClassLoader loader = Thread.currentThread().getContextClassLoader();
    Class appenderClass = loader.loadClass(appenderName);
    return (Appender)appenderClass.newInstance();
  }
}
```

By default, the context classloader is set to be the normal Java application classloader. That is, the use of the context classloader in normal Java application scenarios is equivalent to using `Class.forName(String)` and there is only one classloader, the application classloader. When running inside Eclipse, however, the code pattern outlined above fails because:

- By default, Eclipse does not consult the application classloader. Eclipse-based applications put their code on dynamic plug-in classpaths rather than on the normal Java application classpath.
- Eclipse cannot detect plug-in context switches and set the context classloader as required. That is, there is no way to tell when execution context shifts from one plug-in to the next as is done in Web application servers.

These characteristics, combined with the compositional nature of Eclipse, mean that the value of the context classloader field is seldom useful.

Clients can, however, explicitly set the context classloader before calling libraries that use the context classloader. The snippet below shows an example of calling log4j using this approach:

```
Thread thread = Thread.currentThread();
ClassLoader loader = thread.getContextClassLoader();
thread.setContextClassLoader(this.getClass().getClassLoader());
try {
  ... log4j library call that calls AppenderHelper.createAppender() ...
} finally {
  thread.setContextClassLoader(loader);
}
```

First the current context classloader is saved. The context classloader is then set to an appropriate value for the current execution and log4j is called. log4j's `AppenderHelper` uses the context classloader, so in this case, it uses the client's classloader (e.g., `this.getClass().getClassLoader()`). When the operation is finished, the original context classloader is restored.

The assumption here is that the client's classloader is able to load all required classes. This may or may not be true. Even if it can, the coding pattern is cumbersome to use and hard to maintain for any significant number of library calls. Ideally, log4j would be able to dynamically discover the context relevant to a particular classloading operation. Eclipse enables this using the *context finder*.

The context finder is a kind of `ClassLoader` that is installed by Eclipse as the default context classloader. When invoked, it searches down the Java execution stack for a classloader other than the system classloader. In the `AppenderHelper` example above, it finds the log4j plug-in's classloader—the one that loaded

AppenderHelper. The context finder then delegates the load request to the discovered classloader.

This mechanism transforms log4j's call to getContextClassLoader().loadClass(String) to the equivalent Class.forName(String) call using log4j's classloader to load the given class. Now the buddy classloading techniques discussed in Section 20.5.1 can be applied to help log4j load the needed appender classes.

The net effect is that clients of log4j do not have to use the cumbersome coding pattern outlined above even though the libraries they call use the context classloader. This approach generalizes to other context classloading situations.

20.5.3 Managing JRE Classes

For various reasons, some libraries include packages that are normally found in the JRE. For example, version 2.6 of Xalan, the XML transformation engine, comes with types from the org.w3c.dom.xpath package in xalan.jar. These types are also included as part of typical JRE distributions. When xalan.jar is used as part of a normal Java application, it is added to the classpath, but its xpath classes are obscured by those in the JRE. Everything is fine.

When you bundle Xalan, the tooling produces Export-Package entries for all packages in xalan.jar. However, the tooling cannot know that it should add imports for the org.w3c.dom.xpath package found in the JRE. Without the import, Xalan uses its own copies of the xpath types and may conflict with those supplied by the JRE.

This happens because Eclipse 3.1 plug-in classloading is highly optimized. These optimizations depend on the plug-in manifest information to know which packages come from which plug-ins. Except for the use of certain buddy policies, the classloaders never search for classes—they always knows exactly where to find them.

For the JRE packages, only java.* packages are assumed to come from the boot classloader. All others must be imported in the consuming plug-in's MANIFEST.MF. The API packages included in the JRE are typically exported by the system bundle (i.e., org.eclipse.osgi or system.bundle). This list is captured in a JRE *profile*. Eclipse includes a number of profiles for common JREs and automatically detects the appropriate one to use. You can control this further by setting the osgi.java.profile property to the URL of a profile file to use.

So, if the Xalan plug-in fails to import the xpath packages, its local copies are used. This may result in ClassCastExceptions because the plug-in's copy of the type is not interchangeable with the copy supplied by the JRE. Changing the plug-in to import the packages tells Eclipse to use the external copy. Alternatively, the offending packages can be removed from Xalan.

20.5.4 Serialization

Serialization of objects occurs in many different situations. Some libraries use the built-in `java.io.Serializable` mechanism directly. Some use it indirectly as a consequence of using Remote Method Invocation (RMI). Others serialize objects using their own marshalling strategies (e.g., Hibernate stores/loads objects to/from relational databases). Regardless of the technique used, these plug-ins have the following characteristics:

- They are typically generic utilities and do not have access to, or knowledge of, your domain classes.
- They do not hold onto the classes they request, but rather use them to load objects and then discard their references.
- They need access to internal classes if instances of internal classes have been serialized.

Buddy classloading and context classloading solve all these problems. In effect, loading a serialized object is equivalent to the log4j appender problem. Appender classes are identified by name to log4j. Classes to load are identified to the serialization plug-in by name in the object stream. In both cases, the loading plug-in needs to search beyond its prerequisite plug-ins to find the desired classes.

20.6 Summary

The Java world includes a wealth of useful code libraries. Eclipse provides multiple techniques for integrating these code libraries into the Runtime environment. This can be as simple as running a wizard—most of the time it is.

In some cases, however, the code in the library uses certain patterns that are at odds with Eclipse. Classloading is the most common bone of contention. Eclipse includes a number of mechanisms and strategies for dealing with these cases. In particular, you can use buddies or `Dynamic-ImportPackage` to provide visibility to classes that would normally not be visible. In this chapter, we described the most significant of these and illustrated their use. The mechanisms outlined here enable you to resolve most remaining classloading issues encountered when integrating code libraries into Eclipse.