CHAPTER 4

Extensions

JDK 1.2 offers a new mechanism for updating code: the Java extension mechanism. From the JDK 1.2 documentation, (jdk1.2/docs/guide/extensions/index.html):

> Extensions are packages of Java classes (and any associated native code) that application developers can use to extend the functionality of the core platform. The extension mechanism allows the Java virtual machine (VM) to use the extension classes in much the same way as the VM uses the system classes. The extension mechanism also provides a way for needed extensions to be retrieved from specified URLs when they are not already installed in the JDK or JRE.

That last sentence should strike a nerve—"provides a way for needed extensions to be retrieved from specified URLs when they are not already installed in the JDK or JRE." That would seem to imply that if a class isn't present within the CLASSPATH or local file system we can grab it from someplace else. That's precisely what it means.

4.1 TYPES OF EXTENSIONS

The Java extension mechanism divides the world of Java extensions into two camps: installed and download. Each carries its own advantages and drawbacks.

4.1.1 Installed extensions

An installed extension is code that resides within the JRE's extension directory, which within the Sun JRE distribution, is the JRE\1.2\lib\ext directory. Any compiled Java code, whether in .class or .jar form, will be silently added to the JVM's CLASSPATH if it resides within this directory.

In this respect, the JRE's extension directory now mimics the same semantics as most modern operating systems and shared libraries. For example, under Win32, a DLL will be found by a `LoadLibrary()` call regardless of the directory in which the application is executing if the DLL resides in the Windows directory or Windows system directory. Most UNIX OSs have something similar using the `LD_LIBRARY_PATH` environment variable.

This makes distribution of Java applications much, much easier. Formerly, installing a Java application to a client's machine required not only the installation of the .class or .jar file to the local file system, but also modification of the user's CLASSPATH environment variable to include the new directory or directories or the .jar file itself. While not a monumental task, users can (and quite frequently do) change their environment variable settings, making Java applications particularly vulnerable. Now, install scripts can copy the code over to the extension directory, and Java will automatically find it.

Unfortunately, Java will look only in that specific directory, and not in any subdirectories underneath it. This means that this directory is likely to become cluttered and crowded as multiple applications install themselves to this one place. It also raises the ugly possibility that versioning issues will begin to appear on user systems as applications using common third-party JAR files (GNU code, or third-party GUI toolkits) which start accidentally overwriting newer versions with older versions on install. The Windows development community has been struggling with this problem for a decade, and accidental overwrites still occur despite their best efforts. Unless Sun quickly takes steps to address this, I would be very careful about how files are named when installed to this directory.

Fortunately, Java doesn't seem to care what the JAR file itself is named; for that reason, I'd suggest any JAR file to be installed to this directory follow a naming convention similar to that of Sun's package names. For example, if I create a .jar file containing the "HelloWorld.class" file, version 1, then I'd rename it "com.javageeks.HelloWorld.jar". That way, in my install scripts, I can check for an earlier version of my application, and search through the .jar file for a text file labeled "version", and read which version of my code I'm thinking about overwriting.

One undocumented[1] trick regarding extensions is the `java.ext.dirs` property. When the Java run time starts, it defaults this property to be the JRE's lib/ext directory. However, by using the `-D` parameter at the command line (or by specifying the equivalent option when using JNI invocation), it's possible to change or add directories to this path list.

As proof, create a simple Hello.java class and put it in the root of your file system; here I'm assuming it's a Wintel PC, on the C: drive. Now fire up the Java interpreter with the `–D` parameter like so:

```
java –Djava.ext.dirs=C:\ Hello
```

UNIX Java users would run:

```
java –Djava.ext.dirs=/ Hello
```

Your Hello class will be loaded and executed although it resides in the root directory instead of in the standard Extensions directory.

This in turn offers some hope for directory management. Each subdirectory (corresponding to a single application, development group, component, whatever) can be added to the `java.ext.dirs` property when the JVM is started. Naturally, this, too, could quickly become unmanageable, but until Sun changes this behavior, it's the best we can do.[2]

What would actually be very cool would be to modify the `java.ext.dirs` property, or its equivalent within the ClassLoader, to add Extension directories as the application executes. Unfortunately, URLClassLoader, which serves as the base class for Launcher$ExtClassLoader, doesn't make its `addURL` method public, so we have no hope of being able to do that. Once the Extension directories are loaded into the ExtClassLoader, they're fixed for the lifetime of the JVM.

4.1.2 Building an installed extension

Building an installed extension is as simple as building a normal JAR file. Begin with standard Java code, compile it, and condense it into a JAR file:

```
// HelloWorld.java (in src/chap2)
//
public class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println("Hello, world!");
    }
}
```

---

[1] As of this writing it only shows up when every property in the JVM is displayed via `System.getProperties`.

[2] If you hold a Sun Community Source License to Java 2, you could modify the source for the Extensions ClassLoader that manages the extensions directory (sun.misc.Launcher$ExtClassLoader). While such a modification would immediately render your environment impure Java, sometimes these sorts of localized source changes are necessary and beneficial in the long run.

```
/*
From the command line, do:

javac HelloWorld.java
jar cvf com.javageeks.HelloWorld.jar HelloWorld.class
copy com.javageeks.HelloWorld.jar your-JRE-directory\lib\ext

or, if you use the GNU make from the CD, edit the makefile.rules
file in the 'src' directory, and do:

make clean all

*/
```

As I described, typically if I'm packaging up a JAR file for release or installation on end-user machines, I'll also include a text file labeled "version" in the JAR:

```
Major 1
Minor 0
```

Then, inside of an install script or install executable, I can look for an existing com.javageeks.HelloWorld.jar file within the Extension directory. If one exists, I can open it using the java.util.zip classes, extract the version file, parse it, and determine if I need to overwrite what's there.

Once the JAR file is created, copy it to the Extension directory, and attempt to execute it:

```
copy HelloWorld.jar C:\prg\jdk1.2\jre\lib\ext
cd \
java HelloWorld
```

That's all there is to it.

4.1.3 Download extensions

For all the power in the installed extension mechanism, download extensions will be the ones in which people will probably be most interested. This is the ability to download code from a URL if it is not already present on the system. However, as the JDK 1.2 extension guide tells us, (jdk1.2/docs/guide/extensions/extensions.html):

> Unlike the case of installed extensions, the location of the JAR files that serve as download extensions is irrelevant. A download extension is an extension because it is specified as the value of the `Class-Path` header in another JAR file's manifest, **not** because it has any particular location.
>
> Another difference between installed and download extensions is that only applets and applications bundled in a JAR file can make use of download extensions. Applets and applications not bundled in a JAR file don't have a manifest from which to reference download extensions.

The key part comes in the second paragraph: "… only applets and applications bundled in a JAR file can make use of download extensions." So, in order to make use

of download extensions, we need to have our application in a JAR file, with the Manifest file indicating where else to look for code that the JVM can't find.

This offers some serious code-reuse capabilities, especially in a corporate intranet. In a sense, this is the DLL or shared library concept taken to a distributed context. Remember, the original idea of the shared library (or DLL, under Windows) was to prevent multiple copies of the same code loaded everywhere. By providing a mechanism by which code could be loaded only once across all processes using it, the shared library/DLL concept not only reduced per-process memory requirements, but also allowed for across-the-board updates of code by simply replacing the shared library.

Java now provides the same possibilities via this download extension mechanism. Suppose a team makes use of the com.javageeks.foobar component library, which happens to be in version 2.0, to do its development. Normally, before the download extension mechanism, the .jar file or .class files for the foobar library would need to be deployed with the development team's application. Should javageeks.com release a new version of foobar (version 3), the development team needs to make a new release with the new foobar .jar/.class files in it, even if no new development has taken place on the application.

Instead, with the download extension mechanism, the development team can mark the application's JAR as being dependent on the foobar library by using javageeks.com's URL to reference it:

```
Class-Path: http://www.javageeks.com/javalib/foobar.jar
```

Now, should javageeks.com release a new version of the foobar library, the development team need not do anything to take advantage of the new version; in fact, it may not even be aware of the new version. Just as DLLs could (in theory) be silently upgraded with newer versions as bug fixes and patches were released, new download extensions can also be silently upgraded without client knowledge.

This, of course, presumes that the download extension always exists at the given URL referenced within the application's .jar file. This may not be the case for commercial or freeware source sites, but on a corporate intranet developers certainly would. Just hang the shared component .jar/.class files from a known location on the corporate or departmental web server, and any application which makes use of that .jar/.class file library will automatically pick up any new updates.

Download extensions do carry some restrictions that installed extensions don't. Each and every time an application or JAR file is run that uses a download extension that resides off of a web server, the code will have to come across the wire in its entirety (jdk1.2\docs\guide\extensions\extensions.html):

> The extension mechanism will not install a download extension in the JRE or JDK directory structure. Download extensions **do not** become installed extensions after they have once been downloaded.
>
> Unlike installed extensions, download extensions cannot have any native code.

This means that each and every time the user fires up the application, it will have to download all of the application's class files over the wire. This can mean long load times, especially if your network bandwidth is tight, or you have a large number of users and/or a low-end intranet Web server.

Additionally, the restriction regarding native code may have more impact than might originally have been estimated. As seen in later chapters, JNI and native code can have some powerful applications in server-side Java applications.

4.1.4 Building a download extension

The Manifest file specification is given in jdk1.2\docs\guide\jar\manifest.html, and the specific headers for Java extensions are given in jdk1.2\docs\guide\extensions\extensions.html. Creating a Manifest file means you create a subdirectory (off the directory in which the JAR file will be built) called META-INF, and in that directory, create a file called MANIFEST.MF. It needs to contain, at a minimum, the following line:

```
Manifest-Version: 1.0.
```

This establishes it as a Manifest file to any JAR-reading utility that works with the JAR file. Optionally, it can also contain a line indicating the creator of the JAR file:

```
Created-By: JavaGeeks.com
```

You can establish this as an executable JAR file with the following line:

```
Main-Class: com.javageeks.ClientApp.Main.
```

This line indicates that when this JAR is specified to the Java interpreter using the -jar flag, this class (com.javageeks.ClientApp.Main, in the above example) contains the main method to execute. Specifying this line effectively allows us to create a standalone JAR file to execute on user machines. Effectively, saying

```
java –jar YourJar.jar
```

where YourJar.jar contains a Main-Class line of ClientApp.Main is the same as

```
set CLASSPATH=%CLASSPATH%;YourJar.jar
java ClientApp.Main
```

As a result, for the first time, Java now has the ability to ship a prepackaged single file that contains all the necessary elements for execution, without requiring modifications to the user's environment settings.

The key to download extensions is the Class-Path manifest setting, as demonstrated in this line:

```
Class-Path: servlet.jar foo.jar footoo.jar
```

Class-Path tells the JRE where else it needs to look for the additional classes that this JAR file references. This line contains the file or URL reference telling the JVM where

to find additional .jar files on which this JAR depends. It will then attempt to use these JAR files to resolve any requested classes during execution of the application code.

Readers familiar with the Java applet model will undoubtedly be curious why download extensions would even be necessary, given that an applet embedded in a Java page offers the same sort of functionality. After all, the applet model allows web page designers to download code as necessary into the client JVM to execute applets. In fact, the two approaches are distinctly related. However, in an application that uses download extensions, no security restrictions are in place—the infamous applet sandbox doesn't exist in a standard Java application unless, of course, it is loaded into the application via Java's SecurityManager. This in turn means that all of those things inaccessible to Java applets is freely available to download extension code.

Additionally, the loading code doesn't come from an HTML page, so no web browser is required to execute the application. This in turn means that the loading application remains independent of web servers, HTTP, or HTML.

Example: HelloDownload

In this particular example, because not all readers will have access to a web server with which to test, we'll create a JAR file that in turn depends on one in a nonstandard location. In this case, we'll be trying to use code from the root directory of the C:\ drive on a PC.

To start, create and compile two simple Java classes:

```
// Download.java
//
public class Download
{
    public void sayHello()
    {
        System.out.println("Hello from Download");
    }
}
```

```
// HelloDownload.java
//
public class HelloDownload
{
    public static void main(String[] args)
    {
        Download dl = new Download();
        dl.sayHello();
    }
}
```

Overly simplistic, but the classes should prove the point. The idea is simple: HelloDownload depends on the class Download to run. Therefore, HelloDownload will be either an installed extension or an executable JAR file (we need to make this front-end a JAR file, as well), and will reference the Download.jar file in its Manifest file:

```
Manifest-Version: 1.0
Created-By: JavaGeeks.com
Class-Path: C:/Download.jar
```
[3]

Create the HelloDownload.jar file with the Manifest file named manifest by specifying the name of the Manifest file on the jar utility command-line:

```
jar cvfm HelloDownload.jar manifest HelloDownload.class
```

Create the Download.jar file in the normal fashion:

```
jar cvf Download.jar Download.class version
```

Copy the Download.jar file to the root directory of the C:\ drive, and the HelloDownload.jar (renaming it to com.javageeks.HelloDownload.jar, if you wish) to the Extension directory. Change directory to someplace other than the current directory, so as to make sure we're not picking up the code in the current directory, and execute:

```
java HelloDownload
```

Given a working JDK 1.2 installation, you should see the "Hello from Download" message on your console window.

The Class-Path header can be a file-relative path or a standard HTTP URL. If you have a web server, change the location of the Download.jar to be a location off your web root, change the `Class-Path` in the manifest file to be that URL, rebuild the HelloDownload.jar file, and try running it. Because ExtClassLoader extends URLClassLoader, any given URL type—file, http, or ftp—are all viable candidates in the Class-Path tag.

4.2 IMPLICATIONS OF THE EXTENSIONS MECHANISM

Using Java extensions carries implications that may or may not be immediately obvious.

4.2.1 Distributed libraries through download extensions

One of the problems with building applications using a dynamic linking mechanism is the inevitable necessity of upgrading the libraries which support the application. If an application uses library "X," there will undoubtedly be other applications also using it, and a subsequent version of one of these libraries may in turn require a new version of the "X" library. Getting this out to all the users of the application can be a much more difficult problem, for the same reasons as those making it difficult to distribute the application in the first place. This becomes even more of an issue when libraries in turn use other libraries. Suddenly, there's an entire tree of dependencies.

The download extension mechanism offers one practical solution to this problem. By marking the .jar files that a library or an application uses, any updates to a dependent

[3] Readers running the example on a UNIX installation will need to change the Class-Path line in the manifest file to read "/Download.jar" or "/~/Download.jar" instead of "C:/Download.jar". There's nothing magical about the root directory; any directory on the file system can be used.

library can be picked up automatically. Two options are now possible—if maximum performance is desired, system administrators can manually copy new versions of the library down to end-users' machines, or a stand-alone daemon process on the end-users' system can check (at startup or every twenty-four hours, or any other practical time) the current versions of its .jar files against a central repository. Alternatively, the extension can use http URLs, and pull them as necessary from the same centralized repository. (Both approaches could be used simultaneously, as best benefits each individual application.)

The one drawback to this approach is that download extensions cannot load native library code. Typically, however, on end-user systems, native code will be less attractive due to the higher administrative support necessary to make it work, especially in heterogeneous networks. In those rare situations where native code needs to be moved to each end-user's workstation, the version-checking download daemon process can pull both .jar files and native code at the same time.

4.2.2 Java EXEs; relation to C++ static linking

The ability of the Java 2 interpreter to execute .jar files directly also makes possible the ability to create stand-alone java executable files, .jar files, that contain all of the necessary .class files to execute a given application. Recall from the start of chapter 2 one of the disadvantages of dynamic linking: an application that uses dynamic linking will always be vulnerable to upgrades of the classes on which it depends. In the C++ environment, this can be avoided by linking all referenced code statically, as part of the compiled executable, so that the necessary dependent code travels with and is never upgraded by a dynamic library upgrade.

This sort of static linking carries another, more practical benefit, in that many popular web browsers do not support more than one .jar file in the `<APPLET>` tag. Because of this, attempting to keep the application's code physically separate from the support code it uses will yield unworkable results when that applet is viewed from a nonconforming browser. Instead, by packaging the entire codebase into a single .jar, that file can be placed on the HTTP server and referenced from the web page. True, it means all of the code must be downloaded each time, and that this may not be a trivial amount of data; however, in this case, only if the web browser caches the downloaded .jar file will any time savings be realized, since the necessary classes will need to be downloaded at least once. By static-linking the .jar file, only those classes used by the application, and not any extraneous code, are downloaded.

Performing this sort of static linking is not pain-free. While it may be a simple matter to identify which code written by the developer needs to be deployed as part of this stand-alone application .jar, doing the same for the Java run-time library[4] or

[4] While this may seem overzealous, it actually helps when trying to deal with different Web browsers implementing different versions of the JDK. For example, most Web browsers aren't JDK 1.2-compliant, and most only supported up to about JDK 1.1.6 or so. Because JDK 1.2 introduced a number of classes not found within JDK 1.1.6, such as the CORBA org.omg.* classes, any CORBA-using applet needs to have those along for the ride.

third-party libraries used by the application can be another thing altogether. To go along with this, code and any resources (graphics, sounds, resource bundles, and so forth) used by the application need to be stored within the .jar file.

Because Java stores any classes used by a particular class within the class' compiled bytecode format, as Class entries in the class's constant pool, we could create an automated tool to scan a particular class' compiled bytecode, pick out all the Class entries found there, and perform the same scan recursively. Such tools exist already, many of which can be found within the Open Source community. This list-of-classes can then be fed into the Sun jar utility to build the .jar file directly.

4.3 PACKAGING EXTENSIONS

If extensions provide an easy path for reusable components and component libraries, then it's natural to make GJAS (as well as other components we develop along the way) an extension. Unfortunately, while parts of GJAS migrate very easily to the extension architecture, the nature of Java's ClassLoader architecture requires additional complexity within the GJAS codebase. Since the extensions' ClassLoader is unavailable for modification or separate instantiation, we need to make sure that any `Service`s loaded by GJAS are first loaded by ClassLoaders other than the extensions' ClassLoader unless all other avenues have been played out. The ClassLoader relationship to our Services is illustrated in figure 4.1.

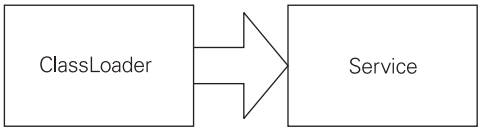

Figure 4.1 ClassLoader-to-classes relationship

To start, the stand-alone components can be bundled up into packages and used independently of the GJAS architecture. This includes the ClassLoader components (com.javageeks.classloader), the thread components (com.javageeks.thread), and socket clients (com.javageeks.client) developed along the way.[5] Because these components will not need to use the change-on-the-fly mechanism ClassLoaders provide and GJAS takes advantage of, we have no problems with storing them as extensions.

The same is true of the Service, Server, and ServerManager classes, the core parts of GJAS itself. Correspondingly, this means that any upgrade of GJAS will require taking down the GJAS process, updating the codebase, and restarting the process. Should developers require the ability to upgrade the GJAS components, then GJAS (or any other component that requires on-the-fly upgrading) cannot be stored in extensions, and will probably want to make use of some other mechanism for easy distribution.

---

[5] The .jar file is created in the "Lib" directory on the publisher's web site; see the makefile there for details on the specifics of how these files are created and stored.

In the source tree on the web site, the entire "com.javageeks" is packaged into a single .jar file. This may not be desirable in large-scale Java applications, since an upgrade to any of the contained packages requires the replacement of the entire .jar file. Instead, each package could be broken out into separate .jar files, with dependencies on other .jar files labeled as download extensions, and upgraded individually as necessary. This approach offers more flexibility in terms of piecemeal upgrades, but sacrifices development ease; developers must now track each "library" separately. This also requires separate versioning of each jar, and some greater testing to verify that various versions of each "library" work together.

4.3.1 The build-time vs. run-time dilemma

Unfortunately, this isn't the only tension between the development and deployment environments. Because Java is both a build-time and run-time interpretive system, it makes no inherent distinction between run time and build time. This seems like double-talk, without further explanation.

One of the first things a Java developer learns is that if the CLASSPATH isn't set to include all of the classes used by an application, the code won't compile. For example, unless the JSDT classes are on the CLASSPATH,[6] any code containing even an `import` statement will not compile.

The reason is simple: the javac compiler is actually implemented in Java, and it uses the CLASSPATH to find the classes to which a particular source file refers in order to carry out its compile-time type checking. The javac compiler, in fact, is a simple wrapper around the class sun.tools.javac.Main, and can be invoked using java sun.tools.javac.Main, assuming the JDK 1.2 tools.jar file is on the CLASSPATH.

All of this doesn't seem to have any relevance, at least not until we get into the build time versus run time dilemma. There will be occasions, within development, when a developer needs to have both a build-time environment and a run-time environment on his/her machine. The classic case is with GJAS itself—even though we need the Service classes we'll be building to be available at build time, we don't want them to be stored in the Extensions directory at run time. If they're on the CLASSPATH or in the Extensions directory, the system ClassLoader (AppClassLoader or ExtClassLoader) will pick up the classes instead of our new ClassLoader instance, and we won't be able to do the load-new-code-on-the-fly trick demonstrated in the last chapter. If the code is stored on the CLASSPATH or in the Extensions directory, testing may be adversely affected, as in the case of GJAS.

Fortunately, this situation arises only on developers' machines, since only developers require both the build-time and run-time environments. Neither the testing nor the production environments require the build-time classes, since they'll be picked up by the individual ClassLoader instances and not by the system loader. Fortunately, as well, most developers won't be faced with this situation, since most developers won't

[6] Or in the Extensions directory.

be facing this sort of situation (where classes need to be picked up by a custom ClassLoader and *not* the system Loader).

Unfortunately, when working with an application server like GJAS, developers will run into this situation head on. One solution is to use multiple JDK environments, one CLASSPATH/extensions setup for compilation, and another for testing/execution. For example, the developer can install the JDK under C:\JDK1.2, and install a standalone JRE under C:\JRE1.2. The developer then runs two distinct Command Prompt shells, one with PATH and CLASSPATH set to the JDK for compilation, and the other with PATH and CLASSPATH set to point to the JRE.

This is awkward for a number of reasons. First, any code compiled within the first shell must be transferred to the second shell's CLASSPATH or extension setup. This can be as simple as specifying a "-d <directory>" option to javac when compiling, but can easily be forgotten or mismatched if the build process isn't completely automated. Secondly, it's often difficult to maintain two separate clean environments, especially if the application uses files or other environment variables, some of which may need to be stored within the Win32 Registry (or other OS-specific centralized storage). This typically isn't too much of an issue since most of these supports are run-time related, not build-time. Lastly, it's not uncommon for developers to get confused, and run the tests from within the wrong shell, and get back results they don't expect.

This build-time/run-time dilemma doesn't rear its ugly head too often, since it only occurs when the multiple-code-loading mechanism needs to be in effect. Within a developer's test arena, once that mechanism has been proven, then all testing is typically geared against one Service class, and not a whole host at once within the same run. For that reason, developers can usually keep the same CLASSPATH for both compilation and testing, and simply know that the code will get picked up by the App- or ExtClassLoader, and not their own custom version.

4.4 THE PLUG-IN

One of the interesting aspects of .jar files is their growing service as the level of atomicity for black-box components. For example, EJB defines a Bean as a .jar, the Servlet 2.2 specification talks about Web-apps being bundled into .jars (with the extension .war), and the Java2 Enterprise Edition specification uses the same approach. On top of all this, as we've seen, the Sun interpreter will examine a .jar's Manifest file for the `Main-Class` attribute for the class name to execute when given a -jar argument to the JVM. If .jar files are going to become the *de facto* standard for Java deployment, certainly we can make use of it, as well.

As we'll see in a moment, allowing end users the ability to drop in new black-box components gives your code tremendous flexibility. Consider a traditional client/server reporting/data-viewing application. Under traditional development approaches, we might code each report or view as a separate class, linking them all into a single .exe

(or .jar), and distributing that to the user. Each time a new report or view was required, we'd have to re-release a new .exe/.jar.

Under an extensible-system approach, however, we'd instead create a basic interface that report or view classes must implement. Instead of building the code into a single .jar file, the application would be a simple shell which in turn looked into a subdirectory (or other location) for the .jar files representing each report type. The user could then pick from a list of the reports found, and the application shell would load the code from that .jar file. If a new report were required, we'd simply code up the new .jar file, and either distribute that to the users, or have the IT staff distribute it via some other form of push to the end-user's machines. Numerous advantages abound:

- *Testing is simpler.*
  Because the existing application shell hasn't been touched, that code doesn't need to be retested before releasing the new report. Your QA department will like you better if they don't have to retest the entire application every other week and your customers will like you even better because of a faster release cycle.
- *Development can be "parallelized."*
  Individual developers (perhaps more junior than would otherwise be required) can be given tasks that involve writing the individual reports. Work can proceed in a more parallel fashion, potentially speeding up the release cycle. In addition, the junior developers won't be able to get into the application shell code where they might introduce additional bugs or violate the basic application design.
- *Promotes encapsulation.*
  If the only way the report can interact with the application is through this well-defined API, then the application knows nothing about the internals of the report, and vice-versa. This promotes encapsulation and allows later maintenance to take place without concern for what else might break.
- *Power-user flexibility.*
  If you happen to have a user who is more technically knowledgeable than his/her peers, he/she can be given the API documentation to allow creation of their own reports without having to bother the developers.

In short, by allowing this kind of drop-in flexibility in your applications, you allow the users to be better served.

4.4.1 The plug-in concept

A class, when loaded, registers itself with some sort of manager which is responsible for calling on the registered class instances when applicable. Usually, in order to support type-safety (and avoid having to use Reflection to discover the plug-in's methods), the plug-in class will implement a common interface that defines the basic behavior required of each plug-in class.

As an example, consider a scripting engine/interpreter. In order to maximize the interpreter's flexibility, we want to allow the engine to interpret different languages

based on the script file's extension—.js for JavaScript, .vbs for VBScript, and so on. Each language-interpreter class will implement a basic LanguageInterpreter interface, which will look like this:

```
public interface LanguageInterpreter
{
    public boolean canInterpret(String filename);
    public int interpret(String filename, String[] args)
        throws Exception;
}
```

(The `throws` declaration is just a cheap way to allow the LanguageInterpreter-implementation class to pass exceptions back to the engine; a production-level application should define more clear-cut exception types, such as SyntaxException, ExecutionException, etc.) The first method, `canInterpret`, is called to see if the LanguageInterpreter-implementation class can, in fact, interpret the given script file. This allows a single LanguageInterpreter-implementation to support more than one scripting language. The second method, `interpret`, is where the LanguageInterpreter-implementation does the actual work of parsing and executing the script file.

Having done this, the ScriptingEngine class becomes ridiculously straightforward. When told to execute a file, it simply iterates through its list of LanguageInterpreters, asking each if it can interpret the file, and if so, orders it to do so. We define the ScriptingEngine class as:

```
public class ScriptingEngine
{
    private LanguageInterpreter[] interps;
        // How this is populated is explained later

    public int interpret(String scriptFile, String[] args)
    {
        for (int i=0; i<interps.length; i++)
        {
            if (interp[i].canInterpret(scriptFile))
                return interp[i].interpret(scriptFile, args);
        }

        return –1; // Nobody recognized it
    }

    public static void main(String[] args)
    {
        ScriptingEngine engine = new ScriptingEngine();
        engine.interpret(args[0], args);
    }
}
```

The ScriptingEngine is trivial; the only question mark comes in regard to the array of LanguageInterpreter instances, `interps`. How does it get initialized?

Conventional design would have each LanguageInterpreter-implementation class defined and stored within the application, and the array initialized within the `ScriptingEngine` code as follows:

```
public ScriptingEngine
{
    private LanguageInterpreter[] interps =
    {
        new JavaScriptInterpreter(),
        new VBScriptInterpreter(),
        new REXXInterpreter()
    };
}
```

Unfortunately, this means that ScriptingEngine now has the sum total of all languages supported by the engine, and cannot be reconfigured at run time to accommodate new languages. This means that if we need to support a new language, we have to ship out an entirely new application. Ick.

Alternatively, we could provide a properties file that the ScriptingEngine examines, parses, and executes `Class.forName()` on each line:

```
# languages.properties file
JavaScriptInterpreter
VBScriptInterpreter
REXXInterpreter
```

Then, the ScriptingEngine parses this languages.properties file (which, presumably, is stored on the user's hard disk) to establish which languages the engine knows about:

```
public ScriptingEngine
{
    private LanguageInterpreter[] interps;

    static
    {
        // Open languages.properties
        // For each line, call Class.forName().newInstance() and
        // store it into the returned array
    }
}
```

While attractive, this approach suffers from one critical flaw: if the languages.properties file is corrupted, deleted, or otherwise rendered unusable, the `ScriptingEngine` is paralyzed. Now it knows about no languages, and will fail every script file handed to it. There must be a better way.

4.4.2 Enter plug-ins

What we really want is for each intalled language interpreter to register itself with the scripting engine. Ideally, this registration (which takes place when we initialize the ScriptingEngine with the array of LanguageInterpreter instances) would be

code-independent, so that users could add new LanguageInterpreters without having to modify code.

This approach isn't a new one. For example, Adobe Photoshop uses this notion of plug-ins extensively, and even built an industry (dominated mostly by Kai's Power Tools) around plug-ins for Photoshop. OLE began life looking to do this sort of plug-in capability, as well, by providing interfaces that allowed those objects to place themselves on the menu bar, provide context-sensitive help, and more. The Emacs text editor is perhaps the crowning glory of this concept, with plug-ins ranging from email clients to full-fledged development-and-debugger modules for just about any language. Jeff Nelson, in his book *Programming Mobile Objects in Java*, shows how even mobile objects can participate in this sort of extend-the-app process by having the extensions download themselves into a text editor.

In a C++ environment, with an operating system that supports shared libraries, we can iterate through a directory that we designate as a plug-in directory, and explicitly load each library into the process' address space. Because each OS provides a method that is called when the shared library is loaded into the process space (DllMain or DllEntryPoint under Win32, for example), the LanguageInterpreter instance can be registered with the ScriptingEngine within this method.

Within Java, however, we have a few hangups. Because Java is already a dynamic-loading system, we don't have to build a custom approach for each platform—the ClassLoading mechanism is already there and in place. Unfortunately, that's the only part that Java gives us; the rest gets tricky.

Remember that one of the Java ClassLoading buzzwords is lazy. This means that even if a .jar file or directory containing .class files is specified in the user's CLASSPATH, the classes stored within that .jar or directory aren't loaded until the system needs the class. Recall, also, that needing a class comes when another class depends on the class in question, or the class is explicitly loaded using `Class.forName` or ClassLoader.loadClass.

In the case of our ScriptingEngine, we could get each LanguageInterpreter to register itself with the ScriptingEngine as follows:

```
public class ScriptingEngine
{
    // Everything else, as before

    private static List interps = new Vector();
    public static void register(LanguageInterpreter interp)
    {
        interps.add(interp);
    }

    public int interpret(String scriptFile, String[] args)
    {
        for (Iterator i = interps.iterator(); i.hasNext(); )
        {
            LanguageInterpreter interp =
```

```
                (LanguageInterpreter)i.next();
            if (interp.canInterpret(scriptFile))
                return interp.interpret(scriptFile, args);
        }

        return –1; // Nobody recognized it
    }
}
```

Now, all we need to do is get each LanguageInterpreter to register an instance of itself with the ScriptingEngine. Usually, this means that the ScriptingEngine (or, more generically, the plug-in manager, where the LanguageInterpreter is the plug-in) is a Singleton, or else uses a static list of plug-ins, as demonstrated in the code snippet above. Within the LanguageInterpreter-derived classes, one of two approaches can be used: either register the instance in a base class,

```
public abstract class LanguageInterpreterBase
    implements LanguageInterpreter
{
    public LanguageInterpreterBase()
    {
        // ... other initialization, as necessary

        ScriptingEngine.register(this);
    }
}
```

or the derived class can register an instance of itself in a static initializer block:

```
public class PerlInterpreter
    implements LanguageInterpreter
{
    static
    {
        ScriptingEngine.register(new PerlInterpreter());
    }
}
```

I prefer the second approach, since the first approach requires that the class in question must be loaded, and then a new instance of it created, before the registration with the plug-in manager takes place. In the second approach, the registration takes place as soon as the class (PerlInterpreter, in this case) is loaded into the JVM.

Furthermore, if a single plug-in can handle more than one type of call, the plug-in's static initializer block can make as many registrations as necessary:

```
public class ShellInterpreter
    implements LanguageInterpreter
{
    static
    {
        ScriptingEngine.register(new ShellInterpreter(), ".bat");
        ScriptingEngine.register(new ShellInterpreter(), ".cmd");
```

```
        ScriptingEngine.register(new ShellInterpreter(), ".sh");
        // ... and so on
    }
}
```

In this way, we're preserving the encapsulation of the plug-in by not having to know anything about what needs to happen to register it with its manager—the plug-in does that as soon as it's loaded into the VM.

If we designate a given directory into which plug-ins must be dropped in order to be loaded, we're going to run into two problems in short order. Remember that Java .class files are stored in directories corresponding to package names, so if we want to allow plug-ins to be packaged like other Java classes, we have to recursively scan through all directories under our plug-in directory.

The greater problem is that most plug-ins of a nontrivial nature are going to use more than one .class file to implement their behavior. Unfortunately, when they're all stored in the same directory, we're not going to know which ones are the plug-in class, and which ones are the supporting class. As a result, we'll have to load each and every one of them—whether or not they'll be used—into the VM. This violates one of the basic precepts of Java's ClassLoading mechanism—if you don't use it, it never gets loaded. It also means a huge performance hit as each and every one of those classes is loaded at plug-in registration time.

If, on the other hand, we require the plug-ins to come in a .jar or .zip file, we have another option.

4.4.3 Marking a .jar file as a plug-in

One of the little-known facts about .jar files (or their ancestors, the .zip file) is that every class used to open, examine, retrieve, and create a .jar file is already part of the JDK run-time library. The `java.util.zip` and `java.util.jar` packages contain all of the code used by the jar utility and the java.net.URLClassLoader class. To examine the contents of a .jar's Manifest file, it's as simple as the following:

```
import java.io.*;
import java.util.*;
import java.util.jar.*;
import java.util.zip.*;

public class JarLister
{
    public static void main (String args[])
        throws Exception
    {
        JarInputStream fin =
            new JarInputStream(new FileInputStream(args[0]));
```

We need to open the .jar file, so we use the JarInputStream class, which, like all Java stream classes, decorates (as in the Decorator pattern sense) another InputStream, which in this case will be a FileInputStream.

```
        Manifest manifest = fin.getManifest();
        if (manifest != null)
        {
```

Next, we obtain the .jar's "META-INF/MANIFEST.MF" file, if it exists. Note that not all .jar files have a Manifest file, since .zip files are technically .jar files and many, if not all, .zip files created before the release of JDK 1.1 (and many long after that) didn't have a Manifest file. Hence, we have to check for a `null` return value from `getManifest`.

```
            Attributes attribs = manifest.getMainAttributes();
```

Attributes is the class representing the attributes that can be attached to either the .jar file or each of the entries within it. By calling `getMainAttributes`, we're asking for the attributes that apply to the .jar file itself (such as the Main-Class or Created-By attributes discussed earlier).

```
            Set set = attribs.keySet();
            for (Iterator i = set.iterator(); i.hasNext(); )
            {
                Attributes.Name key = (Attributes.Name)i.next();
                System.out.println(key + ": " +
                                   attribs.getValue(key));
            }
        }
    }
}
```

And, as you might guess, the last block of code iterates through each of the entries in the Attributes object, printing each one out to the console. Note that the Iterator returned from the Set obtained from the Attributes object is not iterating over String objects, but instead over Attributes.Name objects. If you attempt to cast the returned object from the Iterator to a String, you'll get a ClassCastException.

Now that we know how to get the attributes of the .jar file's manifest, we can introduce our own custom .jar tags. We'll create a custom tag within the manifest, Plugin-Class, that contains the class name (fully qualified) of the plug-in class itself. Then the PluginClassLoader only needs to find this attribute, get the name of the class, and do a `ClassLoader.loadClass` using that value. This will load the plug-in class, which will fire off the plug-in's static initializer block(s), which will in turn register the plug-in with its manager.

4.4.4 PluginClassLoader

The code for PluginClassLoader, from the com.javageeks.classloader package, follows:

```
package com.javageeks.classloader;

import java.io.*;
import java.net.*;
import java.util.*;
import java.util.jar.*;
```

```
/**
 * PluginClassLoader is not an actual ClassLoader, but serves a role
 * of preloading "plugin" classes into the JVM, so that the Plugins
 * can register themselves with whatever "plugin manager" they use.
 *
 * See Chapter 4 of Server-Side Java for a detailed
 * description of how it all works together.
 */
public class PluginClassLoader
{
    /**
     * Interface to allow interested clients to be notified each
     * time a new plugin class is loaded into the JVM.
     */
    public static interface Listener
    {
        public void pluginLoaded(String pluginName);
        public void exception(Exception ex);
    }
```

Listener is simply an interface that allows interested parties, when they construct the PluginClassLoader, to be called back when a new plug-in is loaded. This allows GUIs, for example, to display a status bar that flashes "loading plugin XYZ…" to the user while starting up. The `exception` method is called when an exception is thrown during the load-up process.

```
    // Private data
    //
    private URLClassLoader urlClassLoader;
```

We use a URLClassLoader to load the classes from the list of .jar files we'll be building later in the code because it already has that functionality built within it. By not extending URLClassLoader, and instead containing an instance of it, we can also drop the URLClassLoader (and, implicitly, any classes loaded by it if they're not referenced elsewhere) and reload the plug-ins.

```
    /**
     *
     */
    public PluginClassLoader(String dir)
    {
        this(dir, new Listener()
        {
            public void pluginLoaded(String pluginName) { }
            public void exception(Exception ex) { }
        });
    }
    /**
     *
     */
    public PluginClassLoader(String dir, Listener listener)
    {
```

```
        File file = new File(dir);
        reload(file, listener);
    }
    /**
     *
     */
    public PluginClassLoader(File dir)
    {
        this(dir, new Listener()
        {
            public void pluginLoaded(String pluginName) { }
            public void exception(Exception ex) { }
        });
    }
    /**
     *
     */
    public PluginClassLoader(File dir, Listener listener)
    {
        reload(dir, listener);
    }
```

These four constructors are really just convenience wrappers around the `reload` method. Where no Listener is passed in, the constructor builds a NullObject[7] Listener, which does nothing when called on; that way, the actual implementation in `reload` needs never to check for a null Listener object, and can call on it without worrying.

```
    /**
     * Reload the plug ins; note that the old URLClassLoader held
     * internally is released, so if the plug-in classes loaded
     * earlier aren't in use within the app, they'll get GC'ed.
     *
     * HOWEVER, if an instance of an earlier-loaded
     * plugin class is still in existence, it will remain an
     * entirely separate and distinct type from the type loaded
     * in on this plass, even if the .class files are identical!
     * This is because classes loaded into two separate (non-
     * parentally-related) ClassLoaders are considered separate
     * and unrelated types, even if their contents are identical.
     */
    public void reload(String dir, Listener listener)
    {
        reload(new File(dir), listener);
    }
    /**
     * Reload the plugins; note that the old URLClassLoader held
     * internally is released, so if the plugin classes loaded
     * earlier aren't in use within the app, they'll get GC'ed.
     *
```

[7] This is called the NullObject pattern (*Pattern Languages of Program Design 3*, p. 5).

```
     * HOWEVER, if an instance of an earlier-loaded
     * plugin class is still in existence, it will remain an
     * entirely separate and distinct type from the type loaded
     * in on this plass, even if the .class files are identical!
     * This is because classes loaded into two separate (non-
     * parentally related) ClassLoaders are considered separate
     * and unrelated types, even if their contents are identical.
     */
    public void reload(File dir, Listener listener)
    {
```

The `reload` method is the heart-and-soul of the entire PluginClassLoader, so we'll take it in easy chunks.

```
        String[] contents = getPluginDirContents(dir);
```

The `getPluginDirContents` method simply obtains a list of all the .jar and .zip files in the directory specified by the File object dir. As we'll see later, it guarantees that it will always return a String array of some length, even if that length is zero, so no null-check is necessary.

```
        Vector urls = new Vector();
        Vector plugins = new Vector();
        for (int i=0; i<contents.length; i++)
        {
            try
            {
                File jarFile = new File(dir, contents[i]);

                Attributes attribs =
                    new JarFile(jarFile).getManifest().getMainAttributes();

                if (attribs.getValue("Plugin-Class") != null)
                {
                    String pluginClass =
                        attribs.getValue("Plugin-Class");

                    urls.add(jarFile.toURL());
                    plugins.add(pluginClass.trim());
                        // Need the trim(); getValue() has the
                        // annoying habit of leaving a trailing
                        // space on the end of the class, which will
                        // cause the loadClass() to fail later.
                }
            }
            catch (IOException ioEx)
            {
                // Just continue; ignore the file and move on
            }
            catch (NullPointerException npEx)
            {
                // No manifest, perhaps?
            }
        }
```

This seemingly complex piece of code is doing one thing: checking each .jar/.zip file for that Plugin-Class manifest entry we talked about earlier. If it's found, we add the URL of the .jar/.zip file to the Vector `urls`, and the value of the Plugin-Class attribute to the Vector `plugins`. We need the URL of the .jar/.zip file to pass into the URLClassLoader constructor, and we'll need the name of the class so that we can preload it into the JVM (which will force it to register with the rest of the system).

```
        urlClassLoader =
            URLClassLoader.newInstance(
                convertUrlVectorToArray(urls),
                getClass().getClassLoader());
```

This is simply another way of calling a new `URLClassLoader(…)`. The `convertUrlVectorToArray` method is a convenience method to convert the Vector `urls` to an array of URL objects, which is what URLClassLoader expects. Notice also how we explicitly pass in the ClassLoader that loaded this (the PluginClassLoader) class as our delegating parent—again, this is because we want to preserve the parent-child ClassLoader relationship appropriately, as discussed in chapter 2.

```
        // Preload each of the plugins, giving them the chance to
        // register (in their static initializer block) with whatever
        // "PluginManager" they choose to.
        //
        for (int i=0; i<plugins.size(); i++)
        {
            String plugin = (String)plugins.elementAt(i);
            try
            {
                Class.forName(plugin, true, urlClassLoader);

                listener.pluginLoaded(plugin);
            }
            catch (Exception ex)
            {
                listener.exception(ex);
            }
        }
    }
```

Now that we've constructed the URLClassLoader around the Plugin-Class-marked .jar/.zip files, we need to load each plug-in class into the JVM, which in turn allows those classes, in a static-initializer block, to register instances of themselves with the appropriate plug-in manager. Notice, as the comment points out, that we have to call `newInstance` on the loaded class before it is loaded into the JVM; this requires that the plug-in has a default constructor that can be called by outside clients, or an Exception will be thrown.

```
    /**
     * Releases the handle on the URLClassLoader used internally;
     * this will have the effect of allowing all the plug in classes,
     * if not referenced anywhere else within the application, to be
```

```
     * GC'ed the next time GC takes place.
     */
    public void unload()
    {
        urlClassLoader = null;
    }

    /**
     * Returns a String array of filenames in the directory which are
     * potential plug-in files.
     *
     * @param dir The File object representing the directory to iterate
     *            through
     */
    private String[] getPluginDirContents(File dir)
    {
        // sanity-check--does the directory exist?
        if ( (!dir.exists()) ||
             (!dir.isDirectory()) )
        {
            return new String[0];
        }

        String[] contents = dir.list(new FilenameFilter()
        {
            public boolean accept(File dir, String name)
            {
                if (name.endsWith(".jar") ||
                    name.endsWith(".zip"))
                {
                    return true;
                }
                else
                    return false;
            }
        });
        return contents;
    }
    /**
     * Returns a String array of filenames in the directory which are
     * .class files.
     *
     * @param dir The File object representing the directory to iterate
     *            through
     */
    private String[] getPluginDirClasses(File dir)
    {
        String[] contents = dir.list(new FilenameFilter()
        {
            public boolean accept(File dir, String name)
            {
                if (name.endsWith(".class"))
                    return true;
```

```
                else
                    return false;
            }
        });
        return contents;
    }
    /**
     * Simple helper method to convert a Vector of URL objects into an
     * array of URL objects (required by URLClassLoader)
     */
    private URL[] convertUrlVectorToArray(Vector urls)
    {
        URL[] urlArray = new URL[urls.size()];
        for (int i=0; i<urlArray.length; i++)
        {
            urlArray[i] = (URL)urls.elementAt(i);
        }
        return urlArray;
    }

    /**
     * Test suite--just load whatever plugins happen to be in the
     * current directory.
     */
    public static void main(String[] args)
        throws Exception
    {
        PluginClassLoader pcl =
            new PluginClassLoader(".", new Listener ()
            {
                public void pluginLoaded(String pluginName)
                {
                    System.out.println(pluginName + " loaded.");
                }
                public void exception(Exception ex)
                {
                    System.out.println("Exception:");
                    ex.printStackTrace();
                }
            });
    }
}
```

The remainder of the code entails the convenience methods mentioned earlier, and a `main` method for testing. `Main` simply builds a PluginClassLoader on the current directory, where presumably a collection of some plug-in .jars can be found and loaded.[8]

[8] The Extensions directory contains three .jar files, PluginOne.jar, PluginTwo.jar, and PluginThree.jar, all of which register themselves with the PluginManager class; they simply spit a string to the console when they're registered, just to prove that they are, in fact, loaded and registered.

4.4.5 Example: PluginApp

Let's demonstrate the concept by building a simple, useless GUI application that can be extended by plug-in .jars; by itself, the application does absolutely nothing—it displays a File menu and a Help menu. The File menu has two options: Exit, which is self-explanatory, and Reload, which will call the PluginClassLoader's method to reload the plug-ins found; this will allow us to test PluginClassLoader's dynamic-reload capability. The Help menu has just one option, About.

There's not much to it. The code to produce this application, complete with plug-in support, is also not very large or complicated:

```
import java.awt.*;
import java.awt.event.*;
import java.util.Iterator;
import java.util.Vector;
import javax.swing.*;
import com.javageeks.classloader.PluginClassLoader;

/**
 *
 */
public class PluginApp
{
    // Private data
    //
    private JFrame frame;
    private static Vector plugins = new Vector();
    private transient boolean canQuit;
        // State variable used in method exit(); should be modified
        // *only* within that context and not used elsewhere.
```

These are the `private` data members of PluginApp; of these, only one is of real importance—`plugins` is the Vector of registered plug-ins that the application will use during its run. The `frame` object is the JFrame this application uses as its main window, and `canQuit` is a state variable used later.

```
    /**
     * Plug ins must implement this interface; the app will call
     * the plug in when appropriate.
     */
    public static interface Plugin
    {
        public void addToMenuBar(JMenuBar menu);
        public boolean canQuit();
    }
```

The Plugin interface, here, is the basic interface any of our sample plug-ins should use if they want to "hook into" this application—it defines two methods, `addToMenuBar`, which gives each plug-in a chance to add a menu item or menu to the application's menu bar, and `canQuit`, which gives each plug-in a chance to cancel a user's request to quit. (This is where the traditional "File is not saved—still quit?" message would go.)

```
/**
 * Plug ins make themselves known to the App by calling this
 * method.
 */
public static void registerPlugin(PluginApp.Plugin plugin)
{
    // Just keep a reference to it for future use
    plugins.add(plugin);
}

/**
 * This is an interface to ease calling across all the plug ins
 * in the system.
 */
protected static interface PluginAction
{
    public void action(Plugin plugin);
}
/**
 * General-purpose method for calling an action across all the
 * currently registered plugins.
 */
private void doPlugins(PluginAction pluginAction)
{
    for (Iterator iter = plugins.iterator(); iter.hasNext(); )
    {
        Plugin p = (Plugin)iter.next();
        pluginAction.action(p);
    }
}
```

This is a shorthand version for iterating across all plug-ins to do something. When we want to make a call across all the registered plug-ins on this application, we create an anonymous PluginAction class/object on the spot, and pass it into `doPlugins`; we'll see this used in just a bit.

```
/**
 *
 */
public PluginApp()
{
}

/**
 *
 */
public PluginClassLoader.Listener getPluginListener()
{
    return new PluginClassLoader.Listener()
    {
        public void pluginLoaded(String pluginName)
        {
            System.out.println(pluginName + " loaded.");
```

```
            }
            public void exception(Exception ex)
            {
                System.out.println("Exception:");
                ex.printStackTrace();
            }
        };
    }
```

This method creates the usual console-output PluginClassLoader.Listener that we've seen before. In a production-quality application, however, this is where you would update the splash screen or status bar with messages such as "Loading plug-in XYZ…."

```
    /**
     * Display the application
     */
    public void show()
    {
        frame = new JFrame("PluginApp Example");
        frame.addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent e)
            {
                exit();
            }
        });

        JPanel contentPanel = new JPanel();
        contentPanel.add("North", createMenubar());

        frame.getContentPane().add(contentPanel);

        frame.pack();
        frame.show();
    }
```

The `show` method is unremarkable, with one exception—the call to `createMenubar`, which will iterate across all the plug-ins asking them if they wish to modify the menu bar.

```
    /**
     *
     */
    public void exit()
    {
        canQuit = true;
        doPlugins(new PluginAction()
        {
            public void action(Plugin plugin)
            {
                if (plugin.canQuit() == false)
                {
                    canQuit = false;
```

```
            }
        }
    });

    if (canQuit)
    {
        System.exit(0);
    }
}
```

This is the first of two samples demonstrating how `doPlugins` works. We create an anonymous PluginAction class that calls each plug-in's `canQuit` method, setting the PluginApp state variable `canQuit` to `false` if any indicate that we can't quit yet. (Presumably this is the user telling us this, but perhaps we want to allow plug-ins the capability to prevent the user from quitting without performing some necessary task first.)

```
/**
 * Build the application-shell's menu bar; just "File" and "Help"
 */
private JMenuBar createMenubar()
{
    final JMenuBar mb = new JMenuBar();
    JMenu menu;
    JMenuItem mi;

    // "File"--"Reload"
    menu = new JMenu("File");
    mi = new JMenuItem("Reload");
    mi.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            pluginCL.reload(“.”,getPluginListener());
        }
    });
    menu.add(mi);

    // "File"--"Exit"
    mi = new JMenuItem("Exit");
    mi.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            exit();
        }
    });
    menu.add(mi);

    mb.add(menu);

    // "Help"--"About"
    menu = new JMenu("Help");
    mi = new JMenuItem("About");
    mi.addActionListener(new ActionListener()
```

```
        {
            public void actionPerformed(ActionEvent e)
            {
            }
        });
        menu.add(mi);
        mb.add(menu);

        // Allow the Plugins to register themselves
        doPlugins(new PluginAction()
        {
            public void action(Plugin plugin)
            {
                plugin.addToMenuBar(mb);
            }
        });

        return mb;
    }
```

Finally, the `createMenubar` method builds the JMenuBar instance that will be added to the application's main window. Notice, however, at the end of the method, that we iterate through each installed plug-in, calling on its `addToMenuBar` method (passing in the JMenuBar we just created). This is the mechanism by which the plug-ins can allow themselves to be invoked within this application; within other systems, plug-ins may be called with some discriminatory information to discern which plug-in to load (as in the scripting engine example above), or may simply be tried, round robin, until one is found that works.[9]

```
    /**
     *
     */
    public static void main (String args[])
    {
        // Create the basic app object
        PluginApp app = new PluginApp();

        // Display the app
        app.show();
    }
}
```

And `main`, of course, creates an instance of the application and invokes its `show` method.

Next, let's examine a simple example plug-in for this application:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

[9] This is what James O. Coplien called the "exemplar idiom"; *Advanced C++ Programming Styles and Idioms* (Addison-Wesley, 1992).

The usual necessary-for-Swing imports. Nothing new here.

```
public class PluginOne
    implements PluginApp.Plugin
{
    static
    {
        PluginApp.registerPlugin(new PluginOne());
    }
```

As discussed before, when PluginOne is loaded into the JVM, it registers an instance of itself with the `PluginApp` class.

```
    public PluginOne()
    { }

    public void addToMenuBar(JMenuBar menuBar)
    {
        System.out.println("addToMenuBar called");
        final JMenuBar menu = menuBar;

        // Put us into the "File" menu
        for (int i=0; i<menu.getMenuCount(); i++)
        {
            JMenu m = menu.getMenu(i);

            System.out.println(m.getText());
            if ("File".equals(m.getText()))
            {
                System.out.println("Found File menu; adding self");
                JMenuItem mi = new JMenuItem("PluginOne");
                mi.addActionListener(new ActionListener()
                {
                    public void actionPerformed(ActionEvent e)
                    {
                        int result = JOptionPane.showConfirmDialog(
                            null,
                            "Do you like PluginOne?",
                            "information",
                            JOptionPane.YES_NO_CANCEL_OPTION,
                            JOptionPane.INFORMATION_MESSAGE);
                        if (result == JOptionPane.YES_OPTION)
                        {
                            JOptionPane.showMessageDialog(
                                null, "I'm glad");
                        }
                        else if (result == JOptionPane.NO_OPTION)
                        {
                            JOptionPane.showMessageDialog(
                                null, "I'm sorry to hear that");
                        }
                        else if (result == JOptionPane.CANCEL_OPTION)
                        {
                            JOptionPane.showMessageDialog(
```

```
                                null, "Operation cancelled");
                        }
                        else
                        {
                            // How is this possible?!? Swing is broken!
                        }
                    }
                });
                m.add(mi);
            }

            break;
        }
    }
```

This long snippet of code is an exercise in Swing mechanics; for those who aren't Swing gurus, the code simply adds a menu item to the bottom of the File menu, called "PluginOne". When the user picks "PluginOne" from the File menu, a "Yes/No/Cancel" dialog box will be displayed, and a second dialog box will display "I'm glad," "I'm sorry to hear that," or "Operation canceled," depending on which button the user pressed. Nothing overly exciting here.

```
    public boolean canQuit()

    {
        System.out.println("PluginOne sez yes, you may quit");
        return true;
    }
}
```

Lastly, the `canQuit` method spits a message out to the console, informing us that PluginOne was given a chance to cancel the `File-Exit` command, and chose not to do so.

4.4.6 Uses for plug-ins

The plug-in concept can extend in many directions. As discussed earlier, a scripting language engine could use plug-ins as the interpreters of the various script languages it understands, allowing users to drop in support for new languages by simply copying in the appropriate script-language .jar file. A web server could support servlets in much the same way—instead of a Plugin-Class tag, requiring the .jar file to contain a Servlet-Class tag, indicating the Servlet class to load.[10] An application, as demonstrated above, could allow sophisticated end users to create additional functionality for the application. A graphics conversion (or any kind of file-conversion application, for that matter) can use plug-ins to manage each file format the application wants to handle, so long as there is a good interim format that can be handed between the formatters. Even games can make use of this concept. A basic card

[10] The Java2 Enterprise Edition specification uses XML "Deployment Descriptors" instead of attributes in the .jar file, but it's the same concept.

game shell can implement the rules of various card games (cribbage, gin rummy, and poker) as plug-ins loaded when the game shell starts up.

The plug-in concept represents a good marketing strategy, as well—customers can be given the basic application shell for free (available for download, for example), with a simple demo as their only available plug-in. Then, as customers begin using the application and demand greater functionality, more powerful plug-ins can be made available, which customers buy and copy into the application's plug-ins directory. This approach has the advantage of giving the user a free, non-timing-out version of the application that may be good for lightweight use, but requires purchase for heavier use. Customers who require support outside of the existing realm of plug-ins can contact the company for a custom plug-in, which the company can then turn around and resell to other customers, as well.

4.5 SUMMARY

Developers would be well-advised to think of Java's extension mechanism as under the same rules as developing reusable libraries in other languages such as C++. Many of the same concepts, and trade-offs apply. For example, development of code without using libraries means the entire code base can be assumed to be the same version. Breaking up the code into separate, modular, libraries means now that each library, as well as each application, must be versioned, tracked, and tested against the entire application suite before it can be released. Using the library concept also means that developers will be restricted from wholesale replacement of components, since other applications may be dependent on the particular structure and/or usage of components in the library, which restricts developers.[11]

For all its drawbacks, the Java extension mechanism is the first step Java has shown toward building reusable component libraries and toolkits other than those shipped as part of the JDK. It may be argued that the .jar file was the critical step, but the modification of the CLASSPATH necessary to use a given .jar file made it awkward to use .jar files, especially when large numbers were used. CLASSPATHs over 500 characters long aren't uncommon when making use of a half-dozen .jar files at once, which is not unreasonable in any moderately-sized project. The Java extension mechanism makes the modification of the CLASSPATH almost completely unnecessary now.

[11] In an ideal world, each component would have its `public` interface fixed and immutable, but this is an unattainable target. As needs within the development team change, use of particular components grows, and initially acceptable and elegant designs grow more and more unworkable, and wholesale replacement of the design becomes necessary.