CHAPTER TWO

The Benefits of Modular Programming

2.1 Distributed Development

Nobody writes software entirely in-house anymore. Outside the world of embedded systems, almost everyone relies upon libraries and frameworks written by someone else. By using them, it is possible to concentrate on the actual logic of the application while reusing the infrastructure, frameworks, and libraries written and provided by others. Doing so shortens the time needed to develop software.

The rise of open source software over the past decade makes library reuse doubly compelling. For many kinds of programs there are existing solutions for various problems, and those solutions are available at zero monetary cost. The set of open source offerings starts with UNIX kernels, base C libraries, command-line utilities, and continues over Web servers and Web browsers to Java utilities such as Ant, Tomcat, JUnit, Javacc—ad infinitum. Writing modern software is as much a process of assembly as it is creation. Picking available pieces and composing them together is a large part of modern application development. Instead of writing everything from scratch, people who need an HTTP server for their application select Apache or Tomcat. Those who need a database could choose MySQL or PostgreSQL. The application glues these pieces together and adds its own logic. The result is a fully functional, performant application developed in remarkably little time.

Consider how Linux distributions work. RedHat's Fedora, Mandriva, SUSE, and Debian all contain largely the same applications, written by the same people. The distributor simply packages them and provides the "glue" to install them together. Distribution vendors often write only central management and installation software and provide some quality assurance to make sure all the selected components work well together. This process works well enough that Linux has grown considerably in popularity. As evidence of the meaningfulness of such a model, consider that Mac OS X is in fact a FreeBSD UNIX with a bunch of add-ons from Apple. The key thing to note is that the software in question is created through a *distributed development model*. The developers and distributors of the software may not even know or communicate with each other, and are usually not even in the same place geographically.

Such distributed development has specific characteristics. The first thing to notice is that the source code for the application (or operating system) is no longer under a developer's complete control. It is spread all over the world. Building such software is unquestionably different from building an application whose source code is entirely in your in-house repository.

The other thing to realize is that no one fully controls the schedule of the whole product. Not only the source code, but also the developers are spread all over the world and are working on their own schedules. Such a situation is not actually as unusual or dangerous as it sounds. Anyone who has tried to schedule a project with a team of more than fifty people knows that the idea of ever having "full control" over the process is at best a comforting illusion. You always have to be prepared to drop a feature or release an older version of one or another component. The same model works with distributed development.

The basic right everyone has is the freedom to use a newer or older version of a library.

The ability to use external libraries and compose applications out of them results in an ability to create more complex software with less time and work. The trade-off is the need to manage those libraries and ensure their compatibility. That is not a simple task. But there is no other practical, cost-efficient way to assemble systems of today's complexity.

2.2 Modular Applications

The technological solution to the challenges of distributed development is modularization. A modular application, in contrast to one monolithic chunk of tightly coupled code in which every unit may interface directly with any other, is composed of smaller, separated chunks of code that are well isolated. Those chunks can then be developed by separate teams with their own life cycles and their own schedules. The results can then be assembled together by a separate entity—the distributor.

It has long been possible to put a bunch of libraries on the Java classpath and run an application. The NetBeans Platform takes the management of libraries further—by actively taking part in the loading of libraries and enforcing that the minimum version of a library that another library uses is adequate. Such libraries are what we call *modules*. The NetBeans *Module System* is a *runtime container* that ensures the integrity of the system at runtime.

2.2.1 Versioning

Breaking an application into distinct libraries creates a new challenge—one needs to ensure that those independent parts really work together. There are many possible ways to do so. The most popular is *versioning*. Each piece of a modular application has a version number—usually a set of numbers in Dewey decimal format, such as `1.34.8`. When a new version is released, it has an increased version number, for example `1.34.10`, `1.35.1`, or `2.0`. If you think about it, the idea that an incremented version number can encode the difference between two versions of a complex piece of software is patently absurd. But it is simple to explain, and it works well enough that the practice is popular.

The other parts of a modular system can then declare their external dependencies. Most components will have some external requirements. For example, a component in a modular system might rely on an XML parser being present, or on some database driver being installed, or on a text editor or Web browser being present. For each of these, another module can request a specific minimum version of their interfaces. Even if the dependencies on external libraries are minimized, every program in Java depends on a version of Java itself. A true modular system should make it possible to specify the desired minimum

JDK version. A module could require `JDK >= 1.5`, `xmlparser >= 3.0`, and `webbrowser >= 1.5`. At runtime, the code responsible for starting the application must ensure that the requested dependencies are satisfied—that the XML parser is available in a version 3.0 or newer, the Web browser is in version 1.5 or higher, and so forth. The NetBeans Module System does that.

Using such dependency schemas to maintain dependencies between components in a modular system can work only if certain rules are obeyed. The first rule is *backward compatibility*—that if a new version is released, all contracts that worked in the previous version will work with the new one as well. This is easier to say than to achieve. Rule number two is that components of the system need to accurately say what they need. When a module's set of dependencies changes, it needs to say so, so that the system can accurately determine if they are satisfied. So if a piece of a modular system starts to rely on new functionality, such as an HTML editor, it needs to add a new dependency (e.g., `htmleditor >= 1.0`). And if you start to use a new interface to the HTML editor component—one which was only added in version 1.7 of the component—the dependency needs to be updated to require `htmleditor >= 1.7`. The NetBeans Module System makes this second part relatively simple in practice, since a module's compile-time classpath will only include modules it declares a dependency on. So unless the module's list of dependencies is updated, it will not compile.

2.2.2 Secondary Versioning Information

The versioning scheme just discussed refers to the *specification version* of a library. It describes a specific snapshot of the public APIs in that library.

It is a fact of life that some versions of libraries can contain bugs which must be worked around. For this reason, a secondary version identifier—an *implementation version*—should be associated with a component. In contrast to the specification version, this is usually a string like "Build20050611" which can only be tested for equality. This provides a secondary identifier that can be used to determine if a specific piece of code to work around a given bug is needed. The fact that a bug is present in (specification) version 3.1 does not mean it will also be in version 3.2 or even in a different build of 3.1. So, for reasons of bugfixing or special treatment of certain versions, associating an implementation version with a library can be useful.

2.2.3 Dependency Management

The system of versions and dependencies needs a manager that makes sure all requirements of every piece in the system are satisfied. Such a manager can check at each piece's install time that everything in the system remains consistent—this is how RPMs or Debian packages work in Linux distributions. Metadata about such dependencies is also useful at runtime. Such metadata makes it possible for an application to dynamically update its libraries without shutting down. It can also determine if the dependencies of a module it is asked to dynamically load can be satisfied—and if not, it can describe the problem to the user.

NetBeans IDE is a modular application. Its modules—its constituent libraries—are discovered and loaded at runtime. They can install various bits of functionality, such as components, menu items, or services; or they can run code during startup to initialize programmatically; or they can take advantage of declarative registration mechanisms that various parts of the platform and IDE offer to register services and initialize them on demand. The NetBeans Module System uses the declared dependencies of the installed components to set up the parent classloaders for each module's own classloader, determining what JARs will be searched when a module tries to load a class. This ensures that any one module's classpath excludes any module JARs which are not above it in its dependency tree and enforces the declared dependencies of each component—a module cannot call code in a foreign module unless it declares a dependency on that foreign module, so it will not be loaded at all if some of its dependencies cannot be satisfied.

2.3 A Modular Programming Manifesto

No one is surprised anymore that operating systems and distributions are designed in a modular way. The final product is assembled from independently developed components. Modularity is a mechanism to coordinate the work of many people around the world, manage interdependencies between their parts of the project, and assemble very complex systems in a reasonably reliable way.

The value of this approach is finally filtering down to the level of individual applications. Applications are getting more and more complicated, and they are increasingly assembled from pieces developed independently. But they still need to be reliable. Modular coding enables you to achieve and manage that complexity. Since applications are growing in size and functionality, it is necessary to separate them into individual pieces (whether you call them "components," "modules," or "plugins"). Each such separated piece then becomes one element of the modular architecture. Each piece should be isolated and should export and import well-defined interfaces.

Splitting an application into modules has benefits for software quality. It is not surprising that a monolithic piece of code, where every line in any source file can access any other source file, may become increasingly interconnected, unreadable, and ultimately unreliable. If you have worked in software for a few years, you have probably been on a project where there was some piece of code which everyone on the team was afraid to touch—where fixing one bug always seemed to create two new bugs. That is the entropy of software development. There is economic pressure to fix problems in the most expedient way possible—but the most expedient way is not necessarily in the long-term interest of the codebase. Modular software limits the risk of creeping coupledness by requiring that different components of the system interoperate through well-defined API contracts. It's not a silver bullet, but it makes it considerably harder to have the sort of decay that eventually dooms many complex pieces of software.

Comparing modular design and traditional object-oriented design is a lot like the comparisons of *structure programming* with *spaghetti code* from the 1960s. Spaghetti code was the name given to Fortran or BASIC programs where every line of code could use a `GOTO` statement to transfer execution to another place in the program. Such code tended to be written in such a chaotic way that often only the author of a program could understand the program's logic. Structured programming tried to reduce this disorder by introducing blocks of code: `for` loops, `while` loops, `if` statements, procedures, and calls to procedures. Indeed, this improved the situation and the readability and

maintainability of applications increased. If nothing else, one could be sure that a call to a method will return only once.[1]

The classic object-oriented style of programming in some ways resembles the situation before structured programming arrived. With the term "classic object-oriented style," we are referring to the style of programming typically taught today. It is also the sort of code you get from using UML tools: heavy use of inheritance, and almost everything overridable and public. In such an application, any method in any class may potentially call almost any method of any other class. Indeed there are `public`, `private`, and `protected` access modifiers, but the granularity of access permissions is done on the level of a single class or class member. That is far too low-level to serve as a basic building block of application design. Modularity is about the interaction between systems, rather than between small parts of subsystems.

Modular applications are composed of modules. One module is a collection of Java classes in Java packages. Some of these packages are public, and public classes in them serve as an exported API that other modules can call. Other classes are private and cannot be accessed from outside. Moreover, to be a module, a library must list its dependencies on its surrounding environment—other modules, the Java runtime, etc.

Inside a module, one can still apply bad coding practices, but the architecture of an application can be observed by checking the dependencies among all its modules. If one module does not have a dependency on another, then its classes cannot directly access the other module's classes. This keeps the architecture clean by preventing `GOTO`-like coding constructs that could otherwise couple completely unrelated parts of the codebase.

Sometimes people say that their application is too small for modular architecture to be applicable. It may indeed be so. But if it is beyond the level of a student project, then it is likely to evolve over time. As it evolves, it is likely to grow. And as it grows, it is very likely to face the "entropy of software" problem.

1. Except for boundary conditions where it may never return, throw an exception, etc.

The initial step in designing a complex application is to design its architecture. For this, it is necessary to define and understand the dependencies between parts of the application. It is much easier to do this in case of modular applications.

Thus it is always wise to start designing any application in a modular way. Doing so creates an infrastructure that will let you build more robust applications and avoid a great deal of manual bookkeeping. Rewriting messy, interconnected traditional object-oriented applications to give them a good modular design is a hard task. And it is not often that a project can afford the time it takes to be rewritten or rearchitected. Often, programmers have to live with old, monolithic code with ever-increasing maintenance costs—because the code is known to work.

Modular design starts you out in an environment where the architecture cannot slowly decay into unmaintainability without anyone noticing. If you create a new dependency between two parts of a modular application, you need to do some explicit gestures to set up that dependency. It cannot happen by accident. While that is not a cure for messy designs, it is an environment that encourages well-thought-out ones.

Modularity gives systems clearer design and control of module interdependencies; it also gives developers more flexibility in maintenance. Consider that when starting any new project—regardless of the project's initial scope. Modular design will have large benefits for the architecture of the entire application as it grows from its infancy. The real benefits of modular programming might not be apparent in the first version of an application. But they will become obvious later with the reduced cost of creating the 2.0 and 3.0 versions. Since modular programming does not add significant cost to creating the 1.0 version of an application, there is little reason not to use this approach on all projects. Many programmers are surprised (even sometimes horrified) to find something they wrote fifteen years ago still in use. Since we cannot predict the future of our code, we might as well architect it to last from the start.

2.4 Using NetBeans to Do Modular Programming

Creating a skeleton for a new NetBeans module is as easy as creating a plain old Java application project in the IDE. Just start NetBeans IDE, select **File | New Project**, and choose **NetBeans Plug-in Modules | Module Project**. Give the module a name and a location and you will end up with a brand-new project opened in the **Project** window. Create a Java class in the package that has been precreated for you. Then, in the IDE's Source Editor, use any the features of the JDK platform that you are running against. All the classes from the JDK are accessible and can be used without any special setup.

A slight difference compared to plain Java coding is the way one uses additional libraries. Instead of directly choosing a JAR file, one creates a *module dependency* on another module. To do this, open the **Libraries** subnode of your project and invoke the **Add Module Dependency** dialog. NetBeans contains a lot of modules and you can choose which ones to add as libraries. There are also many *library wrapper modules*, each presenting a third-party library as a standard NetBeans module. For example, there is Apache.org's Commons Logging library—just type `common`, select the library, and click **OK** (Figure 2.1). Now the editor knows how to use logging classes so you can safely type, for example, `org.apache.commons.logging.LogFactory.getLog("name.of.your.log")` and the result will be compilable. This may not look like a huge advantage over selecting the JAR file directly; however, notice that it was enough to specify just the identifying name of the module and the **Library Manager** dialog located it for us. Adding explicit dependencies to a module project is quite simple.

It appears that creating a single module and reusing existing resources is fairly easy. However, the power of modular applications is in having multiple modules with dependencies, so the question is, how to create such modular application inside NetBeans IDE? The answer is a *module suite*. A suite is a container for a set of modules which can see each other and communicate among themselves (Figure 2.2). To create a suite, choose **NetBeans Plug-in**

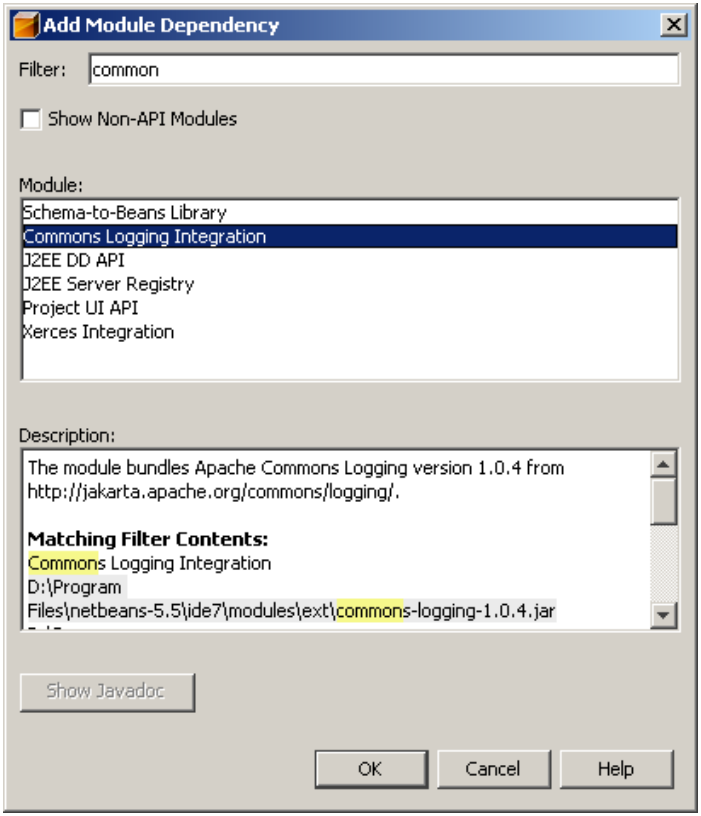

Figure 2.1: Adding a dependency on a module wrapping the Apache's Commons Logging library

Modules/Module Suite in the **New Project** wizard. Then follow the steps in the wizard. After clicking **OK**, a new project be visible in the **Projects** window. In contrast to the regular project, a suite does not have its own sources. It is merely a project to bind together other, interdependent NetBeans module projects. So, choose **Suite/Modules**, invoke a popup menu on that node, and, using the **Add Existing** menu item, add the previously created module into the suite.

A suite can contain more than one module. To demonstrate that, we can convert an existing library into a module in the suite. Choose **New Project**

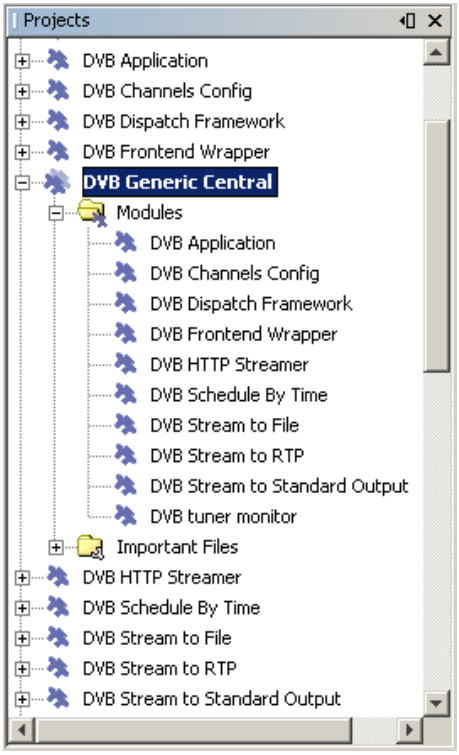

Figure 2.2: A suite can contain many modules

again and select **NetBeans Plug-in Modules/Library Wrapper Module Project**, choose some plain existing Java JAR file, such as an Apache library, and follow the steps to create its wrapper and make it part of the suite we just created. When done, the suite will show the two modules under the **Modules** node and it will be possible to create a dependency between them. Select the first module again, right-click it, and choose **Properties** from the popup menu. In the **Properties** dialog, click the **Libraries** category to show the UI for setting up dependencies. Add a dependency on the just added library wrapper module. When done, classes in the first module will be able to use classes from the library.

Modules vs. Plugins

One point of potential terminology confusion is the use of the terms "plugin" and "module." For most practical intents and purposes, there is no difference. A module is simply a unit of code that you can "plug in" to the platform or the IDE. The term "plugin" has been popularized by various other environments and can easily be applied to modules created for the NetBeans Platform as well.

Traditionally, we have used the term "module" in the NetBeans environment, since the platform itself is composed of modules. (You cannot say that the core platform is composed of "plugins.") On the other hand, if you are creating new features for the IDE or the platform but your feature set is composed of multiple modules, you might prefer to refer to those modules collectively as a single plugin.