CHAPTER 1

Introduction

Before entering the world of open source tools it makes sense to pause and consider what exactly constitutes a "tool" and what "open source" means. This will provide the reader with a context for examining these tools, and it may define ways to use them more effectively.

1.1 On Tools

When you think of a *tool,* you probably picture something like a hammer or a screwdriver—a device that aids in the process of turning raw materials into a finished product. This construction metaphor is useful because the ultimate goal is to make something new out of a collection of algorithms and ideas and Java classes. However, the strict sense of *tool* is applicable only to a Java compiler. To make the definition useful it will have to be expanded. For the purposes of this book a tool will be anything that makes development easier, regardless of how it does so. Applying this definition to the construction of physical objects suggests many new kinds of tools.

A screw, nail, or prebuilt component may be a tool. Certainly one would not want to undertake a project without them. The equivalent in Java terms might be individual classes.

A robotic assembly line greatly simplifies the process of repeatedly building new objects once the first has been built. The Java equivalent of an assembly line is

something like Ant (covered in Chapter 2), which automates the process of assembling classes and other data into an application.

An entire workshop can even be a tool by providing a place and resources in which to build new things. The Java equivalent is an integrated development environment, such as Eclipse, covered in Chapter 3.

Most manufacturing facilities have devices that test the finished products. These can be as simple as a chamber in which objects are repeatedly dropped in order to see how much stress they can take, or they can be as elaborate as automobile crash tests. Such facilities simplify development in a number of ways: by ensuring that individual pieces work, that they work together as expected, and that the result is stable and robust. Java, too, has testing tools, covered in Chapters 4 through 7.

Prefabricated components can also be tools. Someone building a remote-controlled car is more likely to buy an off-the-shelf motor than build one from scratch. It is even possible to build quite elaborate systems by simply connecting prebuilt components. Indeed, this is how most desktop computers are built. Java comes with many such components in the form of the core library, but there are many common needs that are not covered by this library. Chapters 8 through 16 introduce new components.

Finally, something like a breadboard can aid development by providing a framework on top of which a project will be built. Chapters 17 through 20 discuss frameworks for building Web applications.

1.2 On Open Source

The question of what "open source" really means is in many ways an obscure one. There is no universally recognized body that is empowered to define the term. There is not even global consensus that it is a useful term.

A first reasonable attempt at a definition is as follows: A product is open source if the full source code comes included with the product. To put it another way, a set of classes could be considered open if the .java files are provided along with the .class or .jar files.

This definition implies that, at any time, a working version of the product can be reconstructed by compiling the source files, just as the original provider would have done in preparing the class or jar files. The definitional subtleties arise when considering what else the recipient can do with the source code besides compiling it as-is. Is the recipient allowed to make changes to the source code and compile that? Is the recipient allowed to incorporate the source code into its own products? Is the recipient allowed to give original or modified source code to other people?

Each developer who decides to provide source code with their products is free to answer these questions as they see fit. Typically the answers are codified as a license to which the recipient must agree. Even the point at which the agreement happens is subject to variation. Some licenses must be accepted to use the software; others need only be accepted to redistribute the software.

Open source is more than a label that may or may not apply to a particular piece of software. It is also a way to think about development, a philosophy, and, perhaps, a movement. The definitions get even muddier here; a brief history should help clarify them. Please note that this history is very abbreviated, and each of the participants will likely tell it a little differently. See some of the Further Reading at the end of this chapter for more information.

In the early 1980s a developer at MIT named Richard Stallman was facing an ethical dilemma. He felt that if he liked a program he had a moral obligation to share it with other people who might also like it. This sharing would of necessity include not only the program itself but also the source. He has often compared the situation to a recipe: If someone likes certain cookies, they should be free to give some cookies to others to enjoy. But maybe the original recipe uses walnuts, and a lot of people are allergic to nuts. The recipe can be modified to omit nuts, and in the end a lot more people can enjoy the cookies.

The Unix system and tools he was using at the time did not have this property of being sharable. So he gathered a group of volunteers and started the Free Software Foundation (FSF), whose initial goal was the creation of a complete replacement for Unix, called GNU, that would be distributed in accordance with his ethics and ideology.

At no point was this described as "open source," because the openness of the source was a consequence, not a goal. Instead the term "free" was used, and as is often stressed, this refers to liberty, not price. A user of free software is permitted to do anything at all with that software, except restrict other users from doing the same. This idea was encoded as the GPL, the General Public License, under which all FSF code is released.

By 1991 a great many of the elements that comprise a complete Unix system had been completed. The one major piece still missing was the core of the system, also called the "kernel." The GNU project had been working on a kernel called the HURD, which would incorporate a number of cutting-edge concepts in operating system design. Today the HURD is usable, and it continues to be developed, but it was not nearly ready in 1991.

Around that time a programmer named Linus Torvalds developed his own kernel, which while initially not as cutting edge as the HURD, was stable and complete. Torvalds chose to release this kernel under the terms of the GPL, which enabled it to be used in conjunction with the other elements the GNU project had already

developed. The resulting complete system became widely known as Linux, although some, including Stallman, feel that the name GNU/Linux is more appropriate because everything but the kernel originated as part of GNU.[1]

The rise of the GNU/Linux system is nothing less than remarkable, and it is now replacing products from long-established companies like Sun and Microsoft in many places. However, much of its rise can be attributed to pragmatic rather than ideological reasons. Many believe that these reasons are directly attributable to the openness of the source. It is claimed that having the source available makes bug fixes faster and more reliable, makes features easier to add, and makes the whole system easier to customize for particular purposes.

These pragmatic motivations for using and contributing to open source were written up in a document called *The Cathedral and the Bazaar* by Eric S. Raymond. This document had a huge impact. Suddenly people who would never agree with Richard Stallman's ideals saw other reasons to get involved. The number of open source projects, and licenses, exploded exponentially.

This new interest in open source quickly led to the formation of the Open Source Initiative (OSI), which acts as a grassroots campaign for the support of open source. Among other things OSI has drawn up a statement of what constitutes "open source" and maintains a list of what licenses qualify. The FSF also maintains a list of what licenses qualify as free. While neither of these sources is universally accepted, they are the closest thing to standards bodies that exist.

At this point there are still tensions between the "free software" and "open source" communities, even as they work on projects together. Their differences are worth considering, and every developer who benefits from free or open source software should at least consider them and decide for themselves which—if either—view they support and will live by.

These issues will not be discussed again in this book, beyond pointing out that all the code in this book is released under licenses that qualify as "open source" according to OSI and "free" according to the FSF. However, not every license is compatible with the GPL, which may be regarded as the definitive free software license.[2]

The benefits of open source software are particularly relevant in the context of tools. The source code of a tool may itself be considered another tool—pieces and algorithms may be extracted and used in other contexts, reducing development time. It is also possible, if not always easy, to modify a tool for a particular purpose. It is as if a screwdriver can be adapted to use on an unusual screw instead of having to buy a

[1] The convention extends to other systems containing the GNU tools and a kernel. For example, systems using the HURD would be called GNU/HURD.

[2] In particular the Apache 2.0 license, which covers all Apache and Jakarta code released after January 2004, is not compatible with the GPL.

new screwdriver. There are many examples throughout this book of how these ideas are realized in practice.

1.3 The Apache Software Foundation and the Jakarta Project

Much of the software on which the Internet is constructed has always been open source, even before anyone called it that. One of the most important of these was a Web server written at the University of Illinois, which saw the Web from its infancy through 1995 or so. At that point development on this server stopped, and a number of Web masters informally picked it up and started adding new features. Note that this was only possible because the original source code was open. The result was called Apache, a play on words because it was "a patchy" server. To this day Apache is the most frequently used Web server by a sizable margin. It is no longer patchy and is among the most solid and robust code available.

As Apache grew, a number of related projects developed around it, and the Apache Software Foundation was eventually formed to organize and support them all.

Among these related projects was Tomcat, the reference implementation of Sun's first Servlet and JavaServer Pages specifications. Tomcat grew into a full-featured Web server in its own right, and some of its features are discussed in Chapter 17. History repeated itself during the development of Tomcat, and a number of sub- and related projects grew up around it. These were eventually grouped under the name "Jakarta."

There was a time when Jakarta could be thought of as the Java arm of Apache, but that is no longer strictly true. While every Jakarta project is written in Java, many Apache Java projects are no longer part of Jakarta. Apache has continued to grow and now includes subcategories for XML, databases, logging, and much more. Several Jakarta projects have been moved to these new hierarchies or, in some cases, given their own.

1.4 On Application Development

The final element that must be considered before looking at the tools is the general process of development. This will help clarify a number of ways in which a tool can fit into a project.

Engineering is an inherently difficult enterprise; clearly no one person could build a bridge, plane, or skyscraper. The same is true of software engineering beyond fairly small projects. Often a project is so big and complex that it simply cannot all fit into a single human brain.

For a large project to be manageable it must be split into smaller pieces. This is true when there are many members of a team working on the same project so that each knows what part of the whole they will create. It is even true when a project is being undertaken by a single person because it allows the developer to concentrate on one thing at a time. Usually the smaller pieces are easier to design, build, and test, and once built, the individual components can be updated or changed without worrying about how that change will affect the rest of the system.

While using toolkits can be a great time saver, it does require some additional thought at the beginning when deciding how to split up a project. This is because some attention must be given to what pieces the toolkit can provide. If this is not done properly, it may turn out that there is some overlap between what is being developed and what the tool provides. Besides being inefficient, this can lead to problems when integrating the tools into the rest of the code.

Metaphorically, if the whole project is to build a cube, one particular tool may look like a pyramid. This then defines the shape for the remainder of the project that the in-house developers must build.

1.4.1 Modularity in Java

One obvious way to divide a project is by splitting it into individual classes. Classes are Java's natural unit of work and may be thought of as Lego blocks that are ultimately connected together to build the final product. Each individual block can be written by one person or a small team without needing to know the details of how the other blocks work. Each block can also be individually tested to ensure that it exhibits the correct behavior. All that is needed for this approach to work is to ensure that the "connectors" of each block, which typically means the public methods, are stable and well documented.

Indeed, this is how many toolkits work. It is very common for a toolkit to consist of a large collection of classes or, metaphorically, a set of blocks with a variety of shapes and colors. When faced with the need to create one such block, one can save time by using an already-existing block provided as part of a toolkit.

This idea of using existing blocks goes well beyond the question of toolkit use. Such code reuse is fundamental to the whole idea of object-oriented programming. The Java core libraries provide many such blocks. Good developers will try to generalize their tasks and so create additional general-purpose blocks, and other blocks will come as part of toolkits.

What has been called "blocks" would more correctly be called JavaBeans. At one level beans are just classes whose methods conform to certain naming conventions. A bean can expose a property by providing a method that sets the property and/or a

method that obtains the current value. For example, a bean might have a property called "color" and corresponding methods called `getColor()` and `setColor()`. A bean can also designate itself as the source of any number of events that can be registered with other classes that will react to such events.

Note that the notion of bean properties is fairly abstract. They may be simple values, such as a name or color or price, which can be represented as a Java primitive type. A property may also be some compound value represented as another bean or even an array of beans. Beyond this, invoking a get or set method may perform any operation that can be done in Java, from sending an e-mail to updating a database to accessing an external Web page. None of these things are properties in the dictionary sense, but the notion of properties is a useful way of unifying access to beans or toolkits composed of beans.

As powerful as classes are, Java's *interfaces* are even more powerful. A class represents what a component *is,* but an interface specifies what a component *does.* This idea is often referred to as "programming by contract," and it is discussed in many good books on object-oriented programming, including *The Object of Data Abstraction and Structures (Using Java)* by David Riley.

The interface concept is powerful for two reasons. First, a single object may do several things, which is why any class may implement any number of interfaces. More significantly, interfaces provide for a very loose connection between components. While a loose connection may sound like a bad thing, in fact it allows for a great deal of flexibility.

One common design pattern that captures this flexibility is called a *factory*. A factory is a class whose job is to build an instance of a class that implements an interface. Typically there will be many possible implementations to choose from, and the factory will decide which to use based on some configuration file or internal logic.

Such factories are a natural integration point for using toolkits, and this integration can take two forms. The common model where user code calls out to the toolkit can be slightly modified by using a factory as an intermediary. It is also possible to turn this model "inside out." Many of the tools in this book provide complete frameworks into which users plug some of their own code. Often this is accomplished by creating a class that implements an interface provided by the toolkit and then notifying a factory about the presence of this class. When the framework runs, the factory will create an instance of the class, and one of the interface methods will then be invoked automatically.

This is even how servlets work: The configuration files used by application servers associate a URL or URL pattern with a servlet. Internally when a request comes into the server, something akin to a factory uses the configuration information to

determine which class that implements the servlet interface should be invoked, and the request is then passed to this servlet.

1.4.2 Model/View/Controller

Beyond the division into such fine-grained units such as classes and interfaces, it often makes sense to divide projects into major functional units. After all, this is the way most physical engineering is done. When building a car one team is likely to be responsible for the engine, another for the electronics that control everything, and a third with the exterior. These are natural ways to divide up the work, partly because each piece is somewhat independent of the others but also because very different skills are required for each.

There is no set recipe for the way in which a software project should be divided, but there is an important pattern that has emerged over time that advocates identifying three major pieces. The first piece will be responsible for modeling the problem to be solved, so it is called the *model*. This model might be a virtual shopping cart for an online catalog, a database representing a CD collection, or a set of equations representing some complex scientific simulation. In each case, the model contains all the information about how the data is internally stored and the operations that may be performed on that data.

The second piece is responsible for allowing users to interact with this data. This could be a desktop application written in Java using the Swing API (application programming interface), or it could be an applet, or a class that generates HTML, or even a program that controls a huge electronic billboard. The code for this piece contains everything needed in order to navigate through the data, display the values, modify the display as needed, and possibly allow the user to make changes to the model. Ideally, the presentation should also be aesthetically pleasing and intuitive to use. This piece is known as the *view*.

Finally, the third piece acts to mediate between the first two. Although the view allows the user to request particular data, it will not itself load that data into the model. In addition, some data may be restricted to certain users. This kind of information should not reside in either the model or the view. So the third piece, called the *controller,* is responsible for controlling the model based on instructions from the view, and it may also tell the view to hide certain information based on data in the model.

Once these three pieces have been defined, the only remaining work is to ensure that they all fit together and can interoperate. This is done by providing well-defined interfaces between each pair of components. The view will know how to get data from the model, the controller will know how to configure the view and the model, and so on.

Splitting the work in this way is known as the *model/view/controller* paradigm, and it is very powerful. Generally it is used in the context of Web applications, where beans play the role of the model, Java Server Pages and servlets act as the view, and typically a master servlet acts as the controller. Many of the toolkits to be examined are also organized along model/view/controller lines. Some assist in the construction of one of these elements, while others provide entire frameworks comprising integrated implementations of all three into which developers can plug their own code.

This completes the groundwork for this book. In what follows keep in mind the general concept of tools, the motivations and benefits behind open source, and the ways in which tools can integrate with the larger application. And now, on to the tools!

1.5 Further Reading

- The GNU Manifesto: http://www.gnu.org/gnu/manifesto.html
- The Open Source Initiative home page: http://opensource.org/
- The Cathedral and the Bazaar: http://www.catb.org/esr/writings/cathedral-bazaar/cathedral-bazaar/
- A history of Apache: http://httpd.apache.org/ABOUT_APACHE.html