There are many ways to build an application, but most of the time you will pull in a framework or two and a few libraries. Tooling tends to make this easy now, with build tools, such as Maven and Gradle, connecting to a central artifact repository of JAR files. Thanks to the world of open source, many thousands of libraries and frameworks are available to choose from (and most companies have an internal artifact repository with even more). But what makes a good library? How can it be designed well?

Styles of Library

When designing a library, it is useful to bear in mind some common styles that libraries fit into. Back in 2004, I identified two styles within Apache Commons: broad and shallow versus narrow and deep.

The broad and shallow style has many public methods for users to call (the broad part), each of which tends to do relatively little processing (shallow). Using the library focuses on finding the right class to call or create and then following the syntax and operations detailed in the Javadoc. Because the public methods are shallow, they tend to be fairly separate from the others, and it is often possible to split such a library into many smaller libraries. While often this style of library consists of classes with many static methods, it typically includes instantiated classes, too. Examples of this style include Apache Commons Lang, Apache Commons IO, Google Guava, and Joda-Time.

The narrow and deep style has relatively few public methods for users to call (narrow), but each method tends to perform a decent amount of processing (deep). Using such a library tends to involve specific usage patterns that are documented at a high level—often outside the Javadoc. Examples of this style are XML parsers and templating libraries such as Apache FreeMarker. The key to making this approach work is to have an obvious, well-documented public API and to hide the internal classes.

In both styles, the library tends to have relatively small bounds. The result is that if you find the library you chose is buggy or not to your taste, it tends not to be too hard to replace it. This leads to a third style that might best be described as a “business” library. Here, the library is more specific, perhaps used primarily in an industry vertical, and adoption is a major architectural choice for an application. In my day job, I work on Strata, the Duke’s Choice Award–winning library for finance, which is a classic open source example of this style. Most examples of this style are likely to be private and company-specific.

Dependencies
The ease of use and convenience of an artifact repository such as Maven Central makes it all too easy to just pull in dependencies. But when you do, consider how many other dependencies that one dependency has. Too quickly, you find that your application has hundreds of dependencies, and you can face clashes between different versions, a situation termed classpath hell. As such, all good libraries strive to minimize their dependencies.

In my experience with Apache Commons and the Joda projects, I have found that broad and shallow libraries work best if they have no dependencies at all. Commons Lang, Commons IO, and Google Guava all have no dependencies.

There is an interesting case with the Joda-Time and Joda-Money libraries. Both of these broad and shallow libraries do have a dependency — Joda-Convert—but that dependency is optional. Most applications using Joda-Time do not need to have Joda-Convert on the classpath. Only if you use the additional features it provides will you include it.

In my experience, narrow and deep libraries tend to be more complex. As such, they often depend on a few other libraries, which is fine as long as the dependencies are limited. Larger business libraries typically have a larger set of dependencies, but this is usually fine because they are so important to the application that the library drives the dependencies of the application, not the other way around.

It doesn’t make sense to depend on a library for a tiny amount of code, such as a few static utility methods. Instead, consider copying portions of the library into yours with a clear indication as to where the code came from. By keeping track of the copied code, it becomes easier to spot the point at which the additional dependency is worthwhile. Ideally, the code will be package-scoped when copied into your library, as it is not really part of your API.

Finally, you should take extra care using Google Guava in a low-level library, because it tends to be widely used yet incompatible between releases, the classic classpath hell problem.

Integration

One tricky case can be integration, which is when a library needs to provide code to interoperate with other libraries. The most common way to do this is to release a core library and one additional library for each integration. With this approach, the core library is not burdened with the additional dependencies, but the user must pick the correct additional JAR file.

An interesting alternative is to use optional dependencies. With this approach, the library consists of a single JAR file with all the integration code included. However, each integration works only if the user also adds the integration JAR file to their classpath. This can be convenient for the user, as the integration can be made to work transparently.

Best practice normally favors the first approach, with separate JAR files. But when the integration code is relatively small and convenience is important, the second approach can be worthwhile to consider.

Structure

Most libraries consist of just a few packages, and libraries consisting of just one package are quite common. When designing a library, it can be useful to plan the package structure so it has a clear root package to aid first-time users. This is particularly important for narrow and deep libraries. For example, the root package of the library com.foo.shared should contain the most important entry points to the library. Additional packages would contain classes of lower importance, say, com.foo.shared.config and com.foo.shared.model. Any code that should not be called directly by users should go in an internal package, such as com.foo.shared.internal or
In Java releases through Java SE 8, users can, of course, access these internal packages. In Java SE 9 modules, however, it is possible to properly restrict the internal packages so that users cannot access them directly at all.

In addition to modules, library designers should consider using package scope as much as possible. Package scope is hugely underused in Java generally, but it is a great tool for hiding your internal logic. Java SE 8, in particular, enables designers to make much greater use of package scope—thanks to the addition of methods on interfaces. Prior to Java SE 8, your library might have had an interface, a factory for creating instances, and an abstract class to allow for future change. Now, all three features can be combined: Instances can be created using static methods on interfaces, and there is no need for an abstract class with default methods on interfaces. If the whole API can be defined by the interface, it is possible to make the implementation classes package-scoped, that is, created by the static factory methods on the interface. Suddenly, the public API has collapsed from maybe five public classes to one—a huge benefit for later maintenance.

**Features and Growth**

Many libraries start out from a simple need to share code between two projects. The code grows over time and eventually becomes unmanageable, whereas perhaps it should be split. The issue here is that the library grew without a mission statement. Why does this library exist? What problem is it solving? Why should you use this piece of shared code rather than writing it yourself?

By writing something down, often at the top of the homepage of the project or in the README file, you set some boundaries for the library. When requests arrive for new features, it becomes easier to see whether the features are inside or outside the boundaries. This allows you to push back and reject the feature or perhaps create a new library.

If, however, the feature request is within the boundaries for the library, serious consideration should be given to including it. Libraries are shared code, and while perhaps your use cases didn’t need the feature, someone else’s might. But it is important to watch out for bloat, because as more features are added, it becomes harder for new users to learn the library and to find out what it contains. One way to judge whether inclusion warrants the added code is to consider how much code is being shared and what the nearest workaround is for callers. If the workaround is painful and the use case seems sound enough, the added code should probably go into the library.

If you are fortunate enough to be writing a standalone library that isn’t just a sharing of code between two applications, one point to bear in mind is that ‘YAGNI (‘you aren’t gonna need it’) typically does *not* apply. This is because your aim is to serve the needs of the niche that the library sits in so that users are confident that the code they might need will be there when they need it. Doing this may well require additional features or convenience methods beyond those of the minimal use case you have in mind.

Part of managing this growth over time is a plan for compatibility. In most cases, libraries should follow semantic versioning to clearly communicate the compatibility of each version. Tools are available to check this as part of the build process. To avoid classpath hell for your users, it is important to achieve binary compatibility, so that a new version of the library can be just dropped in. This can be painful for a library author, but when many others depend on your library it is a necessity; a key part of the success of Joda-Time is that users can rely on the stable, compatible API, just as they can with the JDK.

**Good Design**

A library generally sits at the bottom of an application. As such, it needs to be reliable and of high quality. When the application calls a library
method, the application needs to be certain that the library will do what is asked of it. It turns out that the best way to arrange this is to use good, modern API design for the library.

Where possible, pass immutable objects into and out of the library. Immutable objects are far clearer for the user: They can be in only one state and will never be affected by any complex concurrency in the application. Of course, for the library designer, immutable objects entail more work. You need to write factory methods and, potentially, a mutable builder class. But the benefits pay off in the form of fewer bugs. (Consider this: If you allow users to pass mutable classes to your library, what happens when your users mutate them while your library is processing?)

The API should also be well defined with regard to null. The simplest approach is to reject null as an input to all methods. Java now has Objects.requireNonNull(), which can help here. An alternative is to accept null and treat it as a no-op or default value, but as I learned with Joda-Time, this approach is usually a very bad idea. As a general rule, methods that might have been defined to return null a few years ago should now return Optional. My experience is that if you follow a strict approach of never returning null, the whole codebase becomes much clearer and safer for users.

Public API methods in the library should also follow sensible and consistent naming. It is vital for users to be able to find the functionality they are looking for using naming only. Consistency is key here. For example, it is fine to use an abbreviation if it makes sense in the context of the library, but use it consistently. And the general advice for initial capitalization of acronyms still applies: You should prefer HttpRequest to HTTPResult.

To aid with compatibility, it is worth considering using a design for key API methods in which a request bean is passed in and a result bean is returned. This has the advantage that when a new feature is needed, you can simply add to the request/response bean rather than create a new method signature on the key API.

**Documentation**

Many developers find documentation to be an annoyance that gets in the way of writing the code. When building a library, you simply can’t think like this. The end users of the library are typically not known to you—they can’t just come ask you a question at your desk. Your only realistic option is to provide the documentation they need.

In practice, this means that the public API must have good and clear Javadocs. In addition, package-level Javadocs, overview Javadocs, and usage documents should be considered, particularly for narrow and deep libraries. These high-level documents should explain how to use the library and what the main entry point is, and they should identify which packages should not be used directly.

One absolutely vital piece of documentation is information about the thread safety of key lifecycle and session classes. For example, when you use an XML or JSON parser, there will typically be a single entry-point class. But should you create a new instance each time? Or should you store it in a static variable? The expected pattern, determined by the thread safety of each class, must be documented. If you don’t do this, your users might conclude that your library doesn’t understand the importance and difficulty of concurrency.

A similar discussion applies to objects that hold external resources, such as streams and buffers. The documentation needs to be clear as to who should close the resource. If the library itself manages resources, for example, through an ExecutorService, it should implement AutoClosable and clearly document the usage pattern.

The final key piece of documentation is the license. While libraries within a company don’t need this, open source libraries must have one. I recommend the Apache License version 2.0 for most libraries. It is a
good, well-written license that is widely used and easy for users to accept.

Conclusion

To design a good library takes time, and it is a task that requires high-quality, clean code. After all, when building an application, all developers can tell whether they are using a good library or a bad one. So, if you are going to build a library, build it well. Your users will thank you.

Learn More

Joda-Time library
Example of Joda-Time’s detailed Javadocs

Stephen Colebourne

Stephen Colebourne (@jodastephen) is a Java Champion who has used Java since version 1.0. He is best known for his work on date and time, through Joda-Time and the Java 8 java.time.* packages. He has many other open source projects under the Joda and ThreeTen brands. Colebourne also writes blogs and speaks at conferences. He works at OpenGamma, producing software for the finance industry.

Share this Page

Facebook
Twitter
LinkedIn
Email

Contact
US Sales: +1.800.633.0738
Global Contacts
Support Directory
Subscribe to Emails

About Us
Careers
Communities
Company Information
Social Responsibility Emails

Downloads and Trials
Java for Developers
Java Runtime Download
Software Downloads
Try Oracle Cloud

News and Events
Acquisitions
Blogs
Events
Newsroom

© Oracle | Site Map | Terms of Use & Privacy | Cookie Preferences | Ad Choices