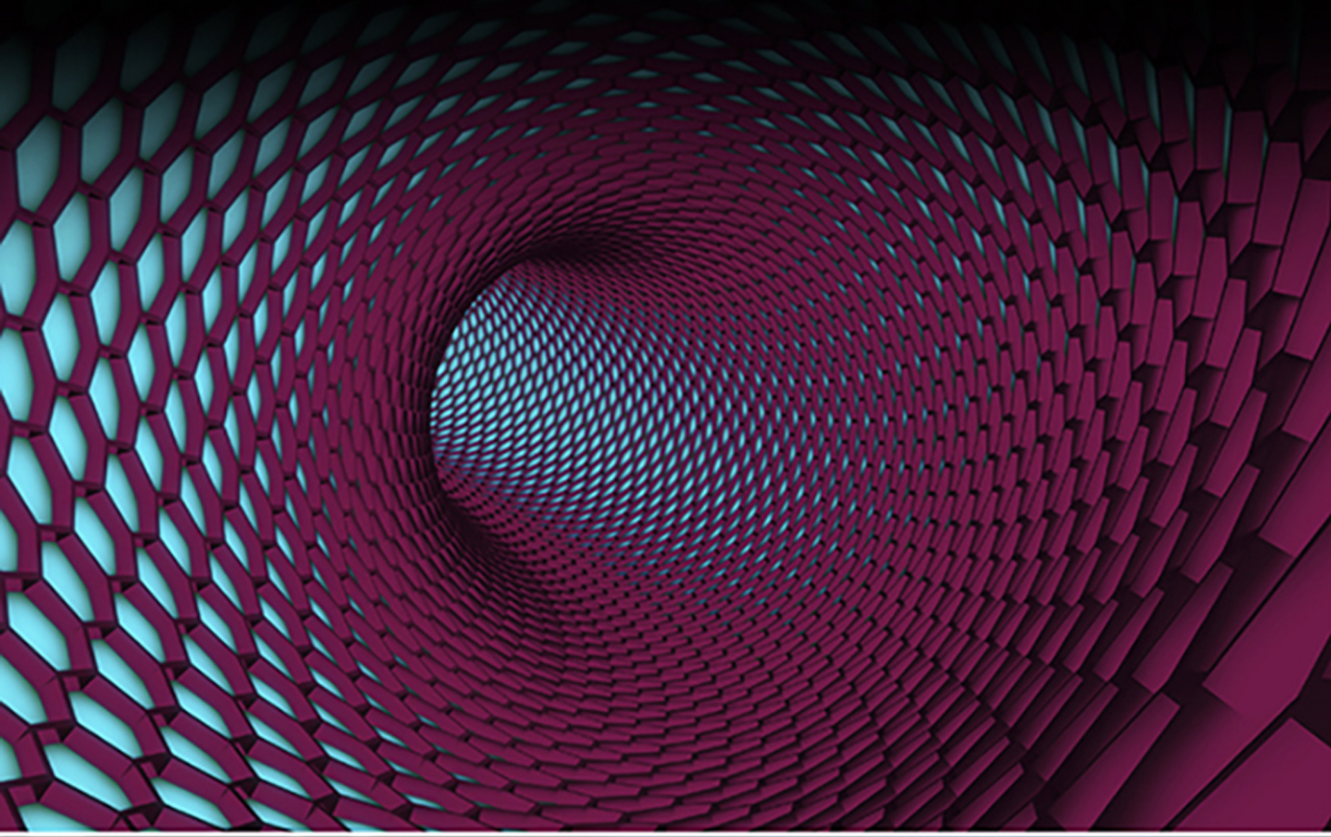
Robert C. Martin Series

Java Application Architecture

Modularity Patterns with Examples Using OSGi



Kirk Knoernschild

Forewords by Robert C. Martin *and* Peter Kriens

Praise for *Java Application Architecture*

"The fundamentals never go out of style, and in this book Kirk returns us to the fundamentals of architecting economically interesting software-intensive systems of quality. You'll find this work to be well-written, timely, and full of pragmatic ideas."

—Grady Booch, IBM Fellow

"Along with GOF's *Design Patterns*, Kirk Knoernschild's *Java Application Architecture* is a must-own for every enterprise developer and architect and on the required reading list for all Paremus engineers."

—Richard Nicholson, Paremus CEO, President of the OSGi Alliance

"In writing this book, Kirk has done the software community a great service: He's captured much of the received wisdom about modularity in a form that can be understood by newcomers, taught in computer science courses, and referred to by experienced programmers. I hope this book finds the wide audience it deserves."

—Glyn Normington, Eclipse Virgo Project Lead

"Our industry needs to start thinking in terms of modules—it needs this book!"

—Chris Chedgey, Founder and CEO, Structure 101

"In this book, Kirk Knoernschild provides us with the design patterns we need to make modular software development work in the real world. While it's true that modularity can help us manage complexity and create more maintainable software, there's no free lunch. If you want to achieve the benefits modularity has to offer, buy this book."

—Patrick Paulin, Consultant and Trainer, Modular Mind

"Kirk has expertly documented the best practices for using OSGi and Eclipse runtime technology. A book any senior Java developer needs to read to better understand how to create great software."

—Mike Milinkovich, Executive Director, Eclipse Foundation

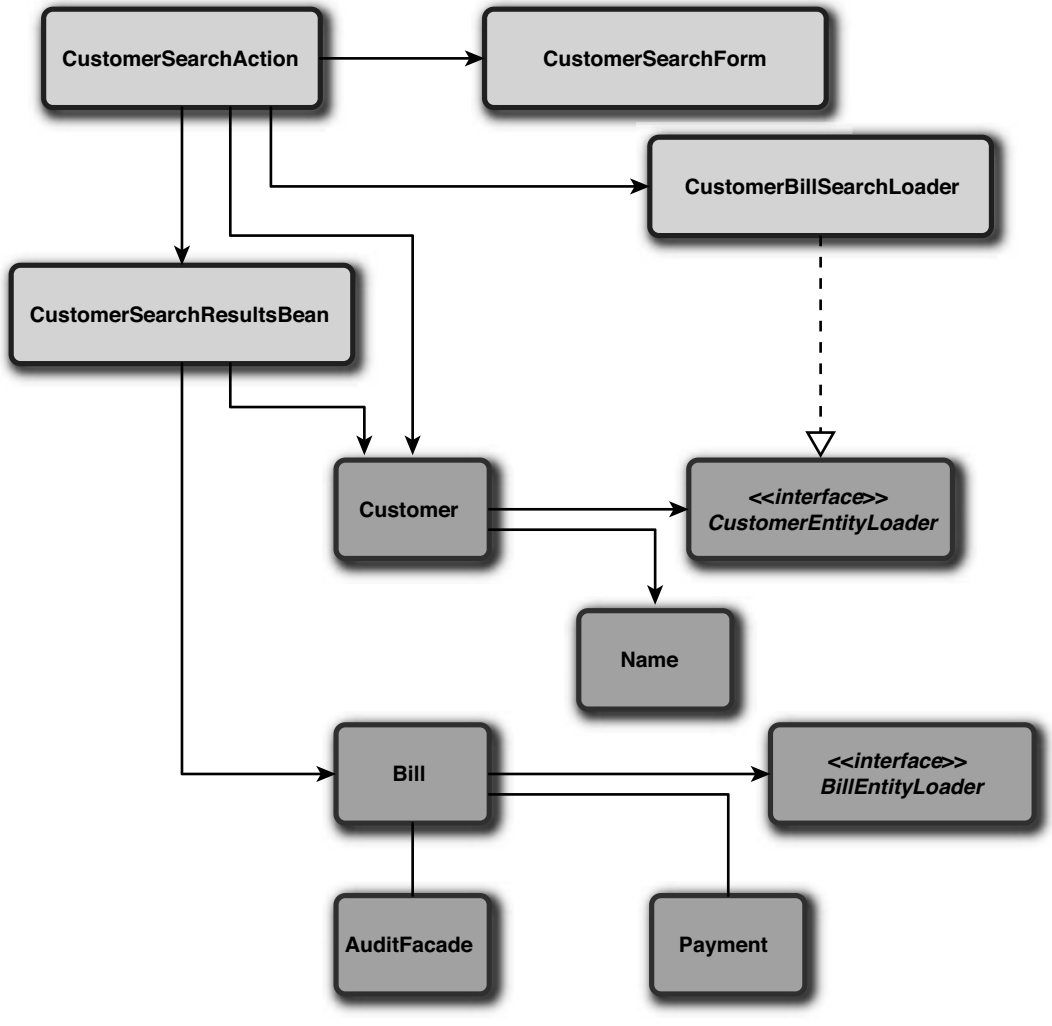


Figure 7.1 Initial version class diagram

backend. It wouldn't be difficult, but it's not what I want to focus on. Although we'll show some code snippets as the example progresses, I set up a Google Code Repository that shows each step in the evolution.[2]

2. The repository can be found at http://code.google.com/p/kcode/source/browse/#svn/trunk/billpayevolution/billpay, and each of the subprojects represents a single step in the example.

7.4 First Refactoring

Physical Layers pattern, 162

Packaging everything into a single WAR file for deployment certainly isn't modular. In most systems we develop, we try to design layers that encapsulate specific behaviors and isolate certain types of change. Typical layers include a UI layer, a business or domain object layer, and a data access layer. In this system, we have these three layers. The Struts `Action` and `Form` classes, along with the JSP, represent part of the UI layer. The UI layer is represented by the upper-portion classes in Figure 7.1. The `Customer`, `Bill`, `Name`, `AuditFacade`, and `Payment` form the business object layer, and the `Loader` classes form the data access layer. These classes are shown in the lower portion of Figure 7.1. Now, here's a key statement that you need to take with you:

> **If I truly have a layered system, then I should be able to break out each layer into a separate module where modules in the upper layers depend on modules in lower layers, but not vice versa.**

If you try this for one of your systems, it's likely you'll find it's not so easy. Most development teams feel they have a layered architecture, but in reality, they don't because somewhere deep within the bowels of the system lies an import or reference to a class in a higher-level layer that we aren't aware of.

In fact, if I really do have a layered system, then I shouldn't have to change anything other than my build script to break the layers into separate JAR files. If I do have to change more than a build script, then I didn't have a layered system to begin with, and I should perform some architectural refactoring to clean things up. Anyway, the end result is relatively simple to understand. No code changes. Only a build script change so that certain classes are allocated to specific modules. Figure 7.2 shows the structure. This refactoring was an example of applying the Physical Layers pattern.

Listing 7.1 illustrates a portion of the initial build script where all of the classes were bundled into the WAR file. Listing 7.2 shows the changes we've made to the build script to create modules for the various layers. Here, we show only the business object layer. We created a new build target where we create the module and then added a new line to the `dist` target where that module is now included in the WAR file.

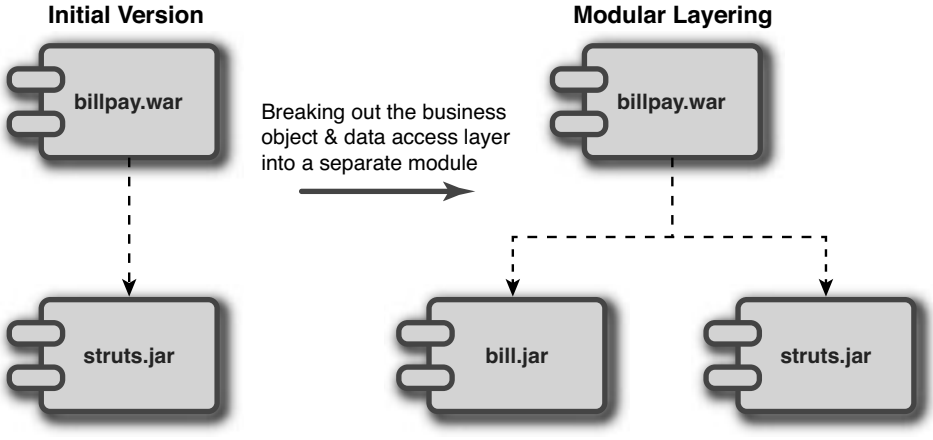


Figure 7.2 Applying the Physical Layers pattern

Listing 7.1 Initial Build Script

```
<target name="compile" depends="init">
   <javac srcdir="${javasrc}:${testsrc}" destdir="${build}">
      <classpath refid="project.class.path"/>
     </javac>
</target>

<target name="bundle" depends="dist">
   <mkdir dir="${deploy}"/>
   <war destfile="${deploy}/billpay.war"
    webxml="WEB-INF/web.xml">
       <fileset dir="jsp"/>
       <webinf dir="WEB-INF">
          <exclude name="web.xml"/>
          <exclude name="lib/servlet-api.jar"/>
       </webinf>

        <classes dir="${build}"/>
    </war>
</target>
```

Listing 7.2 Build Script with Physical Layers

```
<target name="compile" depends="init">
   <javac srcdir="${javasrc}:${testsrc}" destdir="${build}">
      <classpath refid="project.class.path"/>
```

```
   </javac>
</target>

<target name="dist" depends="compile">
    <mkdir dir="${bindist}"/>
    <jar jarfile="${bindist}/bill.jar" basedir="${build}"
     excludes="com/extensiblejava/bill/test/**,
     com/extensiblejava/ui/**"/>
</target>

<target name="bundle" depends="dist">
   <mkdir dir="${deploy}"/>
   <war destfile="${deploy}/billpay.war"
    webxml="WEB-INF/web.xml">
      <fileset dir="jsp"/>
      <webinf dir="WEB-INF">
          <exclude name="web.xml"/>
          <exclude name="lib/servlet-api.jar"/>
      </webinf>
      <lib dir="${bindist}" excludes="test.jar"/>
      <classes dir="${build}"
       includes="com/extensiblejava/ui/**"/>
   </war>
</target>
```

7.4.1 Wrapping Up and Getting Ready for the Next Refactoring

This first refactoring was simple, but it has significant implications. Foremost, it proves that my class-level architecture was decent. I was able to break the system out into modules for the various layers without changing a bunch of code. Really, that's the reason why it was so simple—because the logical design was decent. Had there been violations in the layered structure, it would have been significantly more difficult pulling off this refactoring because I would have been forced to remove the undesired dependencies.

Yet, as we'll see, the existing design may meet the needs of today, but it's going to have to evolve as change emerges. In the second refactoring, we look at what we need to do to integrate with another auditing system and how modularity can help us do this. As we progress, the amazing transformation of a system lacking modularity to a highly modularized version will unfold. In the next few steps, we apply two refactorings using

Abstract Modules pattern, 222

Acyclic Relationships pattern, 146

two different modularity patterns: Abstract Modules and Acyclic Relationships. First, we separate the bill and audit functionality into separate modules so we can independently manage (develop, deploy, and so on) them. Second, we remove the cyclic dependency between these two modules.

7.5 SECOND REFACTORING

In the class diagram shown in Figure 7.1, the `Bill` class has a bidirectional relationship to the `AuditFacade` class. This design has two fundamental flaws. The `Bill` is tightly coupled to the concrete `AuditFacade` class, and the relationship is bidirectional. Bad all around! This can be seen in Listing 7.3, illustrating the `Bill` class's `audit` method.

Listing 7.3 Audit Method of the Bill Class

```
public void audit() {
   AuditFacade auditor = new AuditFacade();
   this.billData.setAuditedAmount(auditor.audit(this));
   this.persist();
}
```

Notice that the `audit` method actually creates the `AuditFacade`, calls the `audit` method, and passes a reference to `Bill`. Ugly. Let's clean this up a little bit. Although there are obvious technology reasons why we need to clean this up, there is also a motivating business force.

> **The system needs to go live with the current vendor's auditing system, but the business has indicated that they aren't renewing the contract with the vendor and are in ongoing negotiation with another vendor. The contract expires in six months, but we deliver the initial version of the system in three months.**

So, three months after deployment, we know we need to swap out auditing systems.

Abstract Modules pattern, 222

We will apply the Abstract Modules pattern, which states that we should *depend upon the abstract elements of a module*. We'll start by refactoring the `AuditFacade` class to an interface and create a separate `AuditFacade1` implementation. This solves the first half of our problem, which

is the tight coupling between the `Bill` and `AuditFacade` implementation. The result is the class diagram shown in Figure 7.3.

Of course, the `Bill` can no longer create the `AuditFacade` implementation. Doing so would compromise the work we've done because `Bill` would still be coupled to `AuditFacade1`. To deal with this challenge, the `AuditFacade` interface is now passed into the Bill's `audit` method, allowing us to swap out `AuditFacade` implementations. Listing 7.4 shows the modified `audit` method.

Listing 7.4 The New Audit Method Accepting the AuditFacade Interface

```
public void audit(AuditFacade auditor) {
   this.billData.setAuditedAmount(auditor.audit(this));
   this.persist();
}
```

If we take this flexible class structure and continue to deploy in the single `bill.jar` module, we have the flexibility to swap out `AuditFacade` implementations at the class level, but we're still required to package everything up into a single bundle and deploy it as a single module. So, what we really must do is separate the audit functionality into a module separate from the bill. Separating the `AuditFacade` interface and `AuditFacade1` implementation into separate modules results in the diagram illustrated in Figure 7.4. Here's where the bidirectional relationship

cyclic dependencies, 50

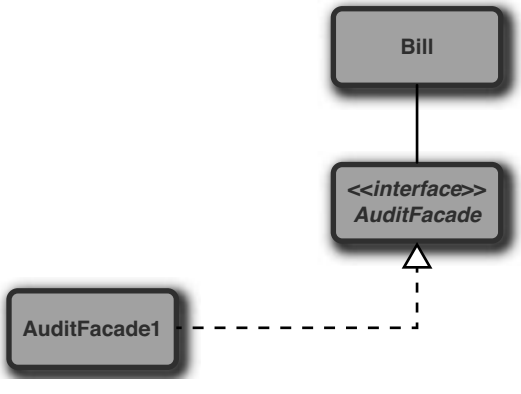


Figure 7.3 Refactoring AuditFacade to an interface