SAMPLE CHAPTER

Effective UNIT TESTING

A guide for Java developers

MANNING

LASSE KOSKELA

Effective Unit Testing

by Lasse Koskela

Chapter 7

brief contents

7 Testable design

In this chapter

- What is testable design?
- What is untestable design?
- How to create testable designs

In the preface of *Implementation Patterns* (Addison Wesley Professional, 2007), Kent Beck compares programming to an American TV show called *Jeopardy*. In the show the host provides answers and the contestants' job is to guess which question that answer was for. *"A short section at the end of a book." "What is an epilogue?" "Correct."*

Kent makes the analogy to programming and points out that Java provides answers in the form of its language constructs, and the programmer's job is to figure out what the questions are and which problems are solved by which language construct. He offers the following example: if the answer is "Declare a field as a set," the question might be, "How can I tell other programmers that a collection contains no duplicates?"

The same happens with design. Throughout our education and especially the first years of our professional careers, we're taught solutions. Our senior colleagues show us "the way things are done around here," and we pick up coding conventions from the code we work with, and sometimes we pick up stuff from books like this.

But it's not enough to know solutions. We also need to learn to identify the problems they solve. That's why this book includes a catalog of test smells.

This chapter aims to identify the common testability problems that stem from design decisions, such as making a method `static`, and the use of certain language constructs like `final`. We'll start with a discussion of what we're looking for when we say we want "testable design," a modular design that likely abides by certain design principles and avoids certain implementation details that make it harder to write tests. From there we'll move on to detailing the specific testability issues we try to avoid and we'll conclude the chapter with simple guidelines for designing testable code.

7.1 What's testable design?

The fundamental value proposition of testable design is being better able to test code. Quoting Roy Osherove, the author of *The Art of Unit Testing with Examples in .NET* (Manning Publications, 2009), "a given piece of code should be easy and quick to write a unit test against." More specifically, testable design makes it easy to instantiate classes, substitute implementations, simulate different scenarios, and invoke particular execution paths from our test code.

Testability isn't an absolute yes-or-no property of a design. As Scott Bellware put it, "Testability is not a term that describes *whether* software can be tested. It refers to software that is *easily* tested."[1] It should be a breeze for a programmer to set up scenarios in unit tests. The less testable the design, the more burdensome it is for the programmer to write those tests. The big question is, what are testable designs made of—how do we end up with such designs?

Though you'll often hear quips about software development being a young profession, we have decades of experience to draw from and that experience has taught us a lot about constructing software in a variety of programming paradigms. One of the things we've learned is the virtues of modular design.

7.1.1 Modular design

What makes a design modular is its nature of being composed of separate modules, each serving a particular purpose in the design. By breaking down a program's overall functionality into distinct responsibilities and assigning those responsibilities to separate components, we end up with a design that has plenty of flexibility.

That flexibility builds on the various seams we've introduced by way of composing the overall design from discrete modules, with each individual module containing everything necessary to fulfill its responsibility. This style of programming aims at having as few dependencies *between* modules as possible, as illustrated in figure 7.1.

Constructing software from small components helps groups of people collaborate on larger products because functional changes to the product tend to be more isolated to a specific part of the code base. This follows from the characteristic of

[1] Scott Bellware, "Good Design is Easily-Learned," Jan. 12, 2009, http://mng.bz/IAMK.

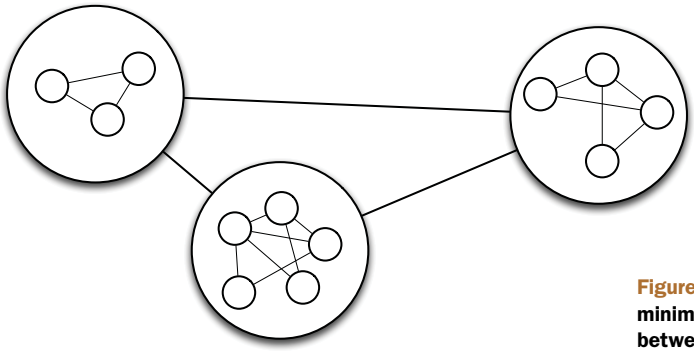

Figure 7.1 Modular designs minimize the dependencies between modules

modular design, where the system is partitioned into discrete functional elements with their respective responsibilities over a specific function or capability.

This structure, encouraged by modular design, enables after-the-fact augmentation of a system, changing or adding new functionality by merely plugging in a new module, as long as the modules are sufficiently self-contained and loosely coupled, and functional concerns are separated within the code base. This also directly aids testability because the properties of modular design are the ones that make code testable.

Generations of programmers have put these ideals in practice and found them a valuable goal to strive for. But the concept of modular design is fairly abstract. Luckily, our elders have also crystallized some of their hard-earned experience into somewhat more concrete design principles for us to keep in mind.

7.1.2 SOLID design principles

Plenty of design principles are documented in literature. One of the more widely spread collections of design principles are the *SOLID* principles. SOLID is an acronym and stands for a set of five design principles documented by Robert C. Martin.

The nice thing about object-oriented design principles such as SOLID is that they mesh well with testability. Keep your code in line with design principles such as SOLID and you're that much more likely to end up with a modular design.

So let's see what the SOLID principles are and how they contribute to the testability of a design.

Single Responsibility Principle

The Single Responsibility Principle (SRP) says, "There should never be more than one reason for a class to change." In other words, classes should be small, focused, and have a high cohesion. It goes further than that, as methods should also have just one reason to change. This is what Sandro Mancuso calls the *inside view*.[2]

Code that follows the SRP is easier to approach and understand, which in turn makes it easier to test because writing tests is essentially an act of specifying the

[2] See "SRP: Simplicity and Complexity," July 26, 2011, http://mng.bz/08ks.

expected behavior, expressing our understanding of the problem the code is intended to solve. Furthermore, taking the inside view, the less complexity a method has, the less complexity is required in a test to thoroughly test that code.

OPEN-CLOSED PRINCIPLE

The Open-Closed Principle (OCP) says that classes should be "open for extension but closed for modification." Though that may sound abstract, all it means is that you want to be able to change what a class does without changing its source code; for instance, by swapping in an alternative strategy.

Classes that delegate specific responsibilities to other objects allow tests to substitute a test double when needed to simulate a specific scenario.

LISKOV SUBSTITUTION PRINCIPLE

The Liskov Substitution Principle (LSP) says, "Subclasses should be substitutable for their base classes." In essence, code that uses an instance of class A should continue to function properly if passed an instance of class B, a subclass of A.

LSP is about class hierarchies that exist for the right reason, embodying a valid abstraction, and not merely as a vehicle of code reuse because it happens to be convenient. Though among the less crucial SOLID principles from a testability point of view, violating the LSP does imply a handicap for the test-infected programmer. Class hierarchies that follow the LSP contribute to testability by enabling the use of *contract tests*—tests written for an interface that can be executed against all implementations of that interface.

INTERFACE SEGREGATION PRINCIPLE

The Interface Segregation Principle (ISP) suggests that "Many client specific interfaces are better than one general purpose interface." In short, you should keep interfaces small and focused.[3]

Small interfaces improve testability by making it easier to write and use test doubles. For instance, one test might want to stub collaborator A, fake collaborator B, and substitute a spy for collaborator C. With each collaborator having its own small interface, it's straightforward to implement the test doubles.

DEPENDENCY INVERSION PRINCIPLE

The fifth SOLID principle is the Dependency Inversion Principle (DIP), which says code should "depend on abstractions, not on concretions." Taken to one extreme, the DIP suggests that a class shouldn't instantiate its own collaborators but rather have their interfaces passed in.

For writing tests, the ability to pass collaborators in rather than selectively override parts of the code under test is huge. Such *dependency injection* is a boon to testability because not only is substituting collaborators possible, but it's also made effortless: the tests use the code just like it's used in production.

It's important for programmers to understand common design principles and to be able to envision concrete implementations and their implications. Over time, you

[3] Sound familiar? I once overheard someone refer to the ISP as "SRP for interfaces."

gather a sufficient collection of mental notes and learn to pattern-match them against the code you're looking at, being able to draw on that knowledge and experience.

7.1.3 Modular design in context

It's not that easy, because though we're pretty good at identifying suitable implementation alternatives for the problem we're solving, we should also be able to envision those solutions as part of the larger whole—from the perspective of the code that pulls together those smaller components. Tim Berners-Lee, the man who invented the World Wide Web, once wrote the following about the principles of design: "It is not only necessary to make sure your own system is designed to be made of modular parts. It is also necessary to realize that your own system, no matter how big and wonderful it seems now, should always be designed to be a part of another larger system."[4]

In other words, modularity as such isn't good unless the design is fit for the way you want to use it and grow it. You're in good fortune—I have a trick up my sleeve that makes it easier to arrive at modular designs that respect common design principles and behave well in their larger context.

7.1.4 Test-driving toward modular design

The fastest way to absorb modular design is to start test-driving your code. The act of writing tests before the implementation they call for is essentially a way to ensure that you're taking the client's view on the code you're shaping. That means that you're that much more likely to end up with a design that's fit for purpose. Plus, there's no question of how testable the design is.

It's not just the test-first facet of TDD that spurs modular design. TDD practitioners also refactor their code frequently so they're constantly looking for too big methods to split, better abstractions to introduce, and duplications to remove. J.B. Rainsberger talks about how minimizing duplication and maximizing clarity leads to modular design in his article on "The Four Elements of Simple Design."[5]

There are many techniques and ways to learn modular design at your disposal. It's only a matter of choosing whether you care about your code enough to adopt disciplined practices, such as test-driven development and merciless refactoring.

Now let's take a look at some of the obstacles that we often find in our way, standing between us and our testable design.

7.2 Testability issues

Most often, a programmer struggling to write a test is facing off against one of a few common show-stoppers. They mostly fall under two categories: restricted access, and inability to substitute certain parts of the implementation. Though there are many

[4] Tim Berners-Lee, "Principles of Design," last updated March 2, 2010, http://www.w3.org/DesignIssues/Principles.html.

[5] J.B. Rainsberger, "The Four Elements of Simple Design," 2009, http://www.jbrains.ca/permalink/the-four-elements-of-simple-design

ways for testability to be less than ideal, a lot of the time the first barrier to get over is as trivial as being unable to instantiate a class in your test.

7.2.1 Can't instantiate a class

One of the first things we as programmers tend to do when writing a test is to instantiate some objects. It could be an object you want to test that you can't instantiate, but most often the culprit is a collaborator you need to pass in to the object you're testing.

This mostly happens with third-party libraries that weren't designed with testability in mind, but sometimes you only have yourself to blame. Typically, you've been overly conservative with your visibility modifiers and now you're paying for it as the compiler points out your short-sightedness.

Another common cause for the inability to instantiate a class with a constructor is a static initialization block that assumes you're running in a full-blown production environment, leading to a totally unexpected exception blowing up in your face when you try to run your test. For instance, consider the following snippet of testability-destruction:

```
public class DocumentRepository {
   private static final String API_KEY = "d869db7fe62fb07c";
   private static String sessionToken;

   static {
      String serverHostName = System.getenv("ACL_SERVER_HOST");
      SessionClient api = new SessionClientImpl(serverHostName);
      sessionToken = api.openSession(API_KEY);
   }

   public DocumentRepository() {
      ...
   }
}
```

Trying to instantiate this class in a test running on your laptop, you could easily stumble on a `NullPointerException` because you hadn't set an environment variable named `ACL_SERVER_HOST`, or a `UnknownHostException` because your laptop fails to connect to the real server running in the lab behind a firewall. Dependencies like this are business as usual; the real problem is that, since the dependency is hardcoded into a static initialization block, there's not much you can do about it.

7.2.2 Can't invoke a method

Even with all objects instantiated and ready to go, your test might stumble on executing the interaction and scenario you want. For instance, you might want to invoke a method that's marked `private`. It's the overly conservative visibility modifier issue again![6]

Another source for the inability to invoke the method you want might be its opacity; you can't figure out what the method expects to receive as its arguments. This is

[6] I'd argue that in this case the true issue is that the private method should really be public on a new class. Clearly it's complicated enough, because otherwise you wouldn't want to test it directly.

especially true with APIs that rely on good old `java.util.Map`. Though that may be a "flexible" API, it comes with some serious issues. You can't intuitively see what the `Map` should contain, so you have to go into the source code or documentation to find out, which grinds your pace to a halt. The same goes with the `private` method you want to test directly. Unless you refactor the design, you're left with two bad alternatives: don't test it or whip up the Reflection API to circumvent the visibility modifier.

7.2.3 Can't observe the outcome

Even if you have the means to invoke the method you want, you might have problems determining whether the right things happened afterward. The "hello world" of unit tests is to invoke a method and assert something about its return value. But if the method is supposed to engage in an interaction with collaborating objects or if it doesn't return anything as a `void` method, it gets more complicated.

Sometimes you find yourself incapable of intercepting the interaction you're interested in. This might be because the collaborator is hard-wired into the method you're testing and can't be substituted with a test double. Other times, your trouble stems from the method under test starting threads to do the work, but our test code can't intercept that thread's execution.

This ties closely with the other major category of testability issues: the trouble you have to go through to selectively substitute parts of the implementation.

7.2.4 Can't substitute a collaborator

Failure to substitute a collaborator is a common testability problem. It might be because the production code has a hardcoded `new Collaborator()` where you want to test its interaction with that particular collaborator. In other words, you're missing a "seam" where you can intercept and observe the interaction; you're technically incapable of substituting the collaborator.

Though such hardcoded collaborator implementation is all too frequent and extremely inconvenient, perhaps even more common an occurrence is a structure that doesn't make it impossible to substitute a collaborator, but that makes it unnecessarily strenuous. That structure is sometimes referred to as a *method chain*: `getCollaborator().doStuff().askForStuff().doMoreStuff()`.[7]

Even if you could substitute the collaborator, such a method chain means that your test double needs to return a test double that needs to return a test double that needs to... Tedious!

7.2.5 Can't override a method

Substituting test doubles for collaborators is a crucially important tool in writing unit tests. But it's not the only tool: sometimes you don't want to substitute a collaborator

[7] This situation calls for a reference to the *Law of Demeter*, which warns us about code knowing too much about other units of code: http://en.wikipedia.org/wiki/Law_of_Demeter.

but rather a *part of the object under test*. Say the method you'd like to invoke and test acquires a collaborator through a method like this:

```
private static final Collaborator getCollaborator() { ... }
```

Now, all three of those keywords (`private`, `final`, `static`) essentially mean that you can't do this in your test:

```
@Test
public void test() {
   final Collaborator collaborator = new TestDouble();
   ObjectUnderTest o = new ObjectUnderTest() {
      @Override
      private static final getCollaborator() {
         return collaborator;
      }
   };
   ...
}
```

You can't do that because the compiler won't let you override a method that's `final`, a method that's `private`, nor a method that's `static`.[8] Again, your options are limited to a range from bad to worse and involve heavy-duty reflection and byte code manipulation—none of which we want to do.

Having faced all of these testability issues again and again over the years, I've come to appreciate a few simple heuristics—guidelines—for testable design. Let's take a look at them.

7.3 Guidelines for testable design

The following is a set of do's and do not's I've gathered and whose importance I've learned through repeated yak shaving.[9] They aren't in any particular order, and none of them is a universal truth—just things to keep in mind so that you think twice before deciding to go against these guidelines.

With that disclaimer, let's start with some guidelines that relate to what we talked about in the previous section.

7.3.1 Avoid complex private methods

There's no easy way to test a `private` method. Therefore, you should strive to make it so that you don't feel the need to test your `private` methods directly.

Note that I didn't say you shouldn't test your `private` methods, but that you shouldn't test them *directly*. As long as those methods are trivial utilities and shorthands to make your `public` methods read well, it should be perfectly fine that they get tested only through those `public` methods.

---

[8] Note that `static` methods can be *hidden* by a subclass, but that's not the same as *overriding*, and largely useless in terms of what we're looking for.

[9] Yak-shaving defined, http://en.wiktionary.org/wiki/yak_shaving.

When the `private` method isn't that straightforward and when you do feel like you want to write tests for it, you should refactor your code so that the logic encapsulated by the `private` method gets moved over to another object responsible for that logic, and where it's a `public` method as it should be.

7.3.2 Avoid final methods

Few programs need `final` methods. The odds are you don't need them either.

The main purpose of marking a method as `final` is to ensure that it isn't overwritten by a subclass. An Oracle Technology Network article on secure coding guidelines for Java[10] suggests that "making a class final prevents a malicious subclass from adding finalizers, cloning and overriding random methods." Though that's true in some contexts, it doesn't mean that *you* should need the `final` modifier.

There are two problems with the preceding logic. First, those potential subclasses are likely written by the people sitting next to you. Second, the Reflection API can be used to remove the `final` modifier. In practice, the only situation where you might reasonably want to make a method `final` is when you load foreign classes at runtime, or you don't trust your colleagues (which sounds like you have much bigger issues to worry about).

You might add a third situation to this list: not trusting yourself; for example, not trusting yourself to not accidentally override the wrong method in a concrete class that's part of a *template method* pattern.[11] Even then, your tests should catch that mistake quite easily. (And if you're really paranoid you could go all out and write a *contract test* that checks for proper use with the Reflection API.)

The big question is this: does that `final` keyword interfere with your tests, and if it does, does the burden of worse testability outweigh the benefit of having the `final` there.

What about the performance of final?

One of the arguments that people sometimes lean on in support of `final` methods is performance. Namely, it's said that since `final` methods can't be overridden, the compiler can optimize the code by inlining the method's implementation.

It turns out that the *compiler* can't safely do this because the method might have a non-`final` declaration at runtime. But a JIT compiler would be able to inline such methods at runtime, which does present a theoretical performance benefit.

Jack Shirazi and Kirk Pepperdine, the authors of *Java Performance Tuning, 2nd Edition* (O'Reilly Media, January 2003), and javaperformancetuning.com have said, "I wouldn't go out of my way to declare a method or class final purely for performance reasons. Only after you've definitely identified a performance problem is this even worth considering."

[10] "Secure Coding Guidelines for the Java Programming Language, Version 4.0," http://www.oracle.com/technetwork/java/seccodeguide-139067.html.

[11] "Template method pattern," http://en.wikipedia.org/wiki/Template_method_pattern.

7.3.3 Avoid static methods

Most `static` methods shouldn't be. The reason to have them is generally either because they don't relate to a specific instance of a class or because we couldn't bother figuring out where it belongs so we made it a static method in this utility class. The former motive is solid but the latter is mere ignorance or lack of interest. Yet another reason, and an unfortunately common one, is to make it easier to provide global access to the method by making it static.

Some methods are naturally `static`. For example, a utility method that performs a calculation on numbers would likely be a good candidate for a `static` method. So what distinguishes a method as one that should or shouldn't be `static`? Perhaps the example in the following listing serves to make the idea more clear.

Listing 7.1 Avoid `static` methods that you might want to stub out

```
public static int rollDie(int sides) {
   return 1 + (int)(Math.random() * sides);
}
```

The `rollDie()` method produces a random result as if you'd rolled a die with a given number of sides. Dealing with random factors in automated tests is generally something you should avoid, so you'll likely want to stub out the `rollDie()` method in your tests.

My rule of thumb is to not make it `static` if you foresee that you might want to stub it out in a unit test one day. It turns out that I rarely need to stub out a pure calculation. On the other hand, I frequently find myself wanting to stub out `static` methods that serve as an entry point to a service or a collaborator object.

Pay attention to the kind of methods you declare `static`. A static method is easy to write, but you'll have a hard time later if you ever need to stub it out in a test. Instead, create an object that provides that functionality through an instance method.

7.3.4 Use new with care

The `new` keyword is the most common form of hardcoding. Every time you "new up" an object you're nailing down its exact implementation. For that reason, your methods should only instantiate objects you won't want to substitute with test doubles. This listing points out this pattern.

Listing 7.2 The `new` keyword hardcodes the implementation class

```
public String createTagName(String topic) {
   Timestamper c = new Timestamper();
   return topic + c.timestamp();
}
```

The method in the listing builds a string based on the given input and a timestamp generated by a `Timestamper`, which is instantiated inside the method.

Many programmers don't think of `new` as the kind of red flag that `static` and `final` are for test-infected programmers. That doesn't make it completely innocent,

as there's no way (other than byte code manipulation) for a test to override the specific class being instantiated within the method under test; for example, the `TimeStamper` in listing 7.2.

When instantiating objects, you should ask yourself: is this object something you'd want to swap out in a test? If it's a true collaborator and you might want to change its implementation on a test-by-test basis, it should be passed into the method somehow rather than instantiated from within that method.

7.3.5 Avoid logic in constructors

Constructors are hard to bypass because a subclass's constructor will always trigger at least one of the superclass constructors. Hence, you should avoid any test-critical code or logic in your constructors.

The next listing is a contrived implementation of a *universally unique identifier* (UUID), which is made up of the generating computer's MAC address and a timestamp.

Listing 7.3 Doing too much in a constructor

```
public class UUID {
   private String value;

   public UUID() {
      // First, obtain the computer's MAC address by
      // running ipconfig.exe and parsing its output
      long macAddress = 0;
      Process p = Runtime.getRuntime().exec(
               new String[] { "ipconfig", "/all" }, null);
      BufferedReader in = new BufferedReader(
               new InputStreamReader(p.getInputStream()));
      String line = null;
      while (macAddress == null &&
               (line = in.readLine()) != null) {
         macAddress = extractMACAddressFrom(line);
      }

      // Obtain the UTC time and rearrange
      // its bytes for a version 1 UUID
      long timeMillis = (System.currentTimeMillis() * 10000)
               + 0x01B21DD213814000L;
      long time = timeMillis << 32;
      time |= (timeMillis & 0xFFFF00000000L) >> 16;
      time |= 0x1000 | ((timeMillis >> 48) & 0x0FFF);

      ...
   }
}
```

Let's say you'd like to test the `UUID` class shown in listing 7.3. Looking at its bloated constructor, you have a testability issue: you can only instantiate this class (or its subclasses) on a Windows computer because it tries to execute `ipconfig /all`. I happen to be running a Mac so I'm screwed!

A much better approach would be to extract all of that code into protected methods that *can* be overridden by subclasses, as shown here.

Listing 7.4 Moved logic from constructor to protected helper methods

```
public class UUID {
   private String value;

   public UUID() {
      long macAddress = acquireMacAddress();
      long timestamp = acquireUuidTimestamp();
      value = composeUuidStringFrom(macAddress, timestamp);
   }

   protected long acquireMacAddress() { ... }
   protected long acquireUuidTimestamp() { ... }

   private static String composeUuidStringFrom(
            long macAddress, long timestamp) { ... }
}
```

With the modifications in listing 7.4, you can instantiate `UUID` objects in your tests regardless of which operating system you're running by overriding the specific bits you want:

```
@Test
public void test() {
   UUID uuid = new UUID() {
      @Override
      protected long acquireMacAddress() {
         return 0;  // bypass actual MAC address resolution
      }
   };
   ...
}
```

To summarize this guideline, whatever code you find in your constructors, make sure that it's not something you'll want to substitute in a unit test. If there is such code, you'd better move it to a method or a parameter object you can override. [12]

7.3.6 Avoid the Singleton

The *Singleton* pattern has cost this industry millions and millions of dollars in bad code. It was once written down as a pattern for "ensuring a class has one instance, and to provide a global point of access to it." I suggest you don't need either half of that proposition.

There are situations where you want just one instance of a class to exist at runtime. But the Singleton pattern tends to prevent *tests* from creating different variants, too. Let's look at the traditional way to implement a Singleton, shown in the next listing.

[12] Overriding a method that's called from the constructor is tricky and could result in unexpected behavior. Refer to the Java Language Specification (http://mng.bz/YFFT) for details.

Listing 7.5 Traditional implementation of the Singleton pattern

```
public class Clock  {
   private static final Clock singletonInstance = new Clock();

   // private constructor prevents instantiation from other classes
   private Clock() { }

   public static Clock getInstance() {
      return singletonInstance;
   }
}
```

Making the constructor `private` and restricting access to the `getInstance()` method essentially means that you can't substitute the `Clock` instance once it's been initialized. This is a big issue because whenever you want to test code that uses the `Clock` Singleton, you're stuck:

```
public class Log {
   public void log(String message) {
      String prefix = "[" + Clock.getInstance().timestamp() + "] ";
      logFile.write(prefix + message);
   }
}
```

Wherever the code you want to test acquires the `Clock` instance through the static `Singleton` accessor, your only option is to inject a new value for the `Singleton` instance in the `Singleton` class using reflection (and you don't want to use reflection in your tests), or by adding a setter method for doing that.

> **SURVIVING A SINGLETON** If you do find yourself with a static Singleton accessor, consider making the singleton `getInstance()` method return an interface instead of the concrete class. An interface is much easier to fake when you don't need to inherit from the concrete class!

The much better—and testable—design would be a singleton with a lowercase 's' that doesn't *enforce* a single instance but rather relies on the programmers' agreement that "we'll only instantiate one of these in production." After all, if you need to protect yourself from sloppy teammates you have bigger problems to worry about.

7.3.7 Favor composition over inheritance

Inheritance for the purpose of reuse is like curing a sore thumb with a butcher knife. Inheritance does allow you to reuse code but it also brings a rigid class hierarchy that inhibits testability.

I'll let Miško Hevery, a clean code evangelist at Google, explain the crux of the issue:[13]

[13] James Sugrue blog, an interview with Miško Hevery, "The Benefits of Testable Code," November 17, 2009, http://java.dzone.com/articles/benefits-testable-code.

> The point of inheritance is to take advantage of polymorphic behavior NOT to reuse code, and people miss that; they see inheritance as a cheap way to add behavior to a class. When I design code I like to think about options. When I inherit, I reduce my options. I am now sub-class of that class and cannot be a sub-class of something else. I have permanently fixed my construction to that of the superclass, and I am at a mercy of the super-class changing APIs. My freedom to change is fixed at compile time.
>
> On the other hand composition gives me options. I don't have to call superclass. I can reuse different implementations (as supposed to reusing your super methods), and I can change my mind at runtime. Which is why if possible I will always solve my problem with composition over inheritance. (But only inheritance can give you polymorphism.)

Note that Miško isn't saying inheritance is bad; inheritance for polymorphism is totally okay. But if your intent is to *reuse functionality* it's often better to do that by means of composition: using another object rather than inheriting from its class.

7.3.8 Wrap external libraries

Not everybody is as good at coming up with testable designs as you are. With that in mind, be extremely wary of inheriting from classes in third-party external libraries, and think twice before scattering direct calls to an external library throughout your code base.

We already touched on the testability problems with inheritance. Inheriting from an external library tends to be worse, as you don't have as much control over the code you're inheriting from. Whether it's through inheritance or direct calls, the more entangled your code is with the external library, the more likely it is that you'll need those external classes to be test-friendly.

Pay attention to the testability of the classes in the external library's API and, if you see red flags, be sure to wrap the library behind your own interface that's test-friendly and makes it easy to substitute the actual implementation.

> **INABILITY TO CHANGE THE DESIGN** Every now and then you'll find yourself wondering what to do because you're stuck in a situation where your design isn't testable *and you can't change the design* to be more testable. Despite the fact that the code you're looking at is technically "yours," this situation is the same as with external libraries, which you also have little control over. Thus, the approach to dealing with this testability challenge is also much alike: wrap the untestable piece of code into a thin layer that *is* testable.

7.3.9 Avoid service lookups

Most service lookups (like acquiring a Singleton instance through a static method call) are a bad trade-off between a seemingly clean interface and testability. It's only *seemingly* clean because the dependency that could've been explicit as a constructor parameter has been hidden within the class. It might not be technically impossible to substitute that dependency in a test, but it's bound to be that much more work to do so.

Let's take a look at an example. The next listing presents a class responsible for making remote search requests to a web service through an `APIClient`.

Listing 7.6 Service lookups are harder to stub than constructor parameters

```
public class Assets {
   public Collection<Asset> search(String... keywords) {
      APIRequest searchRequest = createSearchRequestFrom(keywords);
      APICredentials credentials = Configuration.getCredentials();
      APIClient api = APIClient.getInstance(credentials);
      return api.search(searchRequest);
   }

   private APIRequest createSearchRequestFrom(String... keywords) {
      // omitted for brevity
   }
}
```

Note how the `search()` method acquires the `APIClient` with a service lookup. This design doesn't drive testability as well as we'd like. This listing shows how you might write a test for this object.

Listing 7.7 Stubbing out a service lookup implies an extra step

```
@Test
public void searchingByKeywords() {
   final String[] keywords = {"one", "two", "three"}
   final Collection<Asset> results = createListOfRandomAssets();
   APIClient.setInstance(new FakeAPIClient(keywords, results));
   Assets assets = new Assets();
   assertEquals(results, assets.search(keywords));
}
```

The indirection implied by the service lookup within `Assets` stipulates that we go through the extra step of configuring the service lookup provider (`APIClient`) to return our test double when someone asks for it. Though it's just a one-liner in this test, it implies several lines of code in `APIClient` that only exist to work around a testability issue.

Now, let's contrast this to a situation where we've employed constructor injection to pass our test double directly to the `APIClient`. This listing shows what that test would look like.

Listing 7.8 Constructor parameters are easier to substitute

```
@Test
public void searchByOneKeyword() {
   final String[] keywords = {"one", "two", "three"}
   final Collection<Asset> results = createListOfRandomAssets();
   Assets assets = new Assets(new FakeAPIClient(keywords, results));
   assertEquals(results, assets.search(keywords));
}
```

Again, it's just one line shorter but that's a full 20% less than listing 7.7 and we don't need the workaround setter method in `APIClient` either. Aside from our code being more compact, passing dependencies explicitly through the constructor is a natural, straightforward means to wire up our objects with their collaborators. Friends don't let friends do static service lookups. [14]

7.4 Summary

We started off this chapter by discussing what testable design is, making a corollary to modular design and introducing the SOLID design principles of the *Single Responsibility Principle*, the *Open-Closed Principle*, the *Liskov Substitution Principle*, the *Interface Segregation Principle*, and the *Dependency Inversion Principle*. Abiding to these principles is likely to lead to more modular design and, therefore, more testable design.

Issues with testability boil down to our inability to write tests or the excess trouble we have to go through to get it done. These troubles include:

- Inability to instantiate a class
- Inability to invoke a method
- Inability to observe a method's outcome or side effects
- Inability to substitute a test double
- Inability to override a method

Learning from these issues and drawing from our experience, I described a number of guidelines to avoid the aforementioned testability issues and to arrive at testable designs. Those guidelines teach you to avoid `final`, `static`, and complex `private` methods. You also learned to treat the `new` keyword with care, as it essentially hard-codes an implementation we can't substitute.

You should avoid significant logic in your constructors because they're hard to override. You should avoid the traditional implementation of the Singleton pattern and instead opt for the *just create one* approach. You know to favor composition over inheritance because it's more flexible than the class hierarchy implied by inheritance.

We noted the danger of inheriting from, and making liberal direct use of, external libraries, as such libraries are out of your control and often exhibit a less testable design than your own. Finally, you learned to avoid service lookups and go for passing your dependencies in through constructor parameters instead.

Though they're far from being hard-and-fast rules, keeping these guidelines in mind and stopping to think for a second before going against them is bound to help you devise more testable designs.

---

[14] Worried that your constructor is already taking in six parameters? It sounds like your code might be missing an abstraction or two—that's what such *data clumps* often reveal after a closer look. If that's the case, perhaps introducing a parameter object (http://www.refactoring.com/catalog/introduceParameterObject.html) would help.

And there's more! The internet is chock full of good information related to testable design. My favorites are probably the Google Tech Talk videos. If you have some time to spare, go to youtube.com and search for "The Clean Code Talks." A lot of good stuff there!

When you're done with those videos, we're going to move on to another topic altogether. In the next chapter, you're going to see how you might use other JVM languages for writing tests for your Java code.

JAVA/TESTING

Effective Unit Testing

Lasse Koskela

Test the components before you assemble them into a full application, and you'll get better software. For Java developers, there's now a decade of experience with well-crafted tests that anticipate problems, identify known and unknown dependencies in the code, and allow you to test components both in isolation and in the context of a full application.

Effective Unit Testing teaches Java developers how to write unit tests that are concise, expressive, useful, and maintainable. Offering crisp explanations and easy-to-absorb examples, it introduces emerging techniques like behavior-driven development and specification by example.

What's Inside

- A thorough introduction to unit testing
- Choosing best-of-breed tools
- Writing tests using dynamic languages
- Efficient test automation

Programmers who are already unit testing will learn the current state of the art. Those who are new to the game will learn practices that will serve them well for the rest of their career.

Lasse Koskela is a coach, trainer, consultant, and programmer. He hacks on open source projects, helps companies improve their productivity, and speaks frequently at conferences around the world. Lasse is the author of *Test Driven*, also published by Manning.

"A fantastic, definitive guide. It will boost your productivity and deployment effectiveness."
—Roger Cornejo, GlaxoSmithKline

"Changed the way I look at the Java development process. Highly recommended."
—Phil Hanna, SAS Institute, Inc.

"A common sense approach to writing high quality code."
—Frank Crow
Sr. Progeny Systems Corp.

"Extremely useful, even if you write .NET code."
—J. Bourgeois, Freshly Coded

"If unit tests are a nightmare for you, read this book!"
—Franco Lombardo
Molteni Informatica

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/EffectiveUnitTesting

ISBN 13: 978-1-935182-57-3
ISBN 10: 1-935182-57-9
53999
9 781935 182573

MANNING $39.99 / Can $41.99 [INCLUDING eBOOK]