SAMPLE CHAPTER

Effective UNIT TESTING

A guide for Java developers

MANNING

LASSE KOSKELA

Effective Unit Testing

by Lasse Koskela

Chapter 2

brief contents

PART 1 FOUNDATIONS 1

1 ■ The promise of good tests 3
2 ■ In search of good 15
3 ■ Test doubles 27

PART 2 CATALOG 45

4 ■ Readability 47
5 ■ Maintainability 78
6 ■ Trustworthiness 115

PART 3 DIVERSIONS 137

7 ■ Testable design 139
8 ■ Writing tests in other JVM languages 156
9 ■ Speeding up test execution 170

2 In search of good

In this chapter

- What makes a test "good"?
- Testing relevant behavior
- The importance of reliable tests

We're on a journey of learning about good tests. We want to learn to identify good tests, write good tests, and improve not-so-good tests so they become good tests—or at least closer to being good tests. The question is, What makes a test "good"? What are the magic ingredients? There are several aspects to consider, including:

- The test code's readability and maintainability
- How the code is organized both within the project and within a given source file
- What kinds of things a test is checking for
- How reliable and repeatable a test is
- How a test makes use of test doubles

We'll be taking a closer look at all of these aspects in this chapter.

The preceding list is far from being comprehensive. The range of factors that may tip your test-quality scale either way is endless. Similarly, some of the factors

don't matter that much in all contexts. For some tests, their execution speed may be crucial, whereas for other tests, being extremely focused is key.

Furthermore, some of the quality of test code is in the eye of the beholder. As is the case with code in general, personal preference has a role in defining what "good" is—I'm not going to pretend that it wouldn't. I'm also not going to pretend that I can avoid my bias and preference from coming through in this book. Though I've tried to steer away from pure matter-of-taste questions, you'll find numerous sections where my opinions clearly show through. I think that's fine. After all, the best I can offer is my honest (and opinionated) view of things based on my experience, shaped by the wonderful individuals and software professionals from whom I've learned about code and, specifically, about test code.

With that disclaimer out of the way, let's discuss some of those aspects of test quality and establish what about them makes them relevant to our interests.

2.1 Readable code is maintainable code

Yesterday I popped into our office on my way back from a consulting gig and struck up a conversation with a colleague about an upcoming 1K competition that my colleague was going to attend. Such competitions are an age-old tradition at demo parties —a type of geek gathering where hackers gather at a massive arena for a long weekend with their computers, sleeping bags, and energy drinks. Starting from the first gatherings, these hackers have faced off, wielding their mad skills at producing 3D animations with what most people would today consider antiquated hardware.

A typical constraint for such animations has been size. In the case of the competition my colleague was preparing for, the name *1K* refers to the maximum size of the code compiled into a binary executable, which must be less than 1,024 bytes. Yes, that's right—1,024 bytes. In order to squeeze a meaningful program into such a tiny space, the competitors need to resort to all kinds of tricks. For example, one common trick to pack your code more tightly is to use the same name for many variables—because the resulting code compresses a bit better like that. It's crazy.

What's also crazy is the resulting code. When they're done squeezing the code down to 1,024 bytes, the source code is undecipherable. You can barely recognize which programming language they've used! It's essentially a write-only code base—once you start squeezing and compressing, you can't make functional changes because you wouldn't be able to tell what to edit where and how.

To give you a vivid taste of what such code might look like, here's an actual submission from a recent JS1k competition where the language of choice is JavaScript and it needs to fit into 1,024 bytes:

```
<script>with(document.body.style){margin="0px";overflow="hidden";}
var w=window.innerWidth;var h=window.innerHeight;var ca=document.
getElementById("c");ca.width=w;ca.height=h;var c=ca.getContext("2d");
m=Math;fs=m.sin;fc=m.cos;fm=m.max;setInterval(d,30);function p(x,y,z){
return{x:x,y:y,z:z};}function s(a,z){r=w/10;R=w/3;b=-20*fc(a*5+t);
return p(w/2+(R*fc(a)+r*fs(z+2*t))/z+fc(a)*b,h/2+(R*fs(a))/z+fs(a)*b);
}function q(a,da,z,dz){var v=[s(a,z),s(a+da,z),s(a+da,z+dz),s(a,z+dz)]
```

```
;c.beginPath();c.moveTo(v[0].x,v[0].y);for(i in v)c.lineTo(v[i].x,v[i]
.y);c.fill();}var Z=-0.20;var t=0;function d(){t+=1/30.0;c.fillStyle=
"#000";c.fillRect(0,0,w,h);c.fillStyle="#f00";var n=30;var a=0;var da=
2*Math.PI/n;var dz=0.25;for(var z=Z+8;z>Z;z-=dz){for(var i=0;i<n;i++){
fog=1/(fm((z+0.7)-3,1));if(z<=2){fog=fm(0,z/2*z/2);}var k=(205*(fog*
Math.abs(fs(i/n*2*3.14+t)))>>0;k*=(0.55+0.45*fc((i/n+0.25)*Math.PI*5)
);k=k>>0;c.fillStyle="rgb("+k+","+k+","+k+")";q(a,da,z,dz);if(i%3==0){
c.fillStyle="#000";q(a,da/10,z,dz);}a+=da;}}Z-=0.05;if(Z<=dz)Z+=dz;}
</script>
```

Granted, that is a couple of magnitudes more extreme a situation than what you'd find at a typical software company. But we've all seen code at work that makes our brains hurt. Sometimes we call that kind of code *legacy* because we've inherited it from someone else and now we're supposed to maintain it—except that it's so difficult that our brains hurt every time we try to make sense of it. Maintaining such unreadable code is hard work because we expend so much energy understanding what we're looking at. It's not just that. Studies have shown that poor readability correlates strongly with defect density.[1]

Automated tests are a useful protection against defects. Unfortunately, automated tests are also code and are also vulnerable to bad readability. Code that's difficult to read tends to be difficult to test, too, which leads to fewer tests being written. Furthermore, the tests we do write often turn out to be far from what we consider good tests because we need to kludge our way around the awkwardly structured, difficult-to-understand code with APIs and structures that aren't exactly test-friendly.

We've established (almost to the point of a rant) that code readability has a dire impact on the code's maintainability. Now what about the readability of test code? How is that different, or is it any different? Let's take a look at a not-so-far-fetched example of unreadable test code, shown in this listing:

Listing 2.1 Code doesn't have to be complex to lack readability

```
@Test
public void flatten() throws Exception {
    Env e = Env.getInstance();
    Structure k = e.newStructure();
    Structure v = e.newStructure();
    //int n = 10;
    int n = 10000;
    for (int i = 0; i < n; ++i) {
        k.append(e.newFixnum(i));
        v.append(e.newFixnum(i));
    }
    Structure t = (Structure) k.zip(e.getCurrentContext(),
            new IObject[] {v}, Block.NULL_BLOCK);
    v = (Structure) t.flatten(e.getCurrentContext());
    assertNotNull(v);
}
```

[1] Raymond P.L. Buse, Westley R. Weimer. "Learning a Metric for Code Readability." *IEEE Transactions on Software Engineering*, 09 Nov. 2009. IEEE computer Society Digital Library. IEEE Computer Society, http://doi.ieeecomputersociety.org/10.1109/TSE.2009.70

What is this test checking? Would you say that it's easy to decipher what's going on here? Imagine yourself being the new guy on this team—how long would it take you to figure out the test's intent? What kind of code forensics would you have to go through to grok the situation if this test suddenly starts failing? Based on how I feel about that hideous snippet of code, I'll wager that you immediately identified a handful of things that could be improved about the poor little test—and that readability is a common theme with those improvements.

2.2 Structure helps make sense of things

I've had the pleasure and horror of seeing numerous code bases that weren't full of beautiful strides of genius flowing from one source file to the next. Some of them never jumped to another source file because it was all there—all of the code and logic triggered by, say, the submission of a web form would reside in a single source file. I've had a text editor crash due to the size of a source file I foolishly tried to open. I've seen a web application vomit an error because a JavaServer Pages file had grown so big that the resulting byte code violated the Java class file specification. It's not just that structure would be useful—the lack of structure can be damaging.

What's common among most of these instances of never-ending source listings is that nobody wanted to touch them. Even the simplest conceptual changes would be too difficult to map onto the source code in front of you. There was no structure your brain could rely on. Divide and conquer wasn't an option—you had to juggle the whole thing in your head or be prepared for a lot of contact between your forehead and the proverbial brick wall.

As illustrated by figure 2.1 you don't want just *any* structure to help make sense of things. You need structure that makes sense as such—one that's aligned with the way your brain and your mental models are prepared to slice and dice the world. Blindly externalizing snippets of code into separate source files, classes, or methods does reduce the amount of code you're looking at a given point in time, thereby alleviating the problem of overloading your brain. But it doesn't get you much closer to isolating and understanding the one aspect of the program's logic that we're interested in right now. For that you need a structure that makes sense.

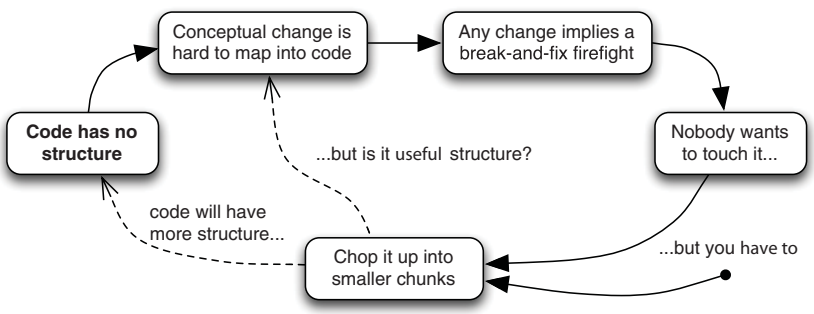

Figure 2.1 It's not just about having structure—it needs to be a useful structure.

When faced with monolithic, never-ending source listings, the obvious solution is to chop them up into smaller pieces, extracting blocks of code into methods. You might go from a huge 500-line method in one class to dozens of methods on 10 classes with the average length dropping below 10 lines per method. That would introduce more structure into the code—at least if you ask a compiler. You'd also be able to see whole methods on your screen at once instead of scrolling back and forth.

But if the boundaries for splitting up that monolith don't make sense—if they don't map to the domain and your abstractions—we might be doing more harm than good because the concepts might now be physically scattered farther from each other than before, increasing the time you spend going back and forth between source files. It's simple. What matters is whether the structure of your code helps you locate the implementation of higher-level concepts quickly and reliably.

Test code is an excellent example of this phenomenon. Let's say you have an application that's fairly well covered by an automated test—one automated test. Imagine that this test exercises all of the application's business logic and behavior through a single monolithic test method that takes half an hour to execute. Now say that the test eventually fails, as illustrated in figure 2.2, because you're making a change in how mailing addresses are represented internally in the application and you mess up something while you're at it. There's a bug. What happens next?

I'd imagine it's going to take a while to pinpoint the exact place in the test code where your programming error manifests itself. There's no structure in the test code to help you see what affects what, where a certain object is instantiated, what the value of a given variable is at the point where things fall apart, and so forth. Eventually, when you've managed to identify and correct your mistake, you have no choice but to run the whole test—all 30 minutes of it—to make sure that you really did fix the problem and that you didn't break something else in the process.

Continuing this thought experiment, fast-forwarding an hour or so, you're about to make another change. This time, having learned from your previous mistake, you're careful to make sure you've understood the current implementation in order to ensure that you'll make the right kind of change. How would you do that? By reading the code, and especially perusing test code that shows you in concrete terms how the production code is expected to behave. Except that you can't find the relevant parts of the test code because there's no structure in place.

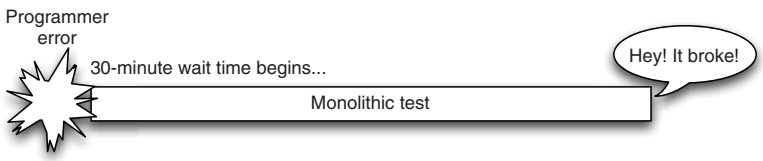

Figure 2.2 Long delay in feedback is a real productivity-killer

What you need are focused tests that are readable, accessible, and comprehensible so that you can:

- Find test classes that are relevant for the task at hand
- Identify the appropriate test methods from those classes
- Understand the lifecycle of objects in those test methods

These are all things that you can get by paying attention to your tests' structure and making sure that it's useful. Having a useful structure is hardly enough, of course.

2.3 It's not good if it's testing the wrong things

More than once I've concluded a journey of reading and debugging code to find the cause for an undesirable system behavior at almost the same place I started looking. A particularly annoying detail overlooked in such a bug-hunt is the contents of a test. The first thing I tend to do when digging into code is to run all the tests to tell me what's working and what's not. Sometimes I make the mistake of trusting what the tests' names tell me they're testing. Sometimes it turns out that those tests are testing something completely different.

This is related to having a good structure—if a test's name misrepresents what it tests, it's akin to driving with all the road signs turned the wrong way. You should be able to trust your tests.

A couple of years ago I was carrying out a code audit for a product that had been under development for more than a decade. It was a big codebase and I could tell from its structure that some parts were clearly newer than others. One of the things that distinguished more recent code from the older was the presence of automated tests. But I quickly found out that I couldn't tell from the tests' names what they were supposed to verify and, looking closer, it turned out that the tests weren't actually testing what they promised to test. It wasn't a Java codebase but I've taken the freedom to translate a representative example to Java:

```
public class TestBmap {
    @Test
    public void mask() {
        Bmap bmap = new Bmap();
        bmap.addParameter(IPSEC_CERT_NAME);
        bmap.addParameter(IPSEC_ACTION_START_DAYS, 0);
        bmap.addParameter(IPSEC_ACTION_START_HOURS, 23);
        assertTrue(bmap.validate());
    }
}
```

Looking at this code you'll immediately notice that the test's name is less than perfect. But on a closer look it turns out that whatever "mask" means for a "Bmap," the test is only checking that certain parameters are a valid combination. Whether the validation works is somewhat irrelevant if the actual behavior isn't correct even when the input is valid.

There's a lot to be said about testing the right things, but it's also crucial to test those right things the right way. Of particular importance from a maintainability point of view is that your tests are checking for the intended behavior and not a specific implementation. That's a topic we'll touch on in the next chapter, so let's leave it at that for now.

2.4 Independent tests run easily in solitude

There's a lot to be said about tests, what they should or shouldn't contain, what they should or shouldn't specify, and how they should be structured in the name of readability. What goes on *around* tests sometimes plays an equally vital role.

Human beings—our brains to be more exact—are enormously powerful information processors. We can make seemingly instant evaluations of what's going on in our physical surroundings and react in a blink. We dodge that incoming snowball before we even realize what we're reacting to. These reactions are in our DNA. They're behavioral recipes that instruct our body to move when our senses observe a familiar pattern. Over time our cookbook of these recipes grows in sophistication, and we're soon lugging around a complex network of interconnected patterns and behaviors.

This happens at work, too. Exploring a foreign code base for the first time, we'll have formed a clear picture of the most prevalent conventions, patterns, *code smells*, and pitfalls within 15 minutes. What makes this possible is our ability to recognize a familiar pattern and be able to tell what else we're likely to see nearby.

> **WHAT'S A CODE SMELL?** A *smell* in code is a hint that something *might* be wrong with the code. To quote the Portland Pattern Repository's Wiki, "If something smells, it definitely needs to be checked out, but it may not actually need fixing or might have to just be tolerated."

For example, one of the first things I pay attention to when introducing myself to a new codebase is the size of methods. If methods are long I *know* that there are a bunch of other issues to deal with in those particular modules, components, or source files. Another signal I'm tuning to is how descriptive the names of the variables, methods and classes are.

Specifically in terms of test code, I pay attention to the tests' *level of independence*, especially near architectural boundaries. I do this because I've found so many code smells by taking a closer look at what's going on in those boundaries, and I've learned to be extra careful when I see dependencies to:

- Time
- Randomness
- Concurrency
- Infrastructure
- Pre-existing data
- Persistence
- Networking

What these things have in common is that they tend to complicate what I consider the most basic litmus test for a project's test infrastructure: can I check out a fresh copy from version control to a brand new computer I just unboxed, run a single command, lean back, and watch a full suite of automated tests run and pass?

Isolation and independence are important because without them it's much harder to run and maintain tests. Everything a developer needs to do to their system in order to run unit tests makes it that much more burdensome.

Whether you need to create an empty directory in a specific location in your filesystem, make sure that you have a specific version of MySQL running at a specific port number, add a database record for the user that the tests use for login, or set a bunch of environment variables—these are all things that a developer shouldn't need to do. All of these small things add up to increased effort and weird test failures.[2]

A characteristic of this type of dependency is that things like the system clock at the time of test execution or the next value to come out from a random number generator are *not in your control*. As a rule of thumb, you want to avoid erratic test failures caused by such dependencies. You want to put your code into a bench vise and control everything by passing it test doubles and otherwise isolating the code to an environment that behaves exactly like you need it to.

Don't rely on test order within a test class

The general context for the advice of not letting tests depend on each other is that you should not let tests in one class depend on the execution or outcome of tests in *another* class. But it really applies to dependencies within a single test class, too.

The canonical example of this mistake is when a programmer sets up the system in a starting state in a `@BeforeClass` method and writes, say, three consecutive `@Test` methods, each modifying the system's state, trusting that the previous test has done its part. Now, when the first test fails, all of the subsequent tests fail, but that's not the biggest issue here—at least you're alerted to something being wrong, right?

The real issue is when some of those tests fail for the *wrong* reason. For instance, say that the test framework decides to invoke the test methods in a different order. False alarm. The JVM vendor decides to change the order in which methods are returned through the Reflection API. False alarm. The test framework authors decide to run tests in alphabetical order. False alarm again.[3]

You don't want false alarms. You don't want your tests failing when the behavior they're checking isn't broken. That's why you shouldn't intentionally make your tests brittle by having them depend on each other's execution.

[2] If you can't find a way to avoid such manual configuration, at least make sure developers need to do it only once.

[3] If this sounds far-fetched, you should know that JUnit doesn't promise to run test methods in any particular order. In fact, several tests in the NetBeans project started breaking when Java 7 changed the order in which declared methods are returned through `Class#getDeclaredMethods()`. Oops. I guess they didn't have independent tests...

One of the most unusual examples of a surprising test failure is a test that passes as part of the whole test suite but fails miserably when it's run alone (or vice versa). Those symptoms reek of interdependent tests. They assume that another test is run before they are, and that the other test leaves the system in a particular state. When that assumption kicks you in the ankle, you have one hellish debugging session ahead.

To summarize, you should be extra careful when writing tests for code that deals with time, randomness, concurrency, infrastructure, persistence, or networking. As a rule of thumb, you should avoid these dependencies as much as you can and localize them into small, isolated units so that most of your test code doesn't need to suffer from the complications and you don't have to be on your toes all the time—just in those few places where you tackle the tricky stuff.

So how would that look in practice? What exactly should you do? For example, you could see if you can find a way to:

- Substitute test doubles for third-party library dependencies, wrapping them with your own adapters where necessary. The tricky stuff is then encapsulated inside those adapters that you can test separately from the rest of application logic.
- Keep test code and the resources they use together, perhaps in the same package.
- Let test code produce the resources it needs rather than keeping them separate from the source code.
- Have your tests set up the context they need. Don't rely on any other tests being run before the one you're writing.
- Use an in-memory database for integration tests that require persistence, as it greatly simplifies the problem of starting tests with a clean data set. Plus, they're generally superfast to boot up.
- Split threaded code into asynchronous and synchronous halves, with all application logic in a regular, synchronous unit of code that you can easily test without complications, leaving the tricky concurrency stuff to a small, dedicated group of tests.

Achieving test isolation can be difficult when working with legacy code that wasn't designed for testability and therefore doesn't have the kind of modularity you'd like to have. But even then, the gain is worth the effort of breaking those nasty dependencies and making your tests independent from their environment and from each other. After all, you need to be able to rely on your tests.

2.5 Reliable tests are reliable

In the previous section I said that sometimes a test is testing something completely different than what you thought it tests. What's even more distracting is that sometimes they don't test a damn thing.

A colleague used to call such tests *happy tests*, referring to a test happily executing a piece of production code—possibly all of its execution paths—without a single assertion being made. Yes, your test coverage reports look awesome as the tests are thoroughly executing every bit of code you've written. The problem is that such tests can only fail if the production code being invoked throws an exception. You can hardly rely on such tests to watch your back, can you? Especially if the programmers have had a convention to encapsulate all of the test methods' bodies into a `try-catch`.[4] This listing shows an example of one such bugger.

Listing 2.2 Can you spot the flaw in this test?

```
@Test
public void shouldRefuseNegativeEntries() {
    int total = record.total();
    try {
        record.add(-1);
    } catch (IllegalArgumentException expected) {
        assertEquals(total, record.total());
    }
}
```

Some tests are less likely to fail than others, and the previous listing is a prime example of the other extreme, where the test is likely to never fail (and likely never has in the past either). If you look carefully, you'll notice that the test won't fail even if `add(-1)` doesn't throw an exception as it's supposed to.

Tests that can hardly ever fail are next to useless. With that said, a test that passes or fails intermittently is an equally blatant violation toward fellow programmers; see figure 2.3.

Some years ago I was consulting on a project and spent most of my days pair programming with the client's technical staff and other consultants. One morning I took on a new task with my pair and ran the related test set as the first thing to do as usual. My pair was intimately familiar with the codebase, having written a significant part of it, and was familiar with its quirks, too. I noticed this when a handful of the tests failed on the first run before we'd touched anything. What tipped me off was how my pair responded to the test failure—he routinely started rerunning the tests again and again until after four or five times all of the tests had passed at least once. I'm not 100% sure, but I don't think those particular tests all passed at the same time even once.

Figure 2.3 Tests have little value if they've never failed or if they're failing all the time.

[4] True story. I spent an hour removing them and the rest of the day fixing or deleting the failed tests that were uncovered by my excavation.

As flabbergasted as I was, I realized that I was looking at a bunch of tests that represented a whole different cast of unreliable tests. These particular tests turned out to be failing randomly because the code under test incorporated nondeterministic logic that screwed up the tests some 50% of the time. In addition to the use of pseudo-random generators in the code being tested, a common cause for such intermittently failing behavior is the use of time-related APIs. My favorite is a call to `System.currentTimeMillis()`, but a close second is a ubiquitous `Thread.sleep(1000)` sprinkled throughout in an attempt to test asynchronous logic.

In order to rely on your tests, they need to be repeatable. If I run a test twice, it must give me the same result. Otherwise, I'll have to resort to manual arbitration after every build I make because there's no way to tell whether 1250/2492 tests means that everything's all right or that all hell's broken loose with that last edit. There's no way to tell.

If your logic incorporates bits that are asynchronous or dependent on current time, be sure to isolate those bits behind an interface you can use for substituting a "test double" and make the test repeatable—a key ingredient of a test being reliable.

2.6 Every trade has its tools and tests are no exception

What's this test double I speak of? If you don't have test doubles in your programmer's toolkit, you're missing out on a lot of testing power. *Test double* is an umbrella term for what some programmers have come to know as *stubs*, *fakes*, or *mocks* (which is short for mock object). Essentially they're objects that you substitute for the real implementation for testing purposes. See figure 2.4.

You could say that test doubles are a test-infected programmer's best friend. That's because they facilitate many improvements and provide many new tools for our disposal, such as:

- Speeding up test execution by simplifying the code that would otherwise be executed
- Simulating exceptional conditions that would otherwise be difficult to create
- Observing state and interaction that would otherwise be invisible to your test code

There's a lot more to test doubles than this and we'll come back to this topic in more detail in the next chapter. But test doubles aren't the only tool of the trade for writing good automated tests.

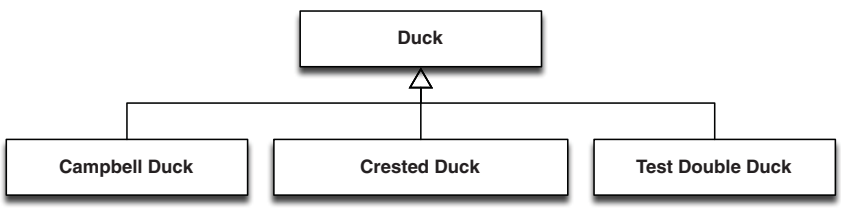

Figure 2.4 A test double for a duck looks just like a duck and quacks *almost* like a duck—but certainly isn't a real duck.

Perhaps the most essential tool of the trade is a test framework such as JUnit. I still remember some of my first attempts at getting a piece of code to work like I wanted it to. When I messed up and got stuck, I'd uncomment half a dozen statements that would print to the console and relaunch the program so I could analyze the console output and figure out what I'd broken and how.

It wasn't more than a couple of months into my professional career when I first bumped into this same practice being upheld by commercial software developers. I trust that I don't need to point out how incredibly unprofessional and wasteful such a practice is compared to writing automated, repeatable tests with tools like JUnit.

In addition to a proper test framework and test doubles, my top three tools of the trade for software developers writing automated tests include one more tool in the chain—the build tool. Whatever your build process looks like, whichever tool or technology your build scripts use under the hood, there's no good excuse for not integrating your automated tests as part of that build.

2.7 Summary

In this chapter we've established several coarse properties for what a good test is. We noted that these things are dependent on context and that there are few absolute truths when it comes to what makes a test "good." We did identify a number of issues that generally have a major impact on how good or appropriate—how fit for its purpose—an automated test is.

We began by noting that one of the essential virtues for a test is its readability, because lacking the ability to be read and understood, test code becomes a maintenance problem that solves itself very soon—by getting deleted because it's too costly to maintain.

We then pointed out how test code's structure makes it usable, allowing the programmer to quickly find their way to the right pieces and helping the programmer understand what's going on—a direct continuation on readability.

Next we shed light on how tests sometimes test the wrong things and how that can create problems by leading you down the wrong path or by muddying the waters, hiding the test's actual logic and making the test itself unreadable.

To conclude the underlying theme of tests sometimes being unreliable, we identified some of the common reasons for such unreliability and how important it is for tests to be repeatable.

Lastly, we identified three essential tools of the trade for writing automated tests—a test framework, an automated build that runs tests written with that framework, and test doubles for improving your tests and ability to test. This third topic is important enough that we've dedicated the next chapter to discussing the use of test doubles for writing good tests.

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]