Sample Chapter

JUnit Recipes

Practical Methods for Programmer Testing

J. B. Rainsberger

with contributions by Scott Stirling

MANNING

JUnit Recipes
Practical Programmer Testing Methods

by J.B. Rainsberger

Chapter 1

1 Fundamentals

This chapter covers

- An introduction to Programmer Testing
- Getting started with JUnit
- A few good practices for JUnit
- Why testing replaces debugging

We hate debugging.

You look up at the clock to see how late it is because you *still* have a handful of defects that need to be fixed *tonight*. Welcome to the "fix" phase of "code and fix," which is now entering its third month. In that time, you have begun to forget what your home looks like. The four walls of your office—assuming you even *have* four walls to look at—are more familiar than you ever wanted them to be. You look at the "hot defects" list and see one problem that keeps coming back. You thought you fixed that last week! These testers...when will they leave you alone?!

Fire up the debugger, start the application server—grab a coffee because you have five minutes to kill—set a breakpoint or two, enter data in 10 text fields and then press the Go button. As the debugger halts at your first breakpoint, your goal is to figure out which object *insists* on sending you bad data. As you step through the code, an involuntary muscle spasm—no doubt from lack of sleep—causes you to accidentally step over the line of code that you *think* causes the problem. Now you have to stop the application server, fire up the debugger again, start the application server again, then grab a stale doughnut to go with your bitter coffee. (It was fresh six hours ago.) Is this really as good as it gets?

Well, no. As a bumper sticker might read, "We'd rather be Programmer Testing."

1.1 What is Programmer Testing?

Programmer Testing is not about testing programmers, but rather about programmers performing tests. In recent years some programmers have rediscovered the benefits of writing their own tests, something we as a community lost when we decided some time ago that "the testing department will take care of it." Fixing defects is expensive, mostly because of the time it takes: it takes time for testers to uncover the defect and describe it in enough detail for the programmers to be able to re-create it. It takes time for programmers to determine the causes of the defects, looking through code they have not seen for months. It takes time for everyone to argue whether something is really a defect, to wonder how the programmers could be so stupid, and to demand that the testers leave the programmers alone to do their job. We could avoid much of this wasted time if the programmers simply tested their own code.

The testing that programmers do is generally called *unit testing*, but we prefer not to use this term. It is overloaded and overused, and it causes more confusion than it provides clarity. As a community, we cannot agree on what a *unit* is—is it a method, a class, or a code path? If we cannot agree on what *unit* means, then there

is little chance that we will agree on what *unit testing* means. This is why we use the term *Programmer Testing* to describe testing done by programmers. It is also why we use the term *Object Tests* to mean tests on individual objects. Testing individual objects in isolation is the kind of testing that concerns us for the majority of this book. It is possible that this is different from what you might think of as testing.

Some programmers test their code by setting breakpoints at specific lines, running the application in debug mode, stepping through code line by line, and examining the values of certain variables. Strictly speaking, this is Programmer Testing, because a programmer is testing her own code. There are several drawbacks to this kind of testing, including:

- It requires a debugging tool, which not everyone has installed (or wants to install).
- It requires someone to set a breakpoint before executing a test and then remove the breakpoint after the test has been completed, adding to the effort needed to execute the test multiple times.
- It requires knowing and remembering the expected values of the variables, making it difficult for others to execute the same tests unless they know and remember those same values.
- It requires executing the entire application in something resembling a real environment, which takes time and knowledge to set up or configure.
- To test any particular code path requires knowing how the entire application works and involves a long, tedious sequence of inputs and mouse clicks, which makes executing a particular test prone to error.

This kind of manual Programmer Testing, while common, is costly. There is a better way.

1.1.1 The goal of Object Testing

We defined the term *Object Testing* as testing objects in isolation. It is the "in isolation" part that makes it different from the manual testing with which you are already familiar. The idea of Object Testing is to take a single object and test it by itself, without worrying about the role it plays in the surrounding system. If you build each object to behave correctly according to a defined specification (or contract), then when you piece those objects together into a larger system, there is a much greater chance that the system will behave the way you want. Writing Object Tests involves writing code to exercise individual objects by invoking their methods directly, rather than testing the entire application "from the outside." So what does an Object Test look like?

1.1.2 The rhythm of an Object Test

When writing an Object Test, a programmer is usually thinking, "If I invoke *this* method on *that* object, it should respond *so*." This gives rise to a certain rhythm—a common, recurring structure, consisting of the following sequence:

1. Create an object.
2. Invoke a method.
3. Check the result.

Bill Wake, author of *Refactoring Workbook*, coined the term the three "A"s to describe this rhythm: "arrange, act, assert." Remembering the three "A"s keeps you focused on writing an effective Object Test with JUnit. This pattern is effective because the resulting tests are repeatable to the extent that they verify *predictable* behavior: if the object is in *this* state and I do *that*, then *this* will happen. Part of the challenge of Object Testing is to reduce all system behavior down to these focused, predictable cases. You could say that this entire book is about finding ways to extract simple, predictable tests from complex software, then writing those tests with JUnit.

So how *do* you write Object Tests?

1.1.3 A framework for unit testing

In a paper called "Simple Smalltalk Testing: With Patterns,"[1] Kent Beck described how to write Object Tests using Smalltalk. This paper presented the evolution of a simple testing framework that became known as *SUnit*. Kent teamed up with Erich Gamma to port the framework to Java and called the result *JUnit*. Since 1999, JUnit has evolved into an industry standard testing and design tool for Java, gaining wide acceptance not only on open source (www.opensource.org) projects, but also in commercial software companies.

Kent Beck's testing framework has been ported to over 30 different programming languages and environments. The concepts behind the framework, known in the abstract as xUnit,[2] grew out of a few simple rules for writing tests.

---

[1] www.xprogramming.com/testfram.htm.

[2] Framework implementations replace *x* with a letter or two denoting the implementation language or platform, so there is SUnit for Smalltalk, JUnit for Java, PyUnit for Python, and others. You can find a more or less complete list of implementations at www.xprogramming.com/software.htm.

Tests must be automated

It is commonplace in the programming community to think of testing as entering text, pushing a button, and watching what happens. Although this *is* testing, it is merely one approach and is best suited for End-to-End Testing through an end-user interface. It is *not* the most effective way to test down at the object level. Manual code-level testing generally consists of setting a breakpoint, running code in a debugger, and then analyzing the value of variables. This process is time consuming, and it interrupts the programmer's flow, taking time away from writing working production code. If you could get the computer to run those tests, it would boost your effectiveness considerably. You *can* get the computer to run those tests if you write them as Java code. Because you're already a Java programmer and the code you want to test is written in Java, it makes sense to write Java code to invoke methods on your objects rather than invoking them by hand.

NOTE *Exploratory testing*—There is a common perception that automated testing and exploratory testing are opposing techniques, but if we examine the definition that James Bach gives in his article "What is Exploratory Testing?"[3] we can see that this is not necessarily the case. Exploratory testing is centered on deciding which test to write, writing it, then using that feedback to decide what to do next. This is similar to Test-Driven Development, a programming technique centered on writing tests to help drive the design of a class. An exploratory tester might perform some manual tests, learn something about the system being tested, keep the knowledge, and discard the tests. He values the knowledge gained more than the tests performed. In Test-Driven Development, a *test driver*[4] uses his tests as a safety net for further changes to the code, so it is important to develop and retain a rich suite of tests. In spite of these differences, the two approaches share a key trait: testing is focused on learning about the software. Exploratory testers learn how the software works, and test drivers learn how the software ought to be designed. We recommend using the exploratory testing approach in general, then automating the results when possible to provide continuous protection against regression. If you are trying to add tests to code that has no tests, you will find the exploratory testing techniques useful to reverse engineer the automated tests you need.

In addition to being automated, tests also need to be repeatable. Executing the same test several times under the same conditions must yield the same results. If a

---

[3] www.satisfice.com/articles/what_is_et.htm.

[4] This is a bit of slang from the Test-Driven Development community: when you are writing code using the techniques of Test-Driven Development, you are said to be *test driving* the code.

test is not repeatable, then you will find yourself spending a considerable amount of time trying to explain why today's test results are different from yesterday's test results, even if there is no defect to fix. You want tests to help you uncover and prevent defects. If running a test costs you time and effort and does a poor job of uncovering or preventing defects, then why use the test?

Tests must verify themselves

Many programmers have already embraced automating their tests. They recognize the value in pressing a button to execute a stable, repeatable test. Because the programmer needs to analyze the value of variables, he often writes code to print the value of key variables to the screen, then looks at those values and judges whether they are correct. This process, while simple and direct, interrupts the programmer's flow; and worse, it relies on the programmer knowing (and remembering) which values to expect. When he is first working on a part of a system, this poses no problem, but four months from now he might not remember whether the value should be 37 or 39—he won't know whether the test passes or fails. To solve this problem, the test itself must know the expected result and tell us whether it passes or fails. This is easy to do: add a line of code to the end of the test that says, "This variable's value should be 39: print OK if it is and Oops if it is not."

Tests must be easy to run simultaneously

As soon as you have automated, self-verifying tests in place, you'll find yourself wanting to run those tests often. You will come to rely on them to give you immediate feedback as to whether the production code you are writing behaves according to your expectations. You will build up sizable collections of tests that verify the simple cases, the boundary conditions, and the exceptional cases that concern you. You will want to run all these tests in a row and let them tell you whether any of them failed. You could run each test one by one—you could even write little scripts to run many of them in succession, but eventually you want to concentrate on writing the tests without worrying about how to execute them. You can do this by grouping tests together into a common place, such as the same Java class, then providing some automatic mechanism for extracting the tests from the class and executing them as a group. You can write this "test extraction" engine once and then use it over and over again.

1.1.4 Enter JUnit

JUnit was created as a framework for writing automated, self-verifying tests in Java, which in JUnit are called *test cases*. JUnit provides a natural grouping mechanism for related tests, which it calls a *test suite*. JUnit also provides *test runners* that you

can use to execute a test suite. The test runner reports on the tests that fail, and if none fail, it simply says "OK." When you write JUnit tests, you put all your knowledge into the tests themselves so that they become entirely programmer independent. This means that anyone can run your tests without knowing what they do, and if they are curious, they only need to read the test code. JUnit tests are written in a familiar language—Java—and in a style that is easy to read, even for someone new to this style of testing.

1.1.5 Understanding Test-Driven Development

Many of the recommendations we make in this book come from our experience with Test-Driven Development. This is a style of programming based on the fundamental idea that if you write a test *before* you write the production code to pass that test, you derive several benefits free of charge:

- Your system is entirely covered by tests.
- You build your system from loosely coupled, highly cohesive objects.
- You make steady progress, improving the system incrementally by making one test pass, then another, then another, and so on.
- A passing test is never more than a few minutes away, giving you confidence and continual positive feedback.

In addition to writing the test first, you *refactor* code as you write; that is, you identify ways to improve the design of your system as you build it with the goal of reducing the cost of building new features. A *well-factored* design is one that is free of duplication, has only the classes it needs, and is self-documenting in the sense that classes and methods have names that make it clear what they are or do. Such a system is easy to change, easy to extend, and easy to understand, all of which make for happy programmers, happy project managers, happy end users, and ultimately, happy CEOs.[5]

There is a large and growing community of Test-Driven Development (TDD) practitioners (or test drivers), including the authors of this book. In spite of our enthusiasm for this style of programming, this is *not* a book on TDD, but a book on using JUnit effectively. We highly recommend Kent Beck's *Test-Driven Development: By Example*[6] for a more thorough treatment of TDD. We hope that this book will serve as a companion to Beck's work, at least for Java programmers.

---

[5] We would also hope it makes for happy stockholders, but that is generally beyond our control.

[6] Kent Beck, *Test-Driven Development: By Example*. Addison-Wesley, 2002.

1.2 Getting started with JUnit

To this point we have decided that there is more to testing than setting breakpoints and looking at the values of variables. We have introduced JUnit, a framework for repeatable, self-verifying Object Tests. The next step is to start writing some JUnit tests, so let us look at downloading and installing JUnit as well as writing and executing tests.

1.2.1 Downloading and installing JUnit

JUnit is easy to install and use. To get JUnit up and running, you must:

1. Download the JUnit package.
2. Unpack JUnit to your file system.
3. Include the JUnit *.jar file on the class path when you want to build or run tests.

Downloading JUnit

At press time, the best place to find JUnit is at www.junit.org. You will find a download link to the latest version of the product. Click the Download link to download the software to your file system.

Unpacking JUnit

You can unpack JUnit to any directory on your file system using your favorite *.zip file utility. Table 1.1 describes the key files and folders in the JUnit distribution.

Table 1.1 What's inside the JUnit distribution

File / Folder	Description
junit.jar	A binary distribution of the JUnit framework, extensions, and test runners
src.jar	The JUnit source, including an Ant buildfile
junit	Samples and JUnit's own tests, written in JUnit
javadoc	Complete API documentation for JUnit
doc	Documentation and articles, including "Test Infected: Programmers Love Writing Tests" and some material to help you get started

To verify your installation, execute the JUnit test suite. That's right: JUnit is distributed with its own tests written using JUnit! To execute these tests, follow these steps:

1. Open a command prompt.
2. Change to the directory that contains JUnit (D:\junit3.8.1 or /opt/junit3.8.1 or whatever you called it).
3. Issue the following command:

```
> java -classpath junit.jar;. junit.textui.TestRunner
≫  junit.tests.AllTests
```

You should see a result similar to the following:

```
.....................................
.....................................
.........
Time: 2.003

OK (91 tests)
```

For each test, the test runner prints a dot to let you know that it is making progress. After it finishes executing the tests, the test runner says "`OK`" and tells you how many tests it executed and how long it took.

*Including the *.jar file on your class path*

Look at the command you used to run the tests:

```
> java -classpath junit.jar;. junit.textui.TestRunner junit.tests.AllTests
```

The class path includes junit.jar and the current directory. This file must be in the class path both when you compile your tests and when you run your tests. This is also the *only* file you need to add to your class path. This is a simple procedure because the current directory—the one where you unpacked JUnit—happens to be the location of the *.class files for the JUnit tests.

The next parameter, `junit.textui.TestRunner`, is the class name of a text-based JUnit test runner. This class executes JUnit tests and reports the results to the console. If you want to save the test results for later review, redirect its output to a file. If the tests do not run properly, see chapter 8, "Troubleshooting JUnit," for details. If you have trouble executing your tests, or you need to execute them a special way, see chapter 6, "Running JUnit Tests," for some solutions.

The last parameter, `junit.tests.AllTests`, is the name of the test suite to run. The JUnit tests include a class `AllTests` that builds a complete test suite containing about 100 tests. Read more about organizing tests in chapter 3, "Organizing and Building JUnit Tests."

1.2.2 Writing a simple test

Now that you can execute tests, you'll want to write one of your own. Let us start with the example in listing 1.1.

Listing 1.1 Your first test

```
package junit.cookbook.gettingstarted.test;

import junit.cookbook.util.Money;
import junit.framework.TestCase;

public class MoneyTest extends TestCase {
    public void testAdd() {
        Money addend = new Money(30, 0);
        Money augend = new Money(20, 0);

        Money sum = addend.add(augend);
        assertEquals(5000, sum.inCents());
    }
}
```

Create a subclass of TestCase

Each test is a method

30 dollars, 0 cents

The parameters should be equal

This test follows the basic Object Test rhythm:

- It creates an object, called `addend`.
- It invokes a method, called `add()`.[7]
- It checks the result by comparing the return value of `inCents()` against the expected value of 5,000.

Without all the jargon, this test says, "If I add $30.00 to $20.00, I should get $50.00, which happens to be 5,000 cents."

This example demonstrates several aspects of JUnit, including:

- To create a test, you write a method that expresses the test. We have named this method `testAdd()`, using a JUnit naming convention that allows JUnit to find and execute your test automatically. This convention states that the name of a method implementing a test must start with "test."
- The test needs a home. You place the method in a class that extends the JUnit framework class `TestCase`. We will describe `TestCase` in detail in a moment.

[7] If you are confused as to what an *augend* is, blame Kent Beck (who, we're sure, will just blame Chet Hendrickson, but that's not our fault). We are simply repeating his discovery that this is the proper term for the second argument in addition: you add the *augend* to the *addend*. We can be an obscure bunch, we programmers.

- To express how you expect the object to behave, you make assertions. An assertion is simply a statement of your expectation. JUnit provides a number of methods for making assertions. Here, you use `assertEquals()`, which tells JUnit, "If these two values are not the same, the test should fail."
- When JUnit executes a test, if the assertion fails—in our case, if `inCents()` does not return 5,000—then the test fails; but if no assertion fails, then the test passes.[8]

These are the highlights, but as always, the devil is in the details.

1.2.3 Understanding the TestCase class

The `TestCase` class is the center of the JUnit framework. You will find `TestCase` in the package `junit.framework`. There is some confusion among JUnit practitioners—even among experienced ones—about the term *test case* and its relation to the `TestCase` class. This is an unfortunate name collision. The term *test case* generally refers to a single test, verifying a specific path through code. It can sound strange, then, to collect *multiple* test cases into a single class, itself a subclass of `TestCase`, with each test case implemented as a method on `TestCase`. If the class contains multiple test cases, then why call it `TestCase` and not something more indicative of a *collection* of tests?

Here is the best answer we can give you: to write a test case, you create a subclass of `TestCase` and implement the test case as a method on the new class; but at runtime, each test case executes as an instance of your subclass of `TestCase`. As a result, each test case is an instance of `TestCase`. Using common object-oriented programming terminology, each test case is a `TestCase` object, so the name fits. Still, a `TestCase` class contains many tests, which causes the confusion of terms. This is why we take great care to differentiate between a test case and a test case *class*: the former is a single test, whereas the latter is the class that contains multiple tests, each implemented as a different method. To make the distinction clearer, we will never (or at least almost never) use the term *test case*, but rather either *test* or *test case class*. As you will see later in this book, we also refer to the test case class as a *fixture*. Rather than cram more information into this short description, we will talk about fixtures later. For now, think of a fixture as a natural way to group tests together so that JUnit can execute them at once. The `TestCase` class provides a default mechanism for identifying which methods are tests, but you can collect the

[8] Exceptions might get in the way, but we'll discuss this in due time.

tests yourself in customized suites. Chapter 4, "Managing Test Suites," describes the various ways to build test suites from your tests.

The class `TestCase` extends a utility class named `Assert` in the JUnit framework. The `Assert` class provides the methods you will use to make assertions about the state of your objects. `TestCase` extends `Assert` so that you can write your assertions without having to refer to an outside class. The basic assertion methods in JUnit are described in table 1.2.

Table 1.2 The JUnit class `Assert` provides several methods for making assertions.

Method	What it does
`assertTrue(boolean condition)`	Fails if `condition` is false; passes otherwise.
`assertEquals(Object expected, Object actual)`	Fails if `expected` and `actual` are not equal, according to the `equals()` method; passes otherwise.
`assertEquals(int expected, int actual)`	Fails if `expected` and `actual` are not equal according to the == operator; passes otherwise. There is an overloaded version of this method for each primitive type: `int`, `float`, `double`, `char`, `byte`, `long`, `short`, and `boolean`. (See Note about `assertEquals()`.)
`assertSame(Object expected, Object actual)`	Fails if `expected` and `actual` refer to different objects in memory; passes if they refer to the same object in memory. Objects that are not the same might still be equal according to the `equals()` method.
`assertNull(Object object)`	Passes if object is `null`; fails otherwise.

JUnit provides additional assertion methods for the logical opposites of the ones listed in the table: `assertFalse()`, `assertNotSame()`, and `assertNotNull()`; but for `assertNotEquals()` you need to explore the various customizations of JUnit, which we describe in part 3, "More Testing Techniques."

NOTE Two of the overloaded versions of `assertEquals()` are slightly different. The versions that compare `double` and `float` values require a third parameter: a *tolerance level*. This tolerance level specifies how close floating-point values need to be before you consider them equal. Because floating-point arithmetic is imprecise at best,[9] you might specify "these two values can be within 0.0001 and that's close enough" by coding `assertEquals (expectedDouble, actualDouble, 0.0001d)`.

[9] See "What Every Computer Scientist Should Know About Floating-Point Arithmetic" (http://docs.sun.com/source/806-3568/ncg_goldberg.html).

1.2.4 Failure messages

When an assertion fails, it is worth including a short message that indicates the nature of the failure, even a reason for it. Each of the assertion methods accepts as an optional first parameter a `String` containing a message to display when the assertion fails. It is a matter of some debate whether the programmer should include a failure message *as a general rule* when writing an assertion. Those in favor claim that it only adds to the self-documenting nature of the code, while others feel that in many situations the context makes clear the nature of the failure. We leave it to you to try both and compare the results.[10]

We will add that `assertEquals()` has its own customized failure message, so that if an equality assertion fails you see this message:

```
junit.framework.AssertionFailedError: expected:<4999> but was:<5000>
```

Here, a custom failure message might not make the cause of the problem any clearer.

1.2.5 How JUnit signals a failed assertion

The key to understanding how JUnit decides when a test passes or fails lies in knowing how these assertion methods signal that an assertion has failed.

In order for JUnit tests to be self-verifying, *you* must make assertions about the state of your objects and *JUnit* must raise a red flag when your production code does not behave according to your assertions. In Java, as in C++ and Smalltalk, the way to raise a red flag is with an exception. When a JUnit assertion fails, the assertion method throws an exception to indicate that the assertion has failed.

To be more precise, when an assertion fails, the assertion method throws an error: an `AssertionFailedError`. The following is the source for `assertTrue()`:[11]

```
static public void assertTrue(boolean condition) {
    if (!condition)
        throw new AssertionFailedError();
}
```

When you assert that a condition is true but it is *not*, then the method throws an `AssertionFailedError` to indicate the failed assertion. The JUnit framework then catches that error, marks the test as failed, remembers that it failed, and moves on to the next test. At the end of the test run, JUnit lists all the tests that failed; the rest are considered as having passed.

---

[10] As Ron Jeffries asks, "Speculation or experimentation—which is more likely to give you the correct answer?"

[11] Well, not exactly, because the code is highly factored. Rather than show you three methods, we have translated them into one that does the same thing.

1.2.6 The difference between failures and errors

You normally don't want the Java code that you write to throw errors, but rather only exceptions. General practice leaves the throwing of errors to the Java Virtual Machine itself, because an error indicates a low-level, unrecoverable problem, such as not being able to load a class. This is the kind of stuff from which we mortals cannot be expected to recover. For that reason, it might seem strange for JUnit to throw an error to indicate an assertion failure.

JUnit throws an error rather than an exception because in Java, errors are unchecked; therefore, not every test method needs to declare that it might throw an error.[12] You might suppose that a `RuntimeException` would have done the same job, but if JUnit threw the kinds of exceptions your production code might throw, then JUnit tests might interfere with your production. Such interference would diminish JUnit's value.

When your test contains a failed assertion, JUnit counts that as a failed test; but when your test throws an exception (and does not catch it), JUnit counts that as an error. The difference is subtle, but useful: a failed assertion usually indicates that the production code is wrong, but an error indicates that there is a problem either with the test itself or in the surrounding environment. Perhaps your test expects the wrong exception object or incorrectly tries to invoke a method on a `null` reference. Perhaps a disk is full, or a network connection is unavailable, or a file is not found. JUnit cannot conclude that your production code is at fault, so it throws up its hands and says, "Something went wrong. I can't tell whether the test would pass or fail. Fix the problem and run this test again." That is the difference between a failure and an error.

JUnit's test runners report the results of a test run in this format: "78 run, 2 failures, 1 error."[13] From this you can conclude that 75 tests passed, 2 failed, and 1 was inconclusive. Our recommendation is to investigate the error first, fix the problem, and then run the tests again. It might be that with the error out of the way, all the tests pass!

---

[12] It is entirely possible that checked exceptions in Java are a waste of time. As we write these words, "checked exceptions are evil" returns 18,500 results from Google. Join the debate!

[13] Looks like you have some work to do!

1.3 A few good practices

The recipes in this book reflect a collection of good practices that we have gathered through hard-won experience. We did not learn them all in one day and neither will you, but we think that the following *good practices* give you a suitable place to start.

1.3.1 Naming conventions for tests and test classes

Names are important. The names of your objects, methods, parameters, and packages all play a significant role in communicating what your system is and does. A programmer ought to be able to sit down at a computer, browse your code, and form an accurate mental model of your system just by reading the names. If not, then the names are wrong. We understand that this is a strong statement. We also understand that finding the right name is not always easy—far from it. We understand that naming the parts of your system is at times difficult, but we believe that finding the right name is always worth the effort.

So it is with your tests: name your tests precisely. The name of a test should summarize the test in a few words, because if it can't, then you need to write additional documentation explaining the test. How much better can you explain what a test does than by pointing to the code that implements it? Your goal is to make that code as easy to read as this book—or easier.

Naming tests

Start with the test itself: the name of a test should describe the behavior you are testing, rather than how the test is implemented. In other words, a test name should *reveal intent*. If you are writing an object in a banking system, then you will write a test for the special case where someone attempts to withdraw too much money. You might name this test `testWithdrawWhenOverdrawn()`, or `testWithdrawWithInsufficientFunds()`. Perhaps even `testWithdrawTooMuch()` suffices. But `testWithdraw200Dollars()` is a doubtful choice: it might describe what the test *does*, but not *why*. Confronted with seeing this test name for the first time, a programmer would ask, "Why is withdrawing $200 so special?" We recommend making that obvious in the name of the test. For the *happy path* cases (the straight-ahead scenario in which nothing goes wrong), we recommend simply naming your test after the behavior you are testing. For the test that withdraws money successfully from an account, the name `testWithdraw()` is perfect. If you do not say otherwise, the reader can assume that you are testing a happy path.

One common convention is to use the underscore character between the name of the behavior and the particular special case. Some programmers prefer names such as `testWithdraw_Overdrawn()` or `testWithdraw_Zero()`. This makes the test methods easier to read: it separates the behavior under test from the special conditions being tested. We endorse this convention, although after three or four special cases for a given behavior, you should consider moving these special case tests to a separate *test fixture*—a class that defines both tests and the objects on which those tests operate. See recipe 3.7, "Move special case tests to a separate test fixture," for a full discussion of this technique.

Naming test case classes

When naming your test case classes, the simplest guideline is to name the test case class after the class under test. In other words, your tests for `Account` go into the class `AccountTest`, your tests for `FileSystem` go into the class `FileSystemTest`, and so on. The principal benefit of this naming convention is that it is easy to find a test, as long as you know the class being tested. While we recommend starting out with this naming convention, it is important to understand that this is *only* a naming convention, and not a requirement of JUnit. We are surprised by the number of programmers who ask us what to do about test case classes that grow unusually large or become difficult to navigate. Any class that becomes too large should be split into smaller classes, JUnit or otherwise. We recommend identifying the tests that share a common fixture and factoring them into a separate test case class, to which the questioner responds, "But don't they all have to be in the same `TestCase` class?" The answer to that is no!

We mentioned earlier that each test is an instance of your subclass of `TestCase`. What we did not mention at the time is that your test case class is simply a container for related tests. How to distribute your tests into the various test case classes is up to you. There are some useful guidelines, which we describe in chapter 3. If you find that three tests belong together and should be separated from the rest of the tests for that class, then move them. We recommend naming the new test fixture after what those tests have in common. If there are six special cases for withdrawing money from an account, then move them into a test case class called `AccountWithdrawalTest` or `WithdrawFromAccountTest`, depending on whether you prefer noun phrases or verb phrases. We prefer verb phrases and would likely choose `WithdrawFromAccountTest`.

1.3.2 Test behavior, not methods

This brings us to another recommendation when writing tests: your tests should focus on the behavior they are testing without worrying about which class is under test. This is why our test names tend to be verbs rather than nouns: we test behavior (verbs) and not classes (nouns). Still, the difference between behavior and methods might not be clear: we implement behavior as methods, so testing behavior *must* be about test methods. But that's not entirely true. We *do* implement behavior as methods, but the way we choose to implement a certain behavior depends on a variety of factors, some of which boil down to personal preference. We make a number of decisions when implementing behavior as methods: their names, their parameter lists, which methods are public and which are private, the classes on which we place the methods, and so on—these are *some* of the ways in which methods might differ, even though the underlying behavior might be the same. The implementation can vary in ways that the tests do not need to determine.

Sometimes a single method implements all the required behavior, and in that case, testing that method directly is all you need. More complex behaviors require the collaboration of several methods or even objects to implement. If you let your tests depend too much on the particulars of the implementation, then you create work for yourself as you try to refactor (improve the design). Furthermore, some methods merely participate in a particular feature, rather than implement it. Testing those methods in isolation might be more trouble than it is worth. Doing so drives up the complexity of your test suite (by using more tests) and makes refactoring more difficult—and all for perhaps not much gain over testing the behavior at a slightly higher level. By focusing on testing behavior rather than each individual method, you can better strike a balance between test coverage and the freedom you need to support refactoring.

To illustrate the point, consider testing a stack. Recall that a stack provides a few basic operations: push (add to the top of the stack), pop (remove from the top of the stack), and peek (look at the top of the stack). When deciding how to test a stack implementation, the following tests spring to mind:

- Popping an empty stack fails somehow.
- Peeking at an empty stack shows nothing.
- Push an object onto the stack, then peek at it, expecting the same object you pushed.

- Push an object onto the stack, then pop it, expecting the same object you pushed.
- Push two different objects onto the stack, then pop twice, expecting the objects in reverse order from the order in which you pushed them.

Notice that these tests focus on *fixture*—the state of the stack when the operation is performed—rather than on the operation itself. Notice also how the methods combine to provide the desired behavior. If `push()` does not work, then there is no good way to verify `pop()` and `peek()`. We can say the same for the other methods. Moreover, when using a stack, you use all three operations, so it does not make sense to test them in isolation, but rather to test that the stack generally behaves correctly, depending on its contents. Does this point to a poor design where methods are overly coupled? No. Instead it reinforces the fact that an object is a cohesive collection of related operations on the same set of data. The object's *overall* behavior—a composite of the behaviors of its methods in a given state—is what is important, so we recommend focusing your test effort on the object as a whole, rather than its parts.

1.4 Summary

We all have the experience of having to track down a defect, whether in our code or someone else's. This horrible act that we call *debugging* generally consists of two different, concomitant activities: reasoning about what *might* have gone wrong and groping in the dark for the slightest clue as to what *actually* went wrong. When we are debugging, we start with the former; but when it becomes clear that the problem is worse than we feared, we end up doing the latter, and at that point there is no way to know when (or if) we will solve the problem.

The most elementary debugging technique has become known as `printf`, after the C library function to print text to the screen. Even Java programmers will say, "Throw some `printf`s in there and see what's going on" (although it might be the fashion to say `println` because that's what Java calls it). The idea is to narrow down the location of the defect to some reasonably small area of the code and then litter it with temporary code that prints the value of variables to the screen. You then run the system, reproduce the defect, and analyze the values of the variables.

Does this sound familiar yet?

What you are doing is exploratory testing, as James Bach described it. The only difference is that you might not have access to a debugger to make the process

smoother: you only have a log file or the console to look at, and you had better look quickly, because the `println` statements whiz past you before you know it!

Why not write a test instead? You have already identified the area of the code that seems to exhibit the problem by narrowing it down to someplace that invokes a method on an object. Write a test that exercises that method. Rather than grope around looking at the value of every variable you can find, determine the input to the method where the defect occurs and then use that input in your test. Make an assertion with the result you expect, and then run the test. Keep adding tests until you find the source of the problem; then when you change the offending code, run your test to verify that the problem is solved. Doing this increases the frequency (and effectiveness!) of the feedback you get each time you think you've solved the problem. After you have solved the problem—and this is the most important part—*keep the test!*

That's right: rather than walking away from this debugging session with only another war story to tell your programmer buddies, walk away with tests that provide insurance against reinjecting the defect into the system at a later date. *You* might forget the cause of the problem or how you solved the problem, but the *tests* will never forget. Save your hard work for posterity and avoid having to go through this again at a customer site or two days before release or during your next product demo to the CEO.

Stop debugging. Write a test instead.

JAVA

JUnit Recipes Practical Methods for Programmer Testing

J. B. Rainsberger with contributions by **Scott Stirling**

FOREWORD BY
Robert C. Martin

When testing becomes a developer's habit good things tend to happen—good productivity, good code, and good job satisfaction. If you want some of that, there's no better way to start your testing habit, nor to continue feeding it, than with *JUnit Recipes*. In this book you will find one hundred and thirty-seven solutions to a range of problems, from simple to complex, selected for you by an experienced developer and master tester. Each recipe follows the same organization giving you the problem and its background before discussing your options in solving it.

JUnit—the unit testing framework for Java—is simple to use, but some code can be tricky to test. When you're facing such code you will be glad to have this book. It is a how-to reference on all issues of testing, from how to name your test case classes to how to test complicated J2EE applications. Its valuable advice includes side matters that can have a big payoff, like how to organize your test data or how to manage expensive test resources.

What's Inside

- How testing *saves* time
- Recipes for servlets, JSPs, EJBs, database code ...
- Difficult-to-test designs, how to fix them
- The right JUnit extension for the job: HTMLUnit, XMLUnit, ServletUnit, EasyMock, and more!

J. B. Rainsberger is a developer and consultant who has been a leader in the JUnit community since 2001. His popular online tutorial *JUnit: A Starter Guide* is read by thousands of new JUnit users each month. Joe lives in Toronto, Canada

"No other unit testing book [offers] as much wisdom, knowledge, and practical advice .... a remarkable compendium ..."

—Robert C. Martin
from the Foreword

"Study this book! It will zoom you along a paved road to expertise."

—Brian Marick, author,
The Craft of Software Testing

"... a compelling argument for how testing increases productivity and quality."

—Michael Rabbior, IBM

"JB's approach: been there, done that, don't do it please."

—Vladimir Ritz Bossicard
JUnit Development Team

"... a 'pattern reference'—it will have stuff to mine for years to come ..."

—Eric Armstrong
author of *JBuilder2 Bible*
consultant for Sun Computing


Ebook edition

www.manning.com/rainsberger

ISBN 1-932394-23-0

$49.95 US/$69.95 Canada