The Pragmatic Programmers

Pragmatic Unit Testing

In Java with JUnit

The Pragmatic Starter Kit – Volume II

Andrew Hunt *David Thomas*

What readers are saying about
***Pragmatic Unit Testing in Java*. . .**

"This book starts out with a nice introduction discussing what unit testing is as well as why we should do it. I like the anecdotes peppered throughout the book illustrating the point of why one should bother. . . . I also really liked the analogies you use. It puts the code into a real-world context."

► **Sharee L. Johnson,**
Project Lead, Applications Development

"I wish I had a copy back when I started doing test-first development as part of Extreme Programming."

► **Al Koscielny,** Software Developer

"I'm not totally new to testing, but I've struggled with many aspects of it. I think this book does a good job of bringing those along who are completely new to unit testing, but still has enough advanced material to assist those of us who have dabbled in testing and floundered once we've hit obstacles."

► **Andrew Thompson,**
Consultant, Greenbrier & Russel

"When I'm on a project that needs to be doing unit testing better (which is often the case), I'd like to have this book available as a simple reference to suggest to the team."

► **Bobby Woolf,** Consulting I/T Specialist,
IBM Software Services for Websphere

"I am a firm believer in unit testing and I would want all team members I work with to be religiously practicing the techniques recommended in this book. I think there is a lot of good, practical information in this book that any professional software engineer should be incorporating into their daily work."

► **James J. O'Connor III,**
Lead System Design Engineer

Pragmatic Unit Testing

in Java with JUnit

Andy Hunt

Dave Thomas

The Pragmatic Bookshelf

Raleigh, North Carolina Dallas, Texas

Our Pragmatic courses, workshops and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at:

`http://www.pragmaticprogrammer.com`

Printed in the United States of America.

ISBN 0-9745140-1-2

Printed on acid-free paper with 85% recycled, 30% post-consumer content.

Fourth printing, December 2004

Version: 2005-6-9

Contents

About the Starter Kit viii

Preface x

1 Introduction 1
- 1.1 Coding With Confidence . . . 2
- 1.2 What is Unit Testing? . . . 3
- 1.3 Why Should I Bother with Unit Testing? . . . 4
- 1.4 What Do I Want to Accomplish? . . . 5
- 1.5 How Do I Do Unit Testing? . . . 7
- 1.6 Excuses For Not Testing . . . 7
- 1.7 Roadmap . . . 12

2 Your First Unit Tests 13
- 2.1 Planning Tests . . . 14
- 2.2 Testing a Simple Method . . . 15
- 2.3 More Tests . . . 20

3 Writing Tests in JUnit 21
- 3.1 Structuring Unit Tests . . . 21
- 3.2 JUnit Asserts . . . 22
- 3.3 JUnit Framework . . . 26
- 3.4 JUnit Test Composition . . . 27
- 3.5 JUnit Custom Asserts . . . 32
- 3.6 JUnit and Exceptions . . . 33
- 3.7 More on Naming . . . 35
- 3.8 JUnit Test Skeleton . . . 35

4 What to Test: The Right-BICEP **37**
4.1 Are the Results Right? . . . 38
4.2 Boundary Conditions . . . 41
4.3 Check Inverse Relationships . . . 42
4.4 Cross-check Using Other Means . . . 42
4.5 Force Error Conditions . . . 43
4.6 Performance Characteristics . . . 44

5 CORRECT Boundary Conditions **46**
5.1 Conformance . . . 47
5.2 Ordering . . . 48
5.3 Range . . . 50
5.4 Reference . . . 53
5.5 Existence . . . 54
5.6 Cardinality . . . 55
5.7 Time . . . 57
5.8 Try It Yourself . . . 59

6 Using Mock Objects **63**
6.1 Simple Stubs . . . 64
6.2 Mock Objects . . . 65
6.3 Testing a Servlet . . . 69
6.4 Easy Mock Objects . . . 72

7 Properties of Good Tests **77**
7.1 Automatic . . . 78
7.2 Thorough . . . 79
7.3 Repeatable . . . 81
7.4 Independent . . . 81
7.5 Professional . . . 82
7.6 Testing the Tests . . . 84

8 Testing on a Project **87**
8.1 Where to Put Test Code . . . 87
8.2 Test Courtesy . . . 91
8.3 Test Frequency . . . 92
8.4 Tests and Legacy Code . . . 93
8.5 Tests and Reviews . . . 96

9 Design Issues **99**
9.1 Designing for Testability 99
9.2 Refactoring for Testing 101
9.3 Testing the Class Invariant 112
9.4 Test-Driven Design 115
9.5 Testing Invalid Parameters 117

A Gotchas **119**
A.1 As Long As The Code Works 119
A.2 "Smoke" Tests 119
A.3 "Works On My Machine" 120
A.4 Floating-Point Problems 120
A.5 Tests Take Too Long 121
A.6 Tests Keep Breaking 121
A.7 Tests Fail on Some Machines 122
A.8 My `main` is Not Being Run 123

B Installing JUnit **124**
B.1 Command-line installation 125
B.2 Does it work? 126

C JUnit Test Skeleton **127**
C.1 Helper Class 129
C.2 Basic Template 129

D Resources **132**
D.1 On The Web 132
D.2 Bibliography 134

E Summary: Pragmatic Unit Testing **135**

F Answers to Exercises **136**

About the Starter Kit

Our first book, *The Pragmatic Programmer: From Journeyman to Master*, is a widely-acclaimed overview of practical topics in modern software development. Since it was first published in 1999, many people have asked us about follow-on books, or sequels. We'll get around to that. But first, we thought we'd go back and offer a *prequel* of sorts.

Over the years, we've found that many of our pragmatic readers who are just starting out need a helping hand to get their development infrastructure in place, so they can begin forming good habits early. Many of our more advanced pragmatic readers understand these topics thoroughly, but need help convincing and educating the rest of their team or organization. We think we've got something that can help.

The *Pragmatic Starter Kit* is a three-volume set that covers the essential basics for modern software development. These volumes include the practices, tools, and philosophies that you need to get a team up and running and super-productive. Armed with this knowledge, you and your team can adopt good habits easily and enjoy the safety and comfort of a well-established "safety net" for your project.

Volume I, *Pragmatic Version Control*, describes how to use version control as the cornerstone of a project. A project without version control is like a word processor without an UNDO button: the more text you enter, the more expensive a mistake will be. Pragmatic Version Control shows you how to use version control systems effectively, with all the benefits and safety but without crippling bureaucracy or lengthy, tedious procedures.

This volume, *Pragmatic Unit Testing*, is the second volume in the series. Unit testing is an essential technique as it provides real-world, real-time feedback for developers as we write code. Many developers misunderstand unit testing, and don't realize that it makes *our* jobs as developers easier. This volume is available in two different language versions: in Java with JUnit, and in C# with NUnit.

Volume III, *Pragmatic Automation*,[1] covers the essential practices and technologies needed to automate your code's build, test, and release procedures. Few projects suffer from having too much time on their hands, so Pragmatic Automation will show you how to get the computer to do more of the mundane tasks by itself, freeing you to concentrate on the more interesting—and difficult—challenges.

These books are created in the same approachable style as our first book, and address specific needs and problems that you face in the trenches every day. But these aren't dummy-level books that only give you part of the picture; they'll give you enough understanding that you'll be able to invent your own solutions to the novel problems you face that we *haven't* addressed specifically.

For up-to-date information on these and other books, as well as related pragmatic resources for developers and managers, please visit us on the web at:

http://www.pragmaticprogrammer.com

Thanks, and remember to make it fun!

[1] Expected to be published in 2004.

Preface

Welcome to the world of developer-centric unit testing! We hope you find this book to be a valuable resource for yourself and your project team. You can tell us how it helped you—or let us know how we can improve—by visiting the *Pragmatic Unit Testing* page on our web site[2] and clicking on "Feedback."

Feedback like that is what makes books great. It's also what makes people and projects great. Pragmatic programming is all about using real-world feedback to fine tune and adjust your approach.

Which brings us to unit testing. As we'll see, unit testing is important to you as a programmer because it provides the feedback you need. Without unit testing, you may as well be writing programs on a yellow legal pad and hoping for the best when they're run.

That's not very pragmatic.

This book can help. It is aimed primarily at the Java programmer who has some experience writing and designing code, but who does not have much experience with unit testing.

But while the examples are in Java, using the JUnit framework, the concepts remain the same whether you are writing in C++, Fortran, Ruby, Smalltalk, or VisualBasic. Testing frameworks similar to JUnit exist for over 60 different languages; these various frameworks can be downloaded for free.[3]

[2] http://www.pragmaticprogrammer.com/sk/ut/
[3] http://www.xprogramming.com/software.htm

For the more advanced programmer, who has done unit testing before, we hope there will be a couple of nice surprises for you here. Skim over the basics of using JUnit and concentrate on how to think about tests, how testing affects design, and how to handle certain team-wide issues you may be having.

And remember that this book is just the beginning. It may be your first book on unit testing, but we hope it won't be your last.

Where To Find The Code

Throughout the book you'll find examples of Java code; some of these are complete programs while others are fragments of programs. If you want to run any of the example code or look at the complete source (instead of just the printed fragment), look in the margin: the filename of each code fragment in the book is printed in the margin next to the code fragment itself.

Some code fragments evolve with the discussion, so you may find the same source code file (with the same name) in the main directory as well as in subdirectories that contain later versions (`rev1`, `rev2`, and so on).

All of the code in this book is available via the *Pragmatic Unit Testing* page on our web site.

Typographic Conventions

italic font	Indicates terms that are being defined, or borrowed from another language.
`computer font`	Indicates method names, file and class names, and various other literal strings.
`xxx xx xx;`	Indicates unimportant portions of source code that are deliberately omitted.
	The "curves ahead" sign warns that this material is more advanced, and can safely be skipped on your first reading.

"Joe the Developer," our cartoon friend, asks a related question that you may find useful.

A break in the text where you should stop and think about what's been asked, or try an experiment live on a computer before continuing.

Language-specific Versions

As of this printing, *Pragmatic Unit Testing* is available in two programming language-specific versions:

- in Java with JUnit
- in C# with NUnit

Acknowledgments

We'd especially like to thank the following Practitioners for their valuable input, suggestions, and stories: Mitch Amiano, Nascif Abousalh-Neto, Andrew C. Oliver, Jared Richardson, and Bobby Woolf.

Thanks also to our reviewers who took the time and energy to point out our errors, omissions, and occasionally-twisted writing: Will Gwaltney, Sharee L. Johnson, Eric Kalendra, Al Koscielny, James J. O'Connor III, Mike Stok, Drew Thompson, and Eric Vought.

Thanks to all of you for your hard work and support.

Andy Hunt and *Dave Thomas*
October, 2003
`pragprog@pragmaticprogrammer.com`

Chapter 1

Introduction

There are lots of different kinds of testing that can and should be performed on a software project. Some of this testing requires extensive involvement from the end users; other forms may require teams of dedicated Quality Assurance personnel or other expensive resources.

But that's not what we're going to talk about here.

Instead, we're talking about *unit testing*: an essential, if often misunderstood, part of project and personal success. Unit testing is a relatively inexpensive, easy way to produce better code, faster.

Many organizations have grand intentions when it comes to testing, but tend to test only toward the end of a project, when the mounting schedule pressures cause testing to be curtailed or eliminated entirely.

Many programmers feel that testing is just a nuisance: an unwanted bother that merely distracts from the real business at hand—cutting code.

Everyone agrees that more testing is needed, in the same way that everyone agrees you should eat your broccoli, stop smoking, get plenty of rest, and exercise regularly. That doesn't mean that any of us actually do these things, however.

But unit testing can be much more than these—while you might consider it to be in the broccoli family, we're here to tell

you that it's more like an awesome sauce that makes everything taste better. Unit testing isn't designed to achieve some corporate quality initiative; it's not a tool for the end-users, or managers, or team leads. Unit testing is done by programmers, for programmers. It's here for our benefit alone, to make our lives easier.

Put simply, unit testing alone can mean the difference between your success and your failure. Consider the following short story.

1.1 Coding With Confidence

Once upon a time—maybe it was last Tuesday—there were two developers, Pat and Dale. They were both up against the same deadline, which was rapidly approaching. Pat was pumping out code pretty fast; developing class after class and method after method, stopping every so often to make sure that the code would compile.

Pat kept up this pace right until the night before the deadline, when it would be time to demonstrate all this code. Pat ran the top-level program, but didn't get any output at all. Nothing. Time to step through using the debugger. Hmm. That can't be right, thought Pat. There's no *way* that this variable could be zero by now. So Pat stepped back through the code, trying to track down the history of this elusive problem.

It was getting late now. That bug was found and fixed, but Pat found several more during the process. And still, there was no output at all. Pat couldn't understand why. It just didn't make any sense.

Dale, meanwhile, wasn't churning out code nearly as fast. Dale would write a new routine and a short test to go along with it. Nothing fancy, just a simple test to see if the routine just written actually did what it was supposed to do. It took a little longer to think of the test, and write it, but Dale refused to move on until the new routine could prove itself. Only then would Dale move up and write the next routine that called it, and so on.

Dale rarely used the debugger, if ever, and was somewhat puzzled at the picture of Pat, head in hands, muttering various evil-sounding curses at the computer with wide, bloodshot eyes staring at all those debugger windows.

The deadline came and went, and Pat didn't make it. Dale's code was integrated and ran almost perfectly. One little glitch came up, but it was pretty easy to see where the problem was. Dale fixed it in just a few minutes.

Now comes the punch line: Dale and Pat are the same age, and have roughly the same coding skills and mental prowess. The only difference is that Dale believes very strongly in unit testing, and tests every newly-crafted method before relying on it or using it from other code.

Pat does not. Pat "knows" that the code should work as written, and doesn't bother to try it until most of the code has been written. But by then it's too late, and it becomes very hard to try to locate the source of bugs, or even determine what's working and what's not.

1.2 What is Unit Testing?

A *unit test* is a piece of code written by a developer that exercises a very small, specific area of functionality of the code being tested. Usually a unit test exercises some particular method in a particular context. For example, you might add a large value to a sorted list, then confirm that this value appears at the end of the list. Or you might delete a pattern of characters from a string and then confirm that they are gone.

Unit tests are performed to prove that a piece of code does what the developer thinks it should do.

The question remains open as to whether that's the right thing to do according to the customer or end-user: that's what acceptance testing is for. We're not really concerned with formal validation and verification or correctness just yet. We're really not even interested in performance testing at this point. All we want to do is prove that code does what we intended, and so we want to test very small, very isolated pieces of functionality. By building up confidence that the individual pieces

work as expected, we can then proceed to assemble and test working systems.

After all, if we aren't sure the code is doing what we think, then any other forms of testing may just be a waste of time. You still need other forms of testing, and perhaps much more formal testing depending on your environment. But testing, as with charity, begins at home.

1.3 Why Should I Bother with Unit Testing?

Unit testing will make your life easier. It will make your designs better and drastically reduce the amount of time you spend debugging.

In our tale above, Pat got into trouble by assuming that lower-level code worked, and then went on to use that in higher-level code, which was in turn used by more code, and so on. Without legitimate confidence in any of the code, Pat was building a "house of cards" of assumptions—one little nudge at the bottom and the whole thing falls down.

When basic, low-level code isn't reliable, the requisite fixes don't stay at the low level. You fix the low level problem, but that impacts code at higher levels, which then need fixing, and so on. Fixes begin to ripple throughout the code, getting larger and more complicated as they go. The house of cards falls down, taking the project with it.

Pat keeps saying things like "that's impossible" or "I don't understand how that could happen." If you find yourself thinking these sorts of thoughts, then that's usually a good indication that you don't have enough confidence in your code—you don't know for sure what's working and what's not.

In order to gain the kind of code confidence that Dale has, you'll need to ask the code itself what it is doing, and check that the result is what you expect it to be.

That simple idea describes the heart of unit testing: the single most effective technique to better coding.

1.4 What Do I Want to Accomplish?

It's easy to get carried away with unit testing because it's so much fun, but at the end of the day we still need to produce production code for customers and end-users, so let's be clear about our goals for unit testing. First and foremost, you want to do this to make your life—and the lives of your teammates—easier.

Does It Do What I Want?

Fundamentally, you want to answer the question: "Is the code fulfilling my intent?" The code might well be doing the wrong thing as far as the requirements are concerned, but that's a separate exercise. You want the code to prove to you that it's doing exactly what **you** think it should.

Does It Do What I Want All of the Time?

Many developers who claim they do testing only ever write one test. That's the test that goes right down the middle, taking the "one right path" through the code where everything goes perfectly.

But of course, life is rarely that cooperative, and things don't always go perfectly: exceptions get thrown, disks get full, network lines drop, buffers overflow, and—heaven forbid—we write bugs. That's the "engineering" part of software development. Civil engineers must consider the load on bridges, the effects of high winds, of earthquakes, floods, and so on. Electrical engineers plan on frequency drift, voltage spikes, noise, even problems with parts availability.

You don't test a bridge by driving a single car over it right down the middle lane on a clear, calm day. That's not sufficient. Similarly, beyond ensuring that the code does what you want, you need to ensure that the code does what you want *all of the time*, even when the winds are high, the parameters are suspect, the disk is full, and the network is sluggish.

Can I Depend On It?

Code that you can't depend on is useless. Worse, code that you *think* you can depend on (but turns out to have bugs) can cost you a lot of time to track down and debug. There are very few projects that can afford to waste time, so you want to avoid that "one step forward two steps back" approach at all costs, and stick to moving forward.

No one writes perfect code, and that's okay—as long as you know where the problems exist. Many of the most spectacular software failures that strand broken spacecraft on distant planets or blow them up in mid-flight could have been avoided simply by knowing the limitations of the software. For instance, the Arianne 5 rocket software re-used a library from an older rocket that simply couldn't handle the larger numbers of the higher-flying new rocket.[1] It exploded 40 seconds into flight, taking $500 million dollars with it into oblivion.

We want to be able to depend on the code we write, and know for certain both its strengths and its limitations.

For example, suppose you've written a routine to reverse a list of numbers. As part of testing, you give it an empty list—and the code blows up. The requirements don't say you have to accept an empty list, so maybe you simply document that fact in the comment block for the method and throw an exception if the routine is called with an empty list. Now you know the limitations of code right away, instead of finding out the hard way (often somewhere inconvenient, such as in the upper atmosphere).

Does it Document my Intent?

One nice side-effect of unit testing is that it helps you communicate the code's intended use. In effect, a unit test behaves as executable documentation, showing how you expect the code to behave under the various conditions you've considered.

[1] For aviation geeks: The numeric overflow was due to a much larger "horizontal bias" due to a different trajectory that increased the horizontal velocity of the rocket.

Team members can look at the tests for examples of how to use your code. If someone comes across a test case that you haven't considered, they'll be alerted quickly to that fact.

And of course, executable documentation has the benefit of being correct. Unlike written documentation, it won't drift away from the code (unless, of course, you stop running the tests).

1.5 How Do I Do Unit Testing?

Unit testing is basically an easy practice to adopt, but there are some guidelines and common steps that you can follow to make it easier and more effective.

The first step is to decide how to test the method in question—before writing the code itself. With at least a rough idea of how to proceed, you proceed to write the test code itself, either before or concurrently with the implementation code.

Next, you run the test itself, and probably all the other tests in that part of the system, or even the entire system's tests if that can be done relatively quickly. It's important that **all the tests pass**, not just the new one. You want to avoid any collateral damage as well as any immediate bugs.

Every test needs to determine whether it passed or not—it doesn't count if you or some other hapless human has to read through a pile of output and decide whether the code worked or not. You want to get into the habit of looking at the test results and telling at a glance whether it all worked. We'll talk more about that when we go over the specifics of using unit testing frameworks.

1.6 Excuses For Not Testing

Despite our rational and impassioned pleas, some developers will still nod their heads and agree with the need for unit testing, but will steadfastly assure us that *they* couldn't possibly do this sort of testing for one of a variety of reasons. Here are some of the most popular excuses we've heard, along with our rebuttals.

Joe Asks...

What's collateral damage?

Collateral damage is what happens when a new feature or a bug fix in one part of the system causes a bug (damage) to another, possibly unrelated part of the system. It's an insidious problem that, if allowed to continue, can quickly render the entire system broken beyond anyone's ability to fix.

We sometime call this the "Whac-a-Mole" effect. In the carnival game of Whac-a-Mole, the player must strike the mechanical mole heads that pop up on the playing field. But they don't keep their heads up for long; as soon as you move to strike one mole, it retreats and another mole pops up on the opposite side of the field. The moles pop up and down fast enough that it can be very frustrating to try to connect with one and score. As a result, players generally flail helplessly at the field as the moles continue to pop up where you least expect them.

Widespread collateral damage to a code base can have a similar effect.

It takes too much time to write the tests This is the number one complaint voiced by most newcomers to unit testing. It's untrue, of course, but to see why we need to take a closer look at where you spend your time when developing code.

Many people view testing of any sort as something that happens toward the end of a project. And yes, if you wait to begin unit testing until then it will definitely take too long. In fact, you may not finish the job until the heat death of the universe itself.

At least it will feel that way: it's like trying to clear a couple of acres of land with a lawn mower. If you start early on when there's just a field of grasses, the job is easy. If you wait until later, when the field contains thick, gnarled trees and dense, tangled undergrowth, then the job becomes impossibly difficult.

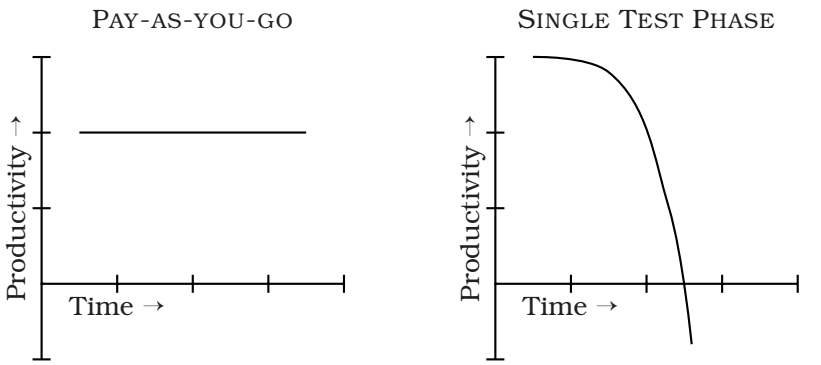

Figure 1.1: Comparison of Paying-as-you-go vs. Having a Single Testing Phase

Instead of waiting until the end, it's far cheaper in the long run to adopt the "pay-as-you-go" model. By writing individual tests with the code itself as you go along, there's no crunch at the end, and you experience fewer overall bugs as you are generally always working with tested code. By taking a little extra time all the time, you minimize the risk of needing a huge amount of time at the end.

You see, the trade-off is not "test now" versus "test later." It's linear work now versus exponential work and complexity trying to fix and rework at the end: not only is the job larger and more complex, but now you have to re-learn the code you wrote some weeks or months ago. All that extra work kills your productivity, as shown in Figure 1.1.

Notice that testing isn't free. In the pay-as-you-go model, the effort is not zero; it will cost you some amount of effort (and time and money). But look at the frightening direction the right-hand curve takes over time—straight down. Your productivity might even become negative. These productivity losses can easily doom a project.

So if you think you don't have time to write tests in addition to the code you're already writing, consider the following questions:

1. How much time do you spend debugging code that you or others have written?
2. How much time do you spend reworking code that you thought was working, but turned out to have major, crippling bugs?
3. How much time do you spend isolating a reported bug to its source?

For most people who work without unit tests, these numbers add up fast, and will continue to add up even faster over the life of the project. Proper unit testing dramatically reduces these times, which frees up enough time so that you'll have the opportunity to write all of the unit tests you want—and maybe even some free time to spare.

It takes too long to run the tests It shouldn't. Most unit tests should execute extremely quickly, so you should be able to run hundreds, even thousands of them in a matter of a few seconds. But sometimes that won't be possible, and you may end up with certain tests that simply take too long to conveniently run all of the time.

In that case, you'll want to separate out the longer-running tests from the short ones. Only run the long tests once a day, or once every few days as appropriate, and run the shorter tests constantly.

It's not my job to test my code Now here's an interesting excuse. Pray tell, what *is* your job, exactly? Presumably your job, at least in part, is to create working code. If you are throwing code over the wall to some testing group without any assurance that it's working, then you're not doing your job. It's not polite to expect others to clean up our own messes, and in extreme cases submitting large volumes of buggy code can become a "career limiting" move.

On the other hand, if the testers or QA group find it very difficult to find fault with your code, your reputation will grow rapidly—along with your job security!

I don't really know how the code is supposed to behave so I can't test it If you truly don't know how the code is supposed to behave, then maybe this isn't the time to be writing it. Maybe a prototype would be more appropriate as a first step to help clarify the requirements.

If you don't know what the code is supposed to do, then how will you know that it does it?

But it compiles! Okay, no one *really* comes out with this as an excuse, at least not out loud. But it's easy to get lulled into thinking that a successful compile is somehow a mark of approval, that you've passed some threshold of goodness.

But the compiler's blessing is a pretty shallow compliment. It can verify that your syntax is correct, but it can't figure out what your code should do. For example, the Java compiler can easily determine that this line is wrong:

```
public statuc void main(String args[]) {
```

It's just a simple typo, and should be `static`, not `statuc`. That's the easy part. But now suppose you've written the following:

```
public void addit(Object anObject){
  ArrayList myList = new ArrayList();
  myList.add(anObject);
  myList.add(anObject);
  // more code...
}
```

Main.java

Did you really mean to add the same object to the same list twice? Maybe, maybe not. The compiler can't tell the difference, only you know what you've intended the code to do.[2]

I'm being paid to write code, not to write tests By that same logic, you're not being paid to spend all day in the debugger, either. Presumably you are being paid to write *working* code, and unit tests are merely a tool toward that end, in the same fashion as an editor, an IDE, or the compiler.

[2]Automated testing tools that generate their own tests based on your existing code fall into this same trap—they can only use what you wrote, not what you meant.

I feel guilty about putting testers and QA staff out of work Not to worry, you won't. Remember we're only talking about *unit testing*, here. It's the barest-bones, lowest-level testing that's designed for us, the programmers. There's plenty of other work to be done in the way of functional testing, acceptance testing, performance and environmental testing, validation and verification, formal analysis, and so on.

My company won't let me run unit tests on the live system Whoa! We're talking about developer unit-testing here. While you might be able to run those same tests in other contexts (on the live, production system, for instance) *they are no longer unit tests*. Run your unit tests on your machine, using your own database, or using a mock object (see Chapter 6).

If the QA department or other testing staff want to run these tests in a production or staging environment, you might be able to coordinate the technical details with them so they can, but realize that they are no longer unit tests in that context.

1.7 Roadmap

Chapter 2, *Your First Unit Tests,* contains an overview of test writing. From there we'll take a look at the specifics of *Writing Tests in JUnit* in Chapter 3. We'll then spend a few chapters on how you come up with *what* things need testing, and how to test them.

Next we'll look at the important properties of good tests in Chapter 7. We then talk about what you need to do to use testing effectively in your project in Chapter 8. This chapter also discusses how to handle existing projects with lots of legacy code. Chapter 9, *Design Issues*. then looks at how testing can influence your application's design (for the better).

The appendices contain additional useful information: a look at common unit testing problems, a note on installing JUnit, a sample JUnit skeleton program, and a list of resources including the bibliography. We finish off with a summary card containing highlights of the book's tips and suggestions.

So sit back, relax, and welcome to the world of better coding.

Chapter 2

Your First Unit Tests

As we said in the introduction, a unit test is just a piece of code. It's a piece of code you write that happens to exercise another piece of code, and determines whether the other piece of code is behaving as expected or not.

How do you do that, exactly?

To check if code is behaving as you expect, you use an *assertion*, a simple method call that verifies that something is true. For instance, the method `assertTrue` checks that the given boolean condition is true, and fails the current test if it is not. It might be implemented like the following.

AssertTrue.java

```
public void assertTrue(boolean condition) {
  if (!condition) {
    abort();
  }
}
```

You could use this assert to check all sorts of things, including whether numbers are equal to each other:

```
int a = 2;
assertTrue(a == 2);
```

If for some reason a does not equal 2 when the method `assertTrue` is called, then the program will abort.

Since we check for equality a lot, it might be easier to have an assert just for numbers. To check that two integers are equal,

for instance, we could write a method that takes two integer parameters:

AssertTrue.java

```
public void assertEquals(int a, int b) {
  assertTrue(a == b);
}
```

Armed with just these two asserts, we can start writing some tests. We'll look at more asserts and describe the details of how you use asserts in unit test code in the next chapter. But first, let's consider what tests might be needed before we write any code at all.

2.1 Planning Tests

We'll start with a simple example, a single, static method designed to find the largest number in a list of numbers:

```
static int largest(int[] list);
```

In other words, given an array of numbers such as `[7, 8, 9]`, this method should return 9. That's a reasonable first test. What other tests can you think of, off the top of your head? Take a minute and write down as many tests as you can think of for this simple method before you continue reading.

Think about this for a moment before reading on...

How many tests did you come up with?

It shouldn't matter what order the given list is in, so right off the bat you've got the following test ideas (which we've written as "what you pass in" → "what you expect").

- `[7, 8, 9]` → `9`
- `[8, 9, 7]` → `9`
- `[9, 7, 8]` → `9`

What happens if there are duplicate largest numbers?

- `[7, 9, 8, 9]` → `9`

Since these are `int` types, not objects, you probably don't care which 9 is returned, as long as one of them is.

What if there's only one number?

- `[1]` → `1`

And what happens with negative numbers:

- `[-9, -8, -7]` → `-7`

It might look odd, but indeed -7 is larger than -9. Glad we straightened that out now, rather than in the debugger or in production code where it might not be so obvious.

To make all this more concrete, lets actually write a "largest" method and test it. Here's the code for our first implementation:

```
Line 1  public class Largest {
-
-     /**
-      * Return the largest element in a list.
5     *
-      * @param list A list of integers
-      * @return The largest number in the given list
-      */
-     public static int largest(int[] list) {
10      int index, max=Integer.MAX_VALUE;
-       for (index = 0; index < list.length-1; index++) {
-         if (list[index] > max) {
-           max = list[index];
-         }
15      }
-       return max;
-     }
-
-   }
```

Largest.java

Now that we've got some ideas for tests, we'll look at writing these tests in Java, using the JUnit framework.

2.2 Testing a Simple Method

Normally you want to make the first test you write incredibly simple, because there is much to be tested the first time besides the code itself: all of that messy business of class names, library locations, and making sure it compiles. You want to get all of that taken care of and out of the way with the very first, simplest test; you won't have to worry about it

anymore after that, and you won't have to debug complex integration issues at the same time you're debugging a complex test!

First, let's just test the simple case of passing in a small array with a couple of unique numbers. Here's the complete source code for the test class. We'll explain all about test classes in the next chapter; for now, just concentrate on the assert statements:

```
import junit.framework.*;
public class TestLargest extends TestCase {
  public TestLargest(String name) {
    super(name);
  }
  public void testOrder() {
    assertEquals(9, Largest.largest(new int[] {8,9,7}));
  }
}
```

TestLargest.java

Java note: the odd-looking syntax to create an anonymous array is just for your authors' benefit, as we are lazy and do not like to type. If you prefer, the test could be written this way instead (although the previous syntax is idiomatic):

```
  public void testOrder2() {
    int[] arr = new int[3];
    arr[0] = 8;
    arr[1] = 9;
    arr[2] = 7;
    assertEquals(9, Largest.largest(arr));
  }
```

TestLargest.java

That's all it takes, and you have your first test. Run it and make sure it passes (see the sidebar on the next page).

Try running this example before reading on. . .

Having just run that code, you probably saw an error similar to the following:

```
There was 1 failure:
1) testOrder(TestLargest)junit.framework.AssertionFailedError:
   expected:<9> but was:<2147483647>
   at TestLargest.testOrder(TestLargest.java:7)
```

Whoops! That didn't go as expected. Why did it return such a huge number instead of our 9? Where could that very large

How To Run A JUnit Test

If JUnit is integrated into your IDE, then running a test may be as easy as pressing a button and selecting from a list of available test classes.

Otherwise, you can always execute a `TestRunner` manually. There are several flavors of test runners. To run a GUI version that lets you pick and choose classes (and which remembers them from session to session), run the following class:

```
java junit.swingui.TestRunner
```

You'll probably be able to run the `junit.swingui.TestRunner` class from your IDE. If not, run it from the command line using the `jre` or `java` command (as shown).

To run a test using the textual UI, as we've shown in this book, use:

```
java junit.textui.TestRunner classname ...
```

For instance, to run the unit tests on the preceding page we'd compile the `Largest` and the `TestLargest` programs, then run the TestRunner on `TestLargest`:

```
javac Largest.java TestLargest.java
java junit.textui.TestRunner TestLargest
```

number have come from? It almost looks like the largest number... oh, it's a small typo: `max=Integer.MAX_VALUE` on line 10 should have been `max=0`. We want to initialize `max` so that any other number instantly becomes the next `max`. Let's fix the code, recompile, and run the test again to make sure that it works.

Next we'll look at what happens when the largest number appears in different places in the list—first or last, and somewhere in the middle. Bugs most often show up at the "edges." In this case, edges occur when the largest number is at the start or end of the array that we pass in. We can lump all three of these asserts together in one test, but let's add the assert statements one at a time. We already have the case

with the largest in the middle:

```
import junit.framework.*;
public class TestLargest extends TestCase {
  public TestLargest(String name) {
    super(name);
  }
  public void testOrder() {
    assertEquals(9, Largest.largest(new int[] {8,9,7}));
  }
}
```

TestLargest.java

Now try it with the 9 as the first value (we'll just add an additional assertion to the existing `testOrder()` method):

```
  public void testOrder() {
    assertEquals(9, Largest.largest(new int[] {9,8,7}));
    assertEquals(9, Largest.largest(new int[] {8,9,7}));
  }
```

TestLargest.java

We're on a roll. One more, just for the sake of completeness, and we can move on to more interesting tests:

```
  public void testOrder() {
    assertEquals(9, Largest.largest(new int[] {9,8,7}));
    assertEquals(9, Largest.largest(new int[] {8,9,7}));
    assertEquals(9, Largest.largest(new int[] {7,8,9}));
  }
```

TestLargest.java

Try running this example before reading on...

Uh oh! This doesn't look good....

```
There was 1 failure:
1) testOrder(TestLargest)junit.framework.AssertionFailedError:
   expected:<9> but was:<8>
   at TestLargest.testOrder(TestLargest.java:9)
```

Why did the test get an `8` as the largest number? It's almost as if the code ignored the last entry in the list. Sure enough, another simple typo: the `for` loop is terminating too early. This is an example of the infamous "off-by-one" error.

Our code has:

```
    for (index = 0; index < list.length-1; index++) {
```

But it should be one of:

```
    for (index = 0; index <= list.length-1; index++) {
    for (index = 0; index < list.length; index++) {
```

The second expression is idiomatic in languages descended from C (including Java and C#), and it's easier to read, so let's go with that one. Make the change and run the tests again.

Let's check for duplicate largest values; type this in and run it (we'll only show the newly added methods from here on):

TestLargest.java

```
public void testDups() {
  assertEquals(9, Largest.largest(new int[] {9,7,9,8}));
}
```

So far, so good. Now the test for just a single integer:

TestLargest.java

```
public void testOne() {
   assertEquals(1, Largest.largest(new int[] {1}));
}
```

Hey, it worked! You're on a roll now, surely all the bugs we planted in this example have been exorcised by now. Just one more check with negative values:

TestLargest.java

```
public void testNegative() {
  int [] negList = new int[] {-9, -8, -7};
  assertEquals(-7, Largest.largest(negList));
}
```

Try running this example before reading on...


```
There was 1 failure:
1) testNegative(TestLargest)junit.framework.AssertionFailedError:
   expected:<-7> but was:<0>
   at TestLargest.testNegative(TestLargest.java:3)
```

Whoops! Where did zero come from?

Looks like choosing 0 to initialize `max` was a bad idea; what we really wanted was `MIN_VALUE`, so as to be less than all negative numbers as well:

```
max = Integer.MIN_VALUE
```

Make that change and try it again—all of the existing tests should continue to pass, and now this one will as well.

Unfortunately, the initial specification for the method "largest" is incomplete, as it doesn't say what should happen if the array is empty. Let's say that it's an error, and add some code at the top of the method that will throw a runtime-exception if the list length is zero:

```
public static int largest(int[] list) {
  int index, max=Integer.MIN_VALUE;
  if (list.length == 0) {
    throw new RuntimeException("largest: Empty list");
  }
  // ...
```

Largest.java

Notice that just by thinking of the tests, we've already realized we need a design change. That's not at all unusual, and in fact is something we want to capitalize on. So for the last test, we need to check that an exception is thrown when passing in an empty array. We'll talk about testing exceptions in depth on page 33, but for now just trust us:

```
public void testEmpty() {
  try {
    Largest.largest(new int[] {});
    fail("Should have thrown an exception");
  } catch (RuntimeException e) {
    assertTrue(true);
  }
}
```

TestLargest.java

Finally, a reminder: all code—test or production—should be clear and simple. Test code *especially* must be easy to understand, even at the expense of performance or verbosity.

2.3 More Tests

We started with a very simple method and came up with a couple of interesting tests that actually found some bugs. Note that we didn't go overboard and blindly try every possible number combination; we picked the interesting cases that might expose problems. But are these all the tests you can think of for this method?

What other tests might be appropriate?

Since we'll need to think up tests all of the time, maybe we need a way to think about code that will help us to come up with good tests regularly and reliably. We'll talk about that after the next chapter, but first, let's take a more in-depth look at using JUnit.

Chapter 3

Writing Tests in JUnit

We've looked at writing tests somewhat informally in the last chapter, but now it's time to take a deeper look at the difference between test code and production code, all the various forms of JUnit's `assert`, the structure and composition of JUnit tests, and so on.

3.1 Structuring Unit Tests

When writing test code, there are some naming conventions you need to follow. If you have a method named `createAccount` that you want to test, then your first test method might be named `testCreateAccount`. The method `testCreateAccount` will call `createAccount` with the necessary parameters and verify that `createAccount` works as advertised. You can, of course, have many test methods that exercise `createAccount`.

The relationship between these two pieces of code is shown in Figure 3.1 on the following page.

The test code is for our internal use only. Customers or end-users will never see it or use it. The production code—that is, the code that will eventually be shipped to a customer and put into production—must therefore not know anything about the test code. Production code will be thrust out into the cold world all alone, without the test code.

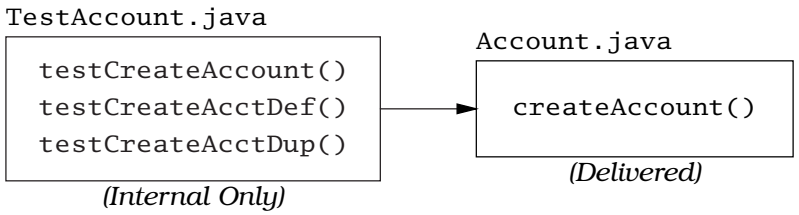

Figure 3.1: Test Code and Production Code

The test code must be written to do a few things:

- Set up all conditions needed for testing (create any required objects, allocate any needed resources, etc.)
- Call the method to be tested
- Verify that the method to be tested functioned as expected
- Clean up after itself

You write test code and compile it in the normal fashion, as you would any other bit of source code in your project. It might happen to use some additional libraries, but otherwise there's no magic—it's just regular code.

When it's time to execute the code, remember that you never actually run the production code directly; at least, not the way a user would. Instead, you run the test code, which in turn exercises the production code under very carefully controlled conditions.

We'll be showing the conventions for JUnit, using Java, in our examples, but the general concepts are the same for any testing framework in any language or environment. If you don't have JUnit installed on your computer yet, see Appendix B on page 124 and then come on back, and we'll take a look at the JUnit-specific method calls and classes.

3.2 JUnit Asserts

As we've seen, there are some helper methods that assist you in determining whether a method under test is performing

correctly or not. Generically, we call all these methods *assertions*. They let you assert that some condition is true; that two bits of data are equal, or not, and so on. We'll take a look at each one of the assert methods that JUnit provides next.

All of the following methods will record failures (that's when the assertion is false) or errors (that's when you get an unexpected exception), and report these through the JUnit classes. For the text version, that means an error message will be printed to the console. The GUI version will show a red bar and supporting details to indicate a failure.

When a failure or error occurs, execution of the current test method is aborted. Other tests within the same test class will still be run.

Asserts are the fundamental building block for unit tests; the JUnit library provides a number of different forms of assert.

assertEquals

```
assertEquals([String message],
             expected,
             actual)
```

This is the most-often used form of assert. *expected* is a value you hope to see (typically hard-coded), and *actual* is a value actually produced by the code under test. *message* is an optional message that will be reported in the event of a failure. You can omit the message argument and simply provide the *expected* and *actual* values.

Any kind of object may be tested for equality; the appropriate equals method will be used for the comparison. In particular, you can compare the contents of strings using this method. Different method signatures are also provided for all the native types (`boolean`, `int`, `short`, etc.) and `Object`. Be aware that the equals method for native arrays, however, does not compare the contents of the arrays, just the array reference itself, which is probably not what you want.

Computers cannot represent all floating-point numbers exactly, and will usually be off a little bit. Because of this, if you are using an assert to compare floating point numbers (floats or doubles in Java), you need to specify one additional piece

of information, the tolerance. This specifies just how close to "equals" you need the result to be. For most business applications, 4 or 5 decimal places is probably enough. For scientific apps, you may need much greater precision.

```
assertEquals([String message],
             expected,
             actual,
             tolerance)
```

For instance, the following assert will check that the actual result is equal to 3.33, but only look at the first two decimal places:

```
assertEquals("Should be 3 1/3", 3.33, 10.0/3.0, 0.01);
```

assertNull

```
assertNull([String message], java.lang.Object object)
assertNotNull([String message], java.lang.Object object)
```

Asserts that the given object is `null` (or not `null`), failing otherwise. The message is optional.

assertSame

```
assertSame([String message], expected, actual)
```

Asserts that *expected* and *actual* refer to the same object, and fails the test if they do not. The message is optional.

```
assertNotSame([String message], expected, actual)
```

Asserts that *expected* and *actual* do **not** refer to the same object, and fails the test if they are the same object. The message is optional.

assertTrue

```
assertTrue([String message], boolean condition)
```

Asserts that the given boolean condition is true, otherwise the test fails. The message is optional.

If you find test code that is littered with the following:

```
assertTrue(true);
```

then you should rightfully be concerned. Unless that construct is used to verify some sort of branching or exception

logic, it's probably a bad idea. In particular, what you really don't want to see is a whole page of "test" code with a single `assertTrue(true)` at the very end (i.e., "the code made it to the very end without blowing up therefore it must work"). That's not testing, that's wishful thinking.

In addition to testing for `true`, you can also test for `false`:

```
assertFalse([String message], boolean condition)
```

Asserts that the given boolean condition is false, otherwise the test fails. The message is optional.

fail

```
fail([String message])
```

Fails the test immediately, with the optional message. Often used to mark sections of code that should not be reached (for instance, after an exception is expected).

Using asserts

You usually have multiple asserts in a given test method, as you prove various aspects and relationships of the method(s) under test. When an assert fails, that test method will be aborted—the remaining assertions in that method will not be executed this time. But that shouldn't be of any concern; you have to fix the failing test before you can proceed anyway. And you fix the next failing test. And the next. And so on.

You should normally expect that all tests pass all of the time. In practice, that means that when you introduce a bug, only one or two tests fail. Isolating the problem is usually pretty easy in that environment.

Under no circumstances should you continue to add features when there are failing tests! Fix any test as soon as it fails, and keep all tests passing all of the time.

To maintain that discipline, you'll need an easy way to run all the tests—or to run groups of tests, particular subsystems, and so on.

3.3 JUnit Framework

So far, we've just looked at the assert methods themselves. But you can't just stick assert methods into a source file and expect it to work; you need a little bit more of a framework than that. Fortunately, it's not too much more.

Here is a very simple piece of test code that illustrates the minimum framework you need to get started.

```
Line 1  import junit.framework.*;
        -
        -  public class TestSimple extends TestCase {
        -
        5    public TestSimple(String name) {
        -      super(name);
        -    }
        -
        -    public void testAdd() {
       10      assertEquals(2, 1+1);
        -    }
        -
        -  }
```

TestSimple.java

This code is pretty straightforward, but let's take a look at each part in turn.

First, the import statement on line 1 brings in the necessary JUnit classes.

Next, we have the class definition itself on line 3: each class that contains tests must extend `TestCase` as shown. The base class `TestCase` provides most of the unit-testing functionality that we'll need, including all of the assert methods we described above.

The base class requires a constructor that takes a `String`, so we have to provide a call to super that passes in a name. We don't know what that name should be just at the moment, so we'll just make our own constructor take a `String` and pass it along on line 5.

Finally, the test class contains individual methods named `test...`. In the example, we've got one test method named `testAdd` on line 9. All methods with names that begin with `test` will be run automatically by JUnit. You can also specify particular methods to run by defining a `suite` method; more on that a bit later.

In the previous example, we showed a single test, using a single assert, in a single test method. Of course, inside a test method, you can place any number of asserts:

```
public void testAdds() {
  assertEquals(2, 1+1);
  assertEquals(4, 2+2);
  assertEquals(-8, -12+4);
}
```

TestSimple.java

Here we have three `assertEquals` inside a test method.

3.4 JUnit Test Composition

As we've seen so far, a test class contains test methods; each method contains one or more `assert` statements. But a test class can also invoke other test classes: individual classes, packages, or even the whole system.

This magic is achieved by creating *test suites*. Any test class can contain a static method named `suite`:

```
public static Test suite();
```

You can provide a `suite()` method to return any collection of tests you want (without a `suite()` method, JUnit runs all of the `test...` methods automatically). But you might want to add particular tests by hand, including the results from other suites.

For instance, suppose you had a normal set of tests as we've already seen in a class named `TestClassOne`:

```
import junit.framework.*;
public class TestClassOne extends TestCase {
  public TestClassOne(String method) {
    super(method);
  }
  public void testAddition() {
    assertEquals(4, 2+2);
  }
  public void testSubtraction() {
    assertEquals(0, 2-2);
  }
}
```

TestClassOne.java

The default action, using Java reflection on this class, would be to run the test methods `testAddition()` and `testSubtraction()`.

Now suppose you've got a second class, `TestClassTwo`. This uses a brute-force algorithm to find the shortest route that our traveling salesman, Bob, can take to visit the top n cities in his territory. The funny thing about the Traveling Salesman algorithm is that for a small number of cities it works just fine, but it's an exponential algorithm—a few hundred cities might take 20,000 years to run, for example. Even 50 cities takes a few hours, so you probably don't want to to include that test by default.

```
import junit.framework.*;
public class TestClassTwo extends TestCase {
  public TestClassTwo(String method) {
    super(method);
  }
  // This one takes a few hours...
  public void testLongRunner() {
    TSP tsp = new TSP(); // Load with default cities
    assertEquals(2300, tsp.shortestPath(50)); // top 50
  }
  public void testShortTest() {
    TSP tsp = new TSP(); // Load with default cities
    assertEquals(140, tsp.shortestPath(5)); // top 5
  }
  public void testAnotherShortTest() {
    TSP tsp = new TSP(); // Load with default cities
    assertEquals(586, tsp.shortestPath(10)); // top 10
  }
  public static Test suite() {
    TestSuite suite = new TestSuite();
    // Only include short tests
    suite.addTest(
          new TestClassTwo("testShortTest"));
    suite.addTest(
          new TestClassTwo("testAnotherShortTest"));
    return suite;
  }
}
```

TestClassTwo.java

The test is still there, but to run it you would have to ask for it explicitly (we'll show one way to do that with a special test skeleton in Appendix C on page 127). Without this special mechanism, only the short-running tests will be run when we invoke the test suite.

Also, now we can see what that `String` parameter to the constructor is for: it lets a `TestCase` return a reference to a named test method. Here we're using it to get references to the two short-running tests to populate our test suite.

You might want to have a higher-level test that is composed of both of these test classes:

```
import junit.framework.*;
public class TestClassComposite extends TestCase {
  public TestClassComposite(String method) {
    super(method);
  }
  static public Test suite() {
    TestSuite suite = new TestSuite();
    // Grab everything:
    suite.addTestSuite(TestClassOne.class);
    // Use the suite method:
    suite.addTest(TestClassTwo.suite());
    return suite;
  }
}
```

TestClassComposite.java

Now if you run `TestClassComposite`, the following individual test methods will be run:

- `testAddition()` from `TestClassOne`
- `testSubtraction()` from `TestClassOne`
- `testShortTest()` from `TestClassTwo`
- `testAnotherShortTest()` from `TestClassTwo`

You can keep going with this scheme; another class might include `TestClassComposite`, which would include the above methods, along with some other composite of tests, and so on.

Per-test Setup and Tear-down

Each test should run independently of every other test; this allows you to run any individual test at any time, in any order.

To accomplish this feat, you may need to reset some parts of the testing environment in between tests, or clean up after a test has run. JUnit's `TestCase` base class provides two methods that you can override to set up and then tear down the test's environment:

```
protected void setUp();
protected void tearDown();
```

The method `setUp()` will be called before each one of the `test...` methods is executed, and the method `tearDown()` will be called after each test method is executed.

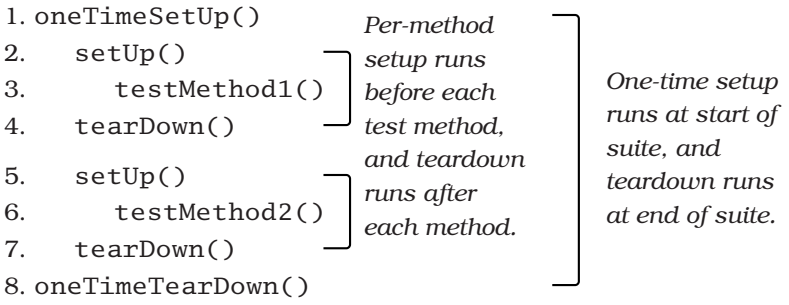

Figure 3.2: Execution Order of Setup Code

For example, suppose you needed some sort of database connection object for each test. Rather than have to include code in each test method that connects to and disconnects from the database, you could take care of that using the setup and teardown methods.

```
public class TestDB extends TestCase {
  private Connection dbConn;
  protected void setUp() {
    dbConn = new Connection("oracle", 1521,
                            "fred", "foobar");
    dbConn.connect();
  }
  protected void tearDown() {
    dbConn.disconnect();
    dbConn = null;
  }
  public void testAccountAccess() {
    // Uses dbConn
    [illegible]
  }
  public void testEmployeeAccess() {
    // Uses dbConn
    [illegible]
  }
}
```

In this example, `setUp()` will be called before `testAccountAccess()`, then `tearDown()` will be called. `setUp()` will be called again, followed by `testEmployeeAccess()` and then `tearDown()` again.

Per-suite Setup and Tear-down

Normally per-test setup is all you need, but in some circumstances you may need to set something up or clean up after the *entire* test suite has run; for that, you need per-suite setup and tear-down (the difference between per-test and per-suite execution order is shown in Figure 3.2 on the page before).

Per-suite setup is a bit more complicated. You need to provide a `suite` of the required tests (by whatever means) and wrap it in a `TestSetup` object. Using our previous example, that might look something like the following:

```
import junit.framework.*;
import junit.extensions.*;
public class TestClassTwo extends TestCase {
  private static TSP tsp;
  public TestClassTwo(String method) {
    super(method);
  }
  // This one takes a few hours...
  public void testLongRunner() {
    assertEquals(2300, tsp.shortestPath(50));
  }
  public void testShortTest() {
    assertEquals(140, tsp.shortestPath(5));
  }
  public void testAnotherShortTest() {
    assertEquals(586, tsp.shortestPath(10));
  }
  public static Test suite() {
    TestSuite suite = new TestSuite();
    // Only include short tests
    suite.addTest(new TestClassTwo("testShortTest"));
    suite.addTest(new TestClassTwo(
                                "testAnotherShortTest"));
    TestSetup wrapper = new TestSetup(suite) {
        protected void setUp() {
          oneTimeSetUp();
        }
        protected void tearDown() {
          oneTimeTearDown();
        }
      };
    return wrapper;
  }
  public static void oneTimeSetUp() {
    // one-time initialization code goes here...
    tsp = new TSP();
    tsp.loadCities("EasternSeaboard");
```

```
  }
  public static void oneTimeTearDown() {
    // one-time cleanup code goes here...
    tsp.releaseCities();
  }
}
```

TestClassTwo.java

Note that you can use both per-suite and per-test `setUp()` methods in the same class.

3.5 JUnit Custom Asserts

The standard asserts that JUnit provides are usually sufficient for most testing. However, you may run into a situation where it would be handy to have your own, customized asserts. Perhaps you've got a special data type, or a common sequence of actions that is done in multiple tests.

Don't copy and paste common code in the tests; tests should be written to the same high standards as regular code, which means honoring good coding practices such as the DRY principle,[1] orthogonality, and so on. Factor out common bits of test harness into real methods, and use those methods in your test cases.

If you have custom asserts or common code that needs to be shared throughout the project, you may want to consider subclassing `TestCase` and using the subclass for all testing. For instance, suppose you are testing a financial application and virtually all of the tests use a data type called `Money`. Instead of having tests subclass `TestCase` directly, you'd create a project-specific base testing class:

```
import junit.framework.*;
/**
 * Project-wide base class for Testing
 */
public class ProjectTest extends TestCase {
  /**
   * Assert that the amount of money is an even
   * number of dollars (no cents)
   *
```

[1] DRY stands for "Don't Repeat Yourself." It's a fundamental technique that demands that every piece of knowledge in a system must have a single, unambiguous, and authoritative representation [HT00].

```
     * @param message Text message to display if the
     *                                assertion fails
     * @param amount Money object to test
     *
     */
    public void assertEvenDollars(String message,
                                  Money amount) {
      assertEquals(message,
            amount.asDouble() - (int)amount.asDouble(),
            0.0,
            0.001);
    }
    /**
     * Assert that the amount of money is an even
     * number of dollars (no cents)
     *
     * @param amount Money object to test
     *
     */
    public void assertEvenDollars(Money amount) {
      assertEvenDollars("", amount);
    }
  }
```

ProjectTest.java

Note that we provide both forms of assert: one that takes a `String` and one that does not. Note also that we didn't copy any code in doing so; we merely forward the call on.

Now all the other test classes in the project will inherit from this base class instead of directly from `TestCase`:

```
public class TestSomething extends ProjectTest {
  ...
```

In fact, it is usually a good idea to start off a new project by *always* inheriting from your own custom base class instead of directly from the JUnit class—even if your base class doesn't add any extra functionality at first. That way, should you need to add a method or capability that every test class needs, you can simply add it to your base class without having to edit every test case in the project.

3.6 JUnit and Exceptions

There are two kinds of exceptions that we might be interested in:

1. Expected exceptions resulting from a test
2. Unexpected exceptions from something that's gone horribly wrong

Contrary to what you might think, exceptions are really good things—they tell us that something is wrong. Sometimes in a test, we *want* the method under test to throw an exception. Consider a method named `sortMyList()`. It's supposed to throw an exception if passed a null list. We must test for that explicitly.

```
Line 1   public void testForException() {
   -       try {
   -         sortMyList(null);
   -         fail("Should have thrown an exception");
   5       } catch (RuntimeException e) {
   -         assertTrue(true);
   -       }
   -     }
```

The method call under test is within a try/catch block on line 3. This method is expected to throw an exception, so if it doesn't—if we proceed past line 3—then we need to fail the test immediately. If the exception fires as expected, the code will proceed to line 6 and record the assertion for statistical purposes.

Now you might ask why bother with the `assertTrue`.It won't do anything, it can't fail, so why bother putting it in? Any use of `assertTrue(true)` translates as "I expect that the control flow should pass through here." That's a strong form of documentation that will help clear up any possible misunderstandings later. However, don't forget that an `assertTrue(true)` that is never called does *not* generate a failure.

In general, you should test a method for every declared exception, and make sure that it throws it when it should. That covers us for expected exceptions, but what about unexpected exceptions?

While you can catch all possible exceptions yourself and call JUnit's `fail()`, you'd be better off letting JUnit do the hard work. For instance, suppose you are reading a file of test data. Rather than catching the possible I/O exceptions yourself, simply declare your test method to throw the possible exceptions:

```
public void testData1() throws FileNotFoundException {
  FileInputStream in = new FileInputStream("data.txt");
  ...
}
```

The JUnit framework will catch any thrown exception and report it as an error, without any extra effort on your part. Even better, JUnit will report the *entire* stack trace right down to the bug itself, not just to the failed assert, which is of tremendous help when trying to figure out why a test failed.

3.7 More on Naming

Normally, you want all tests to pass all of the time. But suppose you've thought up a bunch of tests first, written them, and are now working your way through implementing the code required to pass the tests. What about all those new tests that would fail now?

You can go ahead and write these tests, but you don't want the testing framework to run these tests just yet. Fortunately, most testing frameworks use a naming convention to discover tests automatically. When using JUnit in Java, for instance, methods whose names start with "`test`" (as in `testMyThing`) will be run as tests; all you need to do is name the method something else until you're ready to tackle it. If you named pending tests as "`pendingTestMyThing`", then not only will the testing framework ignore it for now, but you could also search for the string "pendingTest" throughout your code to find easily any tests you may have missed. Of course, the code still has to compile cleanly; if it's not ready for that yet, then you should comment out the offending parts.

What you want to avoid at all costs is the habit of *ignoring* failing test results.

3.8 JUnit Test Skeleton

All you *really* need to write tests using JUnit are just these three things:

1. An import statement to bring in `junit.framework.*`
2. An `extends` statement so your class will inherit from `TestCase`
3. A constructor with a call to `super(`*string*`)`

Many IDE's will supply at least this much for you. Classes written this way can be run using a JUnit test runner, and will automatically execute all of the `test...` methods in the class.

But sometimes it's more convenient to be able to run a test class directly, without running it from a JUnit runner. And what *were* the names of those methods that ran before and after each test?

We can make a skeleton that provides all of these features and more pretty easily. If this would be helpful for your project, take a look at Appendix C on page 127.

Now that you've got a good idea of *how* to write tests, it's time to take a closer look at figuring out *what* to test.

Chapter 4

What to Test: The Right-BICEP

It can be hard to look at a method or a class and try to come up with all the ways it might fail; to anticipate all the bugs that might be lurking in there. With enough experience, you start to get a feel for those things that are "likely to break," and can effectively concentrate on testing in those areas first. But without a lot of experience, it can be hard and frustrating trying to discover possible failure modes. End-users are quite adept at finding our bugs, but that's both embarrassing and damaging to our careers! What we need are some guidelines, some reminders of areas that might be important to test.

Let's take a look at six specific areas to test that will strengthen your testing skills, using your RIGHT-BICEP:

- **Right** — Are the results **right**?
- **B** — Are all the **boundary** conditions CORRECT?
- **I** — Can you check **inverse** relationships?
- **C** — Can you **cross-check** results using other means?
- **E** — Can you force **error conditions** to happen?
- **P** — Are **performance** characteristics within bounds?

4.1 Are the Results Right?

Right BICEP

The first and most obvious area to test is simply to see if the expected results are right—to validate the results.

We've seen simple data validation already: the tests in Chapter 2 that verify that a method returns the largest number from a list.

These are usually the "easy" tests, and many of these sorts of validations may even be specified in the requirements. If they aren't, you'll probably need to ask someone. You need to be able to answer the key question:

If the code ran correctly, how would I know?

If you cannot answer this question satisfactorily, then writing the code—or the test—may be a complete waste of time. "But wait," you may say, "that doesn't sound very agile! What if the requirements are vague or incomplete? Does that mean we can't write code until all the requirements are firm?"

No, not at all. If the requirements are truly not yet known, or complete, you can always invent some as a stake in the ground. They may not be correct from the user's point of view, but you now know what *you* think the code should do, and so you can answer the question.

Of course, you must then arrange for feedback with users to fine-tune your assumptions. The definition of "correct" may change over the lifetime of the code in question, but at any point, you should be able to prove that it's doing what you think it ought.

Using Data Files

For sets of tests with large amounts of test data, you might want to consider putting the test values and/or results in a separate data file that the unit test reads in. This doesn't need to be a very complicated exercise—and you don't even need to use XML.[1] Figure 4.1 on the next page is a version of `TestLargest` that reads in all of the tests from a data file.

[1] This is clearly a joke. XML is mandatory on all projects today, isn't it?

```
import junit.framework.*;
import java.io.*;
import java.util.ArrayList;
import java.util.StringTokenizer;

public class TestLargestDataFile extends TestCase {
  public TestLargestDataFile(String name) {
    super(name);
  }
  /* Run all the tests in testdata.txt (does not test
   * exception case). We'll get an error if any of the
   * file I/O goes wrong.
   */
  public void testFromFile() throws Exception {
    String line;
    BufferedReader rdr = new BufferedReader(
                                    new FileReader(
                                    "testdata.txt"));
    while ((line = rdr.readLine()) != null) {
      if (line.startsWith("#")) { // Ignore comments
        continue;
      }
      StringTokenizer st = new StringTokenizer(line);
      if (!st.hasMoreTokens()) {
        continue; // Blank line
      }
      // Get the expected value
      String val = st.nextToken();
      int expected = Integer.valueOf(val).intValue();
      // And the arguments to Largest
      ArrayList argument_list = new ArrayList();
      while (st.hasMoreTokens()) {
        argument_list.add(Integer.valueOf(
                                    st.nextToken()));
      }
      // Transfer object list into native array
      int[] arguments = new int[argument_list.size()];
      for (int i=0; i < argument_list.size(); i++) {
        arguments[i] = ((Integer)argument_list.
                        get(i)).intValue();
      }
      // And run the assert
      assertEquals(expected,
                   Largest.largest(arguments));
    }
  }
}
```

Figure 4.1: TestLargestDataFile: Reading test specifications from a file.

The data file has a very simple format; each line contains a set of numbers. The first number is the expected answer, the numbers on the rest of the line are the arguments with which to test. We'll allow a pound-sign (#) for comments, so that you can put meaningful descriptions and notes in the test file.

The test file can then be as simple as:

```
#
# Simple tests:
#
9  7 8 9
9  9 8 7
9  9 8 9
#
# Negative number tests:
#
-7  -7 -8 -9
-7  -8 -7 -8
-7  -9 -7 -8
#
# Mixture:
#
7   -9 -7 -8 7 6 4
9   -1 0 9 -7 4
#
# Boundary conditions:
#
1 1
0 0
2147483647 2147483647
-2147483648 -2147483648
```

testdata.txt

For just a handful of tests, as in this example, it's probably not worth the effort. But say this was a more advanced application, with tens or even hundreds of test cases in this form. Then the data file approach becomes a very compelling choice.

Be aware that test data, whether it's in a file or in the test code itself, might well be incorrect. In fact, experience suggests that test data is *more likely* to be incorrect than the code you're testing, especially if the data was hand-calculated or obtained from a system we're replacing (where new features may deliberately cause new results). When test data says you're wrong, double- and triple-check that the test data is right before attacking the code.

Something else to think about: the code as presented does not test any exception cases. How might you implement that?

Do whatever makes it easiest for you to prove that the method is right.

4.2 Boundary Conditions

Right **B** *ICEP*

In the previous "largest number" example, we discovered several boundary conditions: when the largest value was at the end of the array, when the array contained a negative number, an empty array, and so on.

Identifying boundary conditions is one of the most valuable parts of unit testing, because this is where most bugs generally live—at the edges. Some conditions you might want to think about:

- Totally bogus or inconsistent input values, such as a file name of `"!*W:X\&Gi/w~>g/h#WQ@"`.
- Badly formatted data, such as an e-mail address without a top-level domain (`"fred@foobar."`).
- Empty or missing values (such as `0`, `0.0`, `""`, or `null`).
- Values far in excess of reasonable expectations, such as a person's age of 10,000 years.
- Duplicates in lists that shouldn't have duplicates.
- Ordered lists that aren't, and vice-versa. Try handing a pre-sorted list to a sort algorithm, for instance—or even a reverse-sorted list.
- Things that arrive out of order, or happen out of expected order, such as trying to print a document before logging in, for instance.

An easy way to think of possible boundary conditions is to remember the acronym CORRECT. For each of these items, consider whether or not similar conditions may exist in your method that you want to test, and what might happen if these conditions were violated:

- **C**onformance — Does the value conform to an expected format?
- **O**rdering — Is the set of values ordered or unordered as appropriate?
- **R**ange — Is the value within reasonable minimum and maximum values?

- **R**eference — Does the code reference anything external that isn't under direct control of the code itself?
- **E**xistence — Does the value exist (e.g., is non-null, non-zero, present in a set, etc.)?
- **C**ardinality — Are there exactly enough values?
- **T**ime (absolute and relative) — Is everything happening in order? At the right time? In time?

We'll examine all of these boundary conditions in the next chapter.

4.3 Check Inverse Relationships

Right B **I** *CEP*

Some methods can be checked by applying their logical inverse. For instance you might check a method that calculates a square root by squaring the result, and testing that it is tolerably close to the original number:

```
public void testSquareRootUsingInverse() {
  double x = mySquareRoot(4.0);
  assertEquals(4.0, x * x, 0.0001);
}
```

You might check that some data was successfully inserted into a database by then searching for it, and so on.

Be cautious when you've written both the original routine and it's inverse, as some bugs might be masked by a common error in both routines. Where possible, use a different source for the inverse test. In the square root example, we're just using regular multiplication to test our method. For the database search, we'll probably use a vendor-provided search routine to test our insertion.

4.4 Cross-check Using Other Means

Right BI **C** *EP*

You might also be able to cross-check results of your method using different means.

Usually there is more than one way to calculate some quantity; we might pick one algorithm over the others because it performs better, or has other desirable characteristics. That's

the one we'll use in production, but we can use one of the other versions to cross-check our results in the test system. This technique is especially helpful when there's a proven, known way of accomplishing the task that happens to be too slow or too inflexible to use in production code.

We can use that somewhat lesser version to our advantage to check that our new super-spiffy version is producing the same results:[2]

```
public void testSquareRootUsingStd() {
  double number = 3880900.0;
  double root1 = mySquareRoot(number);
  double root2 = Math.sqrt(number);
  assertEquals(root2, root1, 0.0001);
}
```

Another way of looking at this is to use different pieces of data from the class itself to make sure they all "add up." For instance, suppose you were working on a library's database system (that is, a brick-and-mortar library that lends out real books). In this system, the number of copies of a particular book should always balance. That is, the number of copies that are checked out plus the number of copies sitting on the shelves should always equal the total number of copies in the collection. These are separate pieces of data, and may even be reported by objects of different classes, but they still have to agree, and so can be used to cross-check one another.

4.5 Force Error Conditions

Right BIC **E** *P*

In the real world, errors happen. Disks fill up, network lines drop, e-mail goes into a black hole, and programs crash. You should be able to test that your code handles all of these real-world problems by forcing errors to occur.

That's easy enough to do with invalid parameters and the like, but to simulate specific network errors—without unplugging any cables—takes some special techniques. We'll discuss one way to do this using Mock Objects in Chapter 6 on page 63.

[2] Some spreadsheet engines (as found in Microsoft Excel™, etc.) employ similar techniques to check that the models and methods chosen to solve a particular problem are appropriate, and that the answers from different applicable methods agree with each other.

But before we get there, consider what kinds of errors or other environmental constraints you might introduce to test your method? Make a short list before reading further.

Think about this for a moment before reading on...

Here are a few environmental things we've thought of.

- Running out of memory
- Running out of disk space
- Issues with wall-clock time
- Network availability and errors
- System load
- Limited color palette
- Very high or very low video resolution

4.6 Performance Characteristics

Right BICE **P**

One area that might prove beneficial to examine is performance characteristics—not performance itself, but trends as input sizes grow, as problems become more complex, and so on.

What we'd like to achieve is a quick regression test of performance characteristics. All too often, we might release one version of the system that works okay, but somehow by the next release it has become dead-dog slow. We don't know why, or what change was made, or when, or who did it, or anything. And the end users are screaming bloody murder.

To avoid that awkward scenario, you might consider some rough tests just to make sure that the performance curve remains stable. For instance, suppose we've written a filter that identifies web sites that we wish to block (using our new product to view naughty pictures might get us in all sorts of legal trouble, after all.)

The code works fine with a few dozen sample sites, but will it work as well with 10,000? 100,000? Let's write a unit test to find out.

```
public void testURLFilter() {
  Timer timer = new Timer();
  String naughty_url = "http://www.xxxxxxxxxx.com";
  // First, check a bad URL against a small list
  URLFilter filter = new URLFilter(small_list);
  timer.start();
  filter.check(naughty_url);
  timer.end();
  assertTrue(timer.elapsedTime() < 1.0);
  // Next, check a bad URL against a big list
  filter = new URLFilter(big_list);
  timer.start();
  filter.check(naughty_url);
  timer.end();
  assertTrue(timer.elapsedTime() < 2.0);
  // Finally, check a bad URL against a huge list
  filter = new URLFilter(huge_list);
  timer.start();
  filter.check(naughty_url);
  timer.end();
  assertTrue(timer.elapsedTime() < 3.0);
}
```

This gives us some assurance that we're still meeting performance targets. But because this one test takes 6–7 seconds to run, we may not want to run it every time. As long as we run it nightly or every couple of days, we'll quickly be alerted to any problems we may introduce, while there is still time to fix them.

You may want to investigate test decorators that provide better support for timing individual tests, simulating heavy load conditions, and so on, such as the freely available JUnitPerf.[3]

[3] http://www.clarkware.com

Chapter 5

CORRECT Boundary Conditions

Many bugs in code occur around "boundary conditions," that is, under conditions where the code's behavior may be different from the normal, day-to-day routine.

For instance, suppose you have a function that takes two integers:

```
public int calculate(int a, int b) {
  return a / (a + b);
}
```

Most of the time, this code will return a number just as you expect. But if the sum of `a` and `b` happens to equal zero, you will get an `ArithmeticException` instead of a return value. That is a boundary condition—a place where things might suddenly go wrong, or at least behave differently from your expectations.

To help you think of tests for boundary conditions, we'll use the acronym CORRECT:

- **C**onformance — Does the value conform to an expected format?
- **O**rdering — Is the set of values ordered or unordered as appropriate?

- **R**ange — Is the value within reasonable minimum and maximum values?
- **R**eference — Does the code reference anything external that isn't under direct control of the code itself?
- **E**xistence — Does the value exist (e.g., is non-null, non-zero, present in a set, etc.)?
- **C**ardinality — Are there exactly enough values?
- **T**ime (absolute and relative) — Is everything happening in order? At the right time? In time?

Let's look at each one of these in turn. Remember that for each of these areas, you want to consider data that is passed in as arguments to your method as well as internal data that you maintain inside your method and class.

The underlying question that you want to answer fully is:

What* else *can go wrong?

Once you think of something that could go wrong, write a test for it. Once that test passes, again ask yourself, "what else can go wrong?" and write another test, and so on.

5.1 Conformance

Many times you expect or produce data that must conform to some specific format. An e-mail address, for instance, isn't just a simple string. You expect that it must be of the form: [C] *ORRECT*

```
name@somewhere.com
```

With the possibility of extra dotted parts:

```
firstname.lastname@subdomain.somewhere.com
```

And even oddballs like this one:

```
firstname.lastname%somewhere@subdomain.somewhere.com
```

Suppose you are writing a method that will extract the user's name from their e-mail address. You'll expect that the user's name is the portion before the "@" sign. What will your code

do if there *is* no "@" sign? Will it work? Throw a runtime exception? Is this a boundary condition you need to consider?[1]

Validating formatted string data such as e-mail addresses, phone numbers, account numbers, or file names is usually straightforward. But what about more complex structured data? Suppose you are reading some sort of report data that contains a header record linked to some number of data records, and finally to a trailer record. How many conditions might we have to test?

- What if there's no header, just data and a trailer?
- What if there's no data, just a header and trailer?
- What if there's no trailer, just a header and data?
- What if there's just a trailer?
- What if there's just a header?
- What if there's just data?

Just as with the simpler e-mail address example, you have to consider what will happen if the data does not conform to the structure you think it should.

And of course, if you are creating something like an e-mail address (possibly building it up from different sources) or the structured data above, you want to test your result to make sure it conforms.

5.2 Ordering

C**O**RRECT

Another area to consider is the order of data, or the position of one piece of data within a larger collection. For instance, in the `largest()` example in the previous chapter, one bug manifested itself depending on whether the largest number you were searching for was at the beginning or end of the list.

That's one aspect of ordering. Any kind of search routine should be tested for conditions where the search target is first or last, as many common bugs can be found that way.

[1] E-mail addresses are actually very complicated. A close reading of RFC822 may surprise you.

For another aspect of ordering, suppose you are writing a method that is passed a collection containing a restaurant order. You would probably expect that the appetizers will appear first in the order, followed by the salad (and that all-important dressing choice), then the entree and finally a decadent dessert involving lots of chocolate.

What happens to your code if the dessert is first, and the entree is last?

If there's a chance that sort of thing can happen, **and** if it's the responsibility of your method to deal with it if it does, then you need to test for this condition and address the problem. Now, it may be that this is not something your method needs to worry about. Perhaps this needs to be addressed at the user input level (see "Testing Invalid Parameters" later on).

If you're writing a sort routine, what might happen if the set of data is already ordered? Or worse yet, sorted in precisely reverse order? Ask yourself if that could cause trouble—if these are conditions that might be worth testing, too.

If you are supposed to maintain something in order, check that it is. For example, if your method is part of the GUI that is sending the dinner order back to the kitchen, you should have a test that verifies that the items are in the correct serving order:

```
public void testKitchenOrder() {
  Order order = new Order();
  FoodItem dessert = new Dessert("Chocolate Souffle");
  FoodItem entree  = new Entree("Beef Oscar");
  FoodItem salad   = new Salad("Tossed",
                               "Parmesan Peppercorn");

  // Add out of order
  order.addFoodItem(dessert);
  order.addFoodItem(entree);
  order.addFoodItem(salad);

  // But should come out in serving order
  Iterator itr = order.iterator();

  assertEquals(salad, itr.next());
  assertEquals(entree, itr.next());
  assertEquals(dessert, itr.next());

  // No more left
  assertFalse(itr.hasNext());
}
```

Of course, from a human factors standpoint, you'd need to modify the code so that it's flexible enough to allow people to

eat their ice cream first, if so desired. In which case, you'd need to add a test to prove that your four-year old nephew's ice cream comes with everyone else's salads, but Grandma's ice cream comes at the end with your cappuccino.

5.3 Range

CORRECT

Range is a convenient catch-all word for the situation where a variable's type allows it to take on a wider range of values than you need—or want. For instance, a person's age is typically represented as an integer, but no one has ever lived to be 200,000 years old, even though that's a perfectly valid integer value. Similarly, there are only 360 degrees in a circle, even though degrees are commonly stored in an integer.

In good object oriented design, you do not use a raw native type (e.g., an `int` or `Integer`) to store a bounded-integer value such as an age, or a compass heading.

```
/**
 * Compass bearing
 */
public class Bearing {
  protected int bearing; // 0..359
  /**
   * Initialize a bearing to a value from 0..359
   */
  public Bearing(int num_degrees) {
    if (num_degrees < 0 || num_degrees > 359) {
      throw new RuntimeException("Bad bearing");
    }
    bearing = num_degrees;
  }
  /**
   * Return the angle between our bearing and another.
   * May be negative.
   */
  public int angleBetween(Bearing anOther) {
    return bearing - anOther.bearing;
  }
}
```

Bearing.java

Notice that the angle returned is just an int, as we are not placing any range restrictions on the result (it may be negative, etc.)

By encapsulating the concept of a bearing within a class, you've now got one place in the system that can filter out bad

data. You cannot create a `Bearing` object with out of range values. Thus, the rest of the system can use `Bearing` objects and be assured that they contain only reasonable values.

Other ranges may not be as straightforward. For instance, suppose you have a class that maintains two sets of *x, y* coordinates. These are just integers, with arbitrary values, but the constraint on the range is such that the two points must describe a rectangle with no side greater than 100 units. That is, the allowed range of values for both *x, y* pairs is interdependent. You'll want a range test for any method that can affect a coordinate to ensure that the resulting range of the *x, y* pairs remains legitimate. For more information on this topic, see "invariants" in the Design Issues chapter on page 99.

Since you will likely call this from a number of different tests, it probably makes sense to make a new assert method:

```
public static final int MAX_DIST = 100;
public void assertPairRange(String message,
                            Point one, Point two) {
  assertTrue(message,
             Math.abs(one.x - two.x) <= MAX_DIST);
  assertTrue(message,
             Math.abs(one.y - two.y) <= MAX_DIST);
}
```

But the most common ranges you'll want to test probably depend on physical data structure issues, not application domain constraints. Take a simple example like a stack class that implements a stack of `Strings` using an array:

```
public class MyStack {
  public MyStack() {
    stack = new String[100];
    next_index = 0;
  }
  public String pop() {
    return stack[--next_index];
  }
  // Delete n items from the stack en-masse
  public void delete(int n) {
    next_index -= n;
  }
  public void push(String aString) {
    stack[next_index++] = aString;
  }
  public String top() {
    return stack[next_index-1];
  }
```

```
  private int next_index;
  private String[] stack;
}
```

MyStack.java

There are some potential bugs lurking here, as there are no checks at all for either an empty stack or a stack overflow. However we manipulate the index variable `next_index`, one thing is supposed to be always true: `(next_index >= 0 && next_index < stack.length)`. We'd like to check to make sure this expression is true.

Both `next_index` and `stack` are private variables; you don't want to have to expose those just for the sake of testing. There are several ways around this problem; for now we'll just make a special method in `MyStack` named `checkInvariant()`:

```
public void checkInvariant()
  throws InvariantException {
  // JDK 1.4 can use assert() instead
  if (!(next_index >= 0 &&
        next_index  < stack.length)) {
    throw new InvariantException(
                        "next_index out of range: " +
                        next_index +
                        " for stack length " +
                        stack.length);
  }
}
```

MyStack.java

Now a test method can call `checkInvariant()` to ensure that nothing has gone awry inside the guts of the stack class, without having direct access to those same guts.

```
import junit.framework.*;
public class TestMyStack extends TestCase {
  public void testEmpty() throws InvariantException {
    MyStack stack = new MyStack();
    stack.checkInvariant();
    stack.push("sample");
    stack.checkInvariant();
    // Popping last element ok
    assertEquals("sample", stack.pop());
    stack.checkInvariant();
    // Delete from empty stack
    stack.delete(1);
    stack.checkInvariant();
  }
}
```

TestMyStack.java

When you run this test, you'll quickly see that we need to add some range checking!

```
There was 1 error:
1) testEmpty(TestMyStack)InvariantException:
   next_index out of range: -1 for stack length of 100
        at MyStack.checkInvariant(MyStack.java:31)
        at TestMyStack.testEmpty(TestMyStack.java:16)
```

It's much easier to find and fix this sort of error here in a simple testing environment instead of buried in a real application.

Almost any indexing concept (whether it's a genuine integer index or not) should be extensively tested. Here are a few ideas to get you started:

- Start and End index have the same value
- First is greater than Last
- Index is negative
- Index is greater than allowed
- Count doesn't match actual number of items
- . . .

5.4 Reference

CORRECT

What things does your method reference that are outside the scope of the method itself? Any external dependencies? What state does the class have to have? What other conditions must exist in order for the method to work?

For example, a method in a web application to display a customer's account history might require that the customer is first logged on. The method `pop()` for a stack requires a non-empty stack. Shifting the transmission in your car to Park from Drive requires that the car is stopped.

If you have to make assumptions about the state of the class and the state of other objects or the global application, then you need to test your code to make sure that it is well-behaved if those conditions are not met. For example, the code for the microprocessor-controlled transmission might have unit tests that check for that particular condition: the state of the transmission (whether it can shift into Park or not) depends on the state of the car (is it in motion or stopped).

```
public void testJamItIntoPark() {
  transmission.shift(DRIVE);
  car.accelerateTo(35);
  assertEquals(DRIVE, transmission.getGear());
  // should silently ignore
  transmission.shift(PARK);
  assertEquals(DRIVE, transmission.getGear());
  car.accelerateTo(0); // i.e., stop
  car.brakeToStop();
  // should work now
  transmission.shift(PARK);
  assertEquals(PARK, transmission.getGear());
}
```

The *preconditions* for a given method specify what state the world must be in for this method to run. In this case, the precondition for putting the transmission in park is that the car's engine (a separate component elsewhere in the application's world) must be at a stop. That's a documented requirement for the method, so we want to make sure that the method will behave gracefully (in this particular case, just ignore the request silently) in case the precondition is not met.

At the end of the method, *postconditions* are those things that you guarantee your method will make happen. Direct results returned by the method are one obvious thing to check, but if the method has any side-effects then you need to check those as well. In this case, applying the brakes has the side effect of stopping the car.

Some languages even have built-in support for preconditions and postconditions; interested readers might want to read about Eiffel in *Object-Oriented Software Construction* [Mey97].

5.5 Existence

A large number of potential bugs can be discovered by asking the key question "does some given thing exist?". CORR **E** CT

For any value you are passed in or maintain, ask yourself what would happen to the method if the value didn't exist—if it were null, or blank, or zero.

Many Java library methods will throw an exception of some sort when faced with non-existent data. The problem is that it's hard to debug a generic runtime exception buried deep

in some library. But an exception that reports "Age isn't set" makes tracking down the problem much easier.

Most methods will blow up if expected data is not available, and that's probably **not** what you want them to do. So you test for the condition—see what happens if you get a `null` instead of a `CustomerRecord` because some search failed. See what happens if the file doesn't exist, or if the network is unavailable.

Ah, yes: things in the environment can wink out of existence as well—networks, files' URLs, license keys, users, printers—you name it. All of these things may not exist when you expect them to, so be sure to test with plenty of nulls, zeros, empty strings and other nihilist trappings.

Make sure your method can stand up to nothing.

5.6 Cardinality

CORRE C T

Cardinality has nothing to do with either highly-placed religious figures or small red birds, but instead with counting.

Computer programmers (your humble authors included) are really bad at counting, especially past 10 when the fingers can no longer assist us. For instance, answer the following question quickly, off the top of your head, without benefit of fingers, paper, or UML:

> If you've got 12 feet of lawn that you want to fence, and each section of fencing is 3 feet wide, how many fence posts do you need?

If you're like most of us, you probably answered "4" without thinking too hard about it. Pity is, that's wrong—you need five fence posts as shown in Figure 5.1 on page 57. This model, and the subsequent common errors, come up so often that they are graced with the name "fence post errors."

It's one of many ways you can end up being "off by one;" an occasionally fatal condition that afflicts all programmers from time to time. So you need to think about ways to test how well your method counts, and check to see just how many of a thing you may have.

It's a related problem to Existence, but now you want to make sure you have exactly as many as you need, or that you've made exactly as many as needed. In most cases, the count of some set of values is only interesting in these three cases:

1. Zero
2. One
3. More than one

It's called the "0–1–*n* Rule," and it's based on the premise that if you can handle more than one of something, you can probably handle 10, 20, or 1,000 just as easily. Most of the time that's true, so many of our tests for cardinality are concerned with whether we have 2 or more of something. Of course there are situations where an exact count makes a difference—10 might be important to you, or 250.

Suppose you are maintaining a list of the Top-Ten food items ordered in a pancake house. Every time an order is taken, you have to adjust the top-ten list. You also provide the current top-ten list as a real-time data feed to the pancake boss's PDA. What sort of things might you want to test for?

- Can you produce a report when there aren't yet ten items in the list?
- Can you produce a report when there are no items on the list?
- Can you produce a report when there is only one item on the list?
- Can you add an item when there aren't yet ten items in the list?
- Can you add an item when there are no items on the list?
- Can you add an item when there is only one item on the list?
- What if there aren't ten items on the menu?
- What if there are no items on the menu?

Having gone through all that, the boss now changes his mind and wants a top-twenty list instead. What do you have to change?

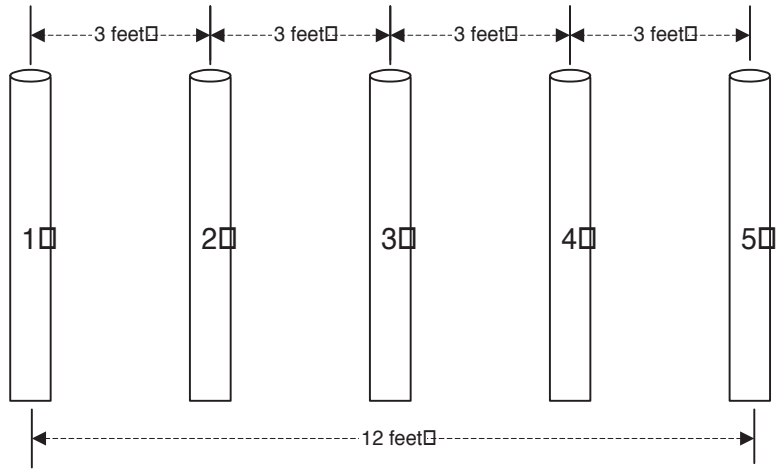

Figure 5.1: A Set of Fence posts

The correct answer is “one line,” something like the following:

```
public int getMaxEntries() {
  return 20;
}
```

Now, when the boss gets overwhelmed and pleads with you to change this to be a top-five report (his PDA is pretty small, after all), you can go back and change this one number. The test should automatically follow suit, because it uses the same accessor.

So in the end, the tests concentrate on boundary conditions of 0, 1, and *n*, where *n* can—and will—change as the business demands.

5.7 Time

CORREC T

The last boundary condition in the CORRECT acronym is Time. There are several aspects to time you need to keep in mind:

- Relative time (ordering in time)
- Absolute time (elapsed and wall clock)
- Concurrency issues

Some interfaces are inherently stateful; you expect that `login()` will be called before `logout()`, that `prepareStatement()` is called before `executeStatement()`, `connect()` before `read()` which is before `close()`, and so on.

What happens if those methods are called out of order? Maybe you should try calling methods out of the expected order. Try skipping the first, last and middle of a sequence. Just as order of data may have mattered to you in the earlier examples (as we described in "Ordering" on page 48), now it's the order of the calling sequence of methods.

Relative time might also include issues of timeouts in the code: how long your method is willing to wait for some ephemeral resource to become available. As we'll discuss shortly, you'll want to exercise possible error conditions in your code, including things such as timeouts. Maybe you've got conditions that aren't guarded by timeouts—can you think of a situation where the code might get "stuck" waiting forever for something that might not happen?

This leads us to issues of elapsed time. What if something you are waiting for takes "too much" time? What if your method takes too much time to return to the caller?

Then there's the actual wall clock time to consider. Most of the time, this makes no difference whatsoever to code. But every now and then, time of day will matter, perhaps in subtle ways. Here's a quick question, true or false: every day of the year is 24 hours long?

The answer is *"it depends."* In UTC (Universal Coordinated Time, the modern version of Greenwich Mean Time, or GMT), the answer is YES. In areas of the world that do not observe Daylight Savings Time (DST), the answer is YES. In most of the U.S. (which does observe DST), the answer is NO. In April, you'll have a day with 23 hours (spring forward) and in October you'll have a day with 25 (fall back). This means that arithmetic won't always work as you expect; 1:45AM plus 30 minutes might equal 1:15, for instance.

But you've tested any time-sensitive code on those boundary days, right? For locations that honor DST and for those that do not?

Oh, and don't assume that any underlying library handles these issues correctly on your behalf. Unfortunately, when it comes to time, there's a lot of broken code out there.

Finally, one of the most insidious problems brought about by time occurs in the context of concurrency and synchronized access issues. It would take an entire book to cover designing, implementing, and debugging multi-threaded, concurrent programs, so we won't take the time now to go into details, except to point out that most code you write in most languages today **will** be run in a multi-threaded environment.

So ask yourself, what will happen if multiple threads use this same object at the same time? Are there global or instance-level data or methods that need to be synchronized? How about external access to files or hardware? Be sure to add the `synchronized` keyword to any data element or method that needs it, and try firing off multiple threads as part of your test.[2]

5.8 Try It Yourself

Now that we've covered the Right-BICEP and CORRECT way to come up with tests, it's your turn to try.

For each of the following examples and scenarios, write down as many possible unit tests as you can think of.

Exercises

1. **A simple stack class.** Push `String` objects onto the stack, and `pop` them off according to normal stack semantics. This class provides the following methods:

Answer on 137

```
public interface StackExercise {
  /**
    * Return and remove the most recent item from
    * the top of the  stack.
    * Throws StackEmptyException
    * if the stack is empty
    */
  public String pop() throws StackEmptyException;
```

[2] JUnit as shipped has some issues with multi-threaded test cases, but there are various fixes available on the net.

```
    /**
     * Add an item to the top of the stack.
     */
    public void push(String item);
    /**
      * Return but do not remove the most recent
      * item from the top of the stack.
      * Throws StackEmptyException
      * if the stack is empty
      */
    public String top() throws StackEmptyException;
    /**
      * Returns true if the stack is empty.
      */
    public boolean isEmpty();
}
```

StackExercise.java

Here are some hints to get you started: what is likely to break? How should the stack behave when it is first initialized? After it's been used for a while? Does it really do what it claims to do?

2. **A shopping cart.** This class lets you add, delete, and count the items in a shopping cart.

Answer on 138

What sort of boundary conditions might come up? Are there any implicit restrictions on what you can delete? Are there any interesting issues if the cart is empty?

```
public interface ShoppingCart {
    /**
     * Add this many of this item to the
     * shopping cart.
     */
    public void addItems(Item anItem, int quantity)
                throws NegativeCountException;
    /**
     * Delete this many of this item from the
     * shopping cart
     */
    public void deleteItems(Item anItem,
                            int quantity)
                throws NegativeCountException,
                       NoSuchItemException;
    /**
     * Count of all items in the cart
     * (that is, all items x qty each)
     */
    public int itemCount();
    /**
     * Return Iterator of all items
     * (see Java Collection's doc)
     */
    public Iterator iterator();
}
```

3. **A fax scheduler.** This code will send faxes from a specified file name to a U.S. phone number. There is a validation requirement; a U.S. phone number with area code must be of the form *xnn-nnn-nnnn*, where *x* must be a digit in the range `[2..9]` and *n* can be `[0..9]`. The following blocks are reserved and are not currently valid area codes: *x*`11`, *x*`9`*n*, `37`*n*, `96`*n*. Answer on 139

   The method's signature is:

```
/**
 * Send the named file as a fax to the
 * given phone number.
 */
public boolean sendFax(String phone,
                       String filename)
               throws MissingOrBadFileException,
                      PhoneFormatException,
                      PhoneAreaCodeException;
```

   Given these requirements, what tests for boundary conditions can you think of?

4. **An automatic sewing machine that does embroidery.** The class that controls it takes a few basic commands. The coordinates (0,0) represent the lower-left corner of the machine. *x* and *y* increase as you move toward the upper-right corner, whose coordinates are x = `getTableSize().width - 1` and y = `getTableSize().height - 1`. Answer on 140

   Coordinates are specified in fractions of centimeters.

```
public void moveTo(float x, float y);
public void sewTo(float x, float y);
public void setWorkpieceSize(float width,
                             float height);
public Size getWorkpieceSize();
public Size getTableSize();
```

   There are some real-world constraints that might be interesting: you can't sew thin air, of course, and you can't sew a workpiece bigger than the machine.

   Given these requirements, what boundary conditions can you think of?

5. **Audio/Video Editing Transport.** A class that provides methods to control a VCR or tape deck. There's the notion of a "current position" that lies somewhere between the beginning of tape (BOT) and the end of tape (EOT). Answer on 141

   You can ask for the current position and move from there to another given position. *Fast-forward* moves from current posi-

tion toward EOT by some amount. *Rewind* moves from current position toward BOT by some amount.

When tapes are first loaded, they are positioned at BOT automatically.

```
public interface AVTransport {
  /**
   * Move the current position ahead by this many
   * seconds. Fast-forwarding past end-of-tape
   * leaves the position at end-of-tape
   */
  public void fastForward(float seconds);
  /**
   * Move the current position backwards by this
   * many seconds. Rewinding past zero leaves
   * the position at zero
   */
  public void rewind(float seconds);
  /**
   * Return current time position in seconds
   */
  public float currentTimePosition();
  /**
   * Mark the current time position with this label
   */
  public void markTimePosition(String name);
  /**
   * Change the current position to the one
   * associated with the marked name
   */
  public void gotoMark(String name);
}
```

AVTransport.java

6. **Audio/Video Editing Transport, Release 2.0.** As above, but now you can position in seconds, minutes, or frames (there are exactly 30 frames per second in this example), and you can move relative to the beginning or the end.

Answer on 142

Chapter 6

Using Mock Objects

The objective of unit testing is to exercise just one method at a time, but what happens when that method depends on other things—hard-to-control things such as the network, or a database, or even a servlet engine?

What if your code depends on other parts of the system—maybe even *many* other parts of the system? If you're not careful, you might find yourself writing tests that end up initializing nearly every system component just to give the tests enough context in which to run. Not only is this time consuming, it also introduces a ridiculous amount of coupling into the testing process: someone goes and changes an interface or a database table, and suddenly the setup code for your poor little unit test dies mysteriously. Even the best-intentioned developers will become discouraged after this happens a few times, and eventually may abandon all testing. But there are techniques we can use to help.

In movie and television production, crews will often use *stand-ins* or *doubles* for the real actors. In particular, while the crews are setting up the lights and camera angles, they'll use *lighting doubles*: inexpensive, unimportant people who are about the same height and complexion as the expensive, important actors lounging safely in their luxurious trailers.

The crew then tests their setup with the lighting doubles, measuring the distance from the camera to the stand-in's nose, adjusting the lighting until there are no unwanted shad-

ows, and so on, while the obedient stand-in just stands there and doesn't whine or complain about "lacking motivation" for their character in this scene.

So what we're going to do in unit testing is similar to the use of lighting doubles in the movies: we'll use a cheap stand-in that is kind of close to the real thing, at least superficially, but that will be easier to work with for our purposes.

6.1 Simple Stubs

What we need to do is to stub out all those uncooperative parts of the rest of the real world and replace each of them with a more complicit ally—our own version of a "lighting double." For instance, perhaps we don't want to test against the real database, or with the real, current, wall-clock time. Let's look at a simple example.

Suppose throughout your code you call your own `getTime()` method to return the current time. It might be defined to look something like this:

```
public long getTime() {
  return System.currentTimeMillis();
}
```

(In general, we usually suggest wrapping calls to facilities outside the scope of the application to better encapsulate them—and this is a good example.) Since the concept of current time is wrapped in a method of your own writing, you can easily change it to make debugging a little easier:

```
public long getTime() {
  if (debug) {
    return debug_cur_time;
  } else {
    return System.currentTimeMillis();
  }
}
```

You might then have other debug routines to manipulate the system's idea of "current time" to cause events to happen that you'd have to wait around for otherwise.

This is one way of stubbing out the real functionality, but it's messy. First of all, it only works if the code consistently calls your own `getTime()` and does not call the Java method

`System.currentTimeMillis()` directly. What we need is a slightly cleaner—and more object-oriented—way to accomplish the same thing.

6.2 Mock Objects

Fortunately, there's a testing pattern that can help: *mock objects*. A mock object is simply a debug replacement for a real-world object. There are a number of situations that come up where mock objects can help us. Tim Mackinnon [MFC01] offers the following list:

- The real object has nondeterministic behavior (it produces unpredictable results, like a stock-market quote feed.)
- The real object is difficult to set up.
- The real object has behavior that is hard to trigger (for example, a network error).
- The real object is slow.
- The real object has (or is) a user interface.
- The test needs to ask the real object about how it was used (for example, a test might need to confirm that a callback function was actually called).
- The real object does not yet exist (a common problem when interfacing with other teams or new hardware systems).

Using mock objects, we can get around all of these problems. The three key steps to using mock objects for testing are:

1. Use an interface to describe the object
2. Implement the interface for production code
3. Implement the interface in a mock object for testing

The code under test only ever refers to the object by its interface, so it can remain blissfully ignorant as to whether it is using the real object or the mock. Let's take another look at our time example. We'll start by creating an interface for

a number of real-world environmental things, one of which is the current time:

Environmental.java
```
public interface Environmental {
  public long getTime();
  // Other methods omitted...
}
```

Next, we create the real implementation:

SystemEnvironment.java
```
public class SystemEnvironment implements Environmental {
  public long getTime() {
    return System.currentTimeMillis();
  }
  // other methods ...
}
```

And finally, the mock implementation:

MockSystemEnvironment.java
```
public class MockSystemEnvironment
  implements Environmental {
  public long getTime() {
    return current_time;
  }
  public void setTime(long aTime) {
    current_time = aTime;
  }
  private long current_time;
  // ...
}
```

Note that in the mock implementation, we've added the additional method `setTime()` (and the corresponding private variable) that allows you to control the mock object.

Now suppose we've written a new method that depends on the `getTime()` method. Some details are omitted, but the part we're interested in looks like this:

```
Line 1  import java.util.Calendar;
-
-       public class Checker {
-
5         public Checker(Environmental anEnv) {
-           env = anEnv;
-         }
-
-         /**
10        * After 5 o'clock, remind people to go home
-         * by playing a whistle
-         */
-        public void reminder() {
-          Calendar cal = Calendar.getInstance();
15        cal.setTimeInMillis(env.getTime());
-          int hour = cal.get(Calendar.HOUR_OF_DAY);
-
-          if (hour >= 17) { // 5:00PM
```

```
        env.playWavFile("quit_whistle.wav");
      }

    }

    // ...

    private Environmental env;
  }
```

Checker.java

In the production environment—the real world code that gets shipped to customers—an object of this class would be initialized by passing in a real `SystemEnvironment`. The test code, on the other hand, uses a `MockSystemEnvironment`.

The code under test that uses `env.getTime()` doesn't know the difference between a test environment and the real environment, as they both implement the same interface. You can now write tests that exploit the mock object by setting the time to known values and checking for the expected behavior.

In addition to the `getTime()` call that we've shown, the `Environmental` interface also supports a `playWavFile()` method call (used on line 19 in `Checker.java` above). With a bit of extra support code in our mock object, we can also add tests to see if the `playWavFile()` method was called without having to listen to the computer's speaker.

```
public void playWavFile(String filename) {
  playedWav = true;
}
public boolean wavWasPlayed() {
  return playedWav;
}
public void resetWav() {
  playedWav = false;
}
private boolean playedWav = false;
```

MockSystemEnvironment.java

Putting all of this together, a test using this setup would go something like this:

```
Line 1  import junit.framework.*;
        import java.util.Calendar;

        public class TestChecker extends TestCase {

          public void testQuittingTime() {

            MockSystemEnvironment env =
              new MockSystemEnvironment();
```

```
-       // Set up a target test time
-       Calendar cal = Calendar.getInstance();
-       cal.set(Calendar.YEAR, 2004);
-       cal.set(Calendar.MONTH, 10);
15      cal.set(Calendar.DAY_OF_MONTH, 1);
-       cal.set(Calendar.HOUR_OF_DAY, 16);
-       cal.set(Calendar.MINUTE, 55);
-       long t1 = cal.getTimeInMillis();
-
20      env.setTime(t1);
-
-       Checker checker = new Checker(env);
-
-       // Run the checker
25      checker.reminder();
-
-       // Nothing should have been played yet
-       assertFalse(env.wavWasPlayed());
-
30      // Advance the time by 5 minutes
-       t1 += (5 * 60 * 1000);
-       env.setTime(t1);
-
-       // Now run the checker
35      checker.reminder();
-
-       // Should have played now
-       assertTrue(env.wavWasPlayed());
-
40      // Reset the flag so we can try again
-       env.resetWav();
-
-       // Advance the time by 2 hours and check
-       t1 += (2 * 60 * 60 * 1000);
45      env.setTime(t1);
-
-       checker.reminder();
-       assertTrue(env.wavWasPlayed());
-     }
50  }
```

TestChecker.java

The code creates a mock version of the application environment at line 9. Lines 12 though 20 set up the fake time that we'll use, and then sets that in the mock environment object.

By line 25 we can run the `reminder()` call, which will (unwittingly) use the mock environment. The `assert` on line 28 makes sure that the `.wav` file has *not* been played yet, as it is not yet quitting time in the mock object environment. But we'll fix that in short order; line 32 puts the mock time exactly equal to quitting time (a good boundary condition, by the way). The `assert` on line 38 makes sure that the `.wav` file did play this time around.

Finally, we'll reset the mock environment's `.wav` file flag at line 41 and test a time two hours later. Notice how easy it is to alter and check conditions in the mock environment—you don't have to bend over and listen to the PC's speaker, or reset

the clock, or pull wires, or anything like that.

Because we've got an established interface to all system functions, people will (hopefully) be more likely to use it instead of calling methods such as `System.currentTimeMillis()` directly, and we now have control over the behavior behind that interface.

And that's all there is to mock objects: faking out parts of the real world so you can concentrate on testing your own code easily. Let's look at a more complicated example next.

6.3 Testing a Servlet

Servlets are chunks of code that a Web server manages: requests to certain URLs are forwarded to a servlet container (or manager) such as Jakarta Tomcat,[1] which in turn invokes the servlet code. The servlet then builds a response that it sends back to the requesting browser. From the end-user's perspective, it's just like accessing any other page.

The listing below shows part of the source of a trivial servlet that converts temperatures from Fahrenheit to Celsius. Let's step quickly through its operation.

```
Line 1   public void doGet(HttpServletRequest req,
                         HttpServletResponse res)
           throws ServletException, IOException
         {
           String str_f = req.getParameter("Fahrenheit");

           res.setContentType("text/html");
           PrintWriter out = res.getWriter();

           try {
             int temp_f = Integer.parseInt(str_f);
             double temp_c = (temp_f - 32)*5.0 /9.0;
             out.println("Fahrenheit: " + temp_f +
                         ", Celsius: " + temp_c);
           } catch (NumberFormatException e) {
             out.println("Invalid temperature: " + str_f);
           }
         }
```

TemperatureServlet.java

When the servlet container receives the request, it automatically invokes the servlet method `doGet()`, passing in two parameters: a request and a response.

[1] http://jakarta.apache.org/tomcat

The request parameter contains information about the request. The servlet uses this parameter to get the contents of the field Fahrenheit. It then converts the value to Celsius before writing the result back to the user. (The response object contains a factory method that returns a `PrintWriter` object, which does the actual writing.) If an error occurs converting the number (perhaps the user typed "boo!" instead of a temperature into the form's temperature field), we catch the exception and report the error in the response.

This snippet of code runs in a fairly complex environment: it needs a Web server and a servlet container, and it requires a user sitting at a browser to interact with it. This is hardly the basis of a good automated unit test. Mock objects to the rescue!

The interface to the servlet code is pretty simple: as we mentioned before, it receives two parameters, a request and a response. The request object must be able to provide a reasonable string when its `getParameter()` method is called, and the response object must support `setContentType()` and `getWriter()`.

Both `HttpServletRequest` and `HttpServletResponse` are interfaces, so all we have to do is whip up a couple of classes that implement the interfaces and we're set. Unfortunately, when we look at the interface, we discover that we'll need to implement dozens of methods just to get the code to compile—it's not as easy as the slightly contrived time/wav-file example above. Fortunately, other folks have already done the hard work for us.

Mackinnon, Freeman, and Craig [MFC01] introduced the formalization of mock objects and have also developed the code for a mock object framework for Java programmers.[2] In addition to the basic framework code that makes it easier to develop mock objects, the mock objects package comes with a number of application-level mock objects.

You'll find mock output objects (including `PrintStream`, and `PrintWriter`), objects that mock the `java.sql` library, and

[2] http://www.mockobjects.com

classes for testing in a servlet environment. In particular, it provides mocked-up versions of `HttpServletRequest` and `HttpServletResponse`, which by an incredible coincidence are the types of the parameters of the method we want to test.

We can use their package to rig the tests, much as we faked out setting the time in the earlier example:

```
Line 1   import junit.framework.*;
       - import com.mockobjects.servlet.*;
       -
       - public class TestTempServlet extends TestCase {
       5
       -   public void test_bad_parameter() throws Exception {
       -     TemperatureServlet s = new TemperatureServlet();
       -     MockHttpServletRequest request =
       -       new MockHttpServletRequest();
      10     MockHttpServletResponse response =
       -       new MockHttpServletResponse();
       -
       -     request.setupAddParameter("Fahrenheit", "boo!");
       -     response.setExpectedContentType("text/html");
      15     s.doGet(request,response);
       -     response.verify();
       -     assertEquals("Invalid temperature: boo!\n",
       -                  response.getOutputStreamContents());
       -   }
      20
       -   public void test_boil() throws Exception {
       -     TemperatureServlet s = new TemperatureServlet();
       -     MockHttpServletRequest request =
       -       new MockHttpServletRequest();
      25     MockHttpServletResponse response =
       -       new MockHttpServletResponse();
       -
       -     request.setupAddParameter("Fahrenheit", "212");
       -     response.setExpectedContentType("text/html");
      30     s.doGet(request,response);
       -     response.verify();
       -     assertEquals("Fahrenheit: 212, Celsius: 100.0\n",
       -                  response.getOutputStreamContents());
       -   }
      35
       - }
```

TestTempServlet.java

We use a `MockHttpServletRequest` object to set up the context in which to run the test. On line 13 of the code, we set the parameter Fahrenheit to the value “boo!” in the request object. This is equivalent to the user entering “boo!” in the corresponding form field in the browser; our mock object eliminates the need for human input when the test runs.

On line 14, we tell the response object that we expect the method under test to set the response’s content type to be `text/html`. Then, on lines 16 and 31, after the method under

test has run, we tell the response object to verify that this happened. Here, the mock object eliminates the need for a human to check the result visually. This example shows a pretty trivial verification; in reality, mock objects can verify that fairly complex sequences of actions have been performed, and they can check that methods have been called the correct number of times.

Mock objects can also record the data that was given to them. In our case, the response object receives the text that our servlet wants to display on the browser. We can query this value (lines 18 and 33) to check that we're returning the text we were expecting.

6.4 Easy Mock Objects

If the thought of writing all these mock object classes is intimidating, you might want to take a look at Easy-Mock,[3] a convenient Java API for creating mock objects dynamically.

Easy-Mock uses a very interesting method to specify which method calls to a mocked-out interface are allowed and what their return values should be: you specify which method calls should exist by calling them! The mock control object lets you specify a *record mode* and a *replay mode* for the corresponding mock object. While the object is in record mode, you go ahead and call the methods you are interested in and set the desired return values. Once that's finished, you switch over to replay mode. Now when you call those methods, you'll get the return values you specified.

Any remaining methods will throw a runtime exception if they are called—but the nice part is that you don't have to define any of that.

For example, suppose we have an interface for a Jukebox hardware controller that has over a dozen methods, but we're only interested in one for this demonstration. (Note that we're not really testing anything here, but are just showing how you set up and use an EasyMock object. Obviously, the whole

[3] http://www.easymock.org

point of using a mock object is to allow you to test something that depends on the object you're mocking up.)

```
Line 1   import junit.framework.*;
         import org.easymock.MockControl;

         public class TestJukebox extends TestCase {
       5
           private Jukebox     mockJukebox;
           private MockControl mockJukebox_control;

           protected void setUp() {
      10     // Create a control handle to the Mock object
             mockJukebox_control =
               MockControl.createControl(Jukebox.class);

             // And create the Mock object itself
      15     mockJukebox =
               (Jukebox) mockJukebox_control.getMock();
           }

           public void testEasyMockDemo() {
      20
             // Set up the mock object by calling
             // methods you want to exist
             mockJukebox.getCurrentSong();
             mockJukebox_control.setReturnValue(
      25                       "King Crimson -- Epitaph");

             // You don't have to worry about the other dozen
             // methods defined in Jukebox...

      30     // Switch from record to playback
             mockJukebox_control.replay();

             // Now it's ready to use:
             assertEquals("King Crimson -- Epitaph",
      35                  mockJukebox.getCurrentSong());
           }

         }
```

TestJukebox.java

The code in the `setUp()` method for this test creates an empty mock object for the `Jukebox` interface, along with its control. The control starts off in record mode, so the call on line 23 will create a mock stub for the `getCurrentSong()` call that we want to use. The next line sets the return value for that method—that's all it takes.

Now we switch the mock object from record to replay mode (line 31), and finally the call on line 35 returns the value we just set up.

There are many other options; you can specify how many times a method should return a particular value, you can verify that methods which return void were actually called, and so on.

There are also alternatives to mock objects, particularly in the servlet environment. The Jakarta Cactus system[4] is a heavier-weight framework for testing server-side components. Compared to the mock objects approach, Cactus runs your tests in the actual target environment and tends to produce less fine-grained tests. Depending on your needs, this might or might not be a good thing.

Exercises

7. Come up with a simple mock object (by hand) for an MP3 player control panel with the following methods:

Answer on 144

```
import java.util.ArrayList;
public interface Mp3Player {
  /**
   * Begin playing the filename at the top of the
   * play list, or do nothing if playlist
   * is empty.
   */
  public void play();
  /**
   * Pause playing. Play will resume at this spot.
   */
  public void pause();
  /**
   * Stop playing. The current song remains at the
   * top of the playlist, but rewinds to the
   * beginning of the song.
   */
  public void stop();
  /** Returns the number of seconds into
   * the current song.
   */
  public double currentPosition();
  /**
   * Returns the currently playing file name.
   */
  public String currentSong();
  /**
   * Advance to the next song in the playlist
   * and begin playing it.
   */
  public void next();
  /**
   * Go back to the previous song in the playlist
   * and begin playing it.
   */
  public void prev();
```

[4] http://jakarta.apache.org/cactus

```
  /**
   * Returns true if a song is currently
   * being played.
   */
  public boolean isPlaying();

  /**
   * Load filenames into the playlist.
   */
  public void loadSongs(ArrayList names);
}
```

Mp3Player.java

It should pass the following unit test:

```
import junit.framework.*;
import java.util.ArrayList;

public class TestMp3Player extends TestCase {

  protected Mp3Player mp3;
  protected ArrayList list = new ArrayList();

  public void setUp() {
    mp3 = new MockMp3Player();

    list = new ArrayList();
    list.add("Bill Chase -- Open Up Wide");
    list.add("Jethro Tull -- Locomotive Breath");
    list.add("The Boomtown Rats -- Monday");
    list.add("Carl Orff -- O Fortuna");
  }

  public void testPlay() {

    mp3.loadSongs(list);
    assertFalse(mp3.isPlaying());
    mp3.play();
    assertTrue(mp3.isPlaying());
    assertTrue(mp3.currentPosition() != 0.0);
    mp3.pause();
    assertTrue(mp3.currentPosition() != 0.0);
    mp3.stop();
    assertEquals(0.0, mp3.currentPosition(), 0.1);

  }

  public void testPlayNoList() {

    // Don't set the list up
    assertFalse(mp3.isPlaying());
    mp3.play();
    assertFalse(mp3.isPlaying());
    assertEquals(0.0, mp3.currentPosition(), 0.1);
    mp3.pause();
    assertEquals(0.0, mp3.currentPosition(), 0.1);
    assertFalse(mp3.isPlaying());
    mp3.stop();
    assertEquals(0.0, mp3.currentPosition(), 0.1);
    assertFalse(mp3.isPlaying());
  }

  public void testAdvance() {

    mp3.loadSongs(list);

    mp3.play();

    assertTrue(mp3.isPlaying());
```

```
        mp3.prev();
        assertEquals(list.get(0), mp3.currentSong());
        assertTrue(mp3.isPlaying());
        mp3.next();
        assertEquals(list.get(1), mp3.currentSong());
        mp3.next();
        assertEquals(list.get(2), mp3.currentSong());
        mp3.prev();
        assertEquals(list.get(1), mp3.currentSong());
        mp3.next();
        assertEquals(list.get(2), mp3.currentSong());
        mp3.next();
        assertEquals(list.get(3), mp3.currentSong());
        mp3.next();
        assertEquals(list.get(3), mp3.currentSong());
        assertTrue(mp3.isPlaying());
    }
}
```

TestMp3Player.java

Chapter 7

Properties of Good Tests

Unit tests are very powerful magic, and if used badly can cause an enormous amount of damage to a project by wasting your time. If unit tests aren't written and implemented properly, you can easily waste so much time maintaining and debugging the tests themselves that the production code—and the whole project—suffers.

We can't let that happen; remember, the whole reason you're doing unit testing in the first place is to make your life easier! Fortunately, there are only a few simple guidelines that you need to follow to keep trouble from brewing on your project.

Good tests have the following properties, which makes them A-TRIP:

- **A**utomatic
- **T**horough
- **R**epeatable
- **I**ndependent
- **P**rofessional

Let's look at what each of these words means to us.

7.1 Automatic

A-TRIP

Unit tests need to be run automatically. We mean "automatically" in at least two ways: invoking the tests and checking the results.

It must be really easy for you to invoke one or more unit tests, as you will be doing it all day long, day in and day out. So it really can't be any more complicated than pressing one button in the IDE or typing in one command at the prompt in order to run the tests you want. Some IDEs can even be set up to run the unit tests continually in the background.

It's important to maintain this environment: don't introduce a test that breaks the automatic model by requiring manual steps. Whatever the test requires (database, network connections, etc.), make these an automatic part of the test itself. Mock objects, as described in Chapter 6, can help insulate you from changes in the real environment.

But you're not the only one running tests. Somewhere a machine should be running all of the unit tests for all checked-in code continuously. This automatic, unattended check acts as a "back stop"; a safety mechanism to ensure that whatever is checked in hasn't broken any tests, anywhere. In an ideal world, this wouldn't be necessary as you could count on every individual developer to run all the necessary tests themselves.

But this isn't an ideal world. Maybe an individual didn't run some necessary test in a remote corner of the project. Perhaps they have some code on their own machine that makes it all work—but they haven't checked that code in, so even though the tests work on their own machine, those same tests fail everywhere else.

You may want to investigate systems such as Cruise Control[1], AntHill[2], or Tinderbox.[3] These are freely available, open source products that manage continuous building and testing. We describe setting up a continuous-build and testing environment in Volume III of the Starter Kit [Cla04].

[1] http://cruisecontrol.sourceforge.net
[2] http://www.cs.unibo.it/projects/anthill
[3] http://www.mozilla.org/tinderbox.html

Finally, by "automatic" we mean that the test must determine for itself whether it passed or failed. Having a person (you or some other hapless victim) read through the test output and determine whether the code is working or not is a recipe for project failure. It's an important feature of consistent regression to have the tests check for themselves. We humans aren't very good at those repetitive tasks, and besides we've got more important things to do—remember the project?

This idea of having the tests run by themselves and check themselves is critical, because it means that you don't have to *think* about it—it just happens as part of the project. Testing can then fulfill its role as a major component of our project's safety net. (Version control and automation are the other two major components of the "safety net.") Tests are there to catch you when you fall, but they're not in your way. You'll need all of your concentration as you cross today's high-wire.

7.2 Thorough

A-T RIP

Good unit tests are thorough; they test everything that's likely to break. But just how thorough? At one extreme, you can aim to test every line of code, every possible branch the code might take, every exception it throws, and so on. At the other extreme, you test just the most likely candidates—boundary conditions, missing and malformed data, and so on. It's a question of judgment, based on the needs of your project.

If you want to aim for more complete coverage, then you may want to invest in code coverage tools to help. (See, for instance, the freely available tools "nounit"[4] and "quilt."[5] Commercial tools, such as Clover, are available as well.)

These tools can help you determine how much of the code under test is actually being exercised.

It's important to realize that bugs are not evenly distributed throughout the source code. Instead, they tend to clump together in problematic areas (for an interesting story along these lines, see the sidebar on the following page).

[4] http://nounit.sourceforge.net
[5] http://quilt.sourceforge.net

Reported Bugs vs. Unit Test Coverage

We had a client recently that didn't quite believe in the power of unit tests. A few members of the team were very good and disciplined at writing unit tests for their own modules, many were somewhat sporadic about it, and a few refused to be bothered with unit tests at all.

As part of the hourly build process, we whipped up a simple Ruby script that performed a quick-and-dirty analysis of test coverage: it tallied up the ratio of test code asserts to production code methods for each module. Well-tested methods may have 3, 4, or more asserts each; untested methods will have none at all. This analysis ran with every build and produced a bar-graph, ranking the most-tested modules at the top and the untested modules at the bottom.

After a few weeks of gathering figures, we showed the bargraph to the project manager, without initial explanation. He was very surprised to see all of the "problem modules" lumped together at the bottom—he thought we had somehow produced this graph based on bug reports from QA and customer support. Indeed, the modules at the top of the graph (well tested) were nearly unknown to him; very few, if any, problems had ever been reported against them. But the clump of modules at the bottom (that had no unit tests) were *very* well known to him, the support managers, and the local drugstore which had resorted to stocking extra-large supplies of antacid.

The results were very nearly linear: the more unit-tested the code, the fewer problems.

This phenomenon leads to the well-known battle cry of "don't patch it, rewrite it." Often, it can be much cheaper and less painful to throw out a piece of code that has a clump of bugs and rewrite it from scratch. And of course, it's much safer to rewrite code from scratch now: you'll have a set of unit tests that can confirm the new code works as it should.

7.3 Repeatable

A-T[R]IP

Just as every test should be independent from every other test, they must be independent of the environment as well. The goal remains that every test should be able to run over and over again, in any order, *and produce the same results*. This means that tests cannot rely on anything in the external environment that isn't under your direct control.

Use mock objects as necessary to isolate the item under test and keep it independent from the environment. If you are forced to use some element from the real world (a database, perhaps), make sure that you won't get interference from any other developer. Each developer needs their own "sandbox" to play in, whether that's their own database instance within Oracle, or their own webserver on some non-standard port.

Without repeatability, you might be in for some surprises at the worst possible moments. What's worse, these sort of surprises are usually bogus—it's not really a bug, it's just a problem with the test. You can't afford to waste time chasing down phantom problems.

Each test should produce the same results every time. If it doesn't, then that should tell you that there's a *real* bug in the code.

7.4 Independent

A-TR[I]P

Tests need to be kept neat and tidy, which means keeping them tightly focused, and independent from the environment and each other (remember, other developers may be running these same tests at the same time).

When writing tests, make sure that you are only testing one thing at a time.

Now that doesn't mean that you use only one assert in a test, but that one test method should concentrate on a single production method, or a small set of production methods that, together, provide some feature.

Sometimes an entire test method might only test one small aspect of a complex production method—you may need multiple test methods to exercise the one production method fully.

Ideally, you'd like to be able to have a traceable correspondence between potential bugs and test code. In other words, when a test fails, it should be obvious where in the code the underlying bug exists.

Independent also means that no test relies on any other test; you should be able to run any individual test at any time, and in any order. You don't want to have to rely on any other test having run first.

We've shown mechanisms to help you do this: the per-test setup and teardown methods and the per-class setup and teardown methods. Use these methods to ensure that every test gets a fresh start—and doesn't impact any test that might run next.

Remember, you aren't guaranteed that JUnit tests will run in any particular order, and as you start combining tests and suites in ever-increasing numbers, you really can't afford to carry ordering dependencies along with you.

John Donne may have been right about people, but not about unit tests: every test *should be* an island.

7.5 Professional

A-TRI P

The code you write for a unit test is real; some may argue it's even more real than the code you ship to customers. This means that it must be written and maintained to the same professional standards as your production code. All the usual rules of good design—maintaining encapsulation, honoring the DRY principle, lowering coupling, etc.—must be followed in test code just as in production code.

It's easy to fall into the trap of writing very linear test code; that is, code that just plods along doing the same thing over and over again, using the same lines of code over and over again, with nary a function or object in sight. That's a bad thing. Test code must be written in the same manner as real code. That means you need to pull out common, repeated bits

of code and put that functionality in a method instead, so it can be called from several different places.

You may find you accumulate several related test methods that should be encapsulated in a class. Don't fight it! Go ahead and create a new class, even if it's only ever used for testing. That's not only okay, it's encouraged: test code is real code. In some cases, you may even need to create a larger framework, or create a data-driven testing facility (remember the simple file reader for `TestLargest` on page 39?).

Don't waste time testing aspects that won't help you. Remember, you don't want to create tests just for the sake of creating tests. Test code must be thorough in that it must test everything interesting about a method that is likely to contain a bug. If it's not likely to contain a bug, don't bother testing it. That means that usually you shouldn't waste time testing things like simple accessors:

```
public Money getBalance() {
  return balance;
}
```

Frankly, there's just not much here to go wrong that the compiler can't catch. Testing methods such as these is just a waste of time. However, if the accessor is doing some work along the way, then suddenly it becomes interesting—and we will want to test it:

```
public Money getBalance() {
  return posted.getBalance() -
         unposted.getDebits() +
         unposted.getCredits();
}
```

That's probably worth testing.

Finally, expect that there will be at least as much test code written as there will be production code. Yup, you read that right. If you've got 20,000 lines of code in your product, then it would be reasonable to expect that there would be 20,000 lines or more of unit test code to exercise it. That's a lot of test code, which is partly why it needs to be kept neat and tidy, well designed and well-factored, just as professional as the production code.

7.6 Testing the Tests

There is one major conceptual weakness in our plans so far. Testing code to make sure it works is a great idea, but you have to write code to perform the tests. What happens when there are bugs in our test code? Does that mean you have to write test code to test the tests that test the code??? Where will it all end?

Fortunately, you don't need to go to that extreme. There are two things you can do to help ensure that the test code is correct:

- Improve tests when fixing bugs
- Prove tests by introducing bugs

How to Fix a Bug

The steps you take when fixing a bug are very important to unit testing. Many times, an existing test will expose a bug in the code, and you can then simply fix the code and watch the vigilant test pass.

When a bug is found "in the wild" and reported back, that means there's a hole in the net—a missing test. This is your opportunity to close the hole, and make sure that this bug never escapes again. All it takes is four simple steps:

1. Identify the bug.
2. Write a test that fails, to prove the bug exists.
3. Fix the code such that the test now passes.
4. Verify that *all* tests still pass (i.e., you didn't break anything else as a result of the fix).

This simple mechanism of applying real-world feedback to help improve the tests is very effective. Over time, you can expect that your test coverage will steadily increase, and the number of bugs that escape into the wild from existing code will decrease.

Of course, as you write new code, you'll undoubtedly introduce new bugs, and new classes of bugs, that aren't being

```
public void testAdd() {
  // Create a new account object
  Account acct = new Account();
  // Populate with our test person
  acct.setPerson(TEST_PERSON_1);
  // Add it to the database
  DatabaseHandler.add(acct);
  // Should find it
  assertTrue(DatabaseHandler.search(TEST_PERSON_1);
}
```

Figure 7.1: Test Adding a Person to a Database

caught by the tests. But when fixing any bug, ask yourself the key question:

> ***Could this same kind of problem happen anywhere else?***

Then it doesn't matter whether you're fixing a bug in an older feature or a new feature; either way, apply what you've just learned to the *whole* project. Encode your new-found knowledge in all the unit tests that are appropriate, and you've done more than just fix one bug. You've caught a whole class of bugs.

Spring the Trap

If you're not sure that a test is written correctly, the easiest thing to do is to "spring the trap": cause the production code to exhibit the very bug you're trying to detect, and verify that the test fails as expected.

For instance, suppose you've got a test method that adds a customer account to the database and then tries to find it, something like the code in Figure 7.1. Perhaps you're not certain that the "finding" part is really working or not—it might be reporting success even if the record wasn't added correctly.

So maybe you'll go into the `add()` method for `DatabaseHandler` and short-circuit it: just return instead of actually adding the record to the database. Now you should see the assertion fail, because the record has not been added.

But wait, you may cry, what about a leftover record from a previous test run? Won't that be in the database? No, it won't, for several reasons:

- You may not really be testing against a live database. The code exercised by the above test case lies between the add method shown and the actual low-level database calls. Those database calls may well be handled by a mock object, whose data is not held persistently in between runs.
- Tests are independent. All tests can be run in any order, and do not depend on each other, so even if a real database is part of this test, the setup and tear-down must ensure that you get a "clean sandbox" to play in. The attempt above to spring the trap can help prove that this is true.

Now the Extreme Programming folks claim that their disciplined practice of test-first development avoids the problem of poor tests that don't fail when they should. In test-first development, you only ever write code to fix a failing test. As soon as the test passes, then you know that the code you just added fixed it. This puts you in the position where you always know with absolute certainty that the code you introduced fixes the failing test that caused you to write the code in the first place.

But there's many a slip 'twixt the cup and the lip, and while test-first development does improve the situation dramatically, there will still be opportunities to be misled by coincidences. For those occasions, you can satisfy any lingering doubts by deliberately "springing the trap" to make sure that all is as you expect.

Finally, remember to write tests that are A-TRIP (Automatic, Thorough, Repeatable, Independent, Professional); keep adding to your unit tests as new bugs and types of bugs are discovered; and check to make sure your tests really do find the bugs they target.

Then sit back and watch problems on your project disappear like magic.

Chapter 8

Testing on a Project

Up to now we've talked about testing as an individual, solitary exercise. But of course, in the real world you'll likely have teammates to work with. You'll all be unit testing together, and that brings up a couple of issues.

8.1 Where to Put Test Code

On a small, one-person project, the location of test code and encapsulation of the production code may not be very important, but on larger projects it can become a critical issue. There are several different ways of structuring your production and test code that we'll look at here.

In general, you don't want to break any encapsulation for the sake of testing (or as Mom used to say, "don't expose your privates!"). Most of the time, you should be able to test a class by exercising its public methods. If there is significant functionality that is hidden behind private or protected access, that might be a warning sign that there's another class in there struggling to get out. When push comes to shove, however, it's probably better to break encapsulation with working, tested code than it is to have good encapsulation of untested, non-working code.

In Java, classes in the same package can have access to each others' *protected* member variables and methods; you

can exploit this feature to allow some non-public access using some of the schemes that follow.

Same directory

Suppose you are writing a class named:

```
com.pragprog.wibble.Account
```

with a corresponding test in:

```
com.pragprog.wibble.TestAccount
```

The first and easiest method of structuring test code is to simply include it right in the same directory alongside the production code.

```
com/
└── pragprog/
    └── wibble/
        ├── Account.java
        └── TestAccount.java
```

This has the advantage that `TestAccount` can access protected member variables and methods of `Account`. But the disadvantage is that the test code is lying around, cluttering up the production code directory. This may or may not be a problem depending on your IDE/compiler and your method of creating a release to ship to customers.

Most of the time, it's enough of a problem that we prefer one of the other solutions. But for small projects, this might be sufficient.

Subdirectories

The next option is to create `test` subdirectories under every production directory.

```
com/
└── pragprog/
    └── wibble/
        ├── Account.java
        └── test/
            └── TestAccount.java
```

This has the advantage of moving the test code a little bit out of the way—not too far, but at least not in the same directory as the production code. The disadvantage is that now the test code is in a different package: `TestAccount` is now in `com.pragprog.wibble.test.TestAccount`. You won't be able to access protected members unless your test code uses a subclass of the production code that exposes the necessary members. For instance, suppose the class you want to test looks like this:

```
package com.acme;
public class Pool {
  protected Date lastCleaned;
  public void xxxx xx {
    xxx xxx xxxx;
  }
  ...
}
```

You need to get at that non-public bit of data that tells you when the pool was last cleaned for testing, but there's no accessor for it. (If there were, the pool association would probably sue us; they don't like to make that information public.) So you make a subclass in the `test` subpackage that exposes it just for testing.

```
package com.acme.test;
import com.acme.Pool;
public class PoolForTesting extends Pool {
  public Date getLastCleaned() {
    return lastCleaned;
  }
  ...
}
```

You then use `PoolForTesting` in the test code instead of using `Pool` directly (see Figure 8.1 on the next page). In fact, you could make this special class private to the test code (to ensure that we don't get sued).

Parallel Trees

Another option is to place your Test classes into the same package as your production code, but in a different source code tree. The trick is to ensure that the root of both trees is in the compiler's `CLASSPATH`. In this case, both `prod` and `test` should be in the `CLASSPATH`.

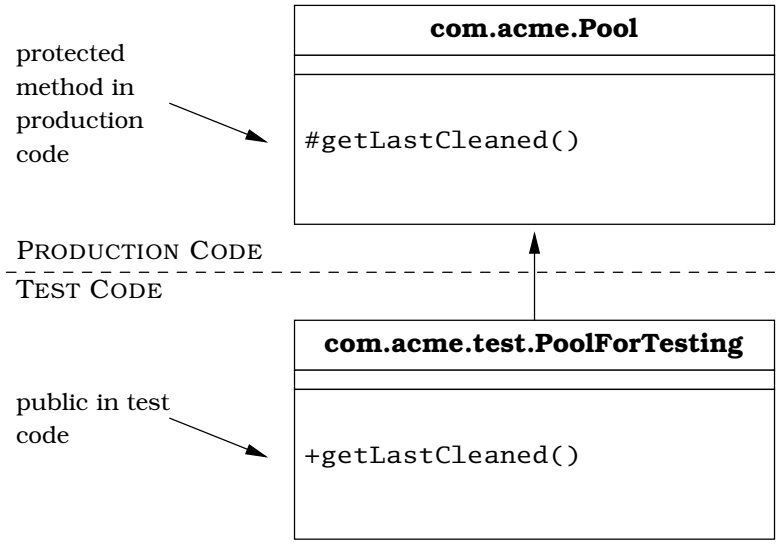

Figure 8.1: Subclasses Expose Methods for Testing

```
prod/
 └── com/
      └── pragprog/
           └── wibble/
                └── Account.java
test/
 └── com/
      └── pragprog/
           └── wibble/
                └── TestAccount.java
```

Now the test code is *really* out of the way—maybe too far out of the way to be convenient depending on your setup. But it certainly isn't lying around near the production code anymore, and because the test code is in the same package, it again enjoys preferential access.

Whatever convention the team decides to adopt, make sure it does so consistently. You cannot have some of the tests in the system set up one way, and other tests elsewhere set up a different way. Pick a style that looks like it will work in your environment and stick with it for all of the system's unit tests.

8.2 Test Courtesy

The biggest difference between testing by yourself and testing with others lies in synchronizing working tests and code.

When working with other members of a team, you will be using some sort of version control system, such as CVS. (If you aren't familiar with version control, or would like some assistance in getting it set up and working correctly, please see [TH03].)

In a team environment (and even in a personal environment) you should make sure that when you check in code (or otherwise make it available to everyone) that it has complete unit tests, and that it passes all of them. In fact, every test in the whole system should continue to pass with your new code.

The rule is very simple: As soon as anyone else can access your code, all tests everywhere need to pass. Since you should normally work in fairly close synchronization with the rest of the team and the version control system, this boils down to **"all tests pass all the time."**

Many teams institute policies to help "remind" developers of the consequences of breaking the build, or breaking the tests. These policies might begin by listing potential infractions involving code that you have checked in (or otherwise made available to other developers):

- Incomplete code (e.g., checking in only one class file but forgetting to check in other files it may depend upon).
- Code that doesn't compile.
- Code that compiles, but breaks existing code such that existing code no longer compiles.
- Code without corresponding unit tests.
- Code with failing unit tests.
- Code that passes its own tests, but causes other tests elsewhere in the system to fail.

If found guilty of any of these heinous crimes, you may be sentenced to providing donuts for the entire team the next morning, or beer or soda, or frozen margaritas, or maybe you'll have

to nursemaid the build machine, or some other token, menial task.

A little lighthearted law enforcement usually provides enough motivation against careless accidents. But what happens if you have to make an incompatible change to the code, or if you make a change that *does* cause other tests to fail elsewhere in the system?

The precise answer depends on the methodology and process you're using on the project, but somehow you need to coordinate your changes with the folks who are responsible for the other pieces of code—which may well be you! The idea is to make all of the necessary changes at once, so the rest of the team sees a coherent picture (that actually works) instead of a fragmented, non-functional "work in progress." (For more information on how to use version control to set up experimental developer branches, see [TH03].)

Sometimes the real world is not so willing, and it might take a few hours or even a few days to work out all of the incompatible bits and pieces, during which time the build is broken. If it can't be helped, then make sure that it is well-communicated. Make sure everyone knows that the build will be broken for the requisite amount of time so that everyone can plan around it as needed. If you're not involved, maybe it would be a good time to take your car in for an oil change or slip off to the beach for a day or two. If you are involved, get it done quickly so everyone else can come back from the beach and get to work!

8.3 Test Frequency

How often should you run unit tests? It depends on what you're doing, and your personal habits, but here are some general guidelines that we find helpful. You want to perform enough testing to make sure you're catching everything you need to catch, but not so much testing that it interferes with producing production code.

Write a new method

Compile and run local unit tests.

Fix a bug
Run tests to demonstrate bug; fix and re-run unit tests.

Any successful compile
Run local unit tests.

Each check-in to version control
Run all module or system unit tests.

Continuously
A dedicated machine should be running a full build and test, from scratch, automatically throughout the day (either periodically or on check-in to version control).

Note that for larger projects, you might not be able to compile and test the whole system in under a few hours. You may only be able to run a full build and test overnight. For even larger projects, it may have to be every couple of days—and that's a shame, because the longer the time between automatic builds the longer the "feedback gap" between creation of a problem and its identification.

The reason to have a more-or-less continuous build is so that it can identify any problems quickly. You don't want to have to wait for another developer to stumble upon a build problem if you can help it. Having a build machine act as a constant developer increases the odds that *it* will find a problem, instead of a real developer.

When the build machine does find a problem, then the whole team can be alerted to the fact that it's not safe to get any new code just yet, and can continue working with what they have. That's better than getting stuck in a situation where you've gotten fresh code that doesn't work.

For more information on setting up automatic build and testing systems, nightly and continuous builds, and automation in general please see [Cla04].

8.4 Tests and Legacy Code

So far, we've talked about performing unit tests in the context of new code. But we haven't said what to do if your project has a lot of code already—code that *doesn't* have unit tests.

It all depends on what kind of state that code is in. If it's reasonably well-factored and modular, such that you can get at all of the individual pieces you need to, then you can add unit tests fairly easily. If, on the other hand, it's just a "big ball of mud" all tangled together, then it might be close to impossible to test without substantial rewriting. Most older projects aren't perfectly factored, but are usually modular enough that you can add unit tests.

For new code that you write, you'll obviously write unit tests as well. This may mean that you'll have to expose or break out parts of the existing system, or create mock objects in order to test your new functionality.

For existing code, you might choose to methodically add unit tests for everything that is testable. But that's not very pragmatic. It's better to add tests for the most broken stuff first, to realize a better return on investment of effort.

The most important aspect of unit tests in this environment is to prevent back-sliding: to avoid the death-spiral where maintenance fixes and enhancements cause bugs in existing features. We use JUnit unit tests as *regression* tests during normal new code development (to make sure new code doesn't break anything that had been working), but regression testing is even more important when dealing with legacy code.

And it doesn't have to cover the entire legacy code base, just the painful parts. Consider the following true story from a pragmatic developer:

Regression Tests Save the Day

"Tibbert Enterprises[1] ships multiple applications, all of which are based on a common Lower Level Library that is used to access the object database.

One day I overheard some application developers talking about a persistent problem they were having. In the product's Lower Level interface, you can look up objects using the object name, which includes a path to the object. Since the application

[1] Not their real name.

has several layers between it and the Lower Level code, and the Lower Level code has several more layers to reach the object database, it takes a while to isolate a problem when the application breaks.

And the application broke. After half the application team spent an entire day tracking down the bug, they discovered the bug was in the Lower Level code that accessed the database. If you had a space in the name, the application died a violent, messy death. After isolating the Lower Level code related to the database access, they presented the bug to the owner of the code, along with a fix. He thanked them, incorporated their fix, and committed the fixed code into the repository.

But the next day, the application died. Once again, a team of application developers tracked it down. It took only a half-a-day this time (as they recognized the code paths by now), and the bug was in the same place. This time, it was a space in the path to the object that was failing, instead of a space in the name itself. Apparently, while integrating the fix, the developer had introduced a new bug. Once again, they tracked it down and presented him with a fix. It's Day Three, and the application is failing again! Apparently the developer in question re-introduced the original bug.

The application manager and I sat down and figured out that the equivalent of nearly two man-months of effort had been spent on this one issue over the course of one week by his team alone (and this likely affected other teams throughout the company). We then developed JUnit tests that tested the Lower Level API calls that the application product was using, and added tests for database access using spaces in both the object name and in the path. We put the product under the control of our continuous-build-and-test program (using CruiseControl) so that the unit tests were run automatically every time code got committed back to the repository.

Sure enough, the following week, the test failed

> on two successive days, at the hands of the original developer. He actually came to my office, shook my hand, and thanked me when he got the automatic notification that the tests had failed.
>
> You see, without the JUnit test, the bad code made it out to the entire company during the nightly builds. But with our continuous build and test, he (and his manager and tester) saw the failure at once, and he was able to fix it immediately before anyone else in the company used the code. In fact, this test has failed half a dozen times since then. But it gets caught, so its not a big deal anymore. The product is now stable because of these tests.
>
> We now have a rule that any issue that pops up twice must have a JUnit test by the end of the week."

In this story, Tibbert Enterprises aren't using JUnit to prove things work so much as they are using it to inoculate against known issues. As they slowly catch up, they'll eventually expand to cover the entire product with unit tests, not just the most broken parts.

When you come into a shop with no automated tests of any kind, this seems to be a very effective approach. Remember, the only way to eat an elephant is one bite at a time.

8.5 Tests and Reviews

Teams that enjoy success often hold code reviews. This can be an informal affair where a senior person just gives a quick look at the code. Or perhaps two people are working on the code together, using Extreme Programming's "Pair Programming" practice. Or maybe it's a very formal affair with checklists and a small committee.

However you perform code reviews (and we suggest that you do), make the test code an integral part of the review process. Since test code is held up to the same high standards as production code, it should be reviewed as well.

In fact, it can sometimes be helpful to expand on the idea of "test-first design" to include both writing and *reviewing* test

code before writing production code. That is, code and review in this order:

1. Write test cases and/or test code.
2. Review test cases and/or test code.
3. Revise test cases and/or test code per review.
4. Write production code that passes the tests.
5. Review production and test code.
6. Revise test and production code per review.

Reviews of the test code are incredibly useful. Not only are reviews more effective than testing at finding bugs in the first place, but by having everyone involved in reviews you can improve team communication. People on the team get to see how others do testing, see what the team's conventions are, and help keep everyone honest.

You can use the checklists on page 135 of this book to help identify possible test cases in reviews. But don't go overboard testing things that aren't likely to break, or repeat essentially similar tests over and over just for the sake of testing.

Finally, you may want to keep track of common problems that come up again and again. These might be areas where more training might be needed, or perhaps something else that should be added to your standard review checklist.

For example, at a client's site several years ago, we discovered that many of the developers misunderstood exception handling. The code base was full of fragments similar to the following:

```
try {
   DatabaseConnection dbc = new DatabaseConnection();
   insertNewRecord(dbc, record);
   dbc.close();
} catch (Exception e) {}
```

That is to say, they simply ignored any exceptions that might have occurred. Not only did this result in random missing records, but the system leaked database connections as well—any error that came up would cause the `close` to be skipped.

We added this to the list of known, typical problems to be checked during reviews. As code was reviewed, any of these infamous `catch` statements that were discovered were first identified, then proper unit tests were put in place to force various error conditions (the "E" in RIGHT-BICEP), and the code was fixed to either propagate or handle the exception. System stability increased tremendously as a result of this simple process.

Chapter 9

Design Issues

So far we have discussed unit testing as it helps you to understand and verify the functional, operational characteristics of your code. But unit testing offers several opportunities to improve the design and architecture of your code as well.

In this chapter, we'll take a look at the following design-level issues:

- Better separation of concerns by designing for testability
- Clarifying design by defining class invariants
- Improving interfaces with test-driven design
- Establishing and localizing validation responsibilities

9.1 Designing for Testability

"Separation of Concerns" is probably the single most important concept in software design and implementation. It's the catch-all phrase that encompasses encapsulation, orthogonality, coupling, and all those other computer science terms that boil down to "write shy code" [HT00].

You can keep your code well-factored (i.e., "shy") and easier to maintain by explicitly designing code to be testable. For example, suppose you are writing a method that will sleep until the top of the next hour. You've got a bunch of calculations and then a `sleep()`:

```
public void sleepUntilNextHour()
            throws InterruptedException {
  int howlong;
  // Calculate how long to wait...
  Thread.sleep(howlong);
  return;
}
```

How will you test that? Wait around for an hour? Set a timer, call the method, wait for the method to return, check the timer, handle the cases when the method doesn't get called when it should—this is starting to get pretty messy.

Perhaps you might refactor this method, just to make testing easier. Instead of combining the calculation of how many milliseconds to sleep with the `sleep()` method itself, split them up:

```
public void sleepUntilNextHour()
            throws InterruptedException {
  int howlong = milliSecondsToNextHour(new Date());
  Thread.sleep(howlong);
  return;
}
```

What's likely to break? The system's `sleep` call? Or our code that calculates the amount of time to wait? It's probably a fair bet to say that Java's `Thread.sleep()` works as advertised (even if it doesn't, our rule is to always suspect our own code first). So for now, you only need to test that the number of milliseconds is calculated correctly, and what might have been a hairy test with timers and all sorts of logic (not to mention an hour's wait) can be expressed very simply as:

```
assertEquals(10000,
             milliSecondsToNextHour(TEST_DATE_10));
```

If we're confident that `milliSecondsToNextHour()` works to our satisfaction, then the odds are that `sleepUntilNextHour()` will be reliable as well—if it is not, then at least we know that the problem must be related to the sleep itself, and not to the numerical calculation. You might even be able to reuse the `milliSecondsToNextHour()` method in some other context.

This is what we mean when we claim that you can improve the design of code by making it easier to test. By changing

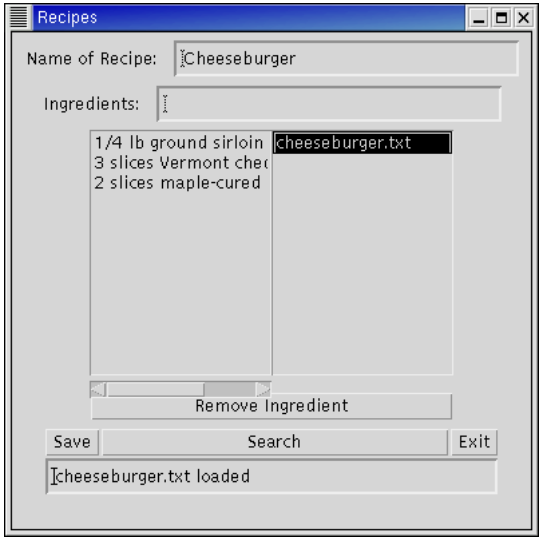

Figure 9.1: Recipes GUI Screen

code so that you can get in there and test it, you'll end up with a cleaner design that's easier to extend and maintain as well as test.

But instead of boring you with examples and techniques, all you really need to do is remember this one fundamental question when writing code:

How am I going to test this?

If the answer is not obvious, or if it looks like the test would be ugly or hard to write, then take that as a warning signal. Your design probably needs to be modified; change things around until the code is easy to test, and your design will end up being far better for the effort.

9.2 Refactoring for Testing

Let's look at a real-life example. Here are excerpts from a novice's first attempt at a recipe management system. The GUI, shown in Figure 9.1, is pretty straightforward. There's only one class, with GUI behavior and file I/O intermixed.

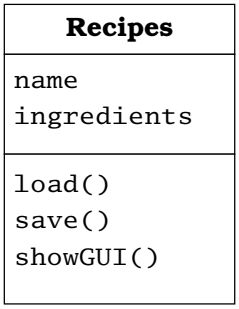

Figure 9.2: Original `Recipes` Static Class Diagram

It reads and writes individual recipes to files, using a line-oriented format, somewhat like an INI or properties file:

cheeseburger.fxt

```
NAME=Cheeseburger
INGREDIENTS=3
1/4 lb ground sirloin
3 slices Vermont cheddar cheese
2 slices maple-cured bacon
```

And here's the code, in its entirety. As is, this is pretty hard to test. You've got to run the whole program and operate the GUI to get at any part of it. All of the file I/O and search routines access the widgets directly, and so are tightly coupled to the GUI code (see, for instance, lines 35, 44, 49, and 60). In fact, the UML diagram for this class, shown in Figure 9.2, is kind of embarrassing—it's just one big class!

```
Line 1   import java.awt.*;
       - import java.awt.event.ActionListener;
       - import java.awt.event.ActionEvent;
       - import java.io.*;
     5 import java.util.ArrayList;
       -
       - class Recipes extends Frame {
       -
       -   private Label titleLabel =
    10     new Label("Name of Recipe:", Label.LEFT);
       -   private TextField titleText = new TextField(30);
       -
       -   private Label ingredientsLabel =
       -     new Label("Ingredients:",Label.LEFT);
    15   private TextField ingredientsText =
       -     new TextField(30);
       -
       -   private Button ingredientsRemoveSel =
       -     new Button("Remove Ingredient");
    20   private List ingredientsList = new List(12, false);
       -   private List searchList = new List(12, false);
       -
       -   private Button saveButton = new Button("Save");
```

```
    private Button searchButton = new Button("Search");
25  private Button exitButton = new Button("Exit");
    private TextField statusText = new TextField(40);

    public void loadFile() {
      statusText.setText(searchList.getSelectedItem());
30
      try {
        BufferedReader in = null;
        String thePath;
        thePath = "recipes/" +
35        searchList.getSelectedItem();
        in = new BufferedReader(
                                  new FileReader(thePath));
        String line;
        while ((line = in.readLine()) != null){
40        int pos = line.indexOf('=');
          String token = line.substring(0, pos);
          String value = line.substring(pos+1);
          if (token.equals("NAME")) {
            titleText.setText(value);
45        } else if (token.equals("INGREDIENTS")) {
            int num_lines = Integer.parseInt(value);
            for (int i=0; i < num_lines; i++) {
              line = in.readLine();
              ingredientsList.add(line);
50          }
          }
        }
        in.close();
      } catch (IOException e) {
55      System.err.println(e);
        statusText.setText(searchList.getSelectedItem()+
                           " is corrupt");
        return;
      }
60    statusText.setText(searchList.getSelectedItem()+
                         " loaded");
    }

    public void removeSel() {
65    int count;
      String str;
      str = ingredientsList.getSelectedItem();
      if (str != null) {
        count = ingredientsList.getSelectedIndex();
70      ingredientsList.remove(count);
      }
    }

    public void addIngredient() {
75    String str;
      str = ingredientsText.getText();

      if (str.length() != 0) {
        ingredientsList.add(str);
80      ingredientsText.setText("");
      }
```

```
    }

    public void exit() {
      dispose();
      System.exit(0);
    }


    public void saveFile(File theFile) {

      try {
        PrintWriter out = new PrintWriter(
                                            new FileWriter(theFile));

        out.println("NAME=" + titleText.getText());

        int counter = ingredientsList.getItemCount();

        out.println("INGREDIENTS=" + counter);
        for (int i = 0; i < counter; i +=1){
          out.println(ingredientsList.getItem(i));
        }

        out.close();
        statusText.setText("Saved "+
                           titleText.getText());
      } catch (IOException e) {
        System.err.println(e);
      }
    }

    public void save() {
      String str = statusText.getText();
      File theFile = new File("recipes/" +
                              titleText.getText() +
                              ".txt");
      if (str.equals(titleText.getText() +
                     ".txt Already exists")) {
        saveFile(theFile);
      } else {
        if (theFile.exists() == true) {
          statusText.setText(titleText.getText() +
                             ".txt Already exists");
        }
        if (theFile.exists() == false) {
          saveFile(theFile);
        }
      }
    }

    public void search() {
      String str;
      String[] dir_list;
      ArrayList matches = new ArrayList();

      str = titleText.getText();

      searchList.removeAll();
      statusText.setText("Partial match: "+
                         titleText.getText());

      File path = new File("recipes/");
```

```
      dir_list = path.list();

      for (int i=0; i < dir_list.length; i++){
        String file_name = dir_list[i];

        // Truncate the ".txt" suffix
        if (file_name.endsWith(".txt")) {
          dir_list[i] = file_name.substring(0,
                        file_name.length() - 4);
        }

        System.err.println("Checking " +
                           file_name +
                           " for " +
                           str);
        if (file_name.indexOf(str) >= 0) {
          matches.add(dir_list[i]);
        }
      }

      for (int i = 0; i < matches.size(); i++){
        searchList.add(matches.get(i) + ".txt");
      }
    }

    public void showGUI() {
      setTitle("Recipes");

      Panel mainp = new Panel(new FlowLayout());

      Panel p1 = new Panel(new BorderLayout());
      p1.add(titleLabel, BorderLayout.WEST);
      p1.add(titleText, BorderLayout.EAST);
      mainp.add(p1);

      Panel p2 = new Panel(new BorderLayout());
      p2.add(ingredientsLabel, BorderLayout.WEST);
      p2.add(ingredientsText, BorderLayout.EAST);
      mainp.add(p2);

      Panel p3 = new Panel(new BorderLayout());
      p3.add(ingredientsList, BorderLayout.CENTER);
      p3.add(searchList, BorderLayout.EAST);
      p3.add(ingredientsRemoveSel, BorderLayout.SOUTH);
      mainp.add(p3);

      Panel p4 = new Panel(new BorderLayout());
      p4.add(saveButton, BorderLayout.WEST);
      p4.add(searchButton, BorderLayout.CENTER);
      p4.add(exitButton, BorderLayout.EAST);
      p4.add(statusText, BorderLayout.SOUTH);
      mainp.add(p4);

      // Add the object listeners
      exitButton.addActionListener(new ActionListener() {
          public void actionPerformed(ActionEvent e) {
            exit();
          }
        });

      ingredientsText.addActionListener(new ActionListener() {
          public void actionPerformed(ActionEvent e) {
```

```
            addIngredient();
          }
        });

    ingredientsRemoveSel.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          removeSel();
        }
      });

    saveButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          save();
        }
      });

    searchButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          search();
        }
      });

    searchList.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          loadFile();
        }
      });

    add(mainp);
    setSize(400,400);
    show();
  }

  public static void main(String args[]){
    Recipes obj = new Recipes();
    obj.showGUI();
  }
}
```

Recipes.java

We clearly need to improve this code. Let's begin by making a separate object to hold a recipe, so that we can construct test recipe data easily and toss it back and forth to the screen, disk, network, or wherever. This is just a simple data holder, with accessors for the data members.

```
Line 1  import java.util.ArrayList;
        import java.util.Iterator;

        public class Recipe {
          protected String name;
          protected ArrayList ingredients;

          public Recipe() {
            name = "";
            ingredients = new ArrayList();
          }

          public Recipe(Recipe another) {
```

```
      name = another.name;
      ingredients = new ArrayList(another.ingredients);
    }

    public void setName(String aName) {
      name = aName;
    }
    public String getName() {
      return name;
    }

    public void addIngredient(String aThing) {
      ingredients.add(aThing);
    }

    public Iterator getIngredients() {
      return ingredients.iterator();
    }

    public int getNumIngredients() {
      return ingredients.size();
    }

}
```

Recipe.java

Next, we need to pull the code out from the original `Recipes` class to save and load a file to disk.

To help separate file I/O from any other kind of I/O, we'll perform the file I/O in a helper class that uses `Recipe`. We want to take out all of the GUI widget references from the original source code, and use instance member variables instead.

```
Line 1  import java.io.IOException;
        import java.io.BufferedReader;
        import java.io.FileReader;
        import java.io.PrintWriter;
        import java.io.FileWriter;
        import java.util.Iterator;


        public class RecipeFile {

          public Recipe load(String fileName)
            throws IOException {

            BufferedReader in = null;
            Recipe result = new Recipe();

            in = new BufferedReader(new FileReader(fileName));
            String line;
            while ((line = in.readLine()) != null){
              int pos = line.indexOf('=');
              String token = line.substring(0, pos);
              String value = line.substring(pos+1);
              if (token.equals("TITLE")) {
                result.setName(value);
              } else if (token.equals("INGREDIENTS")) {
                int num_lines = Integer.parseInt(value);
```

```
            for (int i=0; i < num_lines; i++) {
              line = in.readLine();
              result.addIngredient(line);
            }
          }
        }
        in.close();
        return result;
    }

    public void save(String fileName, Recipe recipe)
      throws IOException {

      PrintWriter out = new PrintWriter(
                              new FileWriter(fileName));

      out.println("NAME=" + recipe.getName());
      out.println("INGREDIENTS=" +
                  recipe.getNumIngredients());

      Iterator itr = recipe.getIngredients();
      while (itr.hasNext()) {
        out.println(itr.next());
      }
      out.close();
    }
  }
```

RecipeFile.java

Now we're in a position where we can write a genuine test case that will test reading and writing to disk, without using any GUI code.

```
Line 1  import junit.framework.*;
  -     import java.io.File;
  -     import java.io.IOException;
  -     import java.util.Iterator;
  5
  -     public class TestRecipe extends TestCase {
  -
  -       public void testSaveandRestore() throws IOException {
  -
 10         final String test_name =
  -           "Cheeseburger";
  -         final String test_ing1 =
  -           "1/4 lb ground sirloin";
  -         final String test_ing2 =
 15           "3 slices Vermont cheddar cheese";
  -         final String test_ing3 =
  -           "2 slices maple-cured bacon";
  -
  -         // Save one out
 20         Recipe rec = new Recipe();
  -         rec.setName(test_name);
  -         rec.addIngredient(test_ing1);
  -         rec.addIngredient(test_ing2);
  -         rec.addIngredient(test_ing3);
 25
  -         RecipeFile filer = new RecipeFile();
  -         filer.save("test.recipe", rec);
  -
  -         try {
```

```
30          // Now get it back
-           Recipe rec2 = new Recipe();
-           filer = new RecipeFile();
-           rec2 = filer.load("test.recipe");
-
35          assertEquals(test_name, rec2.getName());
-           Iterator itr = rec2.getIngredients();
-
-           assertEquals(test_ing1, itr.next());
-           assertEquals(test_ing2, itr.next());
40          assertEquals(test_ing3, itr.next());
-           assertFalse(itr.hasNext());
-         } finally {
-           new File("test.recipe").delete();
-         }
45
-      }
-    }
```

TestRecipe.java

At line 11 we'll declare some constant strings for testing. Then we make a new, empty object and populate it with the test data beginning at line 21. We could just pass literal strings directly into the object instead, and not bother with `final` variables, but since we'll need to check the results against these strings, it makes sense to put them in common constants that we can reference from both spots.

With a `Recipe` data object now fully populated, we'll call the `save()` method to write the recipe to disk at line 27. Now we can make a brand-new `Recipe` object, and ask the helper to load it from that same file at line 33.

With the restored object in hand, we can now proceed to run a whole bunch of `asserts` to make sure that the test data we set in the `rec` object has been restored in the `rec2` object.

Finally, at line 43 we play the part of a good neighbor and delete the temporary file we used for the test. Note that we use a `finally` clause to ensure that the file gets deleted, even if one of our assertions fails.

Now we can run the unit test in the usual fashion to make sure that the code is reading and writing to disk okay.

Try running this example before reading on...


```
There was 1 failure:
1) testSaveandRestore(TestRecipe)junit.framework.ComparisonFailure:
   expected:<Cheeseburger> but was:<>
   at TestRecipe.testSaveandRestore(TestRecipe.java:33)
```

Whoops! Seems that wasn't working as well as we thought—we're not getting the name line of the recipe back. When we save the file out in `RecipeFile.java`, the code is using the key string `"NAME"` to identify the field, but when we read it back in (line 23 of `load()`), it's trying to use the string `"TITLE"`. That's just not going to work. We can easily change that to read `"NAME"`, to match the key used for the save, but stop and ask yourself the critical question:

Could this happen anywhere else in the code?

Using strings as keys is a fine idea, but it does open the door to introduce errors due to misspellings or inconsistent naming as we've seen here. So perhaps this failing test is trying to tell you something more—perhaps you should refactor the code and pull out those literal strings into constants. The class then looks like this:

```
Line 1   import java.io.IOException;
       - import java.io.BufferedReader;
       - import java.io.FileReader;
       - import java.io.PrintWriter;
     5 import java.io.FileWriter;
       - import java.util.Iterator;
       -
       -
       - public class RecipeFile {
    10
       -   // No one else should read or write
       -   // this file, so these strings
       -   // are private.
       -   private final static String NAME_TOK =
    15       "NAME";
       -   private final static String INGREDIENTS_TOK =
       -     "INGREDIENTS";
       -
       -   public Recipe load(String fileName)
    20     throws IOException {
       -
       -     BufferedReader in = null;
       -     Recipe result = new Recipe();
       -
    25     in = new BufferedReader(new FileReader(fileName));
       -     String line;
       -     while ((line = in.readLine()) != null){
       -       int pos = line.indexOf('=');
       -       String token = line.substring(0, pos);
    30       String value = line.substring(pos+1);
       -       if (token.equals(NAME_TOK)) {
       -         result.setName(value);
       -       } else if (token.equals(INGREDIENTS_TOK)) {
       -         int num_lines = Integer.parseInt(value);
    35         for (int i=0; i < num_lines; i++) {
       -           line = in.readLine();
       -           result.addIngredient(line);
```

```
          }
        }
      }
      in.close();
      return result;
    }

    public void save(String fileName, Recipe recipe)
      throws IOException {

      PrintWriter out = new PrintWriter(
                              new FileWriter(fileName));

      out.println(NAME_TOK + "=" +
                  recipe.getName());
      out.println(INGREDIENTS_TOK + "=" +
                  recipe.getNumIngredients());

      Iterator itr = recipe.getIngredients();
      while (itr.hasNext()) {
        out.println(itr.next());
      }
      out.close();
    }
  }
```

RecipeFile.java

We've improved the original program a lot with these simple changes. In order to test the file I/O, we:

- Made `Recipe` a first-class object
- Moved file I/O routines out of the GUI and into `RecipeFile`
- Pulled literals into constants to avoid bugs from typos

Finally, now that we have unit tests that provide the basic capabilities of a `Recipe`, we need to re-integrate the new `Recipe` class into the GUI itself and tend to the file I/O. We'd like to end up with something like Figure 9.3.

Now `RecipeGUI` holds an object of type `Recipe`, and uses the helper class `RecipeFile` to read and write recipes to disk. When the user presses the save button, the GUI will set values from the widgets in the `Recipe` object and call `RecipeFile.save()`. When a new recipe is loaded in, the GUI will get the proper values from the `Recipe` object returned from `RecipeFile.load()`.

Testing GUI's is hard, and isn't always worth the extreme effort. By separating the pure GUI from the guts of the appli-

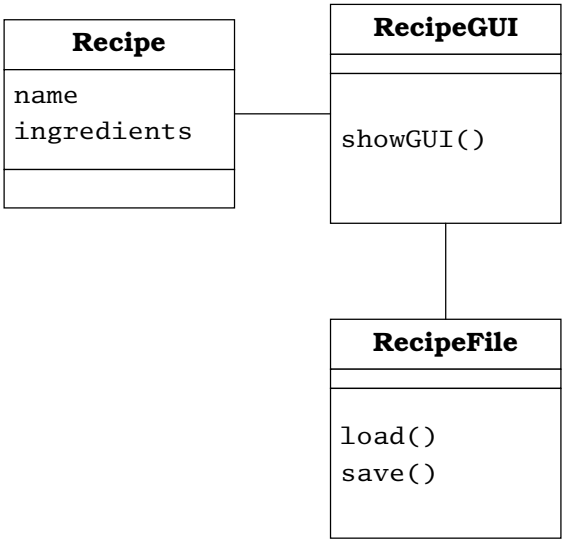

Figure 9.3: Refactored `Recipes` Static Class Diagram

cation, you can easily add and test business features without involving the GUI.

The main GUI class `RecipeGUI` (formerly known as `Recipes`) should now contain nothing but GUI-oriented code: widgets, callbacks, and so on. Thus, all of the "business logic" and file I/O can be in non-GUI, fully testable classes.

And we've got a clean design as an added bonus.

9.3 Testing the Class Invariant

Another way to improve the design of a class is by defining and verifying the "class invariant."[1]

A class invariant is an assertion, or some set of assertions, about objects of a class. For an object to be valid, all of these assertions must be true. They cannot vary.

For instance, a class that implements a sorted list may have the invariant that its contents are in sorted order. That means

[1] For more information on pre-conditions, post-conditions and invariants, see [Mey97].

that no matter what else happens, no matter what methods are called, the list must always be in sorted order—at least as viewed from outside the object. Within a method, of course, the invariant may be momentarily violated as the class performs whatever housekeeping is necessary. But by the time the method returns, or the object is otherwise available for use (as in a multi-threaded environment), the invariant must hold true or else it indicates a bug.

That means it's something you could check for as part of every unit test for this class.

The invariant is generally an artifact of implementation: internal counters, the fact that certain member variables are populated, and so on. The invariant is not the place to check for user input validation or anything of that sort. When writing tests, you want to test just your one thing, but at the same time you want to make sure the overall state of the class is consistent—you want to make sure you have not inflicted any collateral damage.

Here are some possible areas where class invariants might apply.

Structural

The most common invariants are structural in nature. That is, they refer to structural properties of data. For instance, in an order-entry system you might have invariants such as:

- Every line item must belong to an order
- Every order must have one or more line items

When working with arrays of data, you'll typically maintain a member variable that acts as an index into the array. The invariants on that index would include:

- index must be $>= 0$
- index must be $<$ array length

You want to check the invariant if any of these conditions are likely to break. Suppose you are performing some sort of calculation on the index into an array; you'd want to check the invariant throughout your unit tests to make sure the class

is never in an inconsistent state. We showed this in the stack class example on page 51.

Structural errors will usually cause the program to throw an exception and/or terminate abruptly. For that matter, so will failing the invariant check. The difference is that when the invariant is violated, you know about it right away—right at the scene of the crime. You'll probably also know exactly what condition was violated. Without the invariant, the failure may occur far from the original bug, and backtracking to the cause might take you anywhere from a few minutes to a few *days*.

More importantly, checking the invariant makes sure that you aren't passing the tests based just on luck. It may be that there's a bug that the tests aren't catching that will blow up under real conditions. The invariant might help you catch that early, even if an explicit test does not.

Mathematical

Other constraints are more mathematical in nature. Instead of verifying the physical nature of data structures, you may need to consider the logical model. For example:

- Debits and credits on a bank account match the balance.
- Amounts measured in different units match after conversion (an especially popular issue with spacecraft).

This starts to sound a lot like the boundary conditions we discussed earlier, and in a way they are. The difference is that an invariant must always be true for the entire visible state of a class. It's not just a fleeting condition; it's *always* true.

Data Consistency

Often times an object may present the same data in different ways—a list of items in a shopping cart, the total amount of the sale, and the total number of items in the cart are closely related. From a list of items with details, you can derive the other two figures. It must be an invariant that these figures are consistent. If not, then there's a bug.

9.4 Test-Driven Design

Test-driven development is a valuable technique where you always write the tests themselves *before* writing the methods that they test [Bec00]. As a nice side benefit of this style of working, you can enjoy "test-driven design" and significantly improve the design of your interfaces.

You'll get better interfaces (or API's) because you are "eating your own dog food," as the saying goes—you are able to apply feedback to improve the design.

That is, by writing the tests first, you have now placed yourself in the role of a *user* of your code, instead of the *implementor* of your code. From this perspective, you can usually get a much better sense of how an interface will really be used, and might see opportunities to improve its design.

For example, suppose you're writing a routine that does some special formatting for printed pages. There are a bunch of dimensions that need to be specified, so you code up the first version like this:

```
addCropMarks(PSStream str, double paper_width,
                           double paper_height,
                           double body_width,
                           double body_height);
```

Then as you start to write the tests (based on real-world data) you notice that a pattern emerges from the test code:

```
public process() {
  addCropMarks(str, 8.5, 11.0, 6.0, 8.5);
  addCropMarks(str, 8.5, 11.0, 6.0, 8.5);
  addCropMarks(str, 8.5, 11.0, 6.0, 8.5);
  addCropMarks(str, 5.0, 7.0, 4.0, 5.5);
  addCropMarks(str, 5.0, 7.0, 4.0, 5.5);
}
```

As it turns out, there are only a handful of common paper sizes in use, but you still need to allow for odd-ball sizes as

necessary. So the first thing to do—just to make the tests easier, of course—is to factor out the size specification into a separate object.

```
PaperSpec standardPaper1 = new PaperSpec(8.5, 11.0,
                                         6.0, 8.5);
PaperSpec standardPaper2 = new PaperSpec(5.0, 7.0,
                                         4.0, 5.5);
...
addCropMarks(str, standardPaper1);
addCropMarks(str, standardPaper1);
...
addCropMarks(str, standardPaper2);
```

Now the tests are much cleaner and easier to follow, and the application code that uses this will be cleaner as well.

Since these standard paper sizes don't vary, we can make a factory class that will encapsulate the creation of all the standard paper sizes.

```
public class StandardPaperFactory {
  public static PaperSpec getLetterInstance();
  public static PaperSpec getA4Instance();
  public static PaperSpec getLegalInstance();
  ...
  ...
}
```

By making the tests cleaner and easier to write, you will make the real code cleaner and easier to write as well.

Try it

Exercises

8. Design an interest calculator that calculates the amount of interest based on the number of working days in-between two dates. Use test-first design, and take it one step at a time.

Answer on 145

9.5 Testing Invalid Parameters

One question that comes up when folks first start testing is: "Do I have to test whether my class validates it parameters?" The answer, in best consultant fashion, is "it depends. . . ."

Is your class supposed to validate its parameters? If so, then yes, you need to test that this functionality is correct. But there's a larger question here: Who's responsible for validating input data?

In many systems, the answer is mixed, or haphazard at best. You can't really trust that any other part of the system has checked the input data, so you have to check it yourself—or at least, that aspect of the input data that particularly concerns you. In effect, the data ends up being checked by everyone and no one. Besides being a grotesque violation of the DRY principle [HT00], it wastes a lot of time and energy—and we typically don't have that much extra to waste.

In a well-designed system, you establish up-front the parts of the system that need to perform validation, and localize those to a small and well-known part of the system.

So the first question you should ask about a system is, "who is *supposed* to check the validity of input data?"

Generally we find the easiest rule to adopt is the "keep the barbarians out at the gate" approach. Check input at the boundaries of the system, and you won't have to duplicate those tests inside the system. Internal components can trust that if the data has made it this far into the system, then it must be okay.

It's sort of like a hospital operating room or industrial "clean room" approach. You undergo elaborate cleaning rituals before you—or any tools or materials—can enter the room, but once there you are assured of a sterile field. If the field becomes contaminated, it's a major catastrophe; you have to re-sterilize the whole environment.

Any part of the software system that is outward-facing (a UI, or interface to another system) needs to be robust, and not allow any incorrect or unvalidated data through. What defines

"correct" or valid data should be part of specification you're testing against.

What does any of this have to do with unit testing?

It makes a difference with regard to what you need to test against. As we mentioned earlier, if it isn't your code's responsibility to check for input data problems, then don't waste time checking for it. If it *is* your responsibility, then you need to be extra vigilant—because now the rest of the system is potentially relying on you, and you alone.

But that's okay. You've got unit tests.

Appendix A

Gotchas

Here are some popular "gotchas," that is, issues, problems, or misconceptions that have popped up over and over again to trap the unwary.

A.1 As Long As The Code Works

Some folks seem to think that it's okay to live with broken unit tests as long as the code itself works. Code without tests—or code with broken tests—*is* broken. You just don't know where, or when. In this case, you've really got the worst of both worlds: all that effort writing tests in the first place is wasted, and you still have no confidence that the code is doing what it ought.

If the tests are broken, treat it just as if the code were broken.

A.2 "Smoke" Tests

Some developers believe that a "smoke test" is good enough for unit testing. That is, if a method makes it all the way to the end without blowing up, then it passed.

You can readily identify this sort of a test: there are no asserts within the test itself, just one big `assertTrue(true)` at the end. Maybe the slightly more adventurous will have multiple `assertTrue(true)`'s throughout, but no more than that. All they are testing is, "did it make it this far?"

And that's just not enough. Without validating any data or other behavior, all you're doing is lulling yourself into a false sense of security—you might think the code is tested, but it is not.

Watch out for this style of testing, and correct it as soon as possible. **Real testing checks results.** Anything else is just wasting everyone's time.

A.3 "Works On My Machine"

Another pathologic problem that turns up on some projects is that old excuse, "It's not broken, it works on my machine." This points to a bug that has some correlation with the environment. When this happens, ask yourself:

- Is everything under version control?
- Is the development environment consistent on the affected machines?
- Is it a genuine bug that just happens to manifest itself on another machine (because it's faster, or has more or less memory, etc.)?

End users, in particular, don't like to hear that the code works on *your* machine and not theirs.

All tests must pass on *all* machines; otherwise the code is broken.

A.4 Floating-Point Problems

Quite a few developers appear to have missed that one day in class when they talked about floating-point numbers. It's a fact of life that there are floating point numbers that can only be approximately represented in computer hardware. The computer only has so many bits to work with, so something has to give.

This means that `1.333 + 1.333` isn't going to equal `2.666` exactly. It will be close, but not exact. That's why the JUnit floating-point asserts require you to specify a *tolerance* along with the desired values (see the discussion on page 23).

But still you need to be aware that "close enough" may be deceptive at times. Your tests may be too lenient for the real world's requirements, for instance. Or you might puzzle at an error message that says:

```
There was 1 failure:
1) testXyz(TestXyz)junit.framework.AssertionFailedError:
   expected:<1.0000000> but was:<1.0000000>
   at TestXyz.testXyz(TestXyz.java:10)
```

"Gosh, they sure look equal to me!" But they aren't—there must be a difference that's smaller than is being displayed by the print method.

As a side note, you can get a similar problem when using date and time types. Two dates might look equal as they are normally displayed—but maybe the milliseconds aren't equal.

A.5 Tests Take Too Long

Unit tests need to run fairly quickly. After all, you'll be running them a lot. But suddenly you might notice that the tests are taking *too long*. It's slowing you down as you write tests and code during the day.

That means it's time to go through and look at your tests with a fresh eye. Cull out individual tests that take longer than average to run, and group them together somewhere.

You can run these optional, longer-running tests once a day with the build, or when you check in, but not have to run them every single time you change code.

Just don't move them so far out of the way that they *never* get run.

A.6 Tests Keep Breaking

Some teams notice that the tests keep breaking over and over again. Small changes to the code base suddenly break tests all over the place, and it takes a remarkable amount of effort to get everything working again.

This is usually a sign of excessive coupling. Test code might be too tightly-coupled to external data, to other parts of the system, and so on.

As soon as you identify this as a problem, you need to fix it. Isolate the necessary parts of the system to make the tests more robust, using the same techniques you would use to minimize coupling in production code. See [HT00] for more details on orthogonality and coupling, or [FBB+99] for information on refactoring and design smells, and don't forget to use Mock Objects (Chapter 6) to decouple yourself from the real world.

A.7 Tests Fail on Some Machines

Here's a common nightmare scenario: all the tests run fine—on most machines. But on certain machines they fail consistently. Maybe on some machines they even fail intermittently.

What on earth could be going on? What could be different on these different machines?

The obvious answer is differences in the version of the operating system, libraries, the Java runtime engine, the database driver; that sort of thing. Different versions of software have different bugs, workarounds, and features, so it's quite possible that machines configured differently might behave differently.

But what if the machines are configured with identical software, and you still get different results?

It might be that one machine runs a little faster than the other, and the difference in timing reveals a race condition or other problem with concurrency. The same thing can show up on single vs. multiple-processor machines.

It's a real bug, it just happened not to have shown up before. Track it down on the affected machine using the usual methods. Prove the bug exists *on that machine* as best you can, and verify that all tests pass *on all machines* when you are done.

A.8 My `main` is Not Being Run

If you write a custom `main` in your test code it will be run when you run that class (for example, from the command line). However, if run by a JUnit test runner, your `main` will *not* be executed.

In particular, if you are using Ant[1] to run tests using the JUnit Task, `main` will not be called.

That's why you don't want to place any sort of setup code in `main`. Use the JUnit-provided `setUp` and `tearDown` methods (discussed on page 29) instead.

[1] http://ant.apache.org

Appendix B

Installing JUnit

Several IDE's come with JUnit already integrated in the product, right out of the box. If yours is one of these, then you need only consult the documentation that came with your system to begin using JUnit.

If you need to install or integrate JUnit yourself, then please read on.

JUnit for Java can be downloaded for free from the web.[1]

There you will also find instructions for installing and integrating JUnit into popular Java environments, including:

- JDeveloper
- Eclipse
- Forte/Netbeans
- IntelliJ
- JBuilder
- TogetherJ
- VisualAge

Some of these products are available in different versions, ranging from free, entry-level versions to Enterprise-class editions with many bells and whistles. The instructions for inte-

[1] http://junit.org

grating JUnit into your IDE may vary depending on both the product version and particular release, so we will not duplicate them here.

B.1 Command-line installation

JUnit is provided as a JAR file that contains all of the necessary classes. To install JUnit, all you have to do is put the JAR file where your compiler can find it.

If you are not using an IDE, but are just using the JDK directly from the command line, then you have to set your `CLASSPATH` to include the JUnit jar.

On Linux and other Unix-like systems, you just include the path to the JAR file in the `CLASSPATH` environment variable. For instance, suppose the JAR file is located in `/usr/java/packages/junit3.8.1/junit.jar`. You would need to run a command similar to the following:

```
CLASSPATH=$CLASSPATH:/usr/java/packages/junit3.8.1/junit.jar
```

Each entry in the class path is separated by a colon (":").

Typically you would put this command in your shell's startup script (`.bashrc` or `/etc/profile` or the like), so that you always have the modified `CLASSPATH`.

Under Microsoft Windows™ operating systems, go down the following menu path:

```
Start
 └── Settings
      └── Control Panel
           └── System
                └── Advanced Tab
                     └── Environment Variables...
```

Modify the existing `CLASSPATH` variable if there is one, or add a new environment variable named `CLASSPATH`. Suppose the JUnit jar is located in `C:\java\junit3.8.1\junit.jar`. You'd type that value into the dialog box:

```
Variable: CLASSPATH
Variable Value: C:\java\junit3.8.1\junit.jar
```

If you have existing entries in the class path, be aware that each additional entry in the class path is separated by a semicolon (";").

You may have to restart any shell windows or applications to have this change take effect.

B.2 Does it work?

To tell whether JUnit has been installed properly or not, try to compile a source file containing the import statement:

```
import junit.framework.*;
```

If that succeeds, then your compiler can find JUnit. You're ready to go!

Don't forget that your test code needs to inherit from JUnit's `TestCase` base class. See the explanation on page 26 for more information.

Appendix C

JUnit Test Skeleton

Sometimes it can be convenient to add a bit more functionality to the unit tests. In particular, it's nice to add a `main` so that you can simply run the test class directly from the command line. For example, suppose we have a test case named `com.abc.test.TestAlpha`. In this test case we have three test methods, named `testDefault()`, `testWithReal()` and `testWithExtension()`.

```
java com.abc.test.TestAlpha
java com.abc.test.TestAlpha testWithReal
java com.abc.test.TestAlpha testWithReal testDefault
```

The first command will run all three of the test methods in `TestAlpha`. The second line will just run the one method `testWithReal()` and the last will run both `testWithReal()` and `testDefault()`.

Figure C.1 on the following page shows a skeleton template that provides that functionality: you can run all tests or just a named test from the command line. It will use the `suite` method of a test class if one is defined, otherwise it will use reflection to discover the `test...` methods. It also includes stubs for the `setUp` and `tearDown` methods:

You'll notice we put in a default `suite()` method as well. This allows you to always assume that every `TestCase` has a `suite()` method you can call, even if it just does the default dynamic-discovery of test methods. You can change this, of course, to add selected tests or composite suites and so on.

```
// package...
import junit.framework.*;
// Change all occurrences of "Skeleton" below
// as appropriate
public class TestSkeleton extends TestCase {
  /**
   * Per-method test set up
   */
  public void setUp() {
  }
  /**
   * Per-method test tear down
   */
  public void tearDown() {
  }
  /**
   * Add tests here:
   * public void testName() ...
   */
  public TestSkeleton(String name) {
    super(name);
  }
  /**
   * Default suite method
   */
  public static Test suite() {
    return new TestSuite(TestSkeleton.class);
  }
  /** Note -- "main" will only be run when invoked
   * individually  from the command line
   * (not via Ant's JUnit Task, etc.)
   */
  public static void main (String[] args) {
    TestSuite suite = new TestSuite();
    if (args.length != 0) {
      // Run specific tests as indicated from the
      // command line
      for (int i=0; i< args.length; i++) {
        suite.addTest(new TestSkeleton(args[i]));
      }
    } else {
      // Dynamically discover all of them, or use
      // user-defined suite
      suite.addTest(TestSkeleton.suite());
    }
    junit.textui.TestRunner.run(suite);
  }
}
```

Figure C.1: TestSkeleton.java—Basic test Skeleton

This code is a good starting point; however, there's a lot of duplicated code in that skeleton that will be propagated throughout the system. That is, copying this code for each new class throughout the system will result in duplication. And that can cause problems.

For example, we had a client that did exactly that. They were somewhat new to unit testing, and wanted to be able to run tests from the command line. So they had a boilerplate template that provided a suitable `main()` to run the tests. It also contained some startup code to establish database and CORBA connections.

The code was widely copied throughout the system, which was unfortunate when the CORBA connection logic had to change. The developer who had written the template in the first place had to go through all 78 places where it had been copied to make the necessary change. When the database connection details changed, he had to go through all 86 places and change them again (a few more classes had been introduced by then).

So perhaps we'd better refactor this skeleton before it becomes widely used, to make the part that gets copied as small and trivial as possible, with all the real functionality located in one place (instead of the 112 places it's up to by now).

C.1 Helper Class

We'll pull out the execution code into a helper class, shown in Figure C.2 on the next page. This class contains all the functionality we need to run specific tests given as arguments, or all the tests defined by a `suite()` method, or all the tests named `test...` if there is no `suite()` method.

C.2 Basic Template

And now the code template that we need to copy and start off with is much simpler. We show it in Figure C.3 on page 131.

If you think this would be helpful, start each test class you write by copying this template as a starting point. If your

```
import junit.framework.*;
import java.lang.reflect.Method;
public class TestFinder {
  /* Note -- "main" will only be run when invoked individually
   * from the command line (not via Ant, etc.).
   * This code dynamically builds a test suite, based either
   * on command-line  arguments, or on
   * reflection into the specified class.
   */
  public static void run(Class which, String[] args) {
    TestSuite suite = null;
    if (args.length != 0) {
      // Run specific tests as indicated from the command line
      try {
        java.lang.reflect.Constructor ctor;
        ctor = which.getConstructor(new Class[]
          {String.class});
        suite = new TestSuite();
        for (int i=0; i< args.length; i++) {
          suite.addTest((TestCase)ctor.newInstance(
                                    new Object[]{args[i]}));
        }
      } catch (Exception e) {
        System.err.println("Unable to instantiate " +
                           which.getName() +
                           ": " + e.getMessage());
        System.exit(1);
      }
    } else {
      // Call the suite() method of the given class,
      // if there is one
      try {
        Method suite_method = which.getMethod("suite",
                                              new Class[0]);
        suite = (TestSuite) suite_method.invoke(null,
                                                null);
      } catch (Exception e) {
        // Whoops!  No public suite() in that class.
        // Make a default list using reflection:
        suite = new TestSuite(which);
      }
    }
    junit.textui.TestRunner.run(suite);
  }
}
```

Figure C.2: TestFinder.java—Test Helper Class

```
import junit.framework.*;
public class TestSample extends TestCase {
  public TestSample(String name) { super(name); }
  /**
   * Per-test setup
   */
  public void setUp() {
  }
  /**
   * Per-test teardown
   */
  public void tearDown() {
  }
  /*
   * Tests go here...
   */
  public void testMe() {
    assertTrue(true);
  }
  /**
   * Default suite() method discovers all tests...
   */
  public static Test suite() {
    return new TestSuite(TestSample.class);
  }
  public static void main(String[] args) {
    TestFinder.run(TestSample.class, args);
  }
};
```

Figure C.3: TestSample.java—Simplified Test Skeleton

environment allows it, you can arrange to have this skeleton come up whenever you start a unit test for a new class, and just change the name “Sample” to the actual name of the class.

Appendix D

Resources

D.1 On The Web

Ant
⇒ http://ant.apache.org
Java-based, cross-platform build tool, similar to `make`.

Cactus
⇒ http://jakarta.apache.org/cactus
Cactus is a simple test framework for unit testing server-side java code (Servlets, EJBs, Tag Libs, Filters, etc).

CruiseControl
⇒ http://cruisecontrol.sourceforge.net
CruiseControl is a framework for a continuous build process. It includes plugins for email notification, Ant integration, and various source control tools. A web interface is provided to view the details of the current and previous builds.

Easy-Mock
⇒ http://www.easymock.org
Provides an easy way to use Mock Objects in JUnit tests.

JUnit
⇒ http://junit.org
Unit testing framework for Java.

JUnitPerf
⇒ http://www.clarkware.com
JUnitPerf is a collection of JUnit test decorators that help you measure the performance and scalability of those parts of your system that have JUnit tests.

MockObjects
⇒ http://www.mockobjects.com
A core Mock object framework for Java programmers and set of mock implementations for the standard Java platform APIs.

Nounit
⇒ http://nounit.sourceforge.net
Nounit generates a report from your code to graphically show you how many of your project's methods are being tested, and how well. It can graph other aspects of the code base as well.

Pragmatic Programming
⇒ http://www.pragmaticprogrammer.com
Home page for Pragmatic Programming and your authors. Here you'll find all of the source code examples from this book, additional resources, updated URLs and errata, and news on additional volumes in this series and other resources.

Quilt
⇒ http://quilt.sourceforge.net
Quilt provides code coverage statistics based on JUnit Unit Tests. It currently contains Statement and Branch coverage.

Tinderbox
⇒ http://www.mozilla.org/tinderbox.html
Tinderbox allows you to see what is happening in the source tree; it shows you who checked in what (by asking Bonsai); what platforms have built successfully; what platforms are broken and exactly how they are broken (the build logs); and the state of the files that made up the build (cvsblame) so you can figure out who broke the build and how to fix it.

Tomcat
⇒ http://jakarta.apache.org/tomcat
Tomcat is the servlet container that is used in the official Reference Implementation for the Java Servlet and JavaServer Pages technologies.

xUnit
⇒ http://www.xprogramming.com/software.htm
Unit testing frameworks for many, many different languages and environments.

D.2 Bibliography

[Bec00] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, MA, 2000.

[Cla04] Mike Clark. *Pragmatic Automation*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, (planned for) 2004.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley Longman, Reading, MA, 1999.

[HT00] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, Reading, MA, 2000.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, NJ, second edition, 1997.

[MFC01] Tim Mackinnon, Steve Freeman, and Philip Craig. Endo-testing: Unit testing with mock objects. In Giancarlo Succi and Michele Marchesi, editors, *Extreme Programming Examined*, chapter 17, pages 287–302. Addison Wesley Longman, Reading, MA, 2001.

[TH03] Dave Thomas and Andy Hunt. *Pragmatic Version Control*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2003.

Pragmatic Unit Testing: Summary

General Principles:

- ☐ Test anything that might break
- ☐ Test everything that does break
- ☐ New code is guilty until proven innocent
- ☐ Write at least as much test code as production code
- ☐ Run local tests with each compile
- ☐ Run all tests before check-in to repository

Questions to Ask:

- ☐ If the code ran correctly, how would I know?
- ☐ How am I going to test this?
- ☐ What *else* can go wrong?
- ☐ Could this same kind of problem happen anywhere else?

What to Test: Use Your RIGHT-BICEP

- ☐ Are the results **right**?
- ☐ Are all the **boundary** conditions CORRECT?
- ☐ Can you check **inverse** relationships?
- ☐ Can you **cross-check** results using other means?
- ☐ Can you force **error conditions** to happen?
- ☐ Are **performance** characteristics within bounds?

Good tests are A TRIP

- ☐ **A**utomatic
- ☐ **T**horough
- ☐ **R**epeatable
- ☐ **I**ndependent
- ☐ **P**rofessional

CORRECT Boundary Conditions

- ☐ **C**onformance — Does the value conform to an expected format?
- ☐ **O**rdering — Is the set of values ordered or unordered as appropriate?
- ☐ **R**ange — Is the value within reasonable minimum and maximum values?
- ☐ **R**eference — Does the code reference anything external that isn't under direct control of the code itself?
- ☐ **E**xistence — Does the value exist? (e.g., is non-null, non-zero, present in a set, etc.)
- ☐ **C**ardinality — Are there exactly enough values?
- ☐ **T**ime (absolute and relative) — Is everything happening in order? At the right time? In time?

---

http://www.pragmaticprogrammer.com/sk/ut

Appendix F

Answers to Exercises

Exercise 1: *from page 59*

A simple stack class. Push `String` objects onto the stack, and `pop` them off according to normal stack semantics. This class provides the following methods:

```
public interface StackExercise {
  /**
    * Return and remove the most recent item from
    * the top of the  stack.
    * Throws StackEmptyException
    * if the stack is empty
    */
  public String pop() throws StackEmptyException;
  /**
   * Add an item to the top of the stack.
   */
  public void push(String item);
  /**
    * Return but do not remove the most recent
    * item from the top of the stack.
    * Throws StackEmptyException
    * if the stack is empty
    */
  public String top() throws StackEmptyException;
  /**
    * Returns true if the stack is empty.
    */
  public boolean isEmpty();
}
```

StackExercise.java

Here are some hints to get you started: what is likely to break? How should the stack behave when it is first initialized? After it's been used for a while? Does it really do what it claims to do?

Answer 1:

- For a brand-new stack, `isEmpty()` should be `true`, `top()` and `pop()` should throw exceptions.
- Starting with an empty stack, call `push()` to push a test string onto the stack. Verify that `top()` returns that string several times in a row, and that `isEmpty()` returns `false`.
- Call `pop()` to remove the test string, and verify that it is the same string.[1] `isEmpty()` should now be true. Call `pop()` again verify an exception is thrown.
- Now do the same test again, but this time add multiple items to the stack. Make sure you get the rights ones back, in the right order (the most recent item added should be the one returned).
- Push a `null` onto the stack and `pop` it; confirm you get a `null` back.
- Ensure you can use the stack after it has thrown exceptions.

Exercise 2: *from page 60*

A shopping cart. This class lets you add, delete, and count the items in a shopping cart.

What sort of boundary conditions might come up? Are there any implicit restrictions on what you can delete? Are there any interesting issues if the cart is empty?

```
public interface ShoppingCart {
  /**
   * Add this many of this item to the
   * shopping cart.
   */
  public void addItems(Item anItem, int quantity)
              throws NegativeCountException;

  /**
   * Delete this many of this item from the
   * shopping cart
   */
  public void deleteItems(Item anItem,
                          int quantity)
              throws NegativeCountException,
                     NoSuchItemException;

  /**
   * Count of all items in the cart
   * (that is, all items x qty each)
   */
  public int itemCount();
```

[1] In this case, `assertEquals()` isn't good enough; you need `assertSame()` to ensure it's the same object.

```
    /**
     * Return Iterator of all items
     * (see Java Collection's doc)
     */
    public Iterator iterator();
}
```

Answer 2:

- Call `addItems` with quantity of 0 and `itemCount` should remain the same.
- Call `deleteItem` with quantity of 0 and `itemCount` should remain the same.
- Call `addItems` with a negative quantity and it should raise an exception.
- Call `deleteItem` with a negative quantity and it should raise an exception.
- Call `addItems` and the item count should increase, whether the item exists already or not.
- Call `deleteItem` where the item doesn't exist and it should raise an exception.
- Call `deleteItem` when there are no items in the cart and `itemCount` should remain at 0.
- Call `deleteItem` where the quantity is larger than the number of those items in the cart and it should raise an exception.
- Call `iterator` when there are no items in the cart and it should return an empty iterator (i.e., it's a real iterator object (not null) that contains no items).
- Call `addItem` several times for a couple of items and verify that contents of the cart match what was added (as reported via `iterator()` and `itemCount()`).

Hint: you can combine several of these asserts into a single test. For instance, you might start with an empty cart, add 3 of an item, then delete one of them at a time.

Exercise 3: *from page 61*

A fax scheduler. This code will send faxes from a specified file name to a U.S. phone number. There is a validation requirement; a U.S. phone number with area code must be of the form *xnn-nnn-nnnn*, where *x* must be a digit in the range `[2..9]` and *n* can be `[0..9]`. The following blocks are reserved and are not currently valid area codes: *x*`11`, *x*`9`*n*, `37`*n*, `96`*n*.

The method's signature is:

```
/**
 * Send the named file as a fax to the
 * given phone number.
 */
public boolean sendFax(String phone,
                       String filename)
               throws MissingOrBadFileException,
                      PhoneFormatException,
                      PhoneAreaCodeException;
```

Given these requirements, what tests for boundary conditions can you think of?

Answer 3:

- Phone numbers with an area code of 111, 211, up to 911, 290, 291, etc, 999, 370-379, or 960-969 should throw a `PhoneAreaCodeException`.
- A phone number with too many digits (in one of each set of number, area code, prefix, number) should throw a `PhoneFormatException`.
- A phone number with not enough digits (in one of each set) should throw a `PhoneFormatException`.
- A phone number with illegal characters (spaces, letters, etc.) should throw a `PhoneFormatException`.
- A phone number that's missing dashes should throw a `PhoneFormatException`.
- A phone number with multiple dashes should throw a `PhoneFormatException`.
- A null phone number should throw a `PhoneFormatException`.
- A file that doesn't exist should throw a `MissingOrBadFileException`.
- A null filename should also throw that exception.
- An empty file should throw a `MissingOrBadFileException`.
- A file that's not in the correct format should throw a `MissingOrBadFileException`.

Exercise 4: *from page 61*
An automatic sewing machine that does embroidery. The class that controls it takes a few basic commands. The coordinates (0,0) represent the lower-left corner of the machine. x and y increase as you move toward the upper-right corner, whose coordinates are x =

`getTableSize().width - 1` and `y = getTableSize().height - 1`.

Coordinates are specified in fractions of centimeters.

```
public void moveTo(float x, float y);
public void sewTo(float x, float y);
public void setWorkpieceSize(float width,
                             float height);
public Size getWorkpieceSize();
public Size getTableSize();
```

There are some real-world constraints that might be interesting: you can't sew thin air, of course, and you can't sew a workpiece bigger than the machine.

Given these requirements, what boundary conditions can you think of?

Answer 4:

- Huge value for one or both coordinates
- Huge value for workpiece size
- Zero or negative value for one or both coordinates
- Zero or negative value for workpiece size
- Coordinates that move off the workpiece
- Workpiece bigger than the table

Exercise 5: *from page 61*
Audio/Video Editing Transport. A class that provides methods to control a VCR or tape deck. There's the notion of a "current position" that lies somewhere between the beginning of tape (BOT) and the end of tape (EOT).

You can ask for the current position and move from there to another given position. *Fast-forward* moves from current position toward EOT by some amount. *Rewind* moves from current position toward BOT by some amount.

When tapes are first loaded, they are positioned at BOT automatically.

```
public interface AVTransport {
  /**
   * Move the current position ahead by this many
   * seconds. Fast-forwarding past end-of-tape
   * leaves the position at end-of-tape
   */
  public void fastForward(float seconds);
```

```
    /**
     * Move the current position backwards by this
     * many seconds. Rewinding past zero leaves
     * the position at zero
     */
    public void rewind(float seconds);
    /**
     * Return current time position in seconds
     */
    public float currentTimePosition();
    /**
     * Mark the current time position with this label
     */
    public void markTimePosition(String name);
    /**
     * Change the current position to the one
     * associated with the marked name
     */
    public void gotoMark(String name);
}
```

AVTransport.java

Answer 5:

- Verify that the initial position is BOT.
- Fast forward by some allowed amount (not past end of tape), then rewind by same amount. Should be at initial location.
- Rewind by some allowed amount amount (not past beginning of tape), then fast forward by same amount. Should be at initial location.
- Fast forward past end of tape, then rewind by same amount. Should be before the initial location by an appropriate amount to reflect the fact that you can't advance the location past the end of tape.
- Try the same thing in the other direction (rewind past beginning of tape).
- Mark various positions and return to them after moving the current position around.
- Mark a position and return to it *without* moving in between.

Exercise 6: *from page 62*
Audio/Video Editing Transport, Release 2.0. As above, but now you can position in seconds, minutes, or frames (there are exactly 30 frames per second in this example), and you can move relative to the beginning or the end.

Answer 6: Cross-check results using different units: move in one unit and verify your position using another unit; move forward in one unit and back in another, and so on.

Exercise 7: *from page 74*

Come up with a simple mock object (by hand) for an MP3 player control panel with the following methods:

```
import java.util.ArrayList;
public interface Mp3Player {
  /**
   * Begin playing the filename at the top of the
   * play list, or do nothing if playlist
   * is empty.
   */
  public void play();
  /**
   * Pause playing. Play will resume at this spot.
   */
  public void pause();
  /**
   * Stop playing. The current song remains at the
   * top of the playlist, but rewinds to the
   * beginning of the song.
   */
  public void stop();
  /** Returns the number of seconds into
   * the current song.
   */
  public double currentPosition();
  /**
   * Returns the currently playing file name.
   */
  public String currentSong();
  /**
   * Advance to the next song in the playlist
   * and begin playing it.
   */
  public void next();
  /**
   * Go back to the previous song in the playlist
   * and begin playing it.
   */
  public void prev();
  /**
   * Returns true if a song is currently
   * being played.
   */
  public boolean isPlaying();
  /**
   * Load filenames into the playlist.
   */
  public void loadSongs(ArrayList names);
}
```

Mp3Player.java

It should pass the following unit test:

```
import junit.framework.*;
import java.util.ArrayList;

public class TestMp3Player extends TestCase {
  protected Mp3Player mp3;
  protected ArrayList list = new ArrayList();

  public void setUp() {
    mp3 = new MockMp3Player();

    list = new ArrayList();
    list.add("Bill Chase -- Open Up Wide");
    list.add("Jethro Tull -- Locomotive Breath");
    list.add("The Boomtown Rats -- Monday");
    list.add("Carl Orff -- O Fortuna");
  }

  public void testPlay() {

    mp3.loadSongs(list);
    assertFalse(mp3.isPlaying());
    mp3.play();
    assertTrue(mp3.isPlaying());
    assertTrue(mp3.currentPosition() != 0.0);
    mp3.pause();
    assertTrue(mp3.currentPosition() != 0.0);
    mp3.stop();
    assertEquals(0.0, mp3.currentPosition(), 0.1);

  }

  public void testPlayNoList() {

    // Don't set the list up
    assertFalse(mp3.isPlaying());
    mp3.play();
    assertFalse(mp3.isPlaying());
    assertEquals(0.0, mp3.currentPosition(), 0.1);
    mp3.pause();
    assertEquals(0.0, mp3.currentPosition(), 0.1);
    assertFalse(mp3.isPlaying());
    mp3.stop();
    assertEquals(0.0, mp3.currentPosition(), 0.1);
    assertFalse(mp3.isPlaying());
  }

  public void testAdvance() {

    mp3.loadSongs(list);

    mp3.play();

    assertTrue(mp3.isPlaying());

    mp3.prev();
    assertEquals(list.get(0), mp3.currentSong());
    assertTrue(mp3.isPlaying());

    mp3.next();
    assertEquals(list.get(1), mp3.currentSong());
    mp3.next();
    assertEquals(list.get(2), mp3.currentSong());
    mp3.prev();

    assertEquals(list.get(1), mp3.currentSong());
    mp3.next();
    assertEquals(list.get(2), mp3.currentSong());
```

```
    mp3.next();
    assertEquals(list.get(3), mp3.currentSong());
    mp3.next();
    assertEquals(list.get(3), mp3.currentSong());
    assertTrue(mp3.isPlaying());
  }
}
```

TestMp3Player.java

Answer 7:

Here is a simple mock object that handles the semantics of current song and position in that song.

```
import java.util.ArrayList;
public class MockMp3Player implements Mp3Player {
  // State of the player
  private boolean isPlaying = false;
  // Position within the current song
  private double currentPos = 0.0;
  // List of song names
  private ArrayList songList = new ArrayList();
  // Index of current song
  private int currentIndex;
  public void play() {
    if (songList.size() > 0) {
      isPlaying = true;
      // While playing, we're always 1 second
      // into the song.  For a more realistic mock,
      // you could implement a timer in a thread
      // that would advance the position and switch
      // to the next song when needed.
      currentPos = 1.0;
    } else {
      isPlaying = false;
      currentPos = 0.0;
    }
  }
  public void pause() {
    isPlaying = false;
  }
  public void stop() {
    isPlaying = false;
    // Rewind to beginning of current song
    currentPos = 0.0;
  }
  public double currentPosition() {
    return currentPos;
  }
  public String currentSong() {
    if (songList.size() == 0) {
      return null;
    }
    return (String)songList.get(currentIndex);
```

```
  }
  public void next() {
    if (currentIndex < songList.size()-1) {
      currentIndex++;
    }
    currentPos = 0.0;
  }
  public void prev() {
    if (currentIndex > 0) {
      currentIndex--;
    }
    currentPos = 0.0;
  }
  public boolean isPlaying() {
    return isPlaying;
  }
  public void loadSongs(ArrayList names) {
    songList = names;
  }
}
```

MockMp3Player.java

Exercise 8: *from page 116*
Design an interest calculator that calculates the amount of interest based on the number of working days in-between two dates. Use test-first design, and take it one step at a time.

Answer 8: Here's a possible scenario of steps you might take. There is no right answer; this exercise is simply to get you to think about test-first design.

1. Begin by simply calculating the days between any two dates first. The tests might include:
   - Use the same value for first date and last date.
   - Try the normal case where first date < last date.
   - Try the error case where first date > last date.
   - Try dates that span a year boundary (from October 1 2003 to March 1, 2004 for instance).
   - Try dates more than a year apart (from October 1 2003 to December 1, 2006).
2. Next, exclude weekends from the calculation, using the same sorts of tests.
3. Now exclude public and/or corporate holidays. This raises a potentially interesting question: how do you specify holidays? You had to face that issue when writing the tests; do you think doing so improved your interface?

4. Finally, perform the interest calculation itself. You might start off with tests such as:
    - Interest amount should never be negative (an invariant).
    - Interest when first date equals last date should be 0.0.

Index

Symbols

0–1–*n* rule, 56

A

A-TRIP, 77
A/V transport exercise, *see* Exercises, A/V transport
Accessors, 83
Actual, 23
`addCropMarks()`, 115
Agile, 38
Amount of test code, 83
Anonymous array, 16
Ant, 123, 132
AntHill, 78
Arianne 5 rocket, 6
Arrays
    and equality, 23
Assert
    custom, 51
    definition, 23
`assertEquals()`, 23
`assertEquals()`, 13
`assertFalse()`, 25
`assertNotSame()`, 24
`assertNull()`, 24
`assertSame()`, 24
`assertTrue()`, 13, 24, 34, 119
Assumptions, 53
Automatic, 78
Automation, ix, 78

B

Bad magic, 77
`Bearing.java`, 50
Big ball of mud, 94
Blank, 54
Boolean conditions, 13, 24
Boundary conditions, 41, 46
Breaking the build/tests, 91
Broccoli, 1
Bugs
    clumping, 79, 80
    elusive, 122
    fixing, 84
    identifying likely, 37
    in sort routines, 49
    isolating, 10, 25
    list position, 18, 48
    memory, 120
    phantom, 81
    rewriting due to, 10
    traceable to unit tests, 82
Build machine, 78
Business logic, 57, 112

C

Cactus, 74, 132
Cardinality, 55
Career limiting move, 10
`Checker.java`, 66
`checkInvariant()`, 52
Class Invariant, *see* Invariant
CLASSPATH, 89, 125
Clean room, 117
Code examples
    finding the source to, xi
Collateral damage, 113
    definition, 8
Concurrency, 57, 59, 122
Confidence, 3
Conformance, 47
Constructor, 26

Continuous build/integration, 78, 93, 95, 132, 133
Copy and paste, 32, 83, 129
CORRECT, 46
Costs, 10
Coupling, 63, 122
Cross-checking, 42
CruiseControl, 78, 95, 132
CVS, 91

D

Data structures, 51
Daylight savings time, 58
Debugging, 2, 10
Dependencies, 53, 122
Developer sandbox, 81, 86
Donne, John, 82
DRY principle, 82, 117
  definition, 32n

E

E-mail address format, 47
Easy-Mock, 72, 132
Elephant
  how to eat, 96
Encapsulation, 50, 64, 87
Engineering, 5
Environment, *see* Test code, environment
`Environmental.java`, 66
Environmental constraints, 44
Equality, 13
  deceptive, 121
  native arrays, 23
Error conditions, 43
Examples, *see* Code examples
Exception, 20, 23, 25, 33, 34, 54, 97
Excuses, 7
Exercises
  A/V transport, 61, 140
  fax machine, 61, 138
  interest calculator, 116
  MP3 player, 74, 142
  sewing machine, 61, 139
  shopping cart, 60, 137
  stack, 59, 136
Existence, 54
Expected, 23
External dependencies, 53, 122
Extreme Programming, 86, 96

F

Factory class, 116
`fail()`, 25
Failing tests, *see* Test code, broken
Fax machine exercise, *see* Exercises, fax machine
Feedback, x, 84, 93, 115
Fence post errors, 55
Finally, 109
Floating-point numbers, 23, 120
Formal testing, 4

G

`getTime()`, 64–69
GMT, 58
Good neighbor, 109

H

House of cards, 4

I

IDE, 11, 17, 78, 88, 124
Import, 26
Improving tests, 84
Independent, 29, 81
Indexing concepts, 53
Input data validation, 117
Interest calculator exercise, *see* Exercises, interest calculator
Invariant, 51, 52, 112
  on an index, 113
Inverse relationships, 42

J

Java
  library versions, 122
Java Exception, *see* Exception
Java packages, 89
JUnit, 132
  command-line installation, 125
  composing tests, 27
  custom asserts, 32
  directory structure, *see* Test code, locating
  downloading, 124
  and exceptions, 33

import statement, 126
minimum framework, 26, 35
naming conventions, 21, *see* Naming conventions
order of tests, 82
running, 17
JUnitPerf, 45, 132

L

`Largest.java`, 15
`largest()`, 14–20
Legacy code, 93, 94
Lighting doubles, 63
Linux, 125
Long-running tests, 121

M

`main()`, 123
Message, 23
Microsoft Windows, 125
Mock objects, 12, 78, 122, 132, 133
definition, 65
steps to using, 65
`MockHttpServletRequest`, 71
`MockMp3Player.java`, 144
`MockSystemEnvironment.java`, 66
`MockSystemEnvironment`, 67
MP3 player exercise, *see* Exercises, MP3 player
`Mp3Player.java`, 74, 142
`MyStack.java`, 51–53

N

Naming conventions, 21, 26, 35
Nounit, 79, 133
Null, 24, 54
Numeric overflow, 6n

O

Object identity, 24
Off-by-one errors, 18, 55
Ordering, 48

P

Pair programming, 96
Pay-as-you go model, 9
Performance, 44, 132
Phantom bugs, 81
Postconditions
definition, 54
Pragmatic Automation, ix
Pragmatic Programmers
email address, xii
website, xn
Pragmatic Programming, 133
Pragmatic Project Automation, 78
Pragmatic Starter Kit, viii
Pragmatic Version Control, viii, 91
Preconditions
definition, 54
Private access, 87
Production code, 5, 82, 89
definition, 21
Production system, 12
Professional, 82, 96
Project-specific base class, 32
`ProjectTest.java`, 32
Properties file, 102
Protected access, 87
Prototype, 11
Public access, 87

Q

Quilt, 79, 133

R

Range, 50
`Recipe.java`, 106
`RecipeFile.java`, 107
`Recipes.java`, 102–106
Refactoring, 101, 122, 129
Reference, 53
Regression, 44
Repeatable, 81
Requirements, 6, 19, 38, 57, 121
Restaurant order, 48
Results
analyzing, 7, 78, 120
Retrospectives, 96
Return on investment, 94
Reviews, 96
Right, 38
RIGHT-BICEP, 37

S

Sandbox, 81, 86
Scientific applications, 24
`sendFax()`, 61, 138

Separation of concerns, 99
Servlets, 69
setTime(), 66
setUp(), 29, 73, 123, 127
Setup code
    execution order, 30
Sewing machine exercise, *see* Exercises, sewing machine
Shopping cart exercise, *see* Exercises, shopping cart
"Shy" code, 99
Side-effects, 54
Single testing phase, 9
Skeleton template, 127
Sleep, 100
sleepUntilNextHour(), 99
Smoke test, 119
Software engineering, 5
Sort routines, 49
Stack exercise, *see* Exercises, stack machine
Stand-ins, 63
StandardPaperFactory, 116
String constants, 110
Stubs, 64
suite(), 27, 127
Synchronized, 59
Syntax vs. semantics, 11
SystemEnvironment.java, 66

T

Team communication, 97
Team environment, 91
tearDown(), 29, 123, 127
Teardown code
    execution order, 30
Test code
    and accessors, 83
    broken, 25, 119, 121
    cleanup, 109
    compiling, 22
    composing, 27
    correlate to bugs, 82
    coverage analysis, 133
    and data files, 38
    environment, 120
    first test, 15
    invoking, 78
    linear, 82
    locating, 87
    long running, 121
    ordering, 82
    vs. production code, 22, 83
    required actions, 22
    results, 7
    reviewing, 97
    run from command line, 127
    testing, 84
Test coverage analysis tools, 79, 80
Test data, 40
Test setup
    per-suite, 31
    per-test, 29
Test suites
    definition, 27
Test-driven design, 96, 115
testAdd(), 85
TestCase, 26, 29
    subclassing, 32
TestClassComposite.java, 29
TestClassOne.java, 27
TestClassTwo.java, 28
TestClassTwo, 31
TestFinder.java, 130
Testing
    acceptance, 3, 12
    and design, architecture, 20, 99
    courtesy, 91
    environment, 120
    excuses, 7
    formal, 4
    frequency, 92
    functional, 12
    GUI, 111
    metrics, 80
    performance, 3, 12
    regression, 44, 94
    responsibility, 118
testJamItIntoPark(), 53
TestJukebox.java, 73
testKitchenOrder(), 49
TestLargest.java, 16
TestLargest, 20
TestLargestDataFile, 39
TestMp3Player.java, 75, 143
TestMyStack(), 52
TestRecipe.java, 108
TestRunner, 17, 36
TestSample.java, 131

`TestSetup()`, 31
`TestSimple.java`, 26–27
`TestSkeleton.java`, 128
`TestTempServlet.java`, 71
`testURLFilter()`, 45
Thorough, 79
Time, 8, 10, 57, 121
Timeouts, 58
Tinderbox, 78, 133
Tolerance, 120
Tomcat, 69, 133
Traveling salesman algorithm, 28

U

Unit testing
    definition, 3
    intentional sabotage, 85
    potential dangers, 77
UTC, 58

V

Validation, 38
    and verification, 3, 12
    formatted data, 48
    input data, 117
    user input, 117
Version control, viii, 91

W

Wall-clock time, 58
Whac-a-Mole, 8

X

XML, 38
xUnit, 133

Z

Zero, 54

Pragmatic Starter Kit

Version control. **Unit Testing**. **Project Automation**. Three great titles, one objective. To get you up to speed with the essentials for successful project development. Keep your source under control, your bugs in check, and your process repeatable with these three concise, readable books from The Pragmatic Bookshelf.

Visit Us Online

Unit Testing In Java Home Page
`pragmaticprogrammer.com/sk/utj`
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
`pragmaticprogrammer.com/updates`
Be notified when updates and new books become available.

Join the Community
`pragmaticprogrammer.com/community`
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
`pragmaticprogrammer.com/news`
Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: `pragmaticprogrammer.com/sk/utj`.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	`www.pragmaticprogrammer.com/catalog`
Customer Service:	`orders@pragmaticprogrammer.com`
Non-English Versions:	`translations@pragmaticprogrammer.com`
Pragmatic Teaching:	`academic@pragmaticprogrammer.com`
Author Proposals:	`proposals@pragmaticprogrammer.com`