The Pragmatic Programmers

Second Edition

Pragmatic Unit Testing

In C# with NUnit

The Pragmatic Starter Kit—Volume II

Andrew Hunt   David Thomas

with Matt Hargett

Pragmatic Unit Testing

in C# with NUnit, Second Edition

Andy Hunt

Dave Thomas

with Matt Hargett

The Pragmatic Bookshelf

Raleigh, North Carolina   Dallas, Texas

ISBN-10: 0-9776166-7-3

ISBN-13: 978-0-9776166-7-4

Contents

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. Towards that end, we started our own publishing company, the Pragmatic Bookshelf. By now we've got dozens of titles in print and in development, major awards, and many five star reviews.

But the very books we published are still some of the most important ones. Before embarking on any sequels to *The Pragmatic Programmer*, 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*, 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!

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[1] 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 C# 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 C#, using the NUnit framework, the concepts remain the same whether you are writing in C++, Fortran, Ruby, Smalltalk, or VisualBasic. Testing frameworks similar to NUnit exist for over 60 different languages; these various frameworks can be downloaded for free.[2]

[1]http://www.pragmaticprogrammer.com/titles/utc2
[2]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 NUnit 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 C# 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 from the First Edition

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: Gareth Hayter, Dominique Plante, Charlie Poole, Maik Schmidt, and David Starnes.

Matt's Acknowledgments

I would like to first thank my amazing husband, Geoff, for all his patience while writing the book and contributing to various open source projects to fix issues discovered along the way. Second, gratitude to all the people who have been great pairs to program with and illuminated so much: Bryan Siepert, Strick, Mike Muldoon, Edward Hieatt, Aaron Peckham, Luis Miras, Rob Myers, Li Moore, Marcel Prasetya, Anthony Lineberry, Mike Seery, Todd Nagengast, Richard Blaylock, Andre Fonseca, Keith Dreibelbis, Katya Androchina, and Cullen Bryan. Last, I'd like to thank my mom for pair programming with me as a boy, helping to typing in very long `BASIC` programs from various magazines of the day.

Acknowledgments from the Second Edition

Thanks to all of you for your hard work and support. A special thank you goes to Matt Hargett for his contributions to this edition.

Thanks to our early reviewers, Cory Foy, Wes Reisz, and Frédérick Ros.

And since this is a beta book, watch for more acknowledgements in this space.

Andy Hunt
July, 2007
`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.

"Unit testing" is the practice of using small bits of code to exercise the code you've written. In this book, we'll be using the NUnit testing framework to help manage and run these little bits of code.

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[1] 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 completed. 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 in 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

[1]Because Dale had been integrating all along via the unit tests.

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,[2] 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.[3]

Please say that with us, out loud. Unit testing will make your life easier. That's why we're here.

It will make your designs better and drastically reduce the amount of time you spend debugging. We like to write code, and time wasted on debugging is time spent not writing code.

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.

[2] You also need to ensure that you're intending the right thing, see [SH06].

[3] It could also make you wildest dreams come true, but only if you Vote for Pedro.

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. Dale's confidence doesn't come from the fact he knows the code forward and backward at all times; it comes from the fact that he has a safety net of tests that verify things work the way he thought they should.

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 the confidence it instills makes coding 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.

And of course, executable documentation has the benefit of being self-verifiably correct without much effort beyond writing it the first time. Unlike written documentation, it won't drift away from the code (unless, of course, you stop running the tests or let them continuously fail).

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, well-known, "happy 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, and the fact you succeeded is just a coincidence.[4] 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 not particularly useful. 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 num-

[4]See *Programming by Coincidence* in [HT00].

bers of the higher-flying new rocket.[5] 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.

Current and future 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 and making sure they pass).

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.

[5]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.

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 can then write the test code itself, either before or concurrently with the implementation code. If you're writing unit tests for existing code, that's fine too, but you may find you need to refactor it more often than with new code in order to make things testable.

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. This kind of basic regression testing helps you avoid any collateral damage as well as any immediate, local 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. If you can eyeball it, you can use a code assertion to test it.

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.

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

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 easily 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. The root of the problem is usually some kind of inappropriate coupling, coming in forms such as global state via static variables or false singletons, circular object or class dependencies, etc. Eliminate them early on to avoid implicit dependencies on this abhorrent practice in other parts of the code.

unit testing until then it will definitely longer than it would otherwise. 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 by hand—you need bulldozers and lots of heavy equipment.

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 on the following page. These productivity losses can easily doom a project or developer to being perpetually 90% done.

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

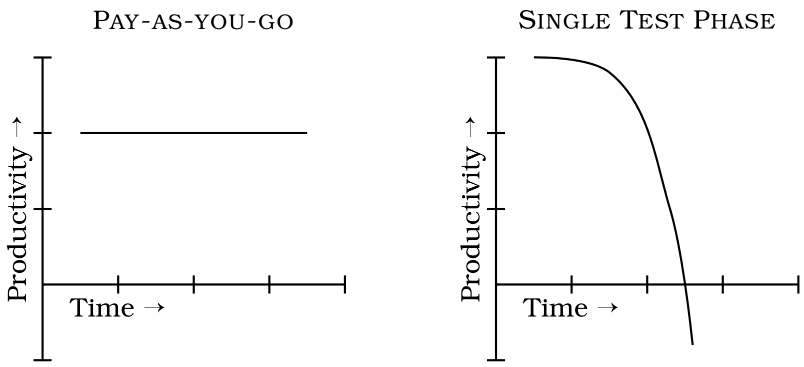


Figure 1.1: Comparison of Paying-as-you-go vs. Having a Single Testing Phase

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 can 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 in the blink of an eye, 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. NUnit has functionality that handles this nicely, which we'll talk about more later. Only run the long tests in the automated build, or manually at the beginning of the day while catching up on email, and run the shorter tests constantly at every significant change or before every commit to your source repository.

My legacy code is impossible to test Many people offer the excuse that they can't possibly do unit testing because the existing, legacy code base is such a tangled mess that it's impossible to get into the middle of it and create an individual test. To test even a small part of the system might mean you have to drag the *entire* system along for the ride, and making any changes is a fragile, risky business.[6]

The problem isn't with unit testing, of course, the problem is with the poorly written legacy code. You'll have to refactor—incrementally re-design and adapt—the legacy code to untangle the mess. Note that this doesn't really qualify as making changes just for the sake of testing. The real power of unit tests is the design feedback that, when acted upon appropriately, will lead to better object-oriented designs.

Coding in a culture of fear because you are paralyzed by legacy code is not productive; it's bad for the project, bad for the programmers, and ultimately bad for business. Introducing unit testing helps break that paralysis.

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, maintainable 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 sup-

[6]See [Fea04] for details on working effectively with legacy code.

posed to behave, then maybe this isn't the time to be writing it.[7] 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 C# compiler can easily determine that this line is wrong:

```
statuc void Main() {
```

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) {
  List myList = new List();
  myList.Add(anObject);
  myList.Add(anObject);
  // more code...
}
```

Main.cs

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.[8]

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.

[7]See [HT00] or [SH06] for more on learning requirements.

[8]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.

Yeah, we unit test already Unit testing is one of the practices that is typically marked by effusive and consistent enthusiasm. If the team isn't enthusiastic, maybe they aren't doing it right. See if you recognize any of the warning signs below.

- Unit tests are in fact integration tests, requiring lots of setup and test code, taking a long time to run, and accessing resources such as databases and services on the network.
- Unit tests are scarce and test only one path, don't test for exceptional conditions (no disk space, etc.), or don't really express what the code is supposed to do.
- Unit tests are not maintained: tests are ignored (or deleted) forever if they start failing, or no new unit tests are added, even when bugs are encountered that illustrate holes in the coverage of the unit tests.

If you find any of these symptoms, then your team is not unit testing effectively or optimally. Have everyone read up on unit testing again, go to some training, or try pair programming to get a fresh perspective.

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 NUnit* 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, followed by 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 legacy code.

We'll then talk about how testing can influence your application's design (for the better) in Chapter 9, *Design Issues*. We then wrap up with an overview of GUI testing in 10.

The appendices contain additional useful information: a look at common unit testing problems, extending NUnit itself, a note on installing NUnit, 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 `IsTrue` checks that the given boolean condition is true, and fails the current test if it is not. It might be implemented like the following.

```
public void IsTrue(bool condition)
{
  if (!condition)
  {
    throw new ArgumentException("Assertion failed");
  }
}
```

AssertTrue.cs

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

```
int a = 2;
xx xxx xx x xxx x;
x x x xx xxx xxxx x;
IsTrue(a == 2);
xxxx xx xx xxx xx;
```

If for some reason `a` does not equal `2` when the method `IsTrue` is called, then the program will throw an exception.

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:

```
public void AreEqual(int a, int b)
{
  IsTrue(a == b);
}
```

AssertTrue.cs

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.

This isn't a comprehensive list by any means, but it's good enough to get started with. To help make all this discussion more concrete, we'll write a "largest" method and test it using these unit tests we just described. Here's the code for our first implementation:

```csharp
using System;

public class Cmp
{
  public static int Largest(int[] list)
  {
    int index, max=Int32.MaxValue;
    for (index = 0; index < list.Length-1; index++)
    {
      if (list[index] > max)
      {
        max = list[index];
      }
    }
    return max;
  }

}
```

Largest.cs

Now that we've got some ideas for tests, we'll look at writing these tests in C#, using the NUnit 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, assembly references, 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:

```
using System;
using NUnit.Framework;
using NUnit.Framework.SyntaxHelpers;

[TestFixture]
public class LargestTest
{
  [Test]
  public void LargestOf3()
  {
    Assert.That(Cmp.Largest(new int[] {8,9,7}), Is.EqualTo(9));
  }
}
```

LargestTest.cs

C# 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):

```
  [Test]
  public void LargestOf3Alt()
  {
    int[] arr = new int[3];
    arr[0] = 8;
    arr[1] = 9;
    arr[2] = 7;
    Assert.That(Cmp.Largest(arr), Is.EqualTo(9));
  }
```

LargestTest.cs

That's all it takes, and you have your first test.

We want to run this simple test and make sure it passes; to do that, we need to take a quick look at running tests using NUnit.

2.3 Running Tests with NUnit

NUnit is a freely available,[1] open source product that provides a testing framework and test runners. It's available as C# source code that you can compile and install yourself, and as a ZIP file of the binaries. The binaries in the ZIP will run on Microsoft .NET on Windows, and possibly other .NET implementations on Linux/UNIX or MacOS X. There is also an MSI package available, but we recommend just using the ZIP file for the least amount of hassle.

Linux and MacOS users may want to look at Mono, an open-source implementation of the ECMA standards upon which C# and .NET are based. While mono ships with its own version of NUnit, we recommend referencing your own copy of NUnit, downloaded separately. This will insulate you from changes to the version of NUnit distributed by the mono team. We discuss more of these project-oriented details in Chapter 8.

Next, you need to compile the code we've shown. If you're using Visual Studio or SharpDevelop, create a new project for this sample code of type *Class Library*. Type our "production" code into a file named `Largest.cs`, and our new test code into a file named `LargestTest.cs`. If you'd rather not type these programs in from scratch, you'll be pleased to know that all of the source code for this book is available from our website.[2])

Notice that the test code uses `NUnit.Framework`; you'll need to add a reference to `nunit.framework.dll` in order to compile this code. In Visual Studio or SharpDevelop, expand the project's node in the Solution Explorer, bring up the context menu on the References folder, then select "Add Reference...". Once there, browse to the `nunit.framework.dll` from the NUnit install directory. Press the `SELECT` button to add the dll to the component list as shown in Figure 2.1. Press `OK`, and now your project will be able to use the functionality of the NUnit framework.

Go ahead and build the project as you normally would (In

[1] http://www.nunit.org
[2] http://www.pragmaticprogrammer.com/titles/utc2

Joe Asks...

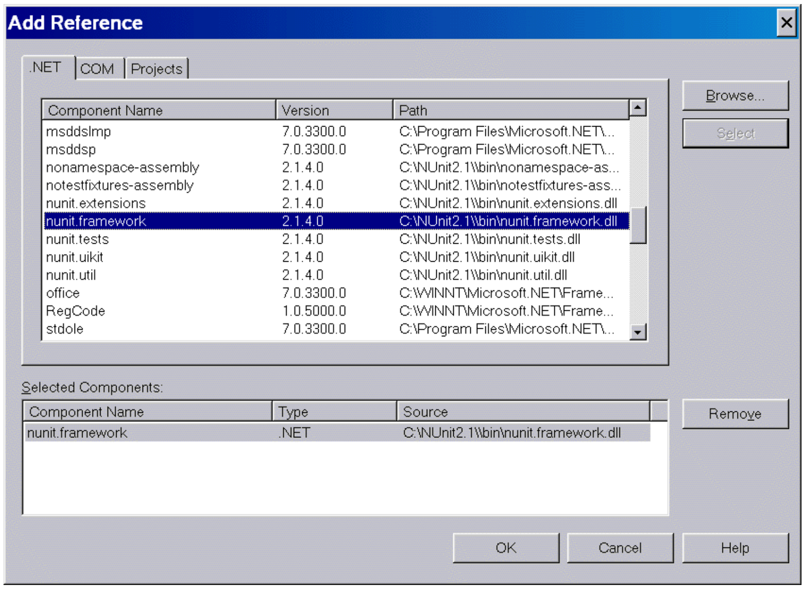
What's the deal with Open Source?

What is open source, exactly? *Open source* refers to software where the source code is made freely available. Typically this means that you can obtain the product for free, and that you are also free to modify it, add to it, give it to your friends, and so on.

Is it safe to use? For the most part, open source products are safer to use than their commercial, closed-source counterparts, because they are open to examination by thousands of other interested developers. Malicious programs, spyware, viruses, and other similar problems are rare to non-existent in the open source community.

Is it legal? Absolutely. Just as you are free to write a song or a book and give it away (or sell it), you are free to write code and give it away (or sell it). There are a variety of open source licenses that clarify the freedoms involved. Before you distribute any software that includes open source components, you should carefully check the particular license agreements involved.

Can I contribute? We certainly hope so! The strength of open source comes from people all over the world: People just like you, who know how to program and have a need for some particular feature. Would you like to add a feature to NUnit? You can! You can edit the source code to the library or one of the test runners and change it, and use those changes yourself. You can e-mail your changes to the maintainers of the product, and they may even incorporate your changes into the next release. You can also submit changes using patch tracker on `sourceforge.net`; that way, even if your change is not included in an official release, other users can take advantage of it.



Component Name	Version	Path
msddslmp	7.0.3300.0	C:\Program Files\Microsoft.NET\...
msddsp	7.0.3300.0	C:\Program Files\Microsoft.NET\...
nonamespace-assembly	2.1.4.0	C:\NUnit2.1\bin\nonamespace-as...
notestfixtures-assembly	2.1.4.0	C:\NUnit2.1\bin\notestfixtures-ass...
nunit.extensions	2.1.4.0	C:\NUnit2.1\bin\nunit.extensions.dll
nunit.framework	2.1.4.0	C:\NUnit2.1\bin\nunit.framework.dll
nunit.tests	2.1.4.0	C:\NUnit2.1\bin\nunit.tests.dll
nunit.uikit	2.1.4.0	C:\NUnit2.1\bin\nunit.uikit.dll
nunit.util	2.1.4.0	C:\NUnit2.1\bin\nunit.util.dll
office	7.0.3300.0	C:\WINNT\Microsoft.NET\Frame...
RegCode	1.0.5000.0	C:\WINNT\Microsoft.NET\Frame...
stdole	7.0.3300.0	C:\Program Files\Microsoft.NET\...

Component Name	Type	Source
nunit.framework	.NET	C:\NUnit2.1\bin\nunit.framework.dll

Figure 2.1: Adding NUnit Assembly Reference

Visual Studio, `CTRL-SHIFT-B` works well). Using Mono, you'd invoke the compiler using something such as:

```
gmcs -debug -t:library -r:System -r:lib/nunit.framework.dll \
                                     -out:Largest.dll Largest.cs LargestTest.cs
```

(The reference to `nunit.framework.dll` will of course be the location where you copied the NUnit distribution.)

Now you've got an assembly. But it's just a library. How can we run it?

Test Runners to the rescue! A test runner knows to look for the `[TestFixture]` attribute of a class, and for the `[Test]` methods within it. The runner will run the tests, accumulate some statistics on which tests passed and failed, and report the results back to you. In this book, we focus on test runners that are easily accessible and freely available.

There are four main ways to use a test runner:

1. NUnit GUI (all platforms)
2. NUnit command line (all platforms)

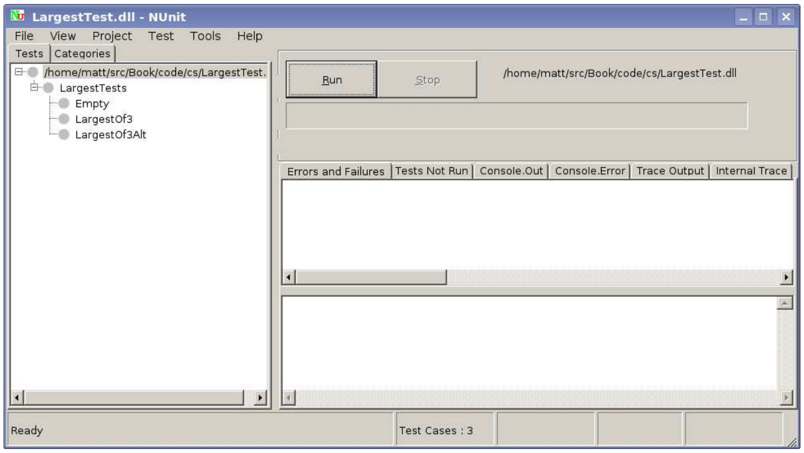


Figure 2.2: NUnit Loaded and Ready

3. TestDriven.NET (Windows-only)
4. SharpDevelop 2.1 runner (Windows-only)
5. MonoDevelop 0.13 runner (all platforms)

NUnit GUI

The NUnit GUI can be started a number of ways: if you unzipped the binaries on Windows, you can just point Windows Explorer at the directory and double-click on `nunit.exe`. If you unzipped the binaries on MacOS or Linux, you can run NUnit GUI via the mono runtime executable (using `mono -debug nunit.exe`). If you used the Windows installer, you can use the shortcuts on your Windows desktop and in the Programs menu of the Start Menu to start the NUnit GUI.

When the GUI comes up, you've got a couple of choices. You can create a new NUnit project as shown in Figure **??** on page ??; navigate to your source directory and create the NUnit project file. Then under the "Project" menu, add assemblies or Visual Studio projects to your NUnit project.[3]

[3]Visual Studio support can be enabled using a preference located under Tools/Options.

Alternatively, you can just Open an assembly (a .dll or .exe file) directly. In Figure 2.2 on the preceding page, we've loaded our tests directly from the dll. It's ready to be tested by pressing the "Run" button.

When you run a selected test, the GUI will display a large, colored, status bar. If all the tests pass, the bar is a happy shade of bright green. If any test fails, the bar becomes an angry red. If the bar is a cautionary yellow, that means some tests were skipped (more on that later).

NUnit Command Line

NUnit can also be run from the command line, which comes in very handy when automating the project build and test. You'll need to add the NUnit bin directory to your path (that is, the directory path to wherever you installed the NUnit application, plus "\bin").

For the current shell, you can set your path variable at the command line, as in the following example on Windows.

```
C:\> set "PATH=%PATH%;C:\Program Files\Nunit V2.4\bin"
```

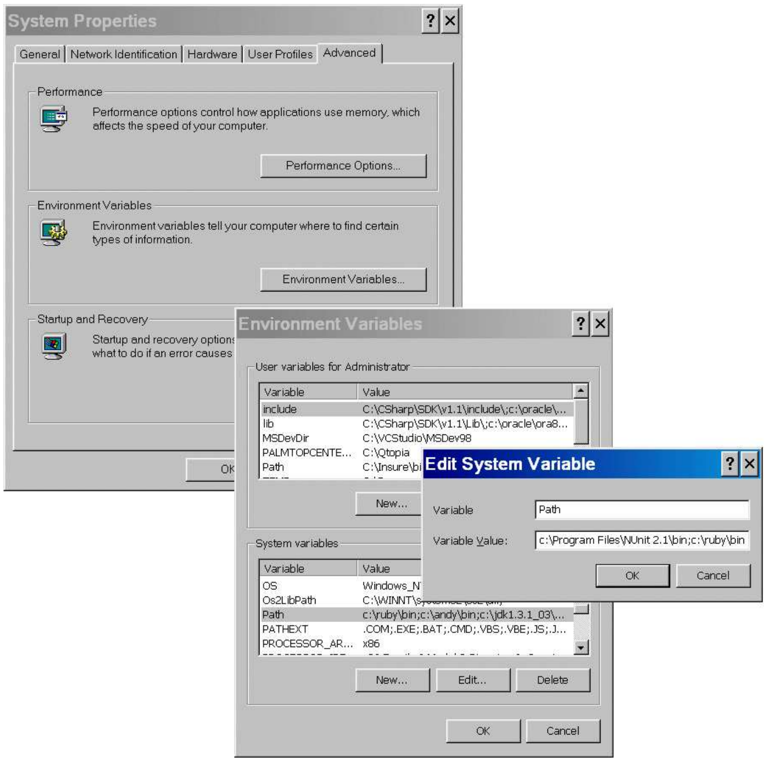
For more permanent use, go to Control Panel/System/Advanced/Environment Variable and add NUnit's bin directory to the Path variable (see Figure 2.3 on the next page).

To run from the command line, type the command nunit-console followed by an NUnit project file or an assembly location. You'll see output something like that shown in Figure 2.4 on page 26.

TestDriven.NET (Visual Studio add-in)

There are several add-ins that integrate NUnit with Visual Studio. The TestDriven.NET[4] add-in adds the ability to run or debug any test just by right-clicking on the source code and selecting "Run Test(s)"; the output from the tests are reported in Visual Studio's output pane, just like compiler warnings or

[4]Such as http://www.testdriven.net/



System Properties

General | Network Identification | Hardware | User Profiles | Advanced

Performance

Performance options control how applications use memory, which affects the speed of your computer.

Performance Options...

Environment Variables

Environment variables tell your computer where to find certain types of information.

Environment Variables...

Startup and Recovery

Startup and recovery options what to do if an error causes

Environment Variables

User variables for Administrator

Variable	Value
include	C:\CSharp\SDK\v1.1\include\;c:\oracle\...
lib	C:\CSharp\SDK\v1.1\Lib\;c:\oracle\ora8...
MSDevDir	C:\VCStudio\MSDev98
PALMTOPCENTE...	C:\Qtopia
Path	C:\Insure\bi

System variables

Variable	Value
OS	Windows_N
Os2LibPath	C:\WINNT\s
Path	c:\ruby\bin;c:\andy\bin;c:\jdk1.3.1_03\...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.J...
PROCESSOR_AR...	x86

Edit System Variable

Variable: Path

Variable Value: c:\Program Files\NUnit 2.1\bin;c:\ruby\bin

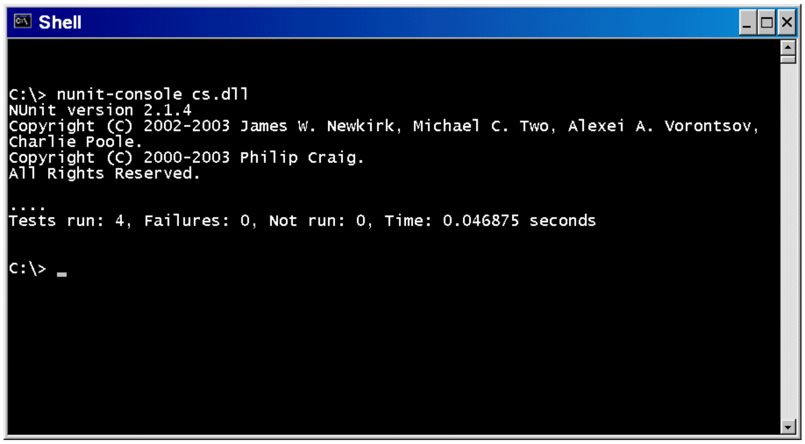
Figure 2.3: Adding to the Windows System Path

errors. You can use this output to quickly browse to failed assertion locations, which is quite handy. Other similar projects add visual reporting of tests and other features.

SharpDevelop

SharpDevelop 2.1 (and above), an open-source IDE written in C#, includes an Eclipse-style integrated test runner. Failed tests come up like compiler errors, allowing for double-clicking on an item and going to the assertion that failed. It also allows for measuring the code coverage of unit tests (using NCover[5]) with source code highlighting that can be en-

[5]http://NCover.org

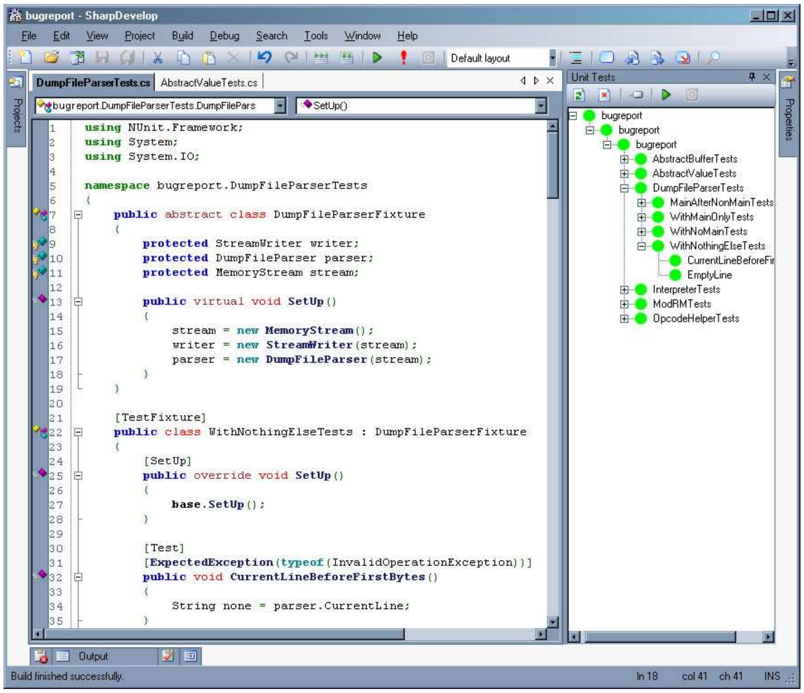


```
C:\> nunit-console cs.dll
NUnit version 2.1.4
Copyright (C) 2002-2003 James W. Newkirk, Michael C. Two, Alexei A. Vorontsov,
Charlie Poole.
Copyright (C) 2000-2003 Philip Craig.
All Rights Reserved.

....
Tests run: 4, Failures: 0, Not run: 0, Time: 0.046875 seconds

C:\> _
```

Figure 2.4: NUnit Command Line Usage



```
using NUnit.Framework;
using System;
using System.IO;

namespace bugreport.DumpFileParserTests
{
    public abstract class DumpFileParserFixture
    {
        protected StreamWriter writer;
        protected DumpFileParser parser;
        protected MemoryStream stream;

        public virtual void SetUp()
        {
            stream = new MemoryStream();
            writer = new StreamWriter(stream);
            parser = new DumpFileParser(stream);
        }
    }

    [TestFixture]
    public class WithNothingElseTests : DumpFileParserFixture
    {
        [SetUp]
        public override void SetUp()
        {
            base.SetUp();
        }

        [Test]
        [ExpectedException(typeof(InvalidOperationException))]
        public void CurrentLineBeforeFirstBytes()
        {
            String none = parser.CurrentLine;
        }
```

Figure 2.5: SharpDevelop's Integrated Unit Testing

abled and disabled. A sample screenshot is shown in Figure 2.5 on the previous page. See SharpDevelop's web page for more details (http://sharpdevelop.net).

MonoDevelop

MonoDevelop 0.13 and above, which is based on SharpDevelop 0.9, also includes an integrated test runner. While not as advanced as SharpDevelop itself, it's a welcome improvement over a flat text editor on platforms where other tools don't run. For more information, see MonoDevelop's web page (http://monodevelop.com).

2.4 Running the Example

You should be ready to run this first test now.

Try running this example before reading on. . .

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

```
Failures:
1) LargestTest.LargestOf3 :
        expected:<9>
         but was:<2147483647>
   at LargestTest.LargestOf3() in c:\largesttests.cs:line 13
```

Whoops! That didn't go as expected. Why did it return such a huge number instead of our 9? Where could that very large number have come from? It almost looks like the largest number. . . oh, it's a small typo: `max=Int32.MaxValue` on line 7 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. Notice that just as in production (non-test) code, you have to exercise care, taste, and restraint when deciding how much code to add to one method, and when to break that up into multiple methods. Since this method is testing variations on a single theme (physical placement of the largest value), let's put them together in a single method.

We already have the case with the largest in the middle:

```
using System;
using NUnit.Framework;
using NUnit.Framework.SyntaxHelpers;

[TestFixture]
public class LargestTest
{
  [Test]
  public void LargestOf3()
  {
    Assert.That(Cmp.Largest(new int[] {8,9,7}), Is.EqualTo(9));
  }
}
```

LargestTest.cs

Now try it with the 9 as the first value (we'll just add an additional assertion to the existing `LargestOf3()` method):

```
  [Test]
  public void LargestOf3()
  {
    Assert.That(Cmp.Largest(new int[] {9,8,7}), Is.EqualTo(9));
    Assert.That(Cmp.Largest(new int[] {8,9,7}), Is.EqualTo(9));
  }
```

LargestTest.cs

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

```
  [Test]
  public void LargestOf3()
  {
    Assert.That(Cmp.Largest(new int[] {9,8,7}), Is.EqualTo(9));
    Assert.That(Cmp.Largest(new int[] {8,9,7}), Is.EqualTo(9));
    Assert.That(Cmp.Largest(new int[] {7,8,9}), Is.EqualTo(9));
  }
```

LargestTest.cs

Try running this example before reading on...

STOP

```
Failures:
1) LargestTest.LargestOf3 :
        expected:<9>
         but was:<8>
   at LargestTest.LargestOf3() in c:\LargestTest.cs:line 14
```

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#), but as you can see, it's prone to off-by-one errors. Make the changes and run the tests again, but consider that this sort of bug is telling you something: it would be better to use an iterator (using the C# foreach statement) here instead. That way you could avoid this kind of off-by-one error in the future.

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

```
[Test]
public void Dups() {
  Assert.That(Cmp.Largest(new int[] {9,7,9,8}), Is.EqualTo(9));
}
```

LargestTest.cs

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

```
[Test]
public void One() {
   Assert.That(Cmp.Largest(new int[] {1}), Is.EqualTo(1));
}
```

LargestTest.cs

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:

```
[Test]
public void Negative() {
  int[] negatives = new int[] {-9, -8, -7};
  Assert.That(Cmp.Largest(negatives), Is.EqualTo(-7));
}
```

LargestTest.cs

Try running this example before reading on. . .

```
Failures:
1) LargestTest.Negative :
        expected:<-7>
         but was:<0>
   at LargestTest.Negative() in c:\LargestTest.cs:line 4
```

Whoops! Where did zero come from?

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

```
    max = Int32.MinValue
```

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=Int32.MinValue;
    if (list.Length == 0) {
      throw new ArgumentException("largest: Empty list");
    }
    // ...
```

Largest.cs

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 54, but for now just trust us:

```
  [Test]
  [ExpectedException(typeof(ArgumentException))]
  public void Empty()
  {
    Cmp.Largest(new int[] {});
  }
```

LargestTest.cs

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.

2.5 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 NUnit.

Chapter 3

Writing Tests in NUnit

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 NUnit's assertions, the structure and composition of NUnit tests, and so on.

3.1 Structuring Unit Tests

Suppose we have a method named `CreateAccount`; the method encapsulates behaviour, and it's behaviour that we want to test. Your first test method might be named something like `CreateSimpleAccount`. The method `CreateSimpleAccount` 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` (not all accounts are simple, after all). Tests should be organized around behaviours, not necessarily individual methods.

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

The test code is for our internal use only; customers or end-users will generally 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 not know anything about the test code. Production code will be thrust out into

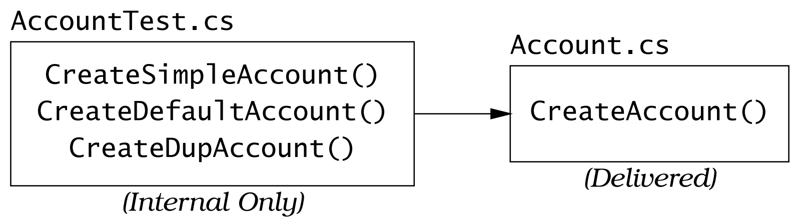


Figure 3.1: Test Code and Production Code

the cold world all alone, without the test code. This typically means that test code is placed under a different project, in its own assembly.

Test code follows a standard formula:

- 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 tested functionality worked as expected
- Clean up after itself[1]

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 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.

Now, although we *could* write all our tests from the ground up, that's not terribly efficient. For the rest of this book we'll assume that you're using the NUnit framework. More specifically, we'll be showing the specific method calls and classes for NUnit 2.4, using C#, in our examples. Earlier or later ver-

[1]This doesn't mean nulling out fields or using `GC.Collect()`. If you find yourself doing either, you may have a race condition due to a misbehaving Finalizer. These issues are almost never limited to test code.

sions may have slight differences from the details presented here, but the general concepts are the same across all versions, and indeed for any testing framework in any language or environment.

3.2 Classic Asserts

As we've seen, there are some helper methods that assist us in determining whether a method under test is performing correctly or not. Generically, we call all these helper methods *assertions*. They let us assert that some condition is true; that two bits of data are equal, or not, and so on. NUnit 2.4 introduced a new constraint-style of assertions while still supporting the classic-style of assertions that more closely match other XUnit frameworks. We'll start off by covering some basic classic-style assertions before diving into the constraint-style assertions.

All of the following methods will report failures (that's when the assertion is false) or errors (that's when we get an unexpected exception), and report these through the NUnit test runner. For the text version of the test runner, that means the details of the failure will be printed to the console. The GUI versions of the test runner will show a red bar and supporting details to indicate a failure. You can also output the test results to an XML file.

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

Asserts are the fundamental building block for unit tests; the NUnit library provides a number of different forms of assert as static methods in the `Assert` class.

AreEqual

```
Assert.AreEqual(expected, actual [, string message])
```

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. We recommend omitting the *message* string for reporting unless you really need to; better that the name of the test method itself expresses your intent, you use the appropriate `Assert` method, or you split the test into two methods to keep it focused. We'll show examples of all of these practices in a bit.

Any kind of object may be tested for equality; the appropriate equals method will be used for the comparison.[2] In particular, you can compare the contents of strings using this method. Different method signatures are also provided for all the native types (`int`, `decimal`, etc.) and `Object`. Strings and Collections also have their own classic-style asserter classes with extra methods, `StringAssert` and `CollectionAssert`, which we'll get into a bit later.

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 C#), you need to specify one additional piece of information, the tolerance. This specifies just how close to "equals" you need the result to be.

```
Assert.AreEqual(expected,
                actual,
                tolerance [, string message])
```

For business applications, 4 or 5 decimal places is probably enough. For scientific apps, you may need greater precision.

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

```
Assert.AreEqual(3.33, 10.0/3.0, 0.01);
```

Less / Greater

```
Assert.Less(x, y)
Assert.Greater(x,y)
```

[2]Remember that the default `Equals()` inherited from `System.Object` only checks to see if the object references themselves are the same—it checks for identity, rather than equality. For value types (structs, enums, etc.) the fields are verified to be equal [Ric06].

Asserts that x < y (or x > y) for numeric types, or any type that is `IComparable`.

GreaterOrEqual / LessOrEqual

```
Assert.GreaterOrEqual(x, y)
Assert.LessOrEqual(x,y)
```

Asserts that x >= y (or x <= y) for numeric types, or any type that is `IComparable`.

IsNull / IsNotNull

```
Assert.IsNull(object [, string message])
Assert.IsNotNull(object [, string message])
```

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

AreSame

```
Assert.AreSame(expected, actual [, string message])
```

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

IsTrue

```
Assert.IsTrue(bool condition [, string message])
```

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:

```
Assert.IsTrue(true);
```

then you should 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 `Assert.IsTrue(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`:

```
Assert.IsFalse(bool condition [, string message])
```

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

Neither `IsTrue` nor `IsFalse` give you any additional information when the test fails; this means you might have to use the debugger or `Console.WriteLine()` statements to diagnose a unit test failure. That's not very efficient. There might be a better assertion you could use, such as `StringAssert.Contains()` or `CollectionAssert.DoesNotContain()`—we'll take a look at these more interesting assertions in just a moment. A more precise assertion like those will give you more precise information on failure so you can concentrate on fixing the code rather than trying to figure out what went wrong.

Fail

```
Assert.Fail([string message])
```

Fails the test immediately, with the optional message. This might be used to mark sections of code that should not be reached, but isn't really used much in practice.

3.3 Constraint-based Asserts

NUnit 2.4 introduced a new style of assertions that are a little less procedural and allow for a more object-oriented underlying implementation. NUnit has a history of innovating on the classic XUnit design, which other frameworks then incorporate later. In this case the NUnit team decided to mimic another innovative framework called NMock2,[3] which we'll discuss later in Chapter 6.

This new assertion style can seem a little odd at first, but we suggest giving it a chance before falling back on the "classic" assertion methods. After all, the classic assertion methods just delegate to the constraint-style assertion methods behind the covers. Let's look at a couple of assertions as they would be written in the new style.

[3]NMock2, in turn, was mimicking jMock.

Joe Asks...

What was wrong with the old syntax?

Well, nothing was particularly wrong with the classic syntax, per se. In fact, there are no plans to remove or deprecate the classic syntax. The classic-style assert methods delegate to the new methods, so there's no duplication. Here's a quick history lesson that may illuminate the progression.[a] In the beginning, test fixture classes had to derive from a class called `TestCase`. Deriving from `TestCase` both told the test runner which classes contained test methods and provided assertion methods, amongst other things.

In those days, we would call `assertEquals()` and other assertions, which were inherited from `TestCase`, from our test methods. The `TestCase` class was also reponsible for providing a `virtual setUp()` and `tearDown()` method. Clearly, the `TestCase` class was a bit overloaded as far as its reponsibilities.

First, NUnit used attributes to mark test fixture classes, as previously discussed. Then, NUnit extracted the growing list of assertion methods into the family of `Assert` classes. This effectively eliminated the `TestCase` class altogether. Several other XUnit frameworks have picked up these ideas in their recent versions. This brings us up to NUnit 2.2.

While developing NUnit 2.4, the NUnit team realised that the `Assert` classes had a few too many reponsibilities. The `Assert` classes had to make sure the actual value matched the expected value, whatever that meant for the given assertion method. On top of this, the `Assert` class needed to format the text to be output by the test runner when the assertion failed.

These responsibilities were broken up, with the `Constraint` objects (returned by syntax helpers such as `Is.EqualTo()`) bearing the responsibility of making sure the actual value met the context-specific constraint of the expected value. Because they are encapsulated in separate objects, multiple constraints can be combined and applied to a single value. That leaves the text formatting when an assertion fails, which falls to the `TextMessageWriter` object that NUnit uses internally.

Is.EqualTo

```
Assert.That(actual, Is.EqualTo(expected))
```

This is equivalant to the `Assert.AreEqual()` classic assertion method we discussed in the last section. The `Is.EqualTo()` method is a syntax helper in the `NUnit.Framework.SyntaxHelpers` namespace. It's a static method that just returns an `EqualConstraint` object. The following code is equivalant, but may not read as smoothly to some folks.

```
Assert.That(actual, new EqualConstraint(expected))
```

To specify a tolerance for floating point numbers like we did previously, we can use a neat feature of the new syntax called *constraint modifiers*. There are several that we'll look at, but here is one called `Within()` that is equivalant to our same example that used the classic-style in the previous section.

```
Assert.That(10.0/3.0, Is.EqualTo(3.33).Within(0.01f));
```

Is.Not.EqualTo

```
Assert.That(actual, Is.Not.EqualTo(expected))
```

This is an example of one of the fun things that the constraint-based syntax allows for and is equivalant to the `Assert.AreNotEqual()` classic assertion that was discussed previously. The usage of `Not` in this context isn't exactly a separate method, as in the other examples. By applying `Not`, it wraps the `EqualConstraint` in a `NotConstraint` object. The following code is equivalant.

```
Assert.That(actual, new NotConstraint(new EqualConstraint(expected)));
```

We can apply `Not` to any `Is` or `Has` syntax helper. As such, you could also wrap the `NotConstraint` object around any other `Constraint` object. Given the verbosity that entails, though, we're probably better off using the syntax helper approach.

Is.AtMost

```
Assert.That(actual, Is.AtMost(expected))
```

This constraint-style assert is equivalant to the `Assert.LessOrEqual()` classic assertion method.

`Is.AtMost()` is just an alias for `Is.LessThenOrEqualTo()`, which returns a `LessThanOrEqualConstraint` object.

Is.Null

```
Assert.That(expected, Is.Null);
```

Asserts that *expected* is `null`, and fails the test if it is not. To assert the opposite, we have two choices of constraint-style syntax.

```
Assert.That(expected, Is.Not.Null);
```

```
Assert.That(expected, !Is.Null);
```

Either of these ways will wrap the constraint in a `NotConstraint` object under the covers. Either style can be applied to any of the constraints. Neat, huh?

Is.Empty

```
Assert.That(expected, Is.Empty);
```

Asserts that *expected* is an empty collection or string, and fails the test if it is not.

Is.AtLeast

```
Assert.That(actual, Is.AtLeast(expected));
```

This is equivalant to `Is.GreaterThanOrEqualTo()`, which asserts that *actual* >= *expected* (or *expected* <= *actual*) for numeric types, or any type that is `IComparable`.

Is.InstanceOfType

```
Assert.That(actual, Is.InstanceOfType(expected));
```

Asserts that *actual* is of type *expected*, or a derivation of that type.

Has.Length

```
Assert.That(actual, Has.Length(expected));
```

Asserts that *actual* has a `Length` property that returns the expected value. Note that it can be any object with a property named “Length”, not just a `string` or `Collection`. We could also just assert the length using `Is.EqualTo()`, but this may be easier to read for some.

In the rest of the examples, we’ll be using this new constraint-style of assertions. If you’re more comfortable with the classic-style, feel free to substitute those into the appropriate places instead.

Using Asserts

We usually have multiple asserts in a given test method, as we 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; we have to fix the failing test before we can proceed anyway. And we 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 we introduce a bug, only one or two tests fail. Isolating the problem is usually pretty easy in that environment.

Under no circumstances should we 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, we’ll need an easy way to run all the tests—or to run groups of tests, particular subsystems, and so on.

3.4 NUnit 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 we need to get started.

```
Line 1  using System;
     -  using NUnit.Framework;
     -  using NUnit.Framework.SyntaxHelpers;
     -
     5  [TestFixture]
     -  public class LargestTest
     -  {
     -    [Test]
     -    public void LargestOf3Alt()
    10    {
     -      int[] arr = new int[3];
     -      arr[0] = 8;
     -      arr[1] = 9;
     -      arr[2] = 7;
    15      Assert.That(Cmp.Largest(arr), Is.EqualTo(9));
     -    }
     -  }
```

LargestTest.cs

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

First, the `using` statement on line **??** brings in the necessary NUnit classes. Remember we'll need to tell the compiler you're referencing `nunit.framework.dll`, otherwise the `using` statement won't be able to find the `NUnit.Framework` namespace.

Next, we have the class definition itself on line **??**: each class that contains tests must be annotated with a `[TestFixture]` attribute as shown. The class must be declared `public` (so that the test runners will run it; by default, classes are `internal`), and it must have a public, no-parameter, constructor (the default implicit constructor is all we need—adding a constructor to a `TestFixture` is generally not necessary).

Finally, the test class contains individual methods annotated with `[Test]` attributes. In the example, we've got one test method named `LargestOf3` on line **??**. Any public, parameterless method specified with a `[Test]` attribute will be run automatically by NUnit. We can include helper methods to support clean code in our tests as well, we just don't mark them as tests.

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:

```
using System;
```

Joe Asks. . .

What's a Fixture?

From the c2.com wiki:[a]

In electronics testing, a fixture is an environment in which you can test a component. Once the circuit board or component is mounted in the text fixture, it is provided with the power and whatever else is needed to drive the behaviour to be tested.

A fixture in the context of unit testing is more about the scenario we're testing than the actual class we're testing. Testing a single class across multiple fixtures is very common.

[a]http://c2.com/cgi/wiki?TestFixture

```
using NUnit.Framework;
using NUnit.Framework.SyntaxHelpers;
[TestFixture]
public class LargestTest
{
  [Test]
  public void LargestOf3()
  {
    Assert.That(Cmp.Largest(new int[] {9,8,7}), Is.EqualTo(9));
    Assert.That(Cmp.Largest(new int[] {8,9,7}), Is.EqualTo(9));
    Assert.That(Cmp.Largest(new int[] {7,8,9}), Is.EqualTo(9));
  }
```

LargestTest.cs

Here we have three calls to `Assert.That` inside a single test method.

3.5 NUnit Test Selection

As we've seen so far, a fixture (that is, a class marked with the `[TestFixture]` attribute) contains test methods; each method contains one or more assertions. Multiple test fixtures can be included into a source code file or a compiled assembly.

You will normally run all of the tests within an assembly just

Organizing fixtures

Following good object-oriented design, a class should be focused on one responsibility. This applies to test fixtures as well—they're just classes, after all. As such, put tests into a fixture that describes the specific scenario they are being tested in.

If there aren't multiple scenarios, then just name the fixture class after the class being tested. You can always extract more focused fixtures from a general fixture once the general fixture starts getting too fat.

Having a fixture class focused on a specific scenario, with a name that documents that scenario, helps avoid duplicating the scenario description in the name of several test methods.

To keep things readable in the test runner output, put the fixture classes under a namespace that includes the name of the class that the fixtures are testing, like so:

```
namespace ZeroBay.Test.ShoppingCartTest
{
  [TestFixture]
  public class NoDataFixture
  {
    [Test]
    public void OverallRateIsZero() {...}
  }
}
```

by specifying the assembly to the test runner. You can also choose to run individual test fixtures within an assembly using either the NUnit command line or GUI.

From the GUI, you can select an individual test, a single test fixture, or the entire assembly by selecting it and clicking the run button, and all the appropriate tests will be run.

From the command line, you can specify the assembly and a particular test fixture as follows:

```
c:\> nunit-console  assemblyname.dll  /fixture:ClassName
```

Given this flexibility, you may want to think a bit about how to organize test methods into individual assemblies and fixtures to make testing easier.

For instance, you may want to run all the database-related tests at once, or all of the tests that Fred wrote (Fred is still on probation from the last project, and you want to keep an eye on him).

Fortunately, NUnit has a mechanism you can use to categorize and classify individual test methods and fixtures.

Categories

NUnit provides an easy way to mark and run individual tests and fixtures by using *categories*. A category is just a name that you define. You can associate different test methods with one or more categories, and then select which categories you want to exclude (or include) when running the tests.

Suppose among your tests you've got a method 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. That means that 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.

You can use NUnit categories to help sort out your usual tests that you can run constantly versus long-running tests that you'd rather only run during the automated build. Categories are generally used for exclusion rather than inclusion.

A category is specified as an attribute. You provide a string to identify the category when you declare the method. Then when you run the tests, you can specify which categories you want to run (you can specify more than one).

For instance, suppose you've got a few methods that only take a few seconds to run, but one method that takes a long time to run. You can annotate them using the category names "Short" and "Long" (you might also consider making a category "Fred" if you still want to keep an eye on him.)

```
using NUnit.Framework;
```

```
 -   using NUnit.Framework.SyntaxHelpers;
 -
 -   [TestFixture]
 5   public class ShortestPathTest
 -   {
 -     TSP tsp;
 -
 -     [SetUp]
10     public void SetUp()
 -     {
 -       tsp = new TSP();
 -     }
 -
15     [Test]
 -     [Category("Short")]
 -     public void Use5Cities()
 -     {
 -       Assert.That(tsp.ShortestPath(5), Is.AtMost(140));
20     }
 -
 -     // This one takes a while...
 -     [Test]
 -     [Category("Long")]
25     [Category("Fred")]
 -     public void Use50Cities()
 -     {
 -       Assert.That(tsp.ShortestPath(50), Is.AtMost(2300));
 -     }
30   }
```

ShortestPathTest.cs

Notice that you can specify multiple attributes (in this case, `Test` and `Category`) on two separate lines as shown around line 26, or combined into one line.

Now if you choose to run just “Short” methods, the two methods `Use2Cities` and `Use10Cities` will be selected to run. If you choose “Long” methods, only `Use50Cities` will be selected. You can also select both categories to run all three of these methods.

In the GUI, you select which categories of tests to include and which to exclude on the tab as shown in Figure 3.2 on the following page. Just select each category you’re interested and press the ADD button.

On a real project, of course, you wouldn’t bother to mark a bunch of tests as “short.” They should *all* be short, except for the ones specifically marked as “Long.”

From the command line, you can specify individual categories to include as well. Just add the following parameter to the

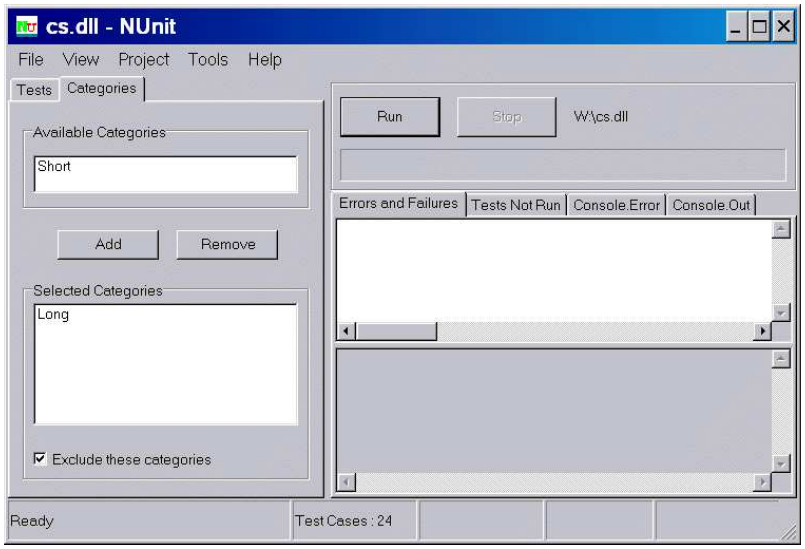


Figure 3.2: NUnit Category Selection

command line:

```
/include=category1;category2;...
```

Note that multiple category names are separated by a semi-colon (";").

You can also choose to *exclude* the listed categories so all other tests except those in the named categories run. There's a check box in the GUI for this; the command line option is, oddly enough, `/exclude`.

But this isn't quite enough: it turns out that some categories of tests should be run when no categories are selected, while others should run only when explicitly selected.

To support this, you can specify the `Explicit` attribute:

```
[Explicit("SpecialEquipmentNeeded")]
```

This syntax automatically excludes the category from a run that doesn't specify any categories. By default, your run will include tests without categories and tests with non-explicit categories. However, if even one category is specified in the

GUI or the command line, then only that single category will be run.

There's a danger here, of course—these tests aren't running all the time. They probably aren't being run in the automated build system, either. This might lull you into a false sense of security, so beware.

In addition to marking individual test methods as belonging to a category, you can also mark entire fixtures. For instance, if we wanted to flag our entire test fixture as long-running (without having to mark each and every test method), we could do so.

```
using NUnit.Framework;
using NUnit.Framework.SyntaxHelpers;

[TestFixture]
[Category("Long")]
public class ShortestPathTest-Revised
{
  TSP tsp;

  [Test]
  public void Use50Cities()
  {
    tsp = new TSP(); // load with default cities
    Assert.That(tsp.ShortestPath(50), Is.AtMost(2300));
  }

  [Test]
  public void Use100Cities()
  {
    tsp = new TSP(); // load with default cities
    Assert.That(tsp.ShortestPath(100), Is.AtMost(4675));
  }

  [Test]
  public void Use150Cities()
  {
    tsp = new TSP(); // load with default cities
    Assert.That(tsp.ShortestPath(150), Is.AtMost(5357));
  }
}
```

ShortestPathTest-Revised.cs

Now you can quickly exclude the whole fixture using a category name.

Of course, not all tests need categories, and you may have entire projects where there are no categories at all. But it's nice to know they are there if you do need them.

Per-method Setup and Teardown

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. NUnit lets you specify two methods to set up and then tear down the environment per test using attributes:

```
[SetUp]
public void PerTestSetup() {
  ...
}
[TearDown]
public void PerTestTeardown() {
  ...
}
```

In this example, the method `PerTestSetup()` is called before each one of the `[Test]` methods is executed, and the method `PerTestTeardown()` is called after each test method is executed, even if the test method throws an exception. This is why we mentioned that constructors in test fixtures generally aren't necessary. Constructors wouldn't work the way you wanted them to anyway, since NUnit doesn't necessarily recreate the `TestFixture` class each time it runs a test; it discovers and runs these methods using reflection.

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

```
[TestFixture]
public class DBTest
{
  private Connection dbConn;
  [SetUp]
  public void PerTestSetup()
  {
    dbConn = new Connection("oracle", 1521, user, pw);
    dbConn.Connect();
  }
  [TearDown]
  public void PerTestTeardown()
  {
    dbConn.Disconnect();
    dbConn.Dispose();
  }
```

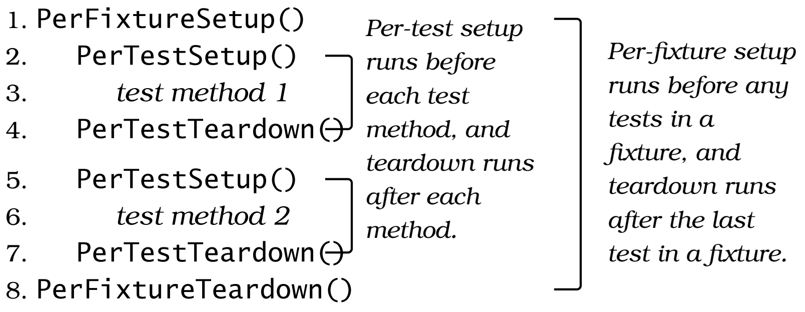

1. PerFixtureSetup()
2. PerTestSetup()
3. test method 1
4. PerTestTeardown()
5. PerTestSetup()
6. test method 2
7. PerTestTeardown()
8. PerFixtureTeardown()
Per-test setup runs before each test method, and teardown runs after each method.
Per-fixture setup runs before any tests in a fixture, and teardown runs after the last test in a fixture.

Figure 3.3: Execution Order of Setup Code

```
    [Test]
    public void AccountAccess()
    {
      // Uses dbConn
      xxx xxx xxxxxx xxx xxxxxxxxx;
      xx xxx xxx xxxx x xx xxxx;
    }
    [Test]
    public void EmployeeAccess()
    {
      // Uses dbConn
      xxx xxx xxxxxx xxx xxxxxxxxx;
      xxxx x x xx xxx xx xxxx;
    }
  }
```

DBTest.cs

In this example, the method `PerTestSetup()` will be called before `TestAccountAccess()`. After `TestAccountAccess()` has finished, `PerTestTearDown()` will be called. `PerTestSetup()` will be called again, followed by `TestEmployeeAccess()` and then `PerTestTeardown()` again.

Per-fixture Setup and Teardown

Normally per-method setup is all you need, but in some circumstances you may need to set something up or clean up after the *entire* test class has run; for that, you need per-fixture setup and teardown (the difference between per-test and per-fixture execution order is shown in Figure 3.3 on the previous page). All you need to do is annotate your setup methods with the following attributes:

```
[TestFixtureSetUp]
public void PerFixtureSetup() {
  ...
}
[TestFixtureTearDown]
public void PerFixtureTeardown() {
  ...
}
```

Note that you can use both per-fixture and per-test methods in the same class. While setup and teardown methods generally come in pairs, they don't have to. Very often, a fixture will have a setup, but no teardown. A teardown without a setup, while rare, is also not unheard of. We can also define set-up methods across inheritance boundaries, in both base classes and derived classes. They will work together as if they were all defined in the same class.

3.6 More NUnit Asserts

In addition to the basic asserts we've seen, NUnit provides additional asserts to aid in testing collections and files. If you prefer the classic-style assertion methods, check out the `StringAssert` and `CollectionAssert` classes as well as the NUnit documentation.

List.Contains

```
Assert.That(actualCollection,
            List.Contains(expectedValue))
Assert.That({5, 3, 2}, List.Contains(2))
```

Tests that the expected value is contained within `actualCollection`.

Is.SubsetOf

```
Assert.That(actualCollection,
            Is.SubsetOf(expectedCollection))
Assert.That(new byte[] {5, 3, 2},
            Is.SubsetOf(new byte[] {1, 2, 3, 4, 5}))
```

Tests that the elements of `actualCollection` are contained within `expectedCollection`, regardless of order.

Text.StartsWith

```
Assert.That(actual,
            Text.StartsWith(expected))
Assert.That("header:data.",
            Text.StartsWith("header:"))
```

Tests that the expected string is at the beginning of `actual`. This is case sensitive by default; to ignore case sensitivity, we need to add the `IgnoreCase` constraint modifier.

```
Assert.That("header:data.",
            Text.StartsWith("HeadeR").IgnoreCase)
```

Text.Matches

```
Assert.That(actual, Text.Matches(expected))
Assert.That("header:data.",
            Text.Matches("$header^\."))
```

Tests that the expected regular expression string matches `actual`. Here we're making sure the actual string starts with "header", and ends with a period character. We could also have used a combination of `Text.StartsWith`, `Text.EndsWith`, or `Text.Contains` constraints.

FileAssert.AreEqual / AreNotEqual

```
FileAssert.AreEqual(FileInfo expected,
               FileInfo actual)
FileAssert.AreEqual(String pathToExpected,
               String pathToActual)
```

Test whether two files are the same, byte for byte. Note that if we do the work of opening a `Stream` (file-based, or not), we can use the `EqualsConstraint` instead, like so:

```
Stream expectedStream = File.OpenRead("expected.bin");
Stream actualStream = File.OpenRead("actual.bin");
Assert.That(
  actualStream,
  Is.EqualTo(expectedStream)
);
```

3.7 NUnit Custom Asserts

The standard asserts that NUnit 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.

The worst thing you can do is slavishly copy the same sequence of test code over and over again. “Copy and paste” of common code in the tests can be a fatal disease.

Instead, tests should be written to the same high standards as regular code, which means honoring good coding practices such as the DRY principle,[4] loose coupling, orthogonality, and so on. Factor out common bits of test harness into real methods, and use those methods in your test cases.

This is real code, and needs to be well-written, and well-factored so you can reuse it and keep it up to date easily as the system grows and evolves.

Don't be afraid to write your own assertion-style methods. For instance, suppose you are testing a financial application and virtually all of the tests use a data type called `Money`.

```
using System;
using NUnit.Framework;
using NUnit.Framework.SyntaxHelpers;

public class MoneyAssert
{
  // Assert that the amount of money is an even
  // number of dollars (no cents)
  public static void AssertNoCents(Money amount,
                                   String message)
  {
    Assert.That(
        Decimal.Truncate(amount.AsDecimal()),
        Is.EqualTo(amount.AsDecimal()),
        message);
  }

  // Assert that the amount of money is an even
  // number of dollars (no cents)
  public static void AssertNoCents(Money amount)
```

[4]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].

```
    {
      AssertNoCents(amount, String.Empty);
    }
  }
```

MoneyAssert.cs

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

Now any other test classes in the project that need to test Money can use our own custom assertion method. If multiple test fixture classes needed to use our custom assertions or other support methods, we could also extract a common base fixture class they would then derive from. We'll talk more about that later in Chapter 8.

```
  using NUnit.Framework;

  [TestFixture]
  public class SomethingTest
  {
    [Test]
    public void CountDeMonet()
    {
      Money m = new Money(42.00);
      m.Add(2);
      MoneyAssert.AssertNoCents(m);
    }
  }
```

SomethingTest.cs

For more examples, take a look at the NUnit source code itself (perhaps the StringAssert code). That's part of the beauty of open source—you can go see the code for yourself, and see how the magic is done.

3.8 NUnit and Exceptions

We might be interested in two different kinds of exceptions:

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 ImportList(). It's supposed to throw an ArgumentException if passed a null list. We must test for that explicitly.

Guarding against bad data is good defensive programming. If a null parameter is passed in and not used immediately, the eventual NullReferenceException becomes a time bomb of sorts. It will go off at an unexpected moment, in some far away corner of the code. You then get the unenviable task of tracking down where in the system the bad data came from originally. But by failing quickly, you'll find the root of the problem quickly, and much more easily. Some people just like pain, but we don't, so we prefer to decrease our time spent debugging by employing this practice.

With what we've learned so far, we can construct the following test to ensure that the exception is thrown as expected.

```
[Test]
public void NullList()
{
  try
  {
     WhitePages.ImportList(null);
     Assert.Fail("ArgumentNullException should have been thrown");
  }
  catch (ArgumentNullException)
  {
  }
}
```

This test will fail if any exception other than ArgumentNullException is thrown, or if no exception is thrown at all. If no exception is thrown, the Assert.Fail() method is called, which fails the test. If an exception other than ArgumentNullException is thrown, it won't be caught by the catch defined, which fails the test. This works, but it's not exactly aesthetically pleasing.

More practically speaking, this style of test just doesn't express our intentions very well, and doesn't scale well to more complicated cases. The NUnit user community and authors agreed, so for expected exceptions, NUnit now provides the [ExpectedException] attribute:

```
[TestFixture]
public class ImportListTests
{
  [Test]
  [ExpectedException(typeof(ArgumentNullException))]
  public void NullList() {
    WhitePages.ImportList(null);
    // Shouldn't get to here
  }
}
```

ExceptionTest.cs

This test method is now expected to throw an exception (from the call to ImportList()). If it doesn't, the test will fail. If the exact exception specified fires as expected, the test passes. If a different exception is thrown (even a super-class of the one specified), the test fails. It might be tempting to just expect the base Exception type, but you're skirting around the fact the tests are telling you your design needs some work.

You want to be as specific with exceptions in this context as you would be in a catch() statement. Otherwise, you'll get tests that pass when a totally different exception is thrown, and you might not know about it until the system starts malfunctioning in the hands of end-users.

Two salient details worth noting: Once the expected exception fires, any remaining code in the test method will be skipped. If the SetUp method throws an exception before a test method's code executes, the test will always be reported as failing even though the actual test code didn't run. Furthermore, even if SetUp throws, TearDown method will still be run (if one is declared).

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

NUnit will take care of those for you. For instance, suppose you are reading a file of test data. Rather than catching the possible I/O exceptions yourself, just let them propagate out to the test framework.

```
[Test]
public void TestData1() {
  StreamReader sr = new StreamReader("data.txt");
  xxx xxx xxxxxxx xxxxx xxxxx;
}
```

Even better, NUnit will report the *entire* stack trace right down to the bug itself, not just to some failed assert, which helps when trying to figure out why a test failed. If you have enabled debugging information during compilation of your assembly under test, it will also give the exact source code line numbers in the stack trace.

When *compiling* under mono's C# compiler (`gmcs`) or Microsoft .NET's C# compiler (`csc`), add `-debug+` to the command line. If you're not working at the command line, this can be accomplished by changing the Project settings in whatever IDE you happen to be using. When *running* under mono, you'll need to use the `-debug` option to the mono runtime executable (`mono -debug`) for it to actually use that generated debug information.[5]

3.9 Temporarily Ignoring Tests

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. NUnit provides the `[Ignore]` attribute:

```
[Test]
[Ignore("Out of time.  Will Continue Monday. --AH")]
public void Something()
{
  xxx xxx xxxxxx xxxxx xxxx;
}
```

ExceptionTest.cs

NUnit will report that this method was skipped (and show a yellow bar in the GUI version), so that you won't forget about it later.

In other testing frameworks and languages, you'd have to either name the method differently or comment it out. When using JUnit in Java, for instance, methods whose names start

[5]Microsoft .NET doesn't require this; hopefully mono will remove this requirement in a future release.

with “test” (as in testSomething) will be run as tests; you have to name the method something else until you’re ready to tackle it. In any language, the code still has to compile cleanly; if it’s not ready for that yet, then you should comment out the offending parts.

It’s a good idea to to put a meaningful message, and perhaps even your initials, into the ignore so that the team knows *why* this test isn’t running. Are you still working on it? Do you need something from someone else in order to finish? Can someone else finish it up for you (in a geographically diverse team, perhaps)? Don’t just Ignore it and forget about it; that’s a Broken Window.[6]

You want to avoid at all costs the habit of *ignoring* failing test results. You don’t see green until they all work: just the absence of a red bar (or error messages) does not mean success.

Ignoring Platform-dependent Tests

There is one small exception to that rule; what to do when certain tests have to be ignored because of the platform on which you are running? This scenario isn’t uncommon and can occur if you’re writing a cross-platform application (whether it be for .NET 2.0 and mono, or specifically for .NET 1.1), some of your tests may only run (or pass) on a specific platform. This was a problem NUnit itself faced, so they introduced the Platform attribute, which is used like this:

```
[Test]
[Platform(Exclude = "Mono")]
public void RemoveOnEmpty() {
    xxx xx xxx xxxxx xx xx xx;
}
[Test, Platform(Exclude = "Net-1.0,Win95")]
public void EmptyStatusBar() {
    xxx xx xxx xxxxx xx xx xx;
}
```

As you can see, Linux-specific tests that don’t work on Solaris, MacOS, or certain Windows or .NET versions can be marked

[6]See [HT00].

as such. [7] When using the `Platform` attribute, you will still get a green bar in the GUI (not yellow) even in the prescense of tests ignored via this attribute. Other than that, it operates similarly to the `Ignore` attribute.

The point again is that you want to avoid any situation where you begin to ignore failing tests out of habit. `Platform` ensures that the proper tests are run only in the proper environment.

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.

[7] A comprehensive list of the platforms can be found in the NUnit documentation on `http://nunit.org`.

Chapter 4

What to Test: The Right-BICEP

Now that you know how to test, we need to spend some chapters looking at what to test; or more precisely, the kinds of things that might need testing.

It can be hard to look at a method or a class and try to come up with all the ways it might fail and 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 help 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?

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

Right *BICEP*

It's a good starting point. 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'll 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 (using automated tests) that the code is 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

```
    using NUnit.Framework;
    using NUnit.Framework.SyntaxHelpers;
    using System;
    using System.IO;
    using System.Collections.Generic;

    [TestFixture]
    public class LargestDataFileTests
    {
      private int[] getNumberList(string line)
      {
        string[] tokens = line.Split(null);

        List<int> numberList = new List<int>();

        for (int i=1; i < tokens.Length; i++)
        {
          numberList.Add(Int32.Parse(tokens[i]));
        }

        return numberList.ToArray();
      }

      private int getLargestNumber(string line)
      {
        string[] tokens = line.Split(null);

        string val = tokens[0];
        int expected = Int32.Parse(val);

        return expected;
      }

      private bool hasComment(string line)
      {
        return line.StartsWith("#");
      }

      // 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.
      [Test]
      public void FromFile()
      {
        string line;
        // most IDEs output the test bi-
nary in bin/[Debug,Release]
        StreamReader reader =
            new StreamReader("../../testdata.txt");

        while ((line = reader.ReadLine()) != null)
        {
          if (hasComment(line))
          {
            continue;
          }

          int[] numberListForLine = getNumberList(line);
          int expectedLargestNumber = getLargestNumber(line);
          int actualLargestNumber = Cmp.Largest(numberListForLine));

          Assert.That(expectedLargestNumber, Is.EqualTo(actualLargestNumber));
        }
      }
```

use XML.[1] Figure 4.1 on the preceding page is a version of TestLargest that reads in all of the tests from a data 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

In this example we're only running one particular test (using one assert), but you could extend that to run as many different tests on the same data as practical.

For just a handful of tests (as in this example), the separate data file approach is probably not worth the effort or the performance overhead of the file I/O. In cases where you can't justify an external file, C#'s string literals paired with a TextReader can provide the same benefits described above without the less palatable aspects:

```
string oneCommentWithTwoSets = @"
```

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

```
# comment line
9 7 8 9
-9 9 8 7
"
```

But say this was a more advanced application, with tens or even hundreds of test cases in this form. Then the 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. Ask a co-worker to take a look, or just take a break (away from the keyboard); sometimes it's difficult to see the woods through the trees.

Something else to think about: the code as presented in this example does not test any exception cases. How might you implement that? Also notice that we wrote a non-test "helper" method to parse the numbers from the data file. It's perfectly okay—even encouraged—to create support methods and classes as needed. We might even extract these support methods into a `TestFileParser` class if we wanted to share this code across different fixtures, or just to unclutter the test class itself.

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

4.2 Boundary Conditions

Right $\boxed{\textbf{\textit{B}}}$ *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. These nether-regions of untested code are where almost all exploitable security vulnerabilities come from.

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 that is missing delimeters or terminators, such as an e-mail address without a top-level domain (`"fred@foobar."`).[2]
- Empty or missing values (such as 0, 0.0, an empty string, an empty array, or `null`), or missing in a sequence (such as a missing TCP packet).
- Values far in excess of reasonable expectations, such as a person's age of 10,000 years or a password string with 10,000 characters in it.
- 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, or getting fragmented IP packets out of order, 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?

[2]A popular mail service suffered from an exploitable bug like this involving a missing '>' in SMTP headers.

- **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?

Because boundary conditions are such an important area to test, we'll examine these in detail in the next chapter (which makes Right-BICEP a nested acronym).

4.3 Check Inverse Relationships

Right B $\boxed{\textit{I}}$ *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:

```
[Test]
public void SquareRootUsingInverse() {
  double x = MyMath.SquareRoot(4.0);
  Assert.That(4.0, Is.EqualTo(x*x).Within(0.0001));
}
```

RootsTest.cs

You might check that some data was successfully inserted into a database, then search for it, and then delete it. You might transfer money into an account, then transfer the same amount out of the account. Any of these operations apply an "inverse" to see if you get back to an original state.

But 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 delete routine to test our insertion.

4.4 Cross-check Using Other Means

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

Right BI C EP

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:[3]

```
[Test]
public void SquareRootUsingStd() {
  double number = 3880900.0;
  double root1 = MyMath.SquareRoot(number);
  double root2 = Math.Sqrt(number);
  Assert.That(root2, Is.EqualTo(root1).Within(0.0001));
}
```

RootsTest.cs

Another way of looking at this issue is to use different pieces of data from the class itself to make sure they all "add up," or reconcile. That counts as a cross-check as well.

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.

[3]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.

As with the inverse checks above, make sure you aren't simply exercising the same underlying code in two different ways—the point of cross-checking is to explicitly use different code to verify the same result.

4.5 Force Error Conditions

Right BIC $\boxed{E}$ *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 90.

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
- Insufficient File or Path permissions
- System load
- Limited color palette
- Very high or very low video resolution

These are just general categories, for each of them there may be more subtle issues worth testing. For instance, you might test that the code can handle the case when the network itself goes down, but what about if the network is up and the DNS server is down? Or the network is up, but slowed to a timeout-inducing crawl due to a denial of service attack? These things happen, and if our code needs to handle these sort of errors, then we need to test for them. If our code isn't supposed to handle these sorts of errors, we should still write a test that validates that behaviour using the `ExpectedException` we previously discussed.

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.

```
[TestFixture]
public class FilterTest
{
  Timer timer;
String naughty_url = "http://www.xxxxxxxxxx.com";
  URLFilter filter;

  [SetUp]
  public void Initialize()
  {
    timer = new Timer();
  }

  [Test]
  public void SmallList()
  {
    filter = new URLFilter(SMALL_LIST);
    timer.Start();
    filter.Check(naughty_url);
    timer.End();
    Assert.That(timer.ElapsedTime, Is.LessThan(1.0));
  }
                                                    }
  [Test]
  [Category("Long")]
  public void HugeList()
  {
    filter = new URLFilter(HUGE_LIST);
    timer.Start();
    filter.Check(naughty_url);
    timer.End();
    Assert.That(timer.ElapsedTime, Is.LessThan(10.0));
  }
}
```

FilterTest.cs

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 in our automated build at least every couple of days, we'll quickly be alerted to any problems we may introduce, while there is still time to fix them.

Chapter 5

CORRECT Boundary Conditions

As we said in the last chapter, boundary conditions are such a vibrant source of bugs that we need a whole chapter to talk about them. 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 we have a function that takes two integers:

RootsTest.cs

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

Most of the time, this code will return a number just as we expect. But if the sum of a and b happens to equal zero, we will get a DivideByZeroException instead of a return value. That is a boundary condition—at the edge of normal expectations. It's a place where things might suddenly go wrong, or at least behave differently from what we wanted.

To help us 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 we 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.

There's always something else that could go wrong, and these are some of the more productive areas to consider.

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 an exception? What about multiple "@" signs or a string of only @" signs? 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 be aware of internationalization issues: not only could there be issues with the format (many countries don't have states or provinces), but issues with character encoding as well.[2] Will you be getting unicode data, and if so, will you be able to handle it?

Then there's more complex, structured data to consider. Suppose you are reading report data that contains a header record linked to a 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. This directly applies to any code that parses file formats or network protocols, avenues by which attacks will come either on purpose or unwittingly. It's best to code defensively and verify the defenses with unit tests,[3] since an attacker will probably end up testing them for you eventually whether we want them to or not.

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

[2]Input validation should always be done on model objects, sometimes in addition to the UI validation.

[3]A fun way to think about this is, "How would I attack this function?"

And of course, if you are creating data (not just validating it) such as 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 as well.

5.2 Ordering

C $\boxed{O}$ 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.

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, and the chapter on GUI testing on page 165. Bear in mind that business logic does not belong in the GUI itself–ever. User interface components (graphical or otherwise) should only contain code for the UI, not for anything else.

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. Then test it anyway, you may be surprised to find it makes a difference.

If you are supposed to maintain something in order, verify 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:

```
[Test]
public void KitchenOrder()
{
  Order order = new Order();
  FoodItem dessert = new Dessert("Chocolate Decadence");
  FoodItem entree = new Entree("Beef Oscar");
  FoodItem salad  = new Salad("Parmesan Peppercorn");
  // Add out of order
  order.AddFoodItem(dessert);
  order.AddFoodItem(entree);
  order.AddFoodItem(salad);
  // But should come out in serving order
  IEnumerator itr = order.GetEnumerator();
  Assert.That(salad, Is.EqualTo(itr.Current));
  itr.MoveNext();
  Assert.That(entree, Is.EqualTo(itr.Current));
  itr.MoveNext();
  Assert.That(dessert, Is.EqualTo(itr.Current));
  itr.MoveNext();
  // No more left
  Assert.That(itr.MoveNext(), Is.False);
}
```

KitchenTest.cs

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

CO $\boxed{R}$ RECT

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 built-in value type (e.g., an `int` or `Int32`) to store a bounded-integer value such as an age, or a compass heading.

```
using System;
//
// 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 ArgumentException("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.cs

Notice that the angle returned is just an int—a plain old number, 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.[4]

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 interdepen-

[4]For types like these, a `struct` might be preferred if you have a deep enough knowledge of the CLR to care.[Ric06]

dent. 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 143.

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

```
public const int MAX_DIST = 100;
static public void AssertPairInRange(Point one,
                                     Point two,
                                     String message)
{
  Assert.That(
    Math.Abs(one.X - two.X),
    Is.AtMost(MAX_DIST),
    message
  );
  Assert.That(
    Math.Abs(one.Y - two.Y),
    Is.AtMost(MAX_DIST),
    message
  );
}
```

PairTest.cs

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()
  {
    elements = new string[100];
    nextIndex = 0;
  }
  public String Pop()
  {
    return elements[--nextIndex];
  }
  // Delete n items from the elements en-masse
  public void Delete(int n)
  {
    nextIndex -= n;
  }
  public void Push(string element)
  {
    elements[nextIndex++] = element;
  }
  public String Top()
```

```
      {
        return elements[nextIndex-1];
      }
      private int nextIndex;
      private string[] elements;
    }
```

MyStack.cs

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 `nextIndex`, 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 `nextIndex` 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()
      {
        if (!(nextIndex >= 0 &&
              nextIndex  < elements.Length))
        {
          throw new InvariantException(
                  "nextIndex out of range: "  +  nextIndex +
                  " for elements length " + elements.Length);
        }
      }
```

MyStack.cs

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.[5]

```
    using NUnit.Framework;
    [TestFixture]
    public class MyStackTest
    {
      [Test]
      public void Empty()
      {
        MyStack stack = new MyStack();
        stack.CheckInvariant();
        stack.Push("sample");
        stack.CheckInvariant();
        // Popping last element ok
```

[5]You'd normally use an `InvalidOperationException`, but in this case we want to reinforce the invariant concept by using a custom exception.

```
        Assert.That(
          stack.Pop(),
          Is.EqualTo("sample")
        );
        stack.CheckInvariant();
        // Delete from empty stack
        stack.Delete(1);
        stack.CheckInvariant();
      }
    }
```

MyStackTest.cs

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

```
TestCase 'MyStackTest.Empty' failed: InvariantException
nextIndex out of range: -1 for stack length 100
        mystack.cs(34,0): at MyStack.CheckInvariant()
        mystacktest.cs(20,0): at MyStackTest.Empty()
```

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

COR $\boxed{\textbf{R}}$ *ECT*

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 be in? 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 nonempty 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).

```
[Test]
public void JamItIntoPark()
{
  transmission.Shift(DRIVE);
  car.AccelerateTo(35);
  Assert.That(
    transmission.CurrentGear,
    Is.EqualTo(DRIVE)
  );
  // should silently ignore
  transmission.Shift(PARK);
  Assert.That(
    transmission.CurrentGear,
    Is.EqualTo(DRIVE)
  );
  car.AccelerateTo(0); // i.e., stop
  car.BrakeToStop();
  // should work now
  transmission.Shift(PARK);
  Assert.That(
    transmission.CurrentGear,
    Is.EqualTo(PARK)
  );
}
```

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 the original Eiffel in *Object-Oriented Software Construction* [Mey97], or take a look at nContract,[6] which can add similar capabilities to C#.[7]

5.5 Existence

CORR $\boxed{E}$ *CT*

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

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, or an empty string, or an empty collection.

Many C# 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 thrown from the depths of some library. But a specific 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, permissions that had been fine last time you checked—you name it. All of these things may not exist when you expect

[6] `http://puzzleware.net/nContract/nContract.html`

[7] There are other efforts for other languages as well, such as `http://dbc.rubyforge.org` for C and `http://icontract2.org` for Java.

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 everything which, funnily enough, includes nothing.

5.6 Cardinality

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

CORRE $\boxed{C}$ *T*

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 84. 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 260. (Why 260? That's the defined value for `MAX_PATH` in `windows.h`, and so turns out to be a good boundary condition for finding string truncation and buffer overflow issues in underlying native code.)

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 (but more than one)?
- Can you add an item when there is only one item on the list?
- Can you add an item when there are already ten items 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?

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

```
public MaxEntries {
  get { 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, af-

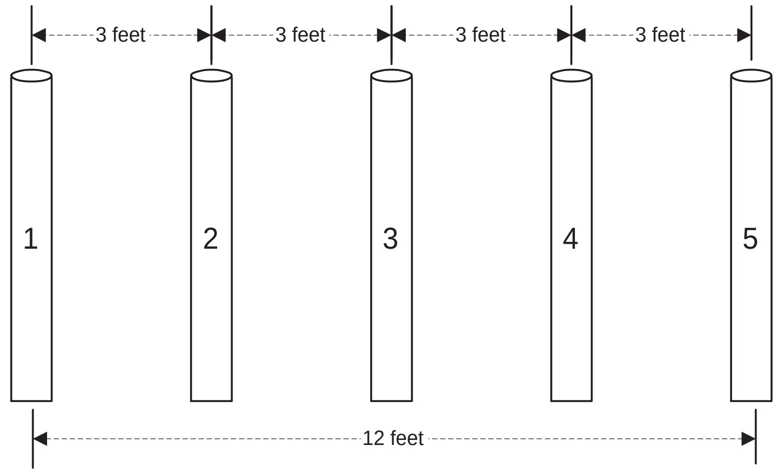


Figure 5.1: A Set of Fence posts

ter all), you can go back and change this one number. The test should automatically follow suit, because it uses the same property.

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

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

CORREC **T**

- 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 to find these kind of temporal dependencies. Just as order of data may have mattered to you in the earlier examples (as we described in "Ordering" on page 74), 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 statement, is it true or false: every day of the year is 24 hours long?[8]

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?

[8]Ignoring leap seconds for now, we're just talking about whole hours.

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. And leap seconds *do* make a difference.

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, multi-processor environment (see the section in on page 187 for an interesting "Gotcha" in C#).

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 `lock` keyword to any property or method that needs it, and try firing off multiple threads as part of your test.

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 196*

```
using System;
public interface StackExercise {
  /// <summary>
  /// Return and remove the most recent item from
  /// the top of the  stack.
  /// </summary>
  /// <exception cref="StackEmptyException">
  /// Throws exception if the stack is empty.
  /// </exception>
  String Pop();

  /// <summary>
  /// Add an item to the top of the stack.
  /// </summary>
  /// <param name="item">A String to push
  /// on the stack</param>
  void Push(String item);

  /// <summary>
  /// Return but do not remove the most recent
  /// item from the top of the stack.
  /// </summary>
  /// <exception cref="StackEmptyException">
  /// Throws exception if the stack is empty.
  /// </exception>
  String Top();

  /// <summary>
  /// Returns true if the stack is empty.
  /// </summary>
  bool IsEmpty();
}
```

StackExercise.cs

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 197

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 {
  /// <summary>
  /// Add this many of this item to the
  /// shopping cart.
  /// </summary>
  /// <exception cref="ArgumentOutOfRangeException">
  /// </exception>
  void AddItems(Item anItem, int quantity);

  /// <summary>
  /// Delete this many of this item from the
  /// shopping cart
  /// </summary>
  /// <exception cref="ArgumentOutOfRangeException">
  /// </exception>
  /// <exception cref="NoSuchItemException">
  /// </exception>
  void DeleteItems(Item anItem, int quantity);
```

```
    /// <summary>
    /// Count of all items in the cart
    /// (that is, all items x qty each)
    /// </summary>
    int ItemCount { get; }

    /// Return iterator of all items
    IEnumerable GetEnumerator();
  }
```

ShoppingCart.cs

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 198

The method's signature is:

```
  ///
  /// Send the named file as a fax to the
  /// given phone number.
  /// <exception cref="MissingOrBadFileException">
  /// </exception>
  /// <exception cref="PhoneFormatException">
  /// </exception>
  /// <exception cref="PhoneAreaCodeException">
  /// </exception>
  public bool SendFax(String phone, String filename)
```

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 = `TableSize.Width - 1` and y = `TableSize.Height - 1`.

Answer on 199

Coordinates are specified in fractions of centimeters.

```
  public void MoveTo(double x, double y);
  public void SewTo(double x, double y);
  public void SetWorkpieceSize(double width,
                               double height);
  public Size WorkpieceSize { get; }
  public Size TableSize { get; }
```

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 200

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.

```
using System;
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
  void FastForward(double seconds);
  /// Move the current position backwards by this
  /// many seconds. Rewinding past zero leaves
  /// the position at zero
  void Rewind(double seconds);
  /// Return current time position in seconds
  double CurrentTimePosition();
  /// Mark the current time position with label
  void MarkTimePosition(String name);
  /// Change the current position to the one
  /// associated with the marked name
  void GotoMark(String name);
}
```

AVTransport.cs

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 201

Chapter 6

Using Mock Objects

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

What if our 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 (directly or indirectly) 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 our poor little unit test dies mysteriously. With this kind of coupling, sometimes simply adding a new test can cause other tests to fail. 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 shadows, 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 nefarious unit testing purposes.

Fortunately, there's a testing pattern that can help: *mock objects*. A mock object is simply a testing 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, like requiring a certain file system, database, or network environment.
- 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 relevant methods on 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 or base class, so it can remain blissfully ignorant as to whether it is using the real object or the mock. Sometimes there's a simpler solution to getting on with our testing, so let's explore that first.

6.1 Stubs

What we need to do is stub out, or fake, all those uncooperative parts of the real world and replace them with more complicit allies—our own version of "lighting doubles." For example, stubs allow us to fake our interaction with a database or the filesystem.

In many cases, stubs just implement an interface and return dummy values for the methods in said interface.[1] In even simpler cases, all the implemented methods in the stub just throw a `NotImplementedException`.[2]

A common scenario is when there is a class that encapsulates database access, but we don't want to actually configure and populate a database to run simple tests.

```
public class MySqlCustomerRepository
{
  public string FindBy(long id)
  {
    xxxx xx xxxxx
  }
}
```

First, we extract an interface for the methods we need to stub and apply that interface to the class we want to mock:

```
public interface CustomerRepository
{
  string FindBy(long id);
}
```

[1]Note that while you can also derive from an abstract class, interfaces are preferred.[**?**]

[2]Most IDEs will fill in this exception for you when told to automatically implement the methods for an interface.

```
public class MySqlCustomerRepository : CustomerRepository
{
  public string FindBy(long id)
  {
    xxxx xx xxxxx
  }
}
```

Then we can return the dummy value that we think will evoke the behaviour we want from the `ProductAdoptionService`.

```
public class StubCustomerRepository : CustomerRepository
{
  public string FindBy(long id)
  {
    return string.Empty;
  }
}
```

We put the code for this stub class in the same file as the test fixture class that will be using it, until we need to move it to a more general area where other test fixture classes can access it. Then we plug in the stub to our unit test like so:

```
namespace WebCRM.Test.ProductAdoptionTest
{
  [TestFixture]
  public class NoDataFixture
  {
    [Test]
    public void OverallRateIsZero()
    {
      StubCustomerRepository customerRepository =
        new StubCustomerRepository();

      ProductAdoptionService service =
        new ProductAdoptionService(customerRepository);

      Assert.That(service.GetPercentage(), Is.EqualTo(0));
    }
  }
}
```

If we're lucky, our test might now pass and we didn't have to touch a database. In fact, this test could have been written before there was ever a schema design, database vendor debate, or anything else. By programming to interfaces, we were able to plug in what we needed without depending on politicking or other non-coding activities that can slow down a project. Note that we not only get to verify the code being tested produces the results that we want, but we also get to verify that it *interacts* with the stubbed class in the way we expect. More on that later.

6.2 Fakes

Sometimes we need to do more than return dummy values to get at the code we're trying to test. What if we have files on the filesystem that conform to a certain format and we want to test that we're parsing them correctly?

```
public class DumpFileParser
{
  FileStream stream;
  public DumpFileParser(string fileName)
  {
    stream = File.Open(fileName);
  }
  xxxx xxx xxxx
}
```

The code above requires a real file on the file system in order to be tested. This can put an unnecessary filesystem layout burden on the person running the tests, and the disk I/O will slow down the tests.[3] What can we do in a situation like this to make it easier to test? The class actually discards the supplied filename after the constructor and just operates on the resulting stream. We'll look at a suboptimal way of making it more testable, then a more optimal way. It's good to understand what the evils in the world are so that we don't accidentally end up evoking any of them.

What if we used `#define` to tell the code when we were testing, then it wouldn't use the filesystem.

```
public class DumpFileParser
{
  FileStream stream;
  public DumpFileParser(string fileName)
  {
#if TESTING
  stream = new MemoryStream();
#else
  stream = File.Open(fileName);
#endif
```

[3]This doesn't seem like a big deal, but little slowdowns like this add up quickly.

```
    }
    [illegible]
  }
```

MemoryStream is a nifty class in the .NET class library that allows us to make, as you may have guessed, an in-memory stream. Now we have a real Stream-derived object that the class can interact with, and it doesn't touch the filesystem. Before we get too far ahead of ourselves, though, we have to realise that an empty stream has limitations. First, an empty stream doesn't really help us if we need to read data from that stream. Many of the tests we write will probably want to supply different data via the stream to make sure the parser behaves correctly. We could figure out various ways to get some test data into place in this scenario, but this approach works around the fact that the code wants the stream to be parameterized; our attempt to test this code has illuminated this. Third, #if statements strewn throughout the code for testing purposes are difficult to maintain. And, in our opinion, they're ugly as well.

It might also be tempting to just add an empty constructor to eliminate the need for any of this deep thinking. While this would "work" in a very narrow sense, there's a good reason there wasn't an empty constructor in the first place: without the Stream being created, the object isn't in a valid state. In this case, invalid state means a probable NullReferenceException whenever we try to do anything with the object. Objects being in a valid state after construction is a core object-oriented design principle, and ignoring it is not the right thing to do in this case. Tests can help drive improvements to the code's design, but this particular example isn't one of them.

Now that we've discussed what won't work, what will work? What if we shifted the responsibility of actually getting the FileStream to the clients and passed in a FileStream instead? Doing this transformation would resolve the design feedback we're getting from testing this in the first place.

```
  public class DumpFileParser
  {
    Stream stream;
    public DumpFileParser(Stream dumpStream)
    {
      this.stream = dumpStream;
    }
    xxxx xxx xxxx
  }
```

This isn't bad at all. Now the consumers of the class, including the tests, could perform the `File.Open()` and pass in a `FileStream`. It may seem like we're just moving the problem around, but we needed to do that to get to the next step. The next step is to make our code a little more shy; specifically, to make it more liberal in what it will accept. In this case, we aren't using any methods specific to `FileStream`, so we can actually accept the base class, `Stream`, instead.

What does that get us? Well, in our tests we can now use the spiffy `MemoryStream` class, like so:

```
  [TestFixture]
  public class DumpFileParserTest
  {
    private StreamWriter writer;
    private DumpFileParser parser;
    private MemoryStream stream;

    [SetUp]
    public void SetUp()
    {
      stream = new MemoryStream();
      writer = new StreamWriter(stream);
    }

    [Test]
    public void EmptyLine()
    {
      writer.WriteLine(string.Empty);
      parser = new Parser(stream);
      Assert.That(xxxx, xxxx);
    }
  }
```

Presto! An instant pseudo-text-file that you can also use to write binary data. Since this operates in-memory, you won't incur the performance penalty of disk I/O.[4] Now we can do the testing we need, quickly and conveniently. A nice side effect is that our code is shyer, yielding a more flexible design that is easier to reuse. One could say that changing the parameter to a `Stream` was a change strictly for the sake of testing and

[4]Note that this technique works just as well with Sockets and other stream-based I/O as well.

that observation would be somewhat correct. The other side of the story is that by not programming against a concrete implementation, the code now has a more flexible design. We were led to this by refactoring a very little bit to make things easier to test. This kind of design feedback is the real magic of unit testing, but this is only one simple example.

Faking collaberators

The DumpFileParser class we were just working on does some pretty complicated collation of the data in the stream. If another class depends on DumpFileParser, we don't want to make the entire fake stream necessary for it to produce the data we're trying to test our other class against. Besides the fact that it would be really tedious, it adds a whole new dimension of coupling and maintenance to the test code. If we used a real DumpFileParser while testing a collaberating class, we're increasing the work we have to do if DumpFileParser changes or gets removed.

That doesn't sound very pragmatic, so how do we decouple DumpFileParser from the tests of a class that requires a DumpFileParser? It's actually very similar to our initial example – we need to abstract things up a level, then we can supply a variation on DumpFileParser that returns whatever dummy values we need for the purpose of testing the other object. This is known in some circles as creating a fake, and in other circles as a static mock. Let's look at some code.

```
public class Analyzer
{
  private DumpFileParser parser;
  private List<string> reportItems;

  public Analyzer(DumpFileParser parser)
  {
    this.parser = parser;
  }

  public bool ExpectationsMet
  {
    get
    {
      return parser.ReportItems.Count == reportItems.Count;
    }
  }

  public byte[] GetNextInstruction()
  {
    xxxxxxxxx
  }
}
```

If we wanted to test the `ExpectationsMet` property, the `ReportItems` property on `parser` will need to be under our control so we can make it return what we want. One way would be to make the `ReportItems` property on `DumpFileParser` virtual. We could then subclass and override it for our testing purposes, and pass an instance of said subclass into the constructor for `Analyzer`. While that would work, there's a better way that yields a more flexible, and interface-oriented, design: extract an `interface` called `Parsable` that contains, for the time being, a declaration for the `ReportItems` property getter.

```
public interface Parsable
{
  List<string> ReportItems
  {
    get;
  }
}
```

Then, we can make DumpFileParser implement the Parsable interface. Next, we change the `Analyzer` constructor's parameter from DumpFileParser to Parsable. Last, we change the `parser` field in `Analyzer` from `DumpFileParser` concrete class to be the `Parsable` interface that `DumpFileParser` now implements. When we try to compile, the compiler might tell us that we're using some methods not defined on the `Parsable` interface. We'll need to add those methods to the interface as well.

```
public DumpFileParser : Parsable
{
  xxxxxxx
}
public class Analyzer
{
  private Parsable parser;
  private List<string> reportItems;

  public Analyzer(Parsable parser)
  {
    this.parser = parser;
  }
  xxxxxxxx
}
```

None of the existing consumers of Analyzer have to change, and yet, we have just made Analyzer easier to test and reuse. If in the future we wanted to add the ability to parse another file format, Analyzer itself wouldn't have to change to accomodate the extra functionality—only the consumers would by passing in a new class that implements the Parsable interface.

This is a good example of the advantage of interface-based design, but the point worth mentioning again is that we arrived at this better design by refactoring toward testability. Besides being more testable and reusable, it also means that we don't need to wait for another set of programmers to finish implementing the concrete class that our class might be collaborating with. We can fully unit test our class by faking the collaborator's interface, which generally makes integrating with the concrete classes developed by others (or even our future selves) significantly less painful.[5]

Fakes are great, especially when they're simple, but it's also easy to outgrow them; like when we need to do more than return a single value, for instance. At some point, we want to return values in a certain order each time a method is called. To accomplish this with a fake, we would need to track a Stack of return values for a given method.

```
public FakeParser : Parsable
{
  private Stack<byte[]> bytesToReturn;
  public Stack<byte[]> BytesToReturn
  {
    get { return bytesToReturn; }
    set { bytesToReturn = value; }
  }
  public Boolean ExpectationsMet
  {
    get { return false; }
  }
  public Byte[] GetNextInstruction()
  {
    return BytesToReturn.Pop();
  }
}
```

[5]In many cases, the usually pandemonious step of integration just works.

While this would work and is a clever way to make a programmable fake, we risk repeating ourselves because we would end up doing this for most methods on our fake. It also gets a little more hairy when we have to make them throw specific exceptions at certain points to test failure modes. Surely, there must be a better way.[6]

6.3 Mock Objects

In the old days, just having the ability to call subroutines was a great advance. Then libraries of code became popular—everything had to be library. Nowadays, libraries aren't good enough. You've got to have a *framework* to be taken seriously.

In the case of `.NET`, there are several alternative mock object frameworks to choose from (a good list can be found at `http://www.mockobjects.com`). NUnit includes its own built-in framework that the NUnit team uses to test NUnit itself. NUnit's mock framework doesn't provide all of the features of some other frameworks, so we'll look at a few other frameworks as well. But before we do, it's worth noting that because we're in .NET's CLR environment, this same framework can be used to mock objects for any code written in any language compliant with the Common Language Specification.

NUnit Mocks

When you think about it, there's really not too much to a mock object: it's simply an object that implements a particular interface, returns values we want it to return, and checks that it was used in a certain way. As a result, the basic frameworks for creating mock objects are also simple.

In the previous section, we saw what we would have to go through to have our fake be somewhat programmable and return multiple values for a given method call. Here is how we would do this using NUnit's mock framework.

[6]Never call us Shirley.

Note that to compile this code, we'll have to add a reference to the nunit.mocks.dll assembly *in addition to* the nunit.framework.dll reference.

```
using NUnit.Framework;
using NUnit.Framework.SyntaxHelpers;
using NUnit.Mocks;

[TestFixture]
public class AnalyzerTest
{
  Analyzer analyzer;
  Parsable parser;

  [Test]
  public void NoBytes()
  {
    DynamicMock controller =
      new DynamicMock(typeof(Parsable));
    parser = controller.MockInstance as Parsable;
    analyzer = new Analyzer(parser);

    controller.ExpectAndReturn(
      "GetNextInstruction",// method name
      new byte[] {},       // return value
      null                 // expected arguments
    );

    controller.ExpectAndReturn(
      "get_ExpectedReportItems",
      new List<string>(),
      null
    );

    analyzer.Run();

    controller.Verify(); // fails the test if expecta-
tions are unmet
    Assert.That(analyzer.ReportItems, Is.Empty);
  }
}
```

Mock object frameworks make it very easy to set multiple method call expectations, with or without accompanying dummy values that should be returned, with or without throwing exceptions, etc. In the code above, we first instantiate a DynamicMock object, passing the Parsable interface's type into the constructor. We can only create a DynamicMock for interfaces or classes that derive from MarshalByRefObject. We highly recommend extracting interfaces whenever possible, not only for the reasons previously discussed in this chapter, but also because of the complex implications of using MarshalByRefObject.[Ric06]

There'll be times when we need to test something that uses an existing interface and there are no pre-written mock objects lying around. Often, we can just jump right on in and create a new mock object. But what if the interface that we're mocking is enormous, with dozens of methods and accessors? That could mean a lot of work producing a mock object that implements the interface. This is particularly galling if we only need one or two methods from the interface to run our tests, and we can't refactor to break up the interface for some reason.

This is where dynamic mock objects come in. They let us create an object that responds as if it implemented a full interface, but in reality it is totally generic. You only need to tell this object how to respond to the method calls that our code uses. This can represent a considerable saving in time. It'll also give you less code to maintain in the future.

The dynamic mock packages operate by creating *proxy objects* in the underlying implementation. These are objects that are designed to stand in for their real-world counterparts. In the dynamic mock object context, this means that we can use a proxy in place of a real object in our tests. However, we still need to be able to control this generated proxy object—we need to be able to tell it how to respond. This is where the controller comes in.

The controller is in charge of a dynamic mock object. You use the controller to create an instance of the mock and to tell the mock what to do. Sometimes the controller is told directly, like in NUnit's mocks, sometimes indirectly as in the NMock2 framework, which we'll discuss later in the chapter.

Once the mock is created, we pass it in to the real object that we are testing the interaction with. If our interaction testing wasn't constrained to the `Run()` method, we would program the mock before we passed it into the constructor for `Analyzer`. Since we're focusing on testing the interaction after `Run()` is called, we start programming it with our expectations right before the call to run, since this expresses our intentions clearly. To put it another way, we *don't* want to program all the expectations for our mock in a big clump that is difficult to read and understand. In the code above, the expectations we set are:

1. The method name
2. The value, if any, that will be returned when the mocked method is called
3. The specific arguments, if any, we expect the mocked method to be called with.

After creating the mock, we tell it to expect a call to `GetNextInstruction()`, and to return an empty `byte` array. The final `null` parameter indicates that there are no specific method argument expectations. In this case, the method in question doesn't have any parameters, but we can also supply `null` when we just don't care. In our experience, checking specific arguments supplied to the mocked method is usually not necessary because that level of detail usually isn't necessary to express the intention of the interaction we're testing.

Next, we tell the mock to expect a call to the getter for the `ExpectedReportItems` property. Note that we had to prepend "get_" to the property name.[7] In the context of this application, a parser having no bytes also means it should also not have any expected report items.

We then get our mock object via the `MockInstance` property. Because NUnit's mock framework doesn't take advantage of generics, we have to cast it—hence the use of the `as` operator. At that point, we can treat that instance like the real object as long as we use it only in the way we programmed it. One way to think about it is the framework provides a kind of API-level record-and-playback mechanism. If we didn't "record" the method calls, the mock can't play them back.

By default, the `DynamicMock` operates fairly loosely. Just by telling the mock to expect the method call once, it will happily do whatever we told it to do even if the method is called multiple times. It also doesn't care about the order in which the expected methods are called by default. While we sometimes don't care about that level of detail, and that makes this default behaviour quite nice, it's a good idea to be vigilant when practical.

[7] For more details on the inner workings of properties, see [Ric06].

When we want that vigilance, we can set the `Strict` property on the mock to `true`. One of the things the strict flag does is that the mock will fail the test immediately if something happens that the mock wasn't expecting. If we're not using strict mode, then we need to ask the mock to `Verify()` that all the expectations were met. The `Verify` method acts as a kind of assertion. If anything we expected didn't happen, the verification will fail. Since the verification generally happens at the end of the test, it can sometimes be difficult to track down where things went wrong. This is another reason to prefer the strict mode on the mock, if practical.

Note that the `ExpectAndReturn` methods take the method name as a `string` parameter. This introduces a gotcha where if you rename the method in the code, but not in the mock expectation, the mock will throw an exception. Other frameworks, which we'll discuss later in this chapter, improve upon this limitation.

Some of the other expectations we can set using NUnit's `DynamicMock` include:

- `ExpectNoCall(string methodName)`, which will cause verification to fail if the method supplied is called. If the mock is in strict mode, the test will fail immediately if specified method is called.
- `ExpectAndThrow(string methodName, Exception exception, params object[] args)`, which operates the same as `ExpectAndReturn`, except the exception specified is thrown. This is great for making sure your exception handling interaction between classes is rock-solid, and stays that way.
- `SetReturnValue(string methodName, object returnValue)`, which will always return the value specified no matter how many times the method in question is called. We generally don't recommend using this, as it can cover up the very interaction feedback that mocks are so good at giving us.

For more information check out NUnit's documentation or just explore a bit with a method-completing code editor.

NMock2 Framework

NMock2,[8] which is based on jMock for Java,[9] inspired NUnit's new style of constraint-based assertions. It is meant to provide a more concise and easily readable syntax in contrast to other mock frameworks. Since we use unit tests as documentation for our code, it's important that the configuration of our mocks be easy to read and understand. In that vein, NMock's syntax reads from left to right, albeit with a syntax that might look a bit strange at first.

```
using NUnit.Framework;
using NMock2;

[TestFixture]
public class AnalyzerTest
{
  Analyzer analyzer;
  Parsable parser;
  Mockery mockery;

  [Test]
  public void NoBytes()
  {
    mockery = new Mockery();
    parser = mocks.NewMock<Parsable>();
    analyzer = new Analyzer(parser);

    Expect.Once.On(parser)
      .Method("GetNextInstruction")
      .Will(Return.Value(new byte[] {}));

    Expect.Once.On(parser)
      .GetProperty("ExpectedReportItems")
      .Will(Return.Value(new List<string>()));

    analyzer.Run();
    mockery.VerifyAllExpectationsHaveBeenMet();
    Assert.That(
      analyzer.ReportItems,
      new EmptyConstraint()
    );
  }
}
```

This code is equivelant to the code from the previous subsection that was written using NUnit's mocks. First, we create a `Mockery` object. `Mockery` acts as a factory for mock objects,[10] via the `NewMock<T>()` generic method. Because it is a generic

[8] http://nmock.org
[9] http://jmock.org
[10] Mocks + Factory = Mockery, get it?

method, whatever type we parameterize it with is the type it will return. This allows us to avoid the casting we had to do in NUnit's mocks.[11] Note that the use of a generic method is a C# 2.0 feature, so NMock2 can't be used on a project that strictly uses an earlier C# version. The `Mockery` object also keeps track of the expectations we are setting.

We then set up the expectations. Remember, we only program the mock with the minimal number of expectations we need to test the interaction triggered by the specific method. We want this to read like a conversation between the mock and the real object, from A to B and back again. We expect that, only once, the `parser`'s `GetNextInstruction` method will be called, and it will return an empty `byte` array. Under the covers, the expectations are communicated to the `Mockery` object, which created the `Parser` mock in the first place.

To our subjective eyes, NMock2 reads a bit more easily than other frameworks. It allows us to focus on only the aspects that we care about. On the other hand, NUnit's mocks require us to always provide the arguments we are expecting; we have to supply `null` to tell it we don't have any argument expectations. In NMock2, we only add the argument constraint matcher if we actually need it by adding `.With(x)` to the chain. This extra flexibility seems small, but it adds up to test code that is easier to maintain.

One major caveat to note is that we can no longer use NUnit's `AssertionHelpers` namespace, which gave us `Is` and `Has`, amongst other nice things, because NMock2 also defines classes with those names. We're not using them in this example, because they relate to argument matching. Because of this conflict, we're using `new EmptyConstraint()` instead of the usual `Is.Empty`. We could also substitute in the classic-style assertion, `CollectionAssert.IsEmpty()`. Hopefully this namespace issue will be resolved in future versions so we can use the advantages of both.

[11]The NUnit 2.4 team purposefully restrained themselves to only using C# 1.1 features so it could be more widely used.

Joe Asks...

How do I mock singletons?

With design patterns becoming popular, many projects have various patterns implemented as part of their design. One of the most commonly misunderstood patterns is the Singleton. Unfortunately, it is usually mis-implemented in such a way that introduces global state, which in turn introduces a large amount of temporal coupling, hidden dependencies, and extremely difficult debugging. Many times, a singleton isn't even necessary, but it can be a tempting way to cheat through actually improving the design.

In the next section, we show how to test around a usage of `DateTime.Now`, which is a static global, just like a singleton class can be. The key is to extract an interface for the methods actually used by the consumers of the singleton class, and then extract a parameter from the class or method that accepts the interface for the singleton. From that point, you can create a mock from the extracted interface and pass in the mock via the parameter.

Even if you aren't unit testing, this is the standard set of refactorings for loosening the hangman's knot of singletons that many projects get themselves into. Once you see the real dependencies you have with the visibility of the extracted parameters, you'll have the information for genuinely improving our design instead of working around it. When applying this design feedback, you may find that the singleton is simply no longer necessary.

RhinoMock

There's a third option in the world of mock frameworks that some of our reviewers asked us to mention, RhinoMock.[a] RhinoMock primarily distinguishes itself by not relying upon strings to specify the method that expectations will be set on. It is this simple feature that makes it work more easily with code completion and refactoring capabilities in modern IDEs.

Each framework has advantages and disadvantages, which one you end up applying on your project is a matter of preference and practicality. We do encourage you to try a couple before settling, though.

[a]http://www.ayende.com/projects/rhino-mocks.aspx

DotNetMock Framework

Some objects are difficult to set up mocks for, regardless of the framework, due to the complexity and girth of their interfaces. ASP.NET and ADO.NET objects can be fairly difficult, in particular. In these cases, a library of static mocks that are engineered specifically for common unit testing scenarios can come in handy. To meet this need, the DotNetMock[12] framework is actually three things in one:

1. It's a framework (not surprisingly), allowing us to create mock objects in a structured way.
2. It contains a (small) set of predefined mock objects that we can use out of the box to test our application.
3. Finally, it comes with a technology, dynamic mocks, that let's us construct mock objects without all that messy coding.

We recommend using one of the aforementioned mock object frameworks for standard mock program-activities, and DotNetMock for when you need to mock one of the messier frame-

[12]http://dotnetmock.sourceforge.net

work classes. As such, we're only going to cover the library of mocks that DotNetMock comes with.

Supplied Mock Objects

One of the nice things about using a standardized framework for testing is that we can start to build a library of standard mock objects and reuse these across projects. In fact, in the open source world, you might even find that other folks have mocked up the interfaces that you need and made them freely available. The DotNetMock package comes with a (small) number of these off-the-shelf mock object packages, available in `DotNetMock.Framework`. While DotNetMock's library of predefined mocks hasn't been updated for .NET 2.0 at the time of this writing, it's still very useful if you're using .NET 1.1 APIs. Here we'll look at one of these, `Data`, which implements many of the interfaces in .NET's `System.Data`.

Let's start by implementing more of our access controller. After verifying that a password has been supplied, we'll now go to a database table and verify that a row exists giving this user, identified with the given password, access to our resource.

```
using System;
using System.Data;
using System.Data.SqlClient;

public class AccessController1 {
  private ILogger       logger;
  private String        resource;
  private IDbConnection conn;

  public static readonly String CHECK_SQL =
    "select count(*) from access where " +
    "user=@user and password=@password " +
    "and resource=@resource";

  public AccessController1(String resource,
                           ILogger logger,
                           IDbConnection conn) {
    this.logger   = logger;
    this.resource = resource;
    this.conn     = conn;
    logger.SetName("AccessControl");
  }

  public bool CanAccess(String user, String password) {
    logger.Log("Checking access for " + user +
      " to " + resource);
    if (password == null || password.Length == 0) {
      logger.Log("Missing password. Access denied");
      return false;
    }
    IDbCommand cmd = conn.CreateCommand();
    cmd.CommandText = CHECK_SQL;
    cmd.Parameters.Add(
      new SqlParameter("@user",     user));
    cmd.Parameters.Add(
      new SqlParameter("@password", password));
    cmd.Parameters.Add(
      new SqlParameter("@resource", resource));
    IDataReader rdr = cmd.ExecuteReader();

    int rows = 0;

    if (rdr.Read())
      rows = rdr.GetInt32(0);

    cmd.Dispose();

    if (rows == 1) {
      logger.Log("Access granted");
      return true;
    }
    else {
      logger.Log("Access denied");
      return false;
    }
  }
}
```

AccessController1.cs

The test code for this is somewhat more complicated than the previous cases, mostly because we want to knit together all the various objects used to access the database (the connection, the command, various parameters, and the reader that returns the result). We also want to set up a reasonable set of expectations to ensure that the underlying code is calling the database layer correctly.

```
using DotNetMock.Framework.Data;
using NUnit.Framework;
using NUnit.Framework.SyntaxHelpers;
using System;

[TestFixture]
public class AnotherAccessControllerTest
{
  [Test]
  public void ValidUser()
  {
    MockLogger3 logger = new MockLogger3();
    logger.ExpectedName = "AccessControl";
-        logger.AddExpectedMsg(
15         "Checking access for dave to secrets");
-        logger.AddExpectedMsg("Access granted");
-
-        // set up the mock database
-        MockDbConnection conn = new MockDbConnection();
20       MockCommand cmd = new MockCommand();
-        MockDataReader rdr = new MockDataReader();
-
-        conn.SetExpectedCommand(cmd);
-        cmd.SetExpectedCommandText(
25         AccessController1.CHECK_SQL);
-        cmd.SetExpectedExecuteCalls(1);
-        cmd.SetExpectedParameter(
-          new MockDataParameter("@user",     "dave"));
-        cmd.SetExpectedParameter(
30         new MockDataParameter("@password", "shhh"));
-        cmd.SetExpectedParameter(
-          new MockDataParameter("@resource", "secrets"));
-
-        cmd.SetExpectedReader(rdr);
35       object [,] rows = new object[1,1];
-        rows[0, 0] = 1;
-        rdr.SetRows(rows);
-
-        AccessController1 access =
40         new AccessController1("secrets", logger, conn);
-
-        Assert.That(
-          access.CanAccess("dave", "shhh"),
-          Is.True
45       );
-        logger.Verify();
-        conn.Verify();
-        cmd.Verify();
-      }
50   }
```

AccessControllerTest1.cs

On line 1 we bring in the DotNetMock framework's `Data` components. In the body of the test method, we start by creating and setting up a mock logger as before. At line 19 we create three mock database objects: the connection, a command (used to issue SQL queries into the database), and a reader (used to return the results of a query).

We now need to associate these three objects together. Line 23 tells the connection object that when it is asked to generate a command object it should return our mock command object, `cmd`. We then set up that command object's expectations: the SQL it should receive, the number of times it will be executed, and the parameters it should expect to receive.

Line 34 starts the stanza that sets up the reader object. It

It isn't all perfect

Observant readers may be wondering why our new AccessController class went to the trouble of using a `Reader` object to get the count back from executing the query. Why didn't we just use the `ExecuteScalar` method of the command object to return the count directly?

Unfortunately, the mock object implementation of `IDbCommand` isn't quite complete (at least at the time of writing). Although `ExecuteScalar` is implemented, it always returns a `null` value. This means that we couldn't use it in our tests.

is first associated with the command (so that when the mock command is executed it will return this reader object). We then set up its result set, a two dimensional array of objects, containing the rows returned by the query and the columns in each row. In our case, the result set contains just a single row containing a single column, the count, but we still need to wrap it in the two-dimensional array.

Finally, on line 39, we create our access controller and check to see if "dave" can access the resource "secrets" by using the password "shhh." Because these values correspond to the values we set up for the query, the access controller will be able to use our mock database objects, which will return a count of "1" and the access will be accepted. At the end of the test, we then verify that the logger, connection, and command mock objects were used correctly by our method under test.

6.4 When Not To Mock

Mock objects are an appealing technology, but because they involve writing code, they represent a definite cost to a project. Whenever you find yourself thinking that you want to write a mock object to help with testing, stop and consider alternatives for a couple of seconds. In particular, ask yourself the simple question: "Do I need to write a mock object at all?" Sometimes we can eliminate the need for a mock object through some simple refactoring.

As a (somewhat contrived) example, let's imagine that we're writing code that downloads files to a handheld device over a relatively slow wire. Because of some hardware restrictions, after we've sent a block of data, we have to wait a while before trying to talk with the device again. The length of time we have to wait depends on the amount of data sent—the hardware guys gave us a table of values to use.

We might start off by writing a routine that waits a length of time dependent on the size of data sent:

```
public void WaitForData(int dataSize)
{
  int timeToWait;
  if (dataSize < 100)
  {
    timeToWait = 50;
  }
  else if (dataSize < 250)
  {
    timeToWait = 100;
  }
  else
  {
    timeToWait = 200;
  }
  Thread.Sleep(timeToWait);
}
```

Example.cs

Now we want to test this method, but there's a problem. The only way to see if it works is to check to see if it sleeps for the right amount of time for various values of the `dataSize` parameters. That's not an easy test to write: we'd have to build in a *fudge factor*, because the time we measure for the wait won't be exact. We might even have to set up some kind of watchdog thread to ensure that the sleep doesn't go on too long. There's also the elapsed time to consider. If running our tests causes `Thread.Sleep` to be called multiple times, our unit tests will take longer to complete—which won't increase our popularity amongst co-workers.

After reading this chapter, your first thought might be to solve these problems using a mock object. If we replace `Thread` with some kind of mock object, we can verify that its `Sleep()`

method was called with the expected values. Class Thread is not an interface, and even if it were, it has a boatload of properties and members.

This is the time to reflect: could we redesign our code slightly to make it easier to test? Of course we can!

```
public int HowLongToWait(int dataSize)
{
  int timeToWait;
  if (dataSize < 100)
  {
    timeToWait = 50;
  }
  else if (dataSize < 250)
  {
    timeToWait = 100;
  }
  else
  {
    timeToWait = 200;
  }
  return timeToWait;
}
public void WaitForData(int dataSize)
{
  Thread.Sleep(HowLongToWait(dataSize));
}
```

Example.cs

In this code we've split the waiting into two methods. One calculates the number of milliseconds to wait based on the data's size, and the other calls it to get the parameter to pass the Thread.Sleep(). If we assume that the framework Sleep() method works, then there's probably no need to test this second method: we can eyeball it and see it does what it says it should. That leaves us with the simple task of testing the method that calculates the time to wait.

```
[Test]
void WaitTimes()
{
  Waiter w = new Waiter();
  Assert.That(w.HowLongToWait(0), Is.EqualTo(50));
  Assert.That(w.HowLongToWait(99), Is.EqualTo(50));
  Assert.That(w.HowLongToWait(100), Is.EqualTo(100));
  Assert.That(w.HowLongToWait(249), Is.EqualTo(100));
  Assert.That(w.HowLongToWait(250), Is.EqualTo(200));
  Assert.That(w.HowLongToWait(251), Is.EqualTo(200));
}
```

Example.cs

A simple refactoring has led us to a better design and eliminated a whole lot of pain associated with coding up the tests.

Testing for time

Here's another, real-world example that shows how a simple refactoring makes for both an easier test and a better, more decoupled design. This is the code to be tested; note the dependency on the current system time.

```
public static string DaysFromNow(DateTime last)
{
  TimeSpan span = DateTime.Now - last;
  switch (span.Days)
  {
    case 0:
      return "Today";
    case 1:
      return "Yesterday";
    default:
      return span.Days + " days ago";
  }
}
```

On this particular project, one senior engineer spent a lot of time trying to invent a good way to fake out or change DateTime.Now. But then an intern from Portugal who learned C# via a few test-driven development books saw the code and made the obvious suggestion of extracting a parameter[FBB+99]. It took some time for the senior engineer to recover from a bad case of "bruised ego," but everyone agreed it was for the best.

The code was refactored to look like the following.

```
public static
string DaysFromNow(DateTime current, DateTime last)
{
  TimeSpan span = current - last;
  switch (span.Days)
  {
    case 0:
      return "Today";
    case 1:
      return "Yesterday";
    default:
      return span.Days + " days ago";
  }
}
```

Notice there is no dependency on the current date or time anywhere in the code; it is passed in from the caller. Now we can use a very simple test to drive this code.

```
[Test]
public void Yesterday()
{
  DateTime date = new DateTime(2007, 9, 27);
  DateTime dateMinusOneDay = new DateTime(2007, 9, 26);

  Assert.That(
    DaysFromNow(date, dateMinusOneDay),
    Is.EqualTo("Yesterday")
  );
}
```

Sometimes you can't change your existing interfaces to accept the parameterized singleton, or just want to do things in a more incremental fashion so we don't have to upheave the entire codebase. In that case, add a new interface that adds the parameterized singleton, then have the original interface delegate to the new one.

```
public static
string DaysFromNow(DateTime last)
{
  return DaysFromNow(DateTime.Now, last);
}
```

We *will* want to eventually get rid of this delegation, when practical, of course.

And that's all there is to mock objects: fake out parts of the real world so we can concentrate on testing our own code, which generally has a nice side-effect of improving our design. Some people might perceive that testing with fakes and mocks makes the testing less "real," but as you can see through the examples, we can actually test *interactions* that would be difficult and slow to reproduce accurately in the real world. Also, remember that it's called unit testing for a reason; we don't drag the whole system along for the ride because we want to test one behavior of a single class.

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 properties means to us.

7.1 Automatic

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

A-TRIP

Automatic Invocation

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 resources 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 if needed.

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] and other open source products that manage continuous building and testing.

[1] http://ccnet.thoughtworks.com

Automatic Checking

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. Also, in the interests of speed, note that any test that spews tons of console I/O (via `Console.WriteLine`, `log4net` or something similar) will slow down the unit tests—sometimes dramatically. We want unit tests to be silent, self-contained, and fast.

It’s an important feature of consistent regression to have the tests check the results for themselves. We humans aren’t very good at those repetitive tasks. We’ll make mistakes in the checking, and waste time investigating a bug that may not exist, or not catch a new bug that will go on to cause additional damage. The computer will not make these inconsistent mistakes; a properly written unit test will check the same thing every time it’s run with perfect consistency. 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

Good unit tests are thorough; they test everything that’s likely to break. But just how thorough? *A-$\boxed{T}$RIP*

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. For instance NCover, at http://ncover.org.[2] NCover produces XML files that describe the coverage, there are a couple of tools to visualize and explore that coverage data:

- NCoverExplorer[3], which is part of the TestDriven.NET extension to Visual Studio.NET.
- CruiseControl.NET comes with a very nice XSL file for transforming the NCover XML into some really cool-looking HTML.
- SharpDevelop 2.1 (and above) has NCover integration that will allow you to browse a tree-view of classes and methods. It also has as an option to highlight lines of code in the IDE that were not covered by the unit tests.

These tools can help you determine how much of the code under test is actually being exercised, as well as help you pinpoint what's *not* being exercised so you can focus your testing efforts.

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 next page).

This phenomenon leads to the well-known battle cry of "don't patch it, rewrite it." Often, it can be cheaper and less painful to throw out a piece of code you've written that has a clump of bugs and rewrite it from scratch. There's nothing that can improve code quite like a good old-fashioned disk crash.

But because it's usually more fun to write new code rather than refactor existing code, be careful with wholesale rewriting—especially if it's someone else's code. Rather than throw it out, first try to refactor someone else's code to make

[2]As of this writing, NCover does not work with Mono due to the way it hooks into Microsoft-specific portions of the CLR.

[3]http://kiwidude.com/blog

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.

it more unit-testable. Then if that's not working, you can go ahead and succumb to the sweet siren song of coding from scratch.

Either way, it will be safer to do: you'll have a set of unit tests that can confirm the new code works as it should.

7.3 Repeatable

A-T$\boxed{\textbf{R}}$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. That includes obvious external entities such as databases, system time, network conditions, but also perhaps less obvious dependents such as global variables. Any global state (in false singletons or otherwise) really isn't under your direct control—it only seems like it.

Something, somewhere, when you least expect it, will alter that state and you'll end spending a lot a quality time in the debugger trying to discover how you got into that state. That's the kind of frustration you just don't need.

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$\boxed{\textbf{I}}$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 you should use only one assert in a test, but that a test method should test only what the name implies—the same as regular methods in production code. If that means stitching a few methods together to accomplish the test, then so be it. Sometimes an entire test method might only test one small aspect of a complex piece of functionality—you may need multiple test methods to exercise the functionality thoroughly.

At any rate, you want to achieve 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 without looking at the test code itself. The name of the test should tell us all we need to know. Otherwise, we've got to go hunting for it, and that will just waste our time.

Independent also means that no test relies on any other test; we should be able to run any individual test at any time, and in any order. We *really* don't want to have to rely on any other test having run first, especially since the ordering will vary between the different test runners.

We've shown mechanisms to help you do this: the per-test setup and teardown methods and the per-fixture 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 NUnit tests will run in any particular order, and as you start combining tests 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

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 *A-TRI* **P** 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 62?).

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 behavior that might break. 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 property accessors:

```
public Money Balance
{
  get { return balance; }
}
```

Frankly, there's just not much here to go wrong that the compiler can't catch.[4] Testing methods such as these is just a waste of time. However, if the property is doing some work along the way, then suddenly it becomes interesting—and we will want to test it:

```
public Money Balance
```

[4]Unless, of course, the IL compiler, JIT compiler, or CLR itself has a bug.

```
    {
      get
      {
        return posted.GetBalance() -
               unposted.GetDebits() +
               unposted.GetCredits();
      }
    }
```

That's probably worth testing.

Finally, expect that, in the end, 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 project, then it would be reasonable to expect that there would be about 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 safety net—a missing test. This is an opportunity to close the hole, and make sure that this particular bug never escapes into the wild again. All it takes is four simple steps:

1. Identify the bug, or bugs, that caused the errant behaviour.
2. Write a test that fails, for each individual bug, to prove the bug exists.[5]
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 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, and potentially found an opportunity to refactor similar code into one place for easier testing, maintenance, and enhancement.

[5]Sometimes a bit of refactoring may need to happen so that tests can be written more easily.

```
[Test]
public void Add()
{
  // 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
  Assert.IsTrue(DatabaseHandler.Search(TEST_PERSON_1);
}
```

Figure 7.1: Test Adding a Person to a Database

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. The practice of pair programming further reduces the chance of these kinds of slip-ups, but you we may not always have someone to pair with. 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.

Same directory

Suppose you are writing a class named:

```
PragProg.Wibble.Account
```

with a corresponding test in:

```
PragProg.Wibble.AccountTest
```

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

This has the advantage that `AccountTest` can access `internal` and `protected internal` 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 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.

Separate Assemblies

The next option is to create your tests in a separate assembly from the production code.

This has the advantage of keeping a clean separation between code that you ship and code for testing.

The disadvantage is that now the test code is in a different assembly; You won't be able to access internal or protected internal 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:

```
namespace FacilitiesManagment {
  public class Pool {
    protected Date lastCleaned;
    public void xxxxx xx {
      xxx xxx xxxxx;
    }
    ...
  }
}
```

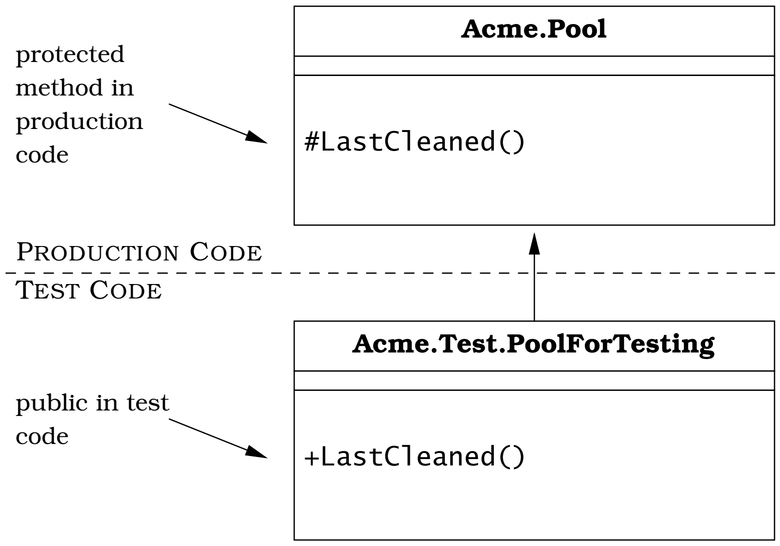


Figure 8.1: Subclasses Expose Methods for Testing

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 that exposes it just for testing.

```
using FacilitiesManagment;

namespace FacilitiesManagmentTesting {
  public class PoolForTesting : Pool {
    public Date LastCleaned {
      get { return lastCleaned; }
    }
  }
}
```

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

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 Where to Put NUnit

One issue that comes up on real projects is how to distribute NUnit itself.

You could have each individual developer install the latest version on their own workstations (as well as on the automated build machine). All the developers would have to install NUnit into the same directory, and make sure to reference that specific `nunit.framework.dll` assembly as not some random one in the GAC or elsewhere via a shortcut or symlink. This is all actually more difficult than it sounds, especially with little bugs that Visual Studio has with assembly references thrown in for fun.

Instead, you should distribute NUnit via your version control system. Many .NET and Java projects define both a `src/` directory and a `lib/` directory. The `src/` directory contains the source code to the project, and the `lib/` directory contains pre-compiled components (usually third-party).

In this context, you'd have a `lib/nunit/` directory that contains the NUnit binary distribution. Your projects and NAnt files would reference the `nunit.framework.dll` in this directory, and developers would run the `nunit.exe` GUI from this directory via a shortcut.

Now keeping developers' versions of NUnit synchronized is easy, as is deploying any upgrades or customizations. It keeps the environment consistent, freeing up time that would otherwise be spent on figuring out mismatched NUnit issues (which usually manifest themselves in odd ways). You may want to discourage developers from installing NUnit on their workstation to reduce confusion. If they do, keep an eye out for changes in the project or NAnt build files that reference NUnit assemblies other than those in the project's `lib/nunit/` directory.

8.3 Test Courtesy

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

Obfuscation and Packaging

Matt tells the following story about packaging, obfuscation, and manual maintenance:

"Recently I worked on a project that used a code obfuscation program, which the team thought helped protect their intellectual property. They packaged their unit tests in the same assembly as their production code, but the unit tests were `#if`'d out in the Release build. Any time a developer added a new test file, they had to remember to add the `#if` or risk violating the obfuscation policy. Or, did they?

Putting the tests into the assembly was reducing the effective design feedback of their packaging (which had major issues), so I proposed to extract the unit tests into a separate assembly so the design issues could be made more obvious. They said this was impossible because the unit tests were testing classes marked 'internal' and thus the tests had to be inside the same assembly as the production code.

I was curious why these classes had to be internal, and this turned up an amusing (albeit embarrassing) misunderstanding: the team thought that in order for the obfuscater to work properly, classes had to be marked as internal. That is, they thought public classes wouldn't be obfuscated in name or in code.

This was a mistake, of course. This particular obfuscater didn't really care, and I was able to configure it to obfuscate everything just fine. One of the neat tricks that came out of this was that this obfuscation product was able to take several assemblies that referenced each other, combine them into one binary, and prune out unused methods and code.

Because the unit tests weren't referenced directly in any of the application code, they were pruned out automatically. The manual `#if` statements they kept using could simply be removed. This also opened the door to making those internal classes public; the unit tests could then be extracted into a separate assembly, and design feedback could be obtained and acted upon appropriately."

When working with other members of a team, you will be using some sort of version control system, such as SubVersion, CVS, or (for the more masochistic among us), Visual SourceSafe. (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!

8.4 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
  Write and run tests that demonstrate bug; fix the bug 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].

8.5 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 NUnit 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 (the team in question happened to be using Java and JUnit for this particular project, but they could just as easily have been using C#, Cobol, C++, Ruby, or any other programming language):

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

[1]Not their real name.

includes a path to the object. Since the application 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 Cruise-Control) 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 unit testing 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.

8.6 Tests and Code 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 194 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) {}
```

That is to say, they simply ignored any exceptions that might have occurred. Not only did this result in random miss-

Delusional Exception Handling

Matt adds this story:

I was working on a project where the company's CTO littered the code with empty `catch`-all statements. When running a run-time analysis tool, I noticed that several dozen exceptions were being thrown and handled. Upon further inspection, I saw a piece of code that would almost always fail because it was—wait for it—dividing by a value that was zero most of the time.

The team found this apalling, so we spent a day cleaning up all the empty catch-all statements. The CTO was upset because the product was now more visibly unstable. He demanded the team put the bad exception handling back in. The team refused—the bugs and broken functionality were always present, the difference was we could *see* how bad it was.

ing 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. For reference, the minimal fix is to close (or dispose) resources in a `finally` clause. That way, they'll be cleaned up when control flow leaves the `try` block—whether an exception is thrown or not.

```
try
{
   DatabaseConnection dbc = new DatabaseConnection();
   InsertNewRecord(dbc, record);
}
finally
{
  dbc.Close()
}
```

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() {
  int howlong;
  xx xxxx x xxxx xx xx xxx;
  // Calculate how long to wait...
  x x xx xxx xxx x x xx;
  xx xxxx x xxxx xx xx xxx;

  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. We saw something similar back in Chapter 6, but this issue is important enough to revisit. Once again, we'll refactor the method in order 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() {
  int howlong = MilliSecondsToNextHour(DateTime.Now);
  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 .NET's `Thread.Sleep()` works as advertised (even if it doesn't, our rule is to always suspect our own code first, see the tip *Select Isn't Broken* in [HT00]). 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:

```
Assert.AreEqual(10000, MilliSecondsToNextHour(DATE_1));
```

If we're confident that `MilliSecondsToNextHour()` works to our satisfaction, then the odds are that `SleepUntilNext-Hour()` 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.

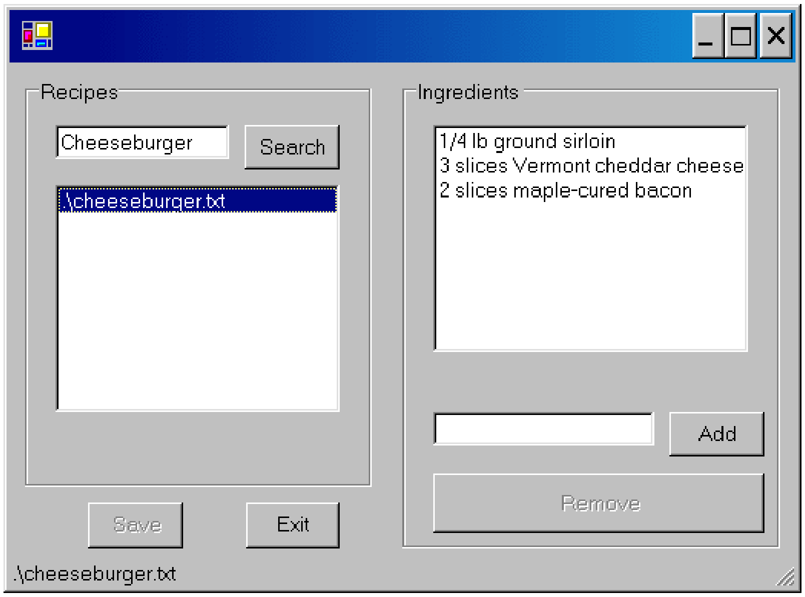


Figure 9.1: Recipes GUI Screen

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

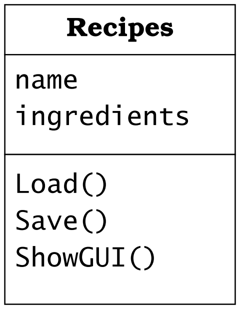


Figure 9.2: Original Recipes Static Class Diagram

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 on the preceding page, is pretty straightforward. There's only one class, with GUI behavior and file I/O intermixed.

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

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

cheeseburger.txt

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 138, 150, 157, and 166). In fact, the UML diagram for this class, shown in Figure 9.2, is kind of embarrassing—it's just one big class! Unfortunately, this kind of code is commonplace in many .NET projects because Visual Studio's designer-generated code gently coerces programmers to add logic directly into the Form or Control class. The forms designers for WinForms and ASP.NET are great tools—just be aware of these sinful temptations.

```
Line 1  using System;
     -  using System.Drawing;
```

```
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.IO;

public class Recipes : Form {
  private Button exitButton = new Button();
  private StatusBar statusBar = new StatusBar();
  private GroupBox groupBox1 = new GroupBox();
  private TextBox titleText = new TextBox();
  private Button searchButton = new Button();
  private ListBox searchList = new ListBox();
  private GroupBox groupBox2 = new GroupBox();
  private ListBox ingredientsList = new ListBox();
  private Button removeButton = new Button();
  private TextBox ingredientsText = new TextBox();
  private Button saveButton = new Button();
  private Button addButton = new Button();

  public Recipes() {
    InitializeComponent();
  }

  private void InitializeComponent() {
    exitButton.Location =
      new System.Drawing.Point(120, 232);
    exitButton.Size = new System.Drawing.Size(48, 24);
    exitButton.Text = "Exit";
    exitButton.Click +=
      new System.EventHandler(exitButton_Click);

    statusBar.Location = new System.Drawing.Point(0, 261);
    statusBar.Size = new System.Drawing.Size(400, 16);

    groupBox1.Controls.Add(searchList);
    groupBox1.Controls.Add(searchButton);
    groupBox1.Controls.Add(titleText);
    groupBox1.Location = new System.Drawing.Point(8, 8);
    groupBox1.Size = new System.Drawing.Size(176, 216);
    groupBox1.TabStop = false;
    groupBox1.Text = "Recipes";

    searchList.Location = new System.Drawing.Point(16, 56);
    searchList.Size = new System.Drawing.Size(144, 147);
    searchList.SelectedIndexChanged +=
      new System.EventHandler(
        searchList_SelectedIndexChanged);

    searchButton.Location = new System.Drawing.Point(112, 24);
    searchButton.Size = new System.Drawing.Size(48, 24);
    searchButton.Text = "Search";
    searchButton.Click +=
      new System.EventHandler(searchButton_Click);

    titleText.Location = new System.Drawing.Point(16, 24);
```

```
      titleText.Size = new System.Drawing.Size(88, 20);

      groupBox2.Controls.Add(addButton);
      groupBox2.Controls.Add(ingredientsText);
      groupBox2.Controls.Add(removeButton);
      groupBox2.Controls.Add(ingredientsList);
      groupBox2.Location = new System.Drawing.Point(200, 8);
      groupBox2.Size = new System.Drawing.Size(192, 248);
      groupBox2.TabStop = false;
      groupBox2.Text = "Ingredients";

      addButton.Location = new System.Drawing.Point(136, 176);
      addButton.Size = new System.Drawing.Size(48, 23);
      addButton.Text = "Add";
      addButton.Click +=
        new System.EventHandler(addButton_Click);

      ingredientsText.Location = new System.Drawing.Point(16, 176);
      ingredientsText.Size = new System.Drawing.Size(112, 20);

      removeButton.Enabled = false;
      removeButton.Location = new System.Drawing.Point(16, 208);
      removeButton.Size = new System.Drawing.Size(168, 32);
      removeButton.Text = "Remove";
      removeButton.Click +=
        new System.EventHandler(removeButton_Click);

      ingredientsList.Location = new System.Drawing.Point(16, 24);
      ingredientsList.Size = new System.Drawing.Size(160, 134);
      ingredientsList.SelectedIndexChanged +=
        new System.EventHandler(
          ingredientsList_SelectedIndexChanged);

      saveButton.Enabled = false;
      saveButton.Location = new System.Drawing.Point(40, 232);
      saveButton.Size = new System.Drawing.Size(48, 24);
      saveButton.Text = "Save";
      saveButton.Click +=
        new System.EventHandler(saveButton_Click);

      AutoScaleBaseSize = new System.Drawing.Size(5, 13);
      ClientSize = new System.Drawing.Size(400, 277);
      Controls.Add(saveButton);
      Controls.Add(groupBox2);
      Controls.Add(groupBox1);
      Controls.Add(statusBar);
      Controls.Add(exitButton);
      groupBox1.ResumeLayout(false);
      groupBox2.ResumeLayout(false);
      ResumeLayout(false);
    }

    [STAThread]
    static void Main() {
      Directory.SetCurrentDirectory(@"..\..\recipes\");
    Application.Run(new Recipes());
  }

  private void exitButton_Click(object sender,
                                System.EventArgs e) {
    Application.Exit();
  }

  private void searchButton_Click(object sender,
                                  System.EventArgs e) {
    String toMatch = "*" + titleText.Text + "*";

    try {
      string [] matchingFiles = Directory.GetFiles(@".", toMatch);
      searchList.DataSource = matchingFiles;
    }
    catch (Exception error) {
      statusBar.Text = error.Message;
    }
  }

  private void
  searchList_SelectedIndexChanged(object sender,
                                  System.EventArgs e) {
    string file = (string)searchList.SelectedItem;
    string line;
    char [] delim = new char[] { '=' };

    statusBar.Text = file;

    using (StreamReader reader =
             new StreamReader(file)) {
      while ((line = reader.ReadLine()) != null) {
        string [] parts = line.Split(delim, 2);
        switch (parts[0]) {
          case "NAME":
            titleText.Text = parts[1];
            break;
          case "INGREDIENTS":
            try {
              int count = Int32.Parse(parts[1]);
              ingredientsList.Items.Clear();
              for (int i = 0; i < count; i++)
                ingredientsList.Items.Add(reader.ReadLine());
            }
            catch (Exception error) {
              statusBar.Text = "Bad ingredient count: " +
                 error.Message;
              return;
            }
            break;
          default:
            statusBar.Text = "Invalid recipe line: " + line;
            return;
        }
      }
      }
      saveButton.Enabled = false;
    }

    private void removeButton_Click(object sender,
                                    System.EventArgs e) {
      int index = ingredientsList.SelectedIndex;
      if (index >= 0) {
        statusBar.Text = "Removed " +
          ingredientsList.SelectedItem;
        ingredientsList.Items.RemoveAt(index);
        saveButton.Enabled = true;
      }
    }

    private void addButton_Click(object sender,
                                 System.EventArgs e) {
      string newIngredient = ingredientsText.Text;
      if (newIngredient.Length > 0) {
        ingredientsList.Items.Add(newIngredient);
        saveButton.Enabled = true;
      }
    }

    private void
    ingredientsList_SelectedIndexChanged(object sender,
                                         System.EventArgs e) {
      int index = ingredientsList.SelectedIndex;
      if (index < 0)
        removeButton.Enabled = false;
      else {
        removeButton.Text = "Remove " +
          ingredientsList.SelectedItem;
        removeButton.Enabled = true;
      }
    }

    private void saveButton_Click(object sender,
                                  System.EventArgs e) {
      string fileName = titleText.Text + ".txt";
      ICollection items = ingredientsList.Items;
      using (StreamWriter file =
               new StreamWriter(fileName, false)) {
        file.WriteLine("NAME={0}", titleText.Text);
        file.WriteLine("INGREDIENTS={0}", items.Count);
        foreach (string line in items) {
          file.WriteLine(line);
        }
      }
      statusBar.Text = "Saved " + fileName;
    }
  }
```

Recipes.cs

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.

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;

public class Recipe
{
  protected string name;
  protected List<string> ingredients;

  public Recipe()
  {
    name = string.Empty;
    ingredients = new List<string>();
  }

  public Recipe(Recipe another)
  {
    name = another.name;
    ingredients = new List<string>(another.ingredients);
  }

  public string Name
  {
    get { return name; }
    set { name = value; }
  }

  public ReadOnlyCollection<string> Ingredients
  {
    get
    {
      return
        new ReadOnlyCollection<string>(ingredients);
    }
  }

  public void AddIngredient(string ingredient)
  {
    ingredients.Add(ingredient);
  }
}
```

Recipe.cs

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.

```
public class RecipeFile
```

```
{
  public Recipe Load(Stream savedRecipe)
  {
    Recipe recipe = new Recipe();
    string line;
    char[] delim = new char[] { '=' };

    using (StreamReader reader = new StreamReader(savedRecipe))
    {
      while ((line = reader.ReadLine()) != null)
      {
        string[] parts = line.Split(delim, 2);

        switch (parts[0]) {
          case "TITLE":
          {
            recipe.Name = parts[1];
            break;
          }
          case "INGREDIENTS":
          {
            try
            {
              int count = Int32.Parse(parts[1]);
              for (int i = 0; i < count; i++)
                recipe.AddIngredient(reader.ReadLine());
            }
            catch (Exception error)
            {
              throw new RecipeFormatException(
                "Bad ingredient count: " + error.Message);
            }
            break;
          }
        }
      }
    }

    return recipe;
  }

  public void Save(Stream savedRecipe, Recipe recipe)
  {
    using (StreamWriter file =
           new StreamWriter(savedRecipe))
    {
      file.WriteLine("NAME={0}", recipe.Name);
      file.WriteLine(
        "INGREDIENTS={0}",
        recipe.Ingredients.Count
      );

      foreach (string line in recipe.Ingredients)
      {
        file.WriteLine(line);
      }
    }
  }
}
```

RecipeFile.cs

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.

```
using NUnit.Framework;
using NUnit.Framework.SyntaxHelpers;
using System;
using System.Collections.Generic;
using System.IO;

[TestFixture]
public class RecipeTest
{
  const string CHEESEBURGER =
    "Cheeseburger";
  const string SIRLOIN =
    "1/4 lb ground sirloin";
  const string CHEESE =
    "3 slices Vermont cheddar cheese";
  const string BACON =
    "2 slices maple-cured bacon";
  const string RECIPE_FILE_NAME =
    "recipe.save";

  [TearDown]
  public void TearDown()
  {
    if (File.Exists(RECIPE_FILE_NAME))
    {
      File.Delete(RECIPE_FILE_NAME);
    }
  }

  [Test]
  public void SaveAndRestore()
  {
    Recipe recipe = new Recipe();
    recipe.Name = CHEESEBURGER;
    recipe.AddIngredient(SIRLOIN);
    recipe.AddIngredient(CHEESE);
    recipe.AddIngredient(BACON);

    Stream recipeStream;
    RecipeFile filer;
    using (recipeStream =
        File.OpenWrite(RECIPE_FILE_NAME))
    {
      filer = new RecipeFile();
      filer.Save(recipeStream, recipe);
    }
```

```
 -
 -        // Now get it back
 -        using (recipeStream =
50            File.OpenRead(RECIPE_FILE_NAME))
 -        {
 -          filer = new RecipeFile();
 -          recipe = filer.Load(recipeStream);
 -        }
55
 -        Assert.That(recipe.Ingredients.Count, Is.EqualTo(3));
 -
 -        Assert.That(
 -          recipe.Name,
60          Is.EqualTo(CHEESEBURGER)
 -        );
 -
 -        Assert.That(
 -          recipe.Ingredients[0],
65          Is.EqualTo(SIRLOIN)
 -        );
 -
 -        Assert.That(
 -          recipe.Ingredients[1],
70          Is.EqualTo(CHEESE)
 -        );
 -
 -        Assert.That(
 -          recipe.Ingredients[2],
75          Is.EqualTo(BACON)
 -        );
 -      }
 -    }
```

RecipeTest.cs

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 34. We could just pass literal strings directly into the object instead, and not bother with `const` data members, 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 45. Now we can make a brand-new `Recipe` object, and ask the helper to load it from that same file at line 53.

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 26 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...

```
Failures:
1) RecipeTest.SaveAndRestore :
   Expected string length 12 but was 0.
   Strings differ at index 0.
  Expected: "Cheeseburger"
  But was:  <string.Empty>
  -----------^
  at RecipeTest.SaveAndRestore() in RecipeTest.cs:58
```

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.cs`, the code is using the key string `"NAME"` to identify the field, but when we read it back in (line 19 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:

```
public class RecipeFile
{
  const string NAME_TOKEN = "NAME";
  const string INGREDIENTS_TOKEN = "INGREDIENTS";

  public Recipe Load(Stream savedRecipe)
  {
    Recipe recipe = new Recipe();
    string line;
    char[] delim = new char[] { '=' };

    using (StreamReader reader = new StreamReader(savedRecipe))
    {
      while ((line = reader.ReadLine()) != null)
      {
        string[] parts = line.Split(delim, 2);

        switch (parts[0]) {
          case NAME_TOKEN:
          {
            recipe.Name = parts[1];
            break;
          }
          case INGREDIENTS_TOKEN:
          {
            try
            {
              int count = Int32.Parse(parts[1]);
              for (int i = 0; i < count; i++)
                recipe.AddIngredient(reader.ReadLine());
            }
            catch (Exception error)
            {
              throw new RecipeFormatException(
                "Bad ingredient count: " + error.Message);
            }
            break;
          }
        }
      }
    }

    return recipe;
  }

  public void Save(Stream savedRecipe, Recipe recipe)
  {
    using (StreamWriter file =
           new StreamWriter(savedRecipe))
    {
      file.WriteLine(
        "{0}={1}",
        NAME_TOKEN,
        recipe.Name
     );

      file.WriteLine(
        "{0}={1}",
        INGREDIENTS_TOKEN,
        recipe.Ingredients.Count
      );

      foreach (string line in recipe.Ingredients)
      {
        file.WriteLine(line);
      }
    }
  }
```

```
    }
```

RecipeFile.cs

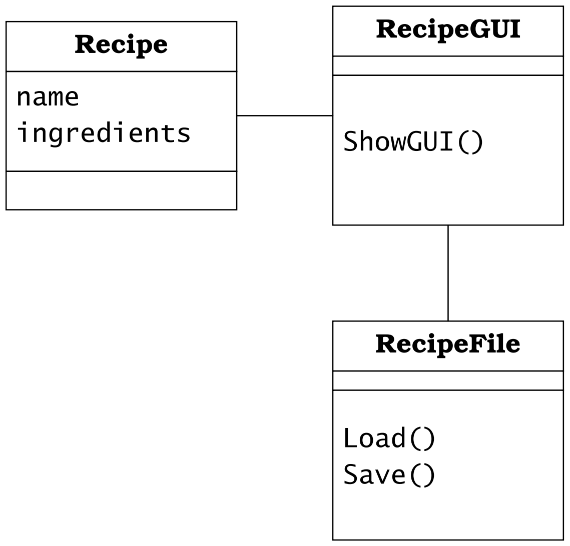


Figure 9.3: Refactored `Recipes` Static Class Diagram

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` to narrow the class' responsibility.
- Pulled literals into constants to avoid bugs from typos and reduce duplication.

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 a GUI can be very hard, but usually because the code is written in such a way as to make it difficult. This kind of code isn't uncommon, either—this is what happens when you use the WinForms designer to generate code and then just integrate logic directly into the generated method's code. It can be tempting to use something like NUnitForms to test this kind of logic, but if we went that route our tests would end up long and complicated.

By separating the pure GUI-related code from the actual logic of the application, you can easily add and test business features without having to worry about how you're going to weave it into the GUI code.

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 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.

[1]For more information on pre-conditions, post-conditions and invariants, see [Mey97].

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 77.

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 [wCA04]. 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() {
  xxx xx xxxxx xxx xx x xx xxx xxx xxxx xx xx;
  x xx x xxx xxxx xx xxx xx xxxxx xxxx;
  AddCropMarks(str, 8.5, 11.0, 6.0, 8.5);
  xx xxx x xxx xxx xx x xxx xxx xxx xxx xx xxx;
  x xxx xxx xxxx x xxx xxx  xxx xxxx xx xxx xx;
  AddCropMarks(str, 8.5, 11.0, 6.0, 8.5);
  xx xx xxxx xx xx xxx xxx  xxx xxxx xx xx xx  xxx;
  x xx xx x xxxx xxx x xxxx xx xx xx xxx xxx xxx;
  AddCropMarks(str, 8.5, 11.0, 6.0, 8.5);
  xxx xx xxxxxxx xxx  xxx xxxxx  x xxx xxxx xx xxxxxxx;
  xx  x xxx xxxx xxxx xxx  xxxx xxxx xx x x xx xxx;
  AddCropMarks(str, 5.0, 7.0, 4.0, 5.5);
  xx xxx xxx xx x xxx xxx  xxx xxxx xx xx xx xxx xx;
  xxx xx xxxxx xxx xx xxx x xxx xxxx xx xx xx xxx;
  AddCropMarks(str, 5.0, 7.0, 4.0, 5.5);
  xx xx xxxxx xx x xx xxx xxx xxxx xx xx;
  x xxx  x xxx xxxx xx xx xxx xxxx xxx;
}
```

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);
    [illegible]
    [illegible]
    AddCropMarks(str, standardPaper1);
    AddCropMarks(str, standardPaper1);
    [illegible]
    [illegible]
    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 LetterInstance;
  public static PaperSpec A4Instance;
  public static PaperSpec LegalInstance;
  [illegible]
  [illegible]
}
```

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

7. 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 201

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.

Chapter 10

GUI Testing

Now that we've separated out the logic from our UI code, what is there left to test in the GUI? And how is this a "unit test" when the GUI is involved?

10.1 Unit testing WinForms

We're going to see how this works in the real world using the NUnitForms framework, which is an extension of NUnit (`http://nunitforms.sourceforge.net`). Alas, NUnitForms uses Win32 native calls to work its magic and therefore doesn't currently work under Mono. Because NUnitForms itself depends upon NUnit, we may find that the version of

Andy's Rant on GUI Testing

"Some people are convinced that they *must* compare bitmaps to do GUI testing. Well, that is simply the most antiquated, 1970's bit of thinking I can imagine. For crying out loud, I wrote a GUI tester based on X11 back in 1992 or so that was object-oriented (i.e., it worked with the 'OK' button object on a form, placement was irrelevant), scriptable, could do live record and playback and then later editing of the event, test composition, etc. And that was some 15 years ago."

nunit.framework.dll we are referencing during compilation of our code isn't the same version as the one NUnitForms was built against. (The compiler will actually warn us of this.) Don't panic, we'll talk more about this later in the chapter.

Let's get started. There's no magic here; remember that unit tests are just code, and controls (forms included) are just objects. For instance, we can create and use additional constructors, and not just be stuck with the default empty constructor. If a Form requires a Recipe object to display, for example, then the Form should have a constructor that takes a Recipe parameter.

Now we've got a first easy unit test that doesn't require NUnitForms—pass null in for the parameter and expect an ArgumentNullException:

```
[TestFixture]
public class RecipeViewFormTests
{
  [Test]
  [ExpectedException(typeof(ArgumentNullException))]
  public void NullRecipe()
  {
    new RecipeViewForm(null);
  }
}
```

What can we test that's actually GUI-related? The RecipeViewForm has two buttons: Save and Cancel. You want to make sure that the Save button calls the Save() method on the Recipe object that is passed to it. (We're only concerned with the GUI functionality of the Save button—the logic behind the Recipe.Save() method is tested elsewhere.) We'll use mock objects to make our lives easier.

```
using NUnit.Framework;
using NUnit.Framework.SyntaxHelpers;
using NUnit.Extensions.Forms;
using RecipeViewer;
using System;

namespace RecipeViewer.Tests
{
  public class FakeRecipe : Recipe
  {
    UInt32 saveCalled = 0;

    public UInt32 SaveCalled
    {
      get { return saveCalled; }
    }
    public override Save()
    {
      saveCalled++;
    }
  }
  [TestFixture]
  public class RecipeViewFormTests
  {
    [Test]
    public void Save()
    {
      FakeRecipe recipe = new FakeRecipe();
      RecipeViewForm recipeView =
        new RecipeViewForm(recipe as Recipe);
      recipeView.Show();
      ButtonTester saveButton = new ButtonTester("Save");
      saveButton.Click();
      Assert.That(recipe.SaveCalled, Is.EqualTo(1));
    }
  }
}
```

First, we create a fake object for `Recipe` which tracks the number of calls to the `Save()` method. Then, we make a new `RecipeViewControl` and give it our `FakeRecipe` object, add the control to a form, and finally call the `Show()` method on the form. When we run the test, the form with the control will pop up quickly (don't blink or you'll miss it).

Next, we create a `ButtonTester` for the Save button. Note that the `ButtonTester` isn't based on the contents of the Button's Text property, but rather the Name property. Make sure you give these sane names and not use the default ones generated by the designer.

We then call the `Click()` method on the ButtonTester, and ask the fake Recipe how many times `Save()` was called. We want to assert that it was called only once.

Pretty cool, huh? By faking the model, we kept the unit test very focused, even though it was testing the GUI. We could also use one of the mock object frameworks discussed in Chapter 6.

Here's another example we want to make sure works: pressing the Cancel button *doesn't* save the recipe.

```
[Test]
public void Cancel()
{
  FakeRecipe recipe = new FakeRecipe();
  RecipeViewControl recipeView =
    new RecipeViewControl(recipe as Recipe);
  Form form = new Form();
  form.Add(recipeView);
  form.Show();
  ButtonTester cancelButton = new ButtonTester("Cancel");
  cancelButton.Click();
  Assert.That(recipe.SaveCalled, Is.EqualTo(0));
}
```

We introduced a little duplication here with the previous test, so it's time to refactor a bit. First, extract `recipe` and `recipeView` to be class-level fields; then extract the intialization of those fields into `SetUp()` so they're fresh for each test method. We've eliminated duplicate code, so we're ready to proceed.

Mocking the User

NUnitForms can also simulate a user changing fields in the GUI via the keyboard, and all other kinds of things, relatively easily. There's only one major exception and that's modal dialogs.

```
[TestFixture]
public class LoginModalDialogTest : NUnitFormTest
{
  const string PASSWORD_FAILURE = "Password Failure";
  [Test]
  public void PasswordFailureClickOK()
  {
    ExpectModal(PASSWORD_FAILURE, "PasswordFailureOkHan-
dler");
    MessageBox.Show("Try again?", PASSWORD_FAILURE);
  }
  public void MessageBoxOkHandler()
  {
    MessageBoxTester messageBox =
      new MessageBoxTester(PASSWORD_FAILURE);
    Assert.That(
      messageBox.Title,
      Is.EqualTo(PASSWORD_FAILURE)
    );
    messageBox.ClickOk();
```

```
        }
    }
```

Modal dialogs are interesting because they suspend the program until they're dismissed. Thankfully, NUnitForms has a way to deal with that, using the `ExpectModal` method in the `NUnitFormTest` class. To use it (or any other NUnitForms methods), we derive our fixture from `NUnitFormTest` and call `ExpectModal`, passing the name of the caption (aka title) of the modal dialog. When a modal dialog is displayed that has the specified caption, the handler method is called. So in our handler method, we do our button clicks, assertions, and so on, and then dismiss the dialog as a user would. Then our tests continue on their merry way.

10.2 Unit testing beyond Windows Forms

What if we're using a UI library that isn't Windows Forms? This isn't unthinkable, and it certainly isn't untestable either. There are some nuances, but many of the concepts presented thus far apply equally. We just won't have a nice framework like NUnitForms to help us along.

For other common GUI toolkits, like Qt# and Gtk#, there is some variance in the ease of testing. Qt# is a .NET binding to the open source C++ native library, Qt. Qt 4.1 and above has a built-in unit testing framework called QTestLib. Gtk# is also a wrapper (in essence), but neither the wrapper nor the native library has a unit testing framework associated with it as of the time of this writing.

What about custom GUIs, like ones that are 3D?[1] That turns out to be easier in some cases, because we have more control over the design and can make it easy to test.

Most 3D applications use a scene graph, which is a tree of nodes where the nodes in the tree represent things to be drawn in 3D space. The nodes know their X, Y, and Z (depth) coordinates, and their length along those planes. A visitor class visits each node in the scene graph and is responsible

[1] For instance, a C# wrapper around OpenGL, like Tao: `http://www.taoframework.com`

Testing code that runs on the GPU

Thanks to Ryan Dy, a programmer on some of Matt's favorite XBox games, for this real-world detail:

Sometimes there are transformations that aren't done on the CPU, they're done on the GPU via shader programs. In that case, we need to expose the shader variables to our test code running on the CPU so we can make assertions. To accomplish that, we would write a shader program that would expose the values we need to assert against in our test. This is a common method for debugging shader code, and it also allows us to make sure our shader perform similarly across different hardware implementations in an automated fashion.

for things like rendering the node in the 3D space or passing messages such as mouse clicks or keyboard interactions.

It's relatively straightforward to see where testing could be introduced in this common 3D scenario. First, we may want to test the nodes themselves, but they are usually data-only—all the behaviour generally goes into the visitor[2] objects that apply transformations to the data contained in the nodes.

Next, we could test the scene graph collection and make sure it is self-balancing based upon Z-order (or whatever other properties we expect). Last, the visitor classes themselves can be unit tested if they are well-encapsulated and loosely coupled with the rest of the design.

This is a lot of talk to be sure; how can you actually code up a test that makes sure that our layout algorithm fits all the nodes onto the rendered screen? We might sketch out a test that looks like this:

Missing: [Code to be written]

What's this magical method referenced? It's sometimes easier

[2]Fun fact: Scene graphs are one of the few places that the Visitor design pattern is commonly applied.

to write the test as you'd like it to read, then work backward and fill in the missing pieces.

Now that we have the test as we'd like it to read, here's the code for the aforementioned magical method:

Missing: [Code to be written]

10.3 Web UIs

An entire book could be written about testing web-based UIs. We'll touch on it briefly here because many people are under the impression this is not possible or requires expensive commercial tools. First, many "Web 2.0" applications have a great deal of their functionality in JavaScript (aka ECMAScript). Unlike the dark ages of JavaScript, it is now an open ECMA standard with frameworks available that make object-orientation and unit testing a snap. In many modern applications, much of the important end-user functionality is on the client-side in JavaScript. The server-side code mostly accepts AJAX requests that either retrieve or store data in the database with some data validation and logging.

As such, unit testing the JavaScript is the first step. We recommend JsUnit,[3] which provides a framework and a test-runner that can run within most modern browsers. You can assert that your JavaScript code is having the correct effect on specific DOM elements, such as adding or removing styles, child nodes, or whatever. This allows us to find and test for bugs that would normally have to be done manually with visual inspection.

We also recommend using a framework like Prototype or JQuery that provide various syntactic and functional helpers that make JavaScript a little easier to code and test. There are all sorts of nifty AJAX libraries and frameworks out there, but we should make sure that they don't hinder our ability to unit test functionality. See the JsUnit web site for examples—many of the concepts from this book can be applied equally between C#, JavaScript, and other languages.

[3] http://www.jsunit.net

To test web applications beyond JavaScript and our server-side objects, there is a free, open source tool called Selenium.[4] With Selenium, you can write code that drives any of the mainstream browsers on the operating system of your choice. It works by running a server that launches a browser, which accepts commands via a socket and translates those commands into browser clicks and keyboard input.

This means we can write NUnit tests that look like this:

```
[Test]
public void AnchoviesNotAvailableInMontana()
{
  ISelenium selenium =
    new DefaultSelenium(
      "localhost", 4444, "firefox2",
      "http://localhost:56789/OrderPizza.aspx"
    );
  selenium.Select(INGREDIENT_DROPDOWN_ID, "anchovies");
  selenium.Type(STATE_TEXT_ID, "montana");
  selenium.Click("submit");
  selenium.WaitForCondition(
        "selenium.isTextPresent('Not Available')"
  );
}
```

We instantiate a new selenium controller, which starts the selenium server. This in turn starts the browser. We tell the selenium controller to select "anchovies" from a list control. Note that the location and style of that control don't really matter—we're just working off the HTML IDs. Because we have stored the HTML IDs into a variable, we only have to change them in one place should the HTML ID in the user interface change. Then, we tell selenium to type 'montana' in the input control. Next, we tell selenium to click a button with the HTML ID of 'submit'. Last, we wait for the validator (or whatever else) text to appear. If the condition isn't met by the default timeout,[5] the test will fail. One interesting side note is that `selenium.isTextPresent` is a snippet of JavaScript that will tell Selenium what to do on the browser-side.

Selenium tests are like any other tests; you tend to do the same things over and over. Being the pragmatic programmers that we are, we don't stand for duplication. When we see

[4]http://www.openqa.org/selenium/index.html
[5]60 seconds in Selenium 0.9

Joe Asks...
Aren't Selenium tests more like system tests?

Yes, Selenium tests aren't really unit tests, even though we are driving the browser in NUnit. ASP.NET doesn't have a good way to isolate the various handlers for testing as of the time of this writing.[a] As such we have a multi-lateral approach to get much of the same benefit. The ASP.NET pages and controls should be a very thin layer on top of other, more easily testable, objects—just like for WinForms or any other widget library. Selenium then helps us test the interaction between the web controls and those underlying model objects. It is slower, mainly due to the overhead of starting and running a real browser, but it is definitely better than manual web UI testing. When using a system testing tool like Selenium, make sure to exclude it from your code coverage measurements. Your unit tests alone should provide high levels of code coverage; measuring the coverage of system-level tests obscures that data.

[a]WebWork and Rails do, which are Java- and Ruby-based respectively.

it, we refactor by extracting methods, extracting a class, and performing other refactorings. A very common pattern with Selenium is to wrap the Selenium instance and delegate to it. By doing this, you can have assertion and helper methods tied to a project-specific Selenium object that can be shared.

```
namespace PizzaWeb.Test.UI
{
  public class MySelenium
  {
    protected ISelenium selenium;
    public MySelenium(string host,
                      int port,
                      string[] browsers,
                      string url)
    {
      selenium =
        new DefaultSelenium(host, port, browsers, url);
    }
```

```
        public Stop()
        {
          selenium.Stop();
        }
        public void waitForText(string expectedText)
        {
          selenium.WaitForCondition(
            "selenium.isTextPresent('" + expectedText + "')"
          );
        }
      }
    }
```

Something else that often gets repeated is the creation of the selenium instance and the closing of the browser. Another common pattern is to have a base class that selenium-oriented fixtures derive from.

```
    namespace PizzaWeb.Test.UI
    {
      public abstract class MySeleniumFixture
      {
        static final uint SELENIUM_SERVER_PORT = 56789;
        protected MySelenium selenium;
        [TestFixtureSetUp]
        public void StartBrowser()
        {
          selenium = new MySelenium(
            "localhost", SELENIUM_SERVER_PORT,
            "firefox2", getInitialUrl()
          );
        }
        [TestFixtureTearDown]
        public void StopBrowser()
        {
          selenium.Stop();
        }
        protected abstract string getInitialUrl();
      }
      [TestFixture]
      public class OrderPizzaTest : MySeleniumFixture
      {
        static final string INGREDIENT_DROPDOWN_ID = "ingredi-
ents";
        static final string STATE_TEXT_ID = "state";
        string getInitialUrl()
        {
          return "http://localhost:7890/OrderPizza.aspx";
        }
        [Test]
```

```
            public void AnchoviesNotAvailableInMontana()
            {
              selenium.Select(
                INGREDIENT_DROPDOWN_ID,
                "anchovies"
              );
              selenium.Type(STATE_TEXT_ID, "montana");
              selenium.Click("submit");
              selenium.waitForText("Not Available");
            }
          }
        }
```

When making the selenium instance, only one thing generally varies from test to test: the initial URL the browser loads. To reuse the creation of the selenium instance, we extracted it into a method and then into a base class that the test fixture itself derives from. We marked that creation method with the `TestFixtureSetUp` attribute so we don't keep closing and opening the browser for every test. Your application may need to close the browser for each test, though, in which case we should use `SetUp` and `TearDown` instead. The base class defines the abstract method called `getInitialUrl()` which the derived class must implement. When we add a new test fixture for a different web page, we'll override that method and get the benefits of reuse.

In our example we set the default browser to 'firefox2'. If you want to test with Internet Explorer as well, you can make sure your tests pass under both browsers by adding "IE6" to the third parameter of the `DefaultSelenium` constructor.

10.4 Command Line UIs

Before we finish talking about GUI Testing, we can't forget about our old friend the command line. Once again, the first step is to make sure that our static `Main()` method is a thin layer that mostly interacts with other, more easily testable, objects. Often, argument parsing is done in a quick and dirty fashion right in the `Main()` method. What do you do if there is a bug in the command line argument parsing, and you want to write a unit test that fails when the bug is present and passes when it is fixed? Say you had code like this:

```
private static void isTracing;
```

```
public static void Main(string[] args)
{
  if (args.Length < 1)
  {
    printUsage();
    Environment.Exit(-1);
  }
  if (args[0] == "--trace")
  {
    isTracing = true;
  }
}
```

There is a bug (or lack of feature, depending on your personal outlook) where the `-trace` command line option is only recognized when it is the first argument. We want to unit test the change, regardless, because text processing is one of those areas in our experience where bugs tend to creep back in as seemingly "safe" changes are made. One way would be to write a test like this:

```
[Test]
public void TraceAsSecondArgument()
{
  TextUI.Main(new String[] {"filename", "--tracing"});
  Assert.That(Main.IsTracing, Is.True);
}
```

This test wouldn't compile as-is—we would have to add a static property called `IsTracing` to our class. If you find yourself thinking this doesn't feel right, we would agree with you. Like the other UI testing paradigms we've discussed, we want `Main()` to be a thin layer that does a little coordination between other objects. Adding a property makes it fatter rather than thinner.

Instead, let's first extract a method which will help highlight some better seams along which we can extract a class that we can then unit test.

```
private static bool hasTracing(string[] args)
{
  return (args[0] == "--trace");
}
```

Now here's something we can unit test more easily. Testing the `Main()` class still feels a little weird, so let's extract that static method into a class called `Args`. Once we do that, we can write a test like this:

```
    [Test]
    public void TraceAsSecondArgument()
    {
      Args args = new Args(new string[] {"filename", "--
tracing"});
      Assert.That(args.IsTracing, Is.True);
    }
```

This example backs up Andy's rant from the beginning of the chapter. Unit testing most GUI code is hard only because people think it is, not because it is that technically challenging.

Once we just approach the problem as though it is solvable and apply those amazing programming skills we posess, it becomes one of the more trivial issues we'll deal with in our professional career.

10.5 GUI Testing Gotchas

GUI testing is fairly straight-forward, save for a couple of gotchas that we'll discuss in this section. Don't be scared—knowing about these issues up front deflates their difficulty quite a bit.

Conflicting NUnit libraries

If NUnitForms was built against a specific version of NUnit and that differs from the version of the NUnit libraries you're referencing in your project, you'll get a compiler warning telling you as such. This usually doesn't present a problem, but if it does there's an easy way to fix it.

Download the NUnitForms source code, and replace it's NUnit libraries with the version of NUnit you're using. Then build NUnitForms and store the custom build in your project's `lib/` directory. This seems like a big deal, but it's a minor annoyance at worst. You'll have to rebuild when there's a new NUnitForms release that you *must* deploy, but that's about it.

Automated build

There's a simple "gotcha" here worth mentioning when using NUnitForms (or similar tools) in an automated build.

If your automated build runs as a service, you will be unable to show any modal dialogs. Attempting to do so will result in an exception that basically says "don't show modal dialogs in a service." This is historical, and does make some sense. Since the service doesn't have a desktop where someone could dismiss a modal dialog, the service would get stuck and the machine would require a reboot. This happened enough times with poorly written commercial applications that Microsoft nipped it in the bud by disallowing it altogether.

There is an easy workaround, and it's only a little messy. Start your automated build controller from a logged-in account, via a batch file or shell script. Then set the build machine to auto-login on boot and run the batch file on start-up for that user. It ends up practically the same as running the service, but you won't run into the aforementioned issue with modal dialogs.

Multithreaded and Complex Controls

Now, for a more advanced gotcha.

If you're testing a Form that contains a control that is multithreaded and does interesting things with the Win32 event loop (such as MSHTML, the Internet Explorer HTML rendering control), you may find it doesn't work correctly when you try to unit test it. This is because the event loop doesn't work the same in this test runner as it does when running under regular Windows.

You can work around this by calling `Application.DoEvents()`, which will suspend your current thread and run the message pump thread until there are no pending Win32 events in the queue.

In the case of some unreasonably complex controls (e.g., MSHTML), you may have to run `Application.DoEvents()` a couple of times in order for you to be able to coerce them into behaving as they would in the real world.

You can definitely unit test most GUIs, but it is a slippery slope that makes it easy to write only what ends up being system tests. Testing only at the system level can sometimes seem much easier because you work around the need to refactor the code to make it more testable. Don't fall into that trap. As we've said a couple of times before, one of the biggest values of unit testing is in making your designs better. On top of that, many unit-level tests can usually run in the same amount of time as a single system-level test. Don't sell yourself, or your project, short by taking the easier way out. Be mindful of the balance between unit-level tests and system-level tests.

Appendix A

Extending NUnit

A.1 Writing NUnit Extensions

In the nunit.extensions framework, there is a Repeat attribute:

```
[Test]
[Repeat (10)]
public void IntermittentFailure() {
  xxx xxx xxxxxx xxxxx xxxx;
}
```

The Repeat attribute will, (as you may have guessed already), repeat the same test the specified number of times. If it fails any of those times, it won't run the remaining times. This can be useful for tests that are sensitive to timing or state issues and you want to make sure you've shaken everything out.

Examining an Existing Attribute

We're going to implement a new kind of attribute that test methods can be decorated with.

We've seen attributes like [Test], [Category], and [Repeat] previously. We're going to implement a new attribute that will let you specify a maximum amount of time a test can take to run. If it doesn't complete in the expected time, the test will fail (the test will also fail if its assertions fail, just as in a regular test.)

You can extend NUnit by adding new attributes; let's see how by exploring the source code to NUnit. Download the NUnit source code[1] (if you haven't already) and uncompress it somewhere convenient on your disk—we'll wait.

To start exploring, we should look at something similar to what we're trying to accomplish. The [Repeat(x)] attribute shown above seems like a good place to start. The source code for the Repeat attribute is in src/NUnitExtensions/framework/RepeatAttribute.cs. If you take a look, all it contains is literally the definition for the Repeat attribute. Something else must interpret the attribute, so let's look for that.

If you grep the source for "RepeatAttribute", it leads you to src/NUnitExtensions/core/RepeatedTestDecorator.cs. The RepeatedTestDecorator provides a couple of static methods; the important one of interest to us right now is the Decorate() method that takes a TestCase as a parameter and returns a TestCase. This follows the Decorator design pattern. The RepeatedTestDecorator's Decorate method for a TestCase does a couple of things; we'll hit the highlights.

First, it ensures that the TestCase method has the RepeatAttribute associated with it. If the TestCase method does not have an associated RepeatAttribute, it skips the decoration and just returns the TestCase as-is. If the TestCase method *does* have an associated RepeatAttribute, it gets the Count value out of the RepeatAttribute's Count property. It then constructs a new RepeatedTestCase with the original TestCase and the Count property. So, if the [Test] was marked with [Repeat(x)], it wraps the underlying TestCase object with a RepeatedTestCase object.

Let's take a look at the RepeatedTestCase object in src/NUnitExtensions/core/RepeatedTestCase.cs. It's pretty obvious how this works: the Run() method runs the original, wrapped TestCase by Count number of times. TestCaseResult is a collection parameter for the results of the test runs.

[1] http://sourceforge.net/projects/nunit

Missing: Review at the unit tests for these objects.

Missing: Finish off this content

Creating a New Attribute

Missing: Finish off this content

A.2 Using NUnit Core Addins

Missing: Content for this section will be added in a later beta.

Appendix B
Gotchas

Here are some popular "gotchas," that is, issues, problems, or misconceptions that have popped up over and over again to trap the unwary.

B.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.

Note that a test that has *no* assert statements or (mock object verification) will count as "passed." This is arguably a bug in NUnit, but at any rate a test without asserts still counts as broken.

If the tests are broken, treat it just as if the code were broken.

B.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 `Assert.IsTrue(true)` at the end. Maybe the slightly more adventurous will have multiple `Assert.IsTrue(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.

B.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.

B.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 NUnit floating-point asserts require you to specify a *tolerance* along with the desired values (see the discussion on page 35).

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:

```
Failures:
1) TestXyz.TestMe :
        expected:<1.00000000>
         but was:<1.00000000>
   at TestXyz.TestMe() in TestXyz.cs:line 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.

B.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 using the `[Category]` attribute discussed on page 45.

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.

B.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. Remember that a singleton is really just a global variable wearing pretty clothes—if other bits of code can muck with its state, they will, and usually when you least expect it.

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.

B.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 C# 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.

B.8 Tests Pass in One Test Runner, Not the Other

You may find a situation where all the tests pass for you using `nunit-gui`, the GUI test runner, but fail in the automated build, which uses `nunit-console`. This can be an indicator of hidden global state, circular object dependencies, or `Finalizer bugs`.

That last one can be especially subtle. For instance, there was a case where the developer wrote their finalizer like it was a C++ destructor—they were accessing the object's fields. But in the .NET environment, the fields may be garbage collected before the `Finalizer` is executed. The result? An intermittent `NullReferenceException`. In this case, it only happened in the `nunit-console` test runner when launched from `nant` via `CruiseControl.NET`, aligning the stars of the garbage collection universe just so. But once diagnosed, it also explained seemingly random, unreproducible crashes in the field as well.[1]

More likely, `nunit-console` and other test runners can run tests in slightly different order, which may expose hidden dependancies.

B.9 Thread state issues

Sometimes we see strange `InvalidOperationException`, `ThreadStateException`, or COM-based exceptions get thrown when code runs under NUnit, but not in production.

[1]For more on garbage collection "gotchas," see [Sub05].

This can sometimes be related to the apartment-type being multi-threaded when it should be single-threaded and nice versa. These are usually referred to by the acronyms MTA and STA, respectively. This is deep .NET voodoo we don't want to get stuck in, but there are a couple of NUnit commandline options to try: `-thread` and `-domain`. See the NUnit documentation for more more information on these options. If you have a burning desire to learn more, a quick perusal through CLR via C#[Ric06] or a web search will give you more information.

B.10 C# 2.0-specific Issues

So far, we haven't mentioned too many things specific to C# 2.0, so how do you use NUnit on a project that uses C# 2.0? Just as you normally would, with a couple of minor exceptions.

First, you'll need to use a version of NUnit that is compiled with a C# 2.0 compiler. There are separate packages on the NUnit web site for .NET 1.1 and 2.0 versions. By the way, it's perfectly safe and okay to use the .NET 2.0-compiled NUnit on a C# 1.1 project; the developers will just have to have .NET 2.0 or mono 1.1 or newer installed to run the tests.

Another notable thing to watch out for is the interaction of `Assert.IsNull()` and `Assert.IsNotNull()` with Nullable types. Nullable types, a C# 2.0 feature, allows value types like int, `DateTime`, enums, or structs to have a "null" value when it is not initialized. This feature was added so that the language could map more closely to the way databases represent data (for more information, see [Ric06]).

If you write a test like this:

```
[Test]
public void NullableInt() {
  int? first;
  Nullable<int> second;

  Assert.IsNull(first);
  Assert.IsNull(second);
}
```

It will fail. The reason why is because the value isn't literally null. The first line and second lines of code are semantically identical; the question mark syntax is just some syntactic sugar to make it easier to consume in C#. Looking at the second declaration, you can see it is a struct of type `Nullable<T>`. This will never be null.

To correctly test whether the Nullable type has a value or not, check it's `HasValue` property:

```
Assert.IsTrue(first.HasValue)
```

Appendix C

Resources

C.1 On The Web

Cruise Control .NET
⇒ http://ccnet.thoughtworks.com
CruiseControl.NET is an automated Continuous Integration server for .NET that integrates with NAnt, NUnit, NCover, and most major open source and proprietary version control systems.

DotGNU
⇒ http://dotgnu.org
An open source implementation of the ECMA standards upon which C# and .NET are based. Sports C# and .NET 1.1 support as well an optimizing JIT compiler as of this writing. Not as complete as mono, another open source implementation.

DotNetMock
⇒ http://sourceforge.net/projects/dotnetmock
A repository for Mock Object information in the .NET environment, as well as testing in general.

mono
⇒ http://mono-project.com
Another open source implementation of the ECMA standards upon which C# and .NET are based. Supports C# and .NET 2.0 as well as an optimizing JIT compiler.

NCover
⇒ http://ncover.org
A simple code coverage tool that runs from the command line and outputs an XML file with the code coverage statistics. Requires de-

bug information for monitored assemblies, and produces line-by-line visit counts. Also includes a simple XSLT transform to make the output readable in a browser.

NCoverExplorer
⇒ `http://kiwidude.com/blog`
A WinForms GUI and commandline UI for summarizing and exploring the XML files that NCover emits. Also includes NAnt tasks, stylesheets and tasks for CruiseControl.NET. Allows for failure of a build if code coverage gathered during the build and test are below a certain watermark.

NMock
⇒ `http://nmock.org`
NMock is a dynamic mock-object library for .NET.

NUnit
⇒ `http://nunit.org`
This xUnit-based unit testing tool for Microsoft .NET is written entirely in C# and has been completely redesigned to take advantage of many .NET language features, including custom attributes and other reflection related capabilities. NUnit brings xUnit to all .NET languages.

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.

SharpDevelop
⇒ `http://www.sharpdevelop.net`
A fully-featured and stable open-source IDE for .NET development. Has tight integration with NAnt, NUnit, NCover, code analysis, and source control.i

TestDriven.NET
⇒ `http://www.testdriven.net`
Visual-Studio integration for NUnit and NCover.

xUnit
⇒ `http://www.xprogramming.com/software.htm`
Unit testing frameworks for many, many different languages and environments.

C.2 Bibliography

[Cla04] Mike Clark. *Pragmatic Project Automation. How to Build, Deploy, and Monitor Java Applications*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 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.

[Fea04] Michael Feathers. *Working Effectively with Legacy Code*. Prentice Hall, Englewood Cliffs, NJ, 2004.

[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.

[Ric06] Jeffrey Richter. *CLR via C*. Microsoft Press, Redmond, WA, second edition, 2006.

[SH06] Venkat Subramaniam and Andy Hunt. *Practices of an Agile Developer: Working in the Real World*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2006.

[Sub05] Venkat Subramaniam. *.NET Gotchas*. O'Reilly & Associates, Inc, Sebastopol, CA, 2005.

[TH03] David Thomas and Andrew Hunt. *Pragmatic Version Control Using CVS*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2003.

[wCA04] Kent Beck with Cynthia Andres. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, MA, second edition, 2004.

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/titles/utc2

Appendix E

Answers to Exercises

Exercise 1: *from page 86*

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:

```
using System;
public interface StackExercise {
  /// <summary>
  /// Return and remove the most recent item from
  /// the top of the  stack.
  /// </summary>
  /// <exception cref="StackEmptyException">
  /// Throws exception if the stack is empty.
  /// </exception>
  String Pop();
  /// <summary>
  /// Add an item to the top of the stack.
  /// </summary>
  /// <param name="item">A String to push
  /// on the stack</param>
  void Push(String item);
  /// <summary>
  /// Return but do not remove the most recent
  /// item from the top of the stack.
  /// </summary>
  /// <exception cref="StackEmptyException">
  /// Throws exception if the stack is empty.
  /// </exception>
  String Top();
  /// <summary>
  /// Returns true if the stack is empty.
```

```
    /// </summary>
    bool IsEmpty();
  }
```

StackExercise.cs

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 right 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 87*
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 {
  /// <summary>
  /// Add this many of this item to the
  /// shopping cart.
  /// </summary>
  /// <exception cref="ArgumentOutOfRangeException">
  /// </exception>
  void AddItems(Item anItem, int quantity);

  /// <summary>
```

[1]In this case, the `Is.EqualTo()` constraint isn't good enough; you need `Is.Same()` to ensure it's the same object.

```
    /// Delete this many of this item from the
    /// shopping cart
    /// </summary>
    /// <exception cref="ArgumentOutOfRangeException">
    /// </exception>
    /// <exception cref="NoSuchItemException">
    /// </exception>
    void DeleteItems(Item anItem, int quantity);

    /// <summary>
    /// Count of all items in the cart
    /// (that is, all items x qty each)
    /// </summary>
    int ItemCount { get; }

    /// Return iterator of all items
    IEnumerable GetEnumerator();
  }
```

ShoppingCart.cs

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 `GetEnumerator` when there are no items in the cart and it should return an empty iterator (i.e., it's a real IEnumerable 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 `GetEnumerator()` 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 88*
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.
/// <exception cref="MissingOrBadFileException">
/// </exception>
/// <exception cref="PhoneFormatException">
/// </exception>
/// <exception cref="PhoneAreaCodeException">
/// </exception>
public bool SendFax(String phone, String filename)
```

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 88*
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 = `TableSize.Width - 1` and y = `TableSize.Height - 1`.

Coordinates are specified in fractions of centimeters.

```
public void MoveTo(double x, double y);
public void SewTo(double x, double y);
public void SetWorkpieceSize(double width,
                             double height);
public Size WorkpieceSize { get; }
public Size TableSize { get; }
```

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 89*
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.

```
using System;

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
  void FastForward(double seconds);

  /// Move the current position backwards by this
  /// many seconds. Rewinding past zero leaves
  /// the position at zero
  void Rewind(double seconds);

  /// Return current time position in seconds
  double CurrentTimePosition();

  /// Mark the current time position with label
  void MarkTimePosition(String name);

  /// Change the current position to the one
  /// associated with the marked name
  void GotoMark(String name);
}
```

AVTransport.cs

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 (not before the 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 89*
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 163*
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 7: 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