Foreword

“...then it began...”

In his introduction to this book, Michael Feathers uses that phrase to describe the start of his passion for software.

“...then it began...”

Do you know that feeling? Can you point to a single moment in your life and say: “...then it began...”? Was there a single event that changed the course of your life and eventually led you to pick up this book and start reading this foreword?

I was in sixth grade when it happened to me. I was interested in science and space and all things technical. My mother found a plastic computer in a catalog and ordered it for me. It was called *Digi-Comp I*. Forty years later that little plastic computer holds a place of honor on my bookshelf. It was the catalyst that sparked my enduring passion for software. It gave me my first inkling of how joyful it is to write programs that solve problems for people. It was just three plastic S-R flip-flops and six plastic and-gates, but it was enough—it served. Then... for me... it began...

But the joy I felt soon became tempered by the realization that software systems almost always degrade into a mess. What starts as a clean crystalline design in the minds of the programmers rots, over time, like a piece of bad meat. The nice little system we built last year turns into a horrible morass of tangled functions and variables next year.

Why does this happen? Why do systems rot? Why can’t they stay clean?

Sometimes we blame our customers. Sometimes we accuse them of changing the requirements. We comfort ourselves with the belief that if the customers had just been happy with what they said they needed, the design would have been fine. It’s the customer’s fault for changing the requirements on us.

Well, here’s a news flash: *Requirements change*. Designs that cannot tolerate changing requirements are poor designs to begin with. It is the goal of every competent software developer to create designs that tolerate change.

This seems to be an intractably hard problem to solve. So hard, in fact, that nearly every system ever produced suffers from slow, debilitating rot. The rot is so pervasive that we’ve come up with a special name for rotten programs. We call them: **Legacy Code**.

Legacy code. The phrase strikes disgust in the hearts of programmers. It conjures images of slogging through a murky swamp of tangled undergrowth with leaches beneath and stinging flies above. It conjures odors of murk, slime, stagnancy, and offal. Although our first joy of programming may have been intense, the misery of dealing with legacy code is often sufficient to extinguish that flame.

Many of us have tried to discover ways to *prevent* code from becoming legacy. We’ve written books on principles, patterns, and practices that can help programmers keep their systems clean. But Michael Feathers had an insight that many of the rest of us missed. Prevention is imperfect. Even the most disciplined development team, knowing the best principles, using the best patterns, and following the best practices will create messes from time to time. The rot still accumulates. It’s not enough to try to prevent the rot—you have to be able to *reverse* it.

That’s what this book is about. It’s about reversing the rot. It’s about taking a tangled, opaque, convoluted system and slowly, gradually, piece by piece, step by step, turning it into a simple, nicely structured, well-designed system. It’s about reversing entropy.

Before you get too excited, I warn you; reversing rot is not easy, and it’s not quick. The techniques, patterns, and tools that Michael presents in this book are effective, but they take work, time, endurance, and *care*. This book is not a magic bullet. It won’t tell you how to eliminate all the accumulated rot in your systems overnight. Rather, this book describes a set of disciplines, concepts, and attitudes that you will carry with you for the rest of your career and that *will help you to turn systems that gradually degrade into systems that gradually improve*.

Robert C. Martin
29 June, 2004

Preface

Do you remember the first program you wrote? I remember mine. It was a little graphics program I wrote on an early PC. I started programming later than most of my friends. Sure, I'd seen computers when I was a kid. I remember being really impressed by a minicomputer I once saw in an office, but for years I never had a chance to even sit at a computer. Later, when I was a teenager, some friends of mine bought a couple of the first TRS-80s. I was interested, but I was actually a bit apprehensive, too. I knew that if I started to play with computers, I'd get sucked into it. It just looked too cool. I don't know why I knew myself so well, but I held back. Later, in college, a roommate of mine had a computer, and I bought a C compiler so that I could teach myself programming. Then it began. I stayed up night after night trying things out, poring through the source code of the emacs editor that came with the compiler. It was addictive, it was challenging, and I loved it.

I hope you've had experiences like this—just the raw joy of making things work on a computer. Nearly every programmer I ask has. That joy is part of what got us into this work, but where is it day to day?

A few years ago, I gave my friend Erik Meade a call after I'd finished work one night. I knew that Erik had just started a consulting gig with a new team, so I asked him, "How are they doing?" He said, "They're writing legacy code, man." That was one of the few times in my life when I was sucker-punched by a coworker's statement. I felt it right in my gut. Erik had given words to the precise feeling that I often get when I visit teams for the first time. They are trying very hard, but at the end of the day, because of schedule pressure, the weight of history, or a lack of any better code to compare their efforts to, many people are writing legacy code.

What is legacy code? I've used the term without defining it. Let's look at the strict definition: Legacy code is code that we've gotten from someone else. Maybe our company acquired code from another company; maybe people on the original team moved on to other projects. Legacy code is somebody else's code. But in programmer-speak, the term means much more than that. The term *legacy code* has taken on more shades of meaning and more weight over time.

What do you think about when you hear the term *legacy code*? If you are at all like me, you think of tangled, unintelligible structure, code that you have to change but don't really understand. You think of sleepless nights trying to add in features that should be easy to add, and you think of demoralization, the sense that everyone on the team is so sick of a code base that it seems beyond care, the sort of code that you just wish would die. Part of you feels bad for even thinking about making it better. It seems unworthy of your efforts. That definition of legacy code has nothing to do with who wrote it. Code can degrade in many ways, and many of them have nothing to do with whether the code came from another team.

In the industry, *legacy code* is often used as a slang term for difficult-to-change code that we don't understand. But over years of working with teams, helping them get past serious code problems, I've arrived at a different definition.

To me, *legacy code* is simply code without tests. I've gotten some grief for this definition. What do tests have to do with whether code is bad? To me, the answer is straightforward, and it is a point that I elaborate throughout the book:

> Code without tests is bad code. It doesn't matter how well written it is; it doesn't matter how pretty or object-oriented or well-encapsulated it is. With tests, we can change the behavior of our code quickly and verifiably. Without them, we really don't know if our code is getting better or worse.

You might think that this is severe. What about clean code? If a code base is very clean and well structured, isn't that enough? Well, make no mistake. I love clean code. I love it more than most people I know, but while clean code is good, it's not enough. Teams take serious chances when they try to make large changes without tests. It is like doing aerial gymnastics without a net. It requires incredible skill and a clear understanding of what can happen at every step. Knowing precisely what will happen if you change a couple of variables is often like knowing whether another gymnast is going to catch your arms after you come out of a somersault. If you are on a team with code that clear, you are in a better position than most programmers. In my work, I've noticed that teams with that degree of clarity in all of their code are rare. They seem like a statistical anomaly. And, you know what? If they don't have supporting tests, their code changes still appear to be slower than those of teams that do.

Yes, teams do get better and start to write clearer code, but it takes a long time for older code to get clearer. In many cases, it will never happen completely. Because of this, I have no problem defining legacy code as code without tests. It is a good working definition, and it points to a solution.

I've been talking about tests quite a bit so far, but this book is not about testing. This book is about being able to confidently make changes in any code base. In the following chapters, I describe techniques that you can use to understand code, get it under test, refactor it, and add features.

One thing that you will notice as you read this book is that it is not a book about pretty code. The examples that I use in the book are fabricated because I work under nondisclosure agreements with clients. But in many of the examples, I've tried to preserve the spirit of code that I've seen in the field. I won't say that the examples are always representative. There certainly are oases of great code out there, but, frankly, there are also pieces of code that are far worse than anything I can use as an example in this book. Aside from client confidentiality, I simply couldn't put code like that in this book without boring you to tears and burying important points in a morass of detail. As a result, many of the examples are relatively brief. If you look at one of them and think "No, he doesn't understand—my methods are much larger than that and much worse," please look at the advice that I am giving at face value and see if it applies, even if the example seems simpler.

The techniques here have been tested on substantially large pieces of code. It is just a limitation of the book format that makes examples smaller. In particular, when you see ellipses (...) in a code fragment like this, you can read them as "insert 500 lines of ugly code here":

```
m_pDispatcher->register(listener);
...
m_nMargins++;
```

If this book is not about pretty code, it is even less about pretty design. Good design should be a goal for all of us, but in legacy code, it is something that we arrive at in discrete steps. In some of the chapters, I describe ways of adding new code to existing code bases and show how to add it with good design principles in mind. You can start to grow areas of very good high-quality code in legacy code bases, but don't be surprised if some of the steps you take to make changes involve making some code slightly uglier. This work is like surgery. We have to make incisions, and we have to move through the guts and suspend some aesthetic judgment. Could this patient's major organs and viscera be better than they are? Yes. So do we just forget about his immediate problem, sew him up again, and tell him to eat right and train for a marathon? We could, but what we really need to do is take the patient as he is, fix what's wrong, and move him to a healthier state. He might never become an Olympic athlete, but we can't let "best" be the enemy of "better." Code bases can become healthier and easier to work in. When a patient feels a little better, often that is the time when you can help him make commitments to a healthier life style. That is what we are shooting for with legacy code. We are trying to get to the point at which we are used to ease; we expect it and actively attempt to make code change easier. When we can sustain that sense on a team, design gets better.

The techniques I describe are ones that I've discovered and learned with coworkers and clients over the course of years working with clients to try to establish control over unruly code bases. I got into this legacy code emphasis accidentally. When I first started working with Object Mentor, the bulk of my work involved helping teams with serious problems develop their skills and interactions to the point that they could regularly deliver quality code. We often used Extreme Programming practices to help teams take control of their work, collaborate intensively, and deliver. I often feel that Extreme Programming is less a way to develop software than it is a way to make a well-jelled work team that just happens to deliver great software every two weeks.

From the beginning, though, there was a problem. Many of the first XP projects were "greenfield" projects. The clients I was seeing had significantly large code bases, and they were in trouble. They needed some way to get control of their work and start to deliver. Over time, I found that I was doing the same things over and over again with clients. This sense culminated in some work I was doing with a team in the financial industry. Before I'd arrived, they'd realized that unit testing was a great thing, but the tests that they were executing were full scenario tests that made multiple trips to a database and exercised large chunks of code. The tests were hard to write, and the team didn't run them very often because they took so long to run. As I sat down with them to break dependencies and get smaller chunks of code under test, I had a terrible sense of déjà vu. It seemed that I was doing this sort of work with every team I met, and it was the sort of thing that no one really wanted to think about. It was just the grunge work that you do when you want to start working with your code in a controlled way, if you know how to do it. I decided then that it was worth really reflecting on how we were solving these problems and writing them down so that teams could get a leg up and start to make their code bases easier to live in.

A note about the examples: I've used examples in several different programming languages. The bulk of the examples are written in Java, C++, and C. I picked Java because it is a very common language, and I included C++ because it presents some special challenges in a legacy environment. I picked C because it highlights many of the problems that come up in procedural legacy code. Among them, these languages cover much of the spectrum of concerns that arise in legacy code. However, if the languages you use are not covered in the examples, take a look at them anyway. Many of the techniques that I cover can be used in other languages, such as Delphi, Visual Basic, COBOL, and FORTRAN.

I hope that you find the techniques in this book helpful and that they allow you to get back to what is fun about programming. Programming can be very rewarding and enjoyable work. If you don't feel that in your day-to-day work, I hope that the techniques I offer you in this book help you find it and grow it on your team.

Acknowledgments

First of all, I owe a serious debt to my wife, Ann, and my children, Deborah and Ryan. Their love and support made this book and all of the learning that preceded it possible. I'd also like to thank "Uncle Bob" Martin, president and founder of Object Mentor. His rigorous pragmatic approach to development and design, separating the critical from the inconsequential, gave me something to latch upon about 10 years ago, back when it seemed that I was about to drown in a wave of unrealistic advice. And thanks, Bob, for giving me the opportunity to see more code and work with more people over the past five years than I ever imagined possible.

I also have to thank Kent Beck, Martin Fowler, Ron Jeffries, and Ward Cunningham for offering me advice at times and teaching me a great deal about team work, design, and programming. Special thanks to all of the people who reviewed the drafts. The official reviewers were Sven Gorts, Robert C. Martin, Erik Meade, and Bill Wake; the unofficial reviewers were Dr. Robert Koss, James Grenning, Lowell

Lindstrom, Micah Martin, Russ Rufer and the Silicon Valley Patterns Group, and James Newkirk.

Thanks also to reviewers of the very early drafts I placed on the Internet. Their feedback significantly affected the direction of the book after I reorganized its format. I apologize in advance to any of you I may have left out. The early reviewers were: Darren Hobbs, Martin Lippert, Keith Nicholas, Phlip Plumlee, C. Keith Ray, Robert Blum, Bill Burris, William Caputo, Brian Marick, Steve Freeman, David Putman, Emily Bache, Dave Astels, Russel Hill, Christian Sepulveda, and Brian Christopher Robinson.

Thanks also to Joshua Kerievsky who gave a key early review and Jeff Langr who helped with advice and spot reviews all through the process.

The reviewers helped me polish the draft considerably, but if there are errors remaining, they are solely mine.

Thanks to Martin Fowler, Ralph Johnson, Bill Opdyke, Don Roberts, and John Brant for their work in the area of refactoring. It has been inspirational.

I also owe a special debt to Jay Packlick, Jacques Morel, and Kelly Mower of Sabre Holdings, and Graham Wright of Workshare Technology for their support and feedback.

Special thanks also to Paul Petralia, Michelle Vincenti, Lori Lyons, Krista Hansing, and the rest of the team at Prentice-Hall. Thank you, Paul, for all of the help and encouragement that this first-time author needed.

Special thanks also to Gary and Joan Feathers, April Roberts, Dr. Raimund Ege, David Lopez de Quintana, Carlos Perez, Carlos M. Rodriguez, and the late Dr. John C. Comfort for help and encouragement over the years. I also have to thank Brian Button for the example in Chapter 21, *I'm Changing the Same Code All Over the Place*. He wrote that code in about an hour when we were developing a refactoring course together, and it's become my favorite piece of teaching code.

Also, special thanks to Janik Top, whose instrumental *De Futura* served as the soundtrack for my last few weeks of work on this book.

Finally, I'd like to thank everyone whom I've worked with over the past few years whose insights and challenges strengthened the material in this book.

Michael Feathers
mfeathers@objectmentor.com
www.objectmentor.com
www.michaelfeathers.com

Introduction

How to Use This Book

I tried several different formats before settling on the current one for this book. Many of the different techniques and practices that are useful when working with legacy code are hard to explain in isolation. The simplest changes often go easier if you can find seams, make fake objects, and break dependencies using a couple of dependency-breaking techniques. I decided that the easiest way to make the book approachable and handy would be to organize the bulk of it (*Part II*, *Changing Software*) in FAQ (frequently asked questions) format. Because specific techniques often require the use of other techniques, the FAQ chapters are heavily interlinked. In nearly every chapter, you'll find references, along with page numbers, for other chapters and sections that describe particular techniques and refactorings. I apologize if this causes you to flip wildly through the book as you attempt to find answers to your questions, but I assumed that you'd rather do that than read the book cover to cover, trying to understand how all the techniques operate.

In *Changing Software*, I've tried to address very common questions that come up in legacy code work. Each of the chapters is named after a specific problem. This does make the chapter titles rather long, but hopefully, they will allow you to quickly find a section that helps you with the particular problems you are having.

Changing Software is bookended by a set of introductory chapters (*Part I*, *The Mechanics of Change*) and a catalog of refactorings, which are very useful in legacy code work (*Part III*, *Dependency-Breaking Techniques*). Please read the introductory chapters, particularly Chapter 4, *The Seam Model*. These chapters provide the context and nomenclature for all the techniques that follow. In addition, if you find a term that isn't described in context, look for it in the Glossary.

The refactorings in *Dependency-Breaking Techniques* are special in that they are meant to be done without tests, in the service of putting tests in place. I encourage you to read each of them so that you can see more possibilities as you start to tame your legacy code.

Part I: The Mechanics of Change

Chapter 1: Changing Software

Changing code is great. It's what we do for a living. But there are ways of changing code that make life difficult, and there are ways that make it much easier. In the industry, we haven't spoken about that much. The closest we've gotten is the literature on refactoring. I think we can broaden the discussion a bit and talk about how to deal with code in the thorniest of situations. To do that, we have to dig deeper into the mechanics of change.

Four Reasons to Change Software

For simplicity's sake, let's look at four primary reasons to change software.

1. Adding a feature
2. Fixing a bug
3. Improving the design
4. Optimizing resource usage

Adding Features and Fixing Bugs

Adding a feature seems like the most straightforward type of change to make. The software behaves one way, and users say that the system needs to do something else also.

Suppose that we are working on a web-based application, and a manager tells us that she wants the company logo moved from the left side of a page to the right side. We talk to her about it and discover it isn't quite so simple. She wants to move the logo, but she wants other changes, too. She'd like to make it animated for the next release. Is this fixing a bug or adding a new feature? It depends on your point of view. From the point of view of the customer, she is definitely asking us to fix a problem. Maybe she saw the site and attended a meeting with people in her department, and they decided to change the logo placement and ask for a bit more functionality. From a developer's point of view, the change could be seen as a completely new feature. "If they just stopped changing their minds, we'd be done by now." But in some organizations the logo move is seen as just a bug fix, regardless of the fact that the team is going to have to do a lot of fresh work.

It is tempting to say that all of this is just subjective. You see it as a bug fix, and I see it as a feature, and that's the end of it. Sadly, though, in many organizations, bug fixes and features have to be tracked and accounted for separately because of contracts or quality initiatives. At the people level, we can go back and forth endlessly about whether we are adding features or fixing bugs, but it is all just changing code and other artifacts. Unfortunately, this talk about bug-fixing and feature addition masks something that is much more important to us technically: behavioral change. There is a big difference between adding new behavior and changing old behavior.

> Behavior is the most important thing about software. It is what users depend on. Users like it when we add behavior (provided it is what they really wanted), but if we change or remove behavior they depend on (introduce bugs), they stop trusting us.

In the company logo example, are we adding behavior? Yes. After the change, the system will display a logo on the right side of the page. Are we getting rid of any behavior? Yes, there won't be a logo on the left side.

Let's look at a harder case. Suppose that a customer wants to add a logo to the right side of a page, but there wasn't one on the left side to start with. Yes, we are adding behavior, but are we removing any? Was anything rendered in the place where the logo is about to be rendered?

Are we changing behavior, adding it, or both?

It turns out that, for us, we can draw a distinction that is more useful to us as programmers. If we have to modify code (and HTML kind of counts as code), we could be changing behavior. If we are only adding code and calling it, we are often adding behavior. Let's look at another example. Here is a method on a Java class:

```
public class CDPlayer
{
    public void addTrackListing(Track track) {
        ...
    }
    ...
}
```

The class has a method that enables us to add track listings. Let's add another method that lets us replace track listings.

```
public class CDPlayer
{
    public void addTrackListing(Track track) {
        ...
    }

    public void replaceTrackListing(String name, Track track) {
        ...
    }
    ...
}
```

When we added that method, did we add new behavior to our application or change it? The answer is: neither. Adding a method doesn't change behavior unless the method is called somehow.

Let's make another code change. Let's put a new button on the user interface for the CD player. The button lets users replace track listings. With that move, we're adding the behavior we specified in `replaceTrackListing` method, but we're also subtly changing behavior. The UI will render differently with that new button. Chances are, the UI will take about a microsecond longer to display. It seems nearly impossible to add behavior without changing it to some degree.

Improving Design

Design improvement is a different kind of software change. When we want to alter software's structure to make it more maintainable, generally we want to keep its behavior intact also. When we drop behavior in that process, we often call that a bug. One of the main reasons why many programmers don't attempt to improve design often is because it is relatively easy to lose behavior or create bad behavior in the process of doing it.

The act of improving design without changing its behavior is called *refactoring*. The idea behind refactoring is that we can make software more maintainable without changing behavior if we write tests to make sure that existing behavior doesn't change and take small steps to verify that all along the process. People have been cleaning up code in systems for years, but only in the last few years has refactoring taken off. Refactoring differs from general cleanup in that we aren't just doing low-risk things such as reformatting source code, or invasive and risky things such as rewriting chunks of it. Instead, we are making a series of small structural modifications, supported by tests to make the code easier to change. The key thing about refactoring from a change point of view is that there aren't supposed to be any functional changes when you refactor (although behavior can change somewhat because the structural changes that you make can alter performance, for better or worse).

Optimization

Optimization is like refactoring, but when we do it, we have a different goal. With both refactoring and optimization, we say, “We’re going to keep functionality exactly the same when we make changes, but we are going to change something else.” In refactoring, the “something else” is program structure; we want to make it easier to maintain. In optimization, the “something else” is some resource used by the program, usually time or memory.

Putting It All Together

It might seem strange that refactoring and optimization are kind of similar. They seem much closer to each other than adding features or fixing bugs. But is this really true? The thing that is common between refactoring and optimization is that we hold functionality invariant while we let something else change.

In general, three different things can change when we do work in a system: structure, functionality, and resource usage.

Let’s look at what usually changes and what stays more or less the same when we make four different kinds of changes (yes, often all three change, but let’s look at what is typical):

	Adding a Feature	Fixing a Bug	Refactoring	Optimizing
Structure	Changes	Changes	Changes	—
Functionality	Changes	Changes	—	—
Resource Usage	—	—	—	Changes

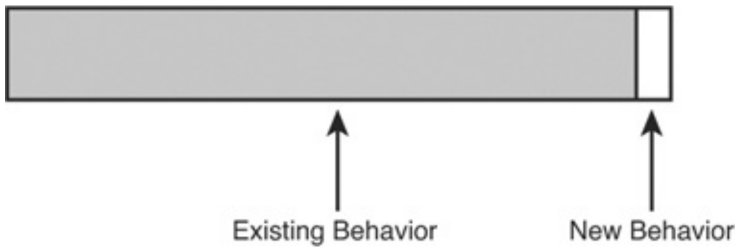
Superficially, refactoring and optimization do look very similar. They hold functionality invariant. But what happens when we account for new functionality separately? When we add a feature often we are adding new functionality, but without changing existing functionality.

	Adding a Feature	Fixing a Bug	Refactoring	Optimizing
Structure	Changes	Changes	Changes	—
New Functionality	Changes	—	—	—
Functionality	—	Changes	—	—
Resource Usage	—	—	—	Changes

Adding features, refactoring, and optimizing all hold existing functionality invariant. In fact, if we scrutinize bug fixing, yes, it does change functionality, but the changes are often very small compared to the amount of existing functionality that is not altered.

Feature addition and bug fixing are very much like refactoring and optimization. In all four cases, we want to change some functionality, some behavior, but we want to preserve much more (see Figure 1.1).

Figure 1.1 *Preserving behavior.*



That’s a nice view of what is supposed to happen when we make changes, but what does it mean for us practically? On the positive side, it seems to tell us what we have to concentrate on. We have to make sure that the small number of things that we change are changed correctly. On the negative side, well, that isn't the only thing we have to concentrate on. We have to figure out how to preserve the rest of the behavior. Unfortunately, preserving it involves more than just leaving the code alone. We have to know that the behavior isn't changing, and that can be tough. The amount of behavior that we have to preserve is usually very large, but that isn't the big deal. The big deal is that we often don't know how much of that behavior is at risk when we make our changes. If we knew, we could concentrate on that behavior and not care about the rest. Understanding is the key thing that we need to make changes safely.

Preserving existing behavior is one of the largest challenges in software development. Even when we are changing primary features, we often have very large areas of behavior that we have to preserve.

Risky Change

Preserving behavior is a large challenge. When we need to make changes and preserve behavior, it can involve considerable risk.

To mitigate risk, we have to ask three questions:

1. What changes do we have to make?
2. How will we know that we've done them correctly?
3. How will we know that we haven't broken anything?

How much change can you afford if changes are risky?

Most teams that I've worked with have tried to manage risk in a very conservative way. They minimize the number of changes that they make to the code base. Sometimes this is a team policy: "If it's not broke, don't fix it." At other times, it isn't anything that anyone articulates. The developers are just very cautious when they make changes. "What? Create another method for that? No, I'll just put the lines of code right here in the method, where I can see them and the rest of the code. It involves less editing, and it's safer."

It's tempting to think that we can minimize software problems by avoiding them, but, unfortunately, it always catches up with us. When we avoid creating new classes and methods, the existing ones grow larger and harder to understand. When you make changes in any large system, you can expect to take a little time to get familiar with the area you are working with. The difference between good systems and bad ones is that, in the good ones, you feel pretty calm after you've done that learning, and you are confident in the change you are about to make. In poorly structured code, the move from figuring things out to making changes feels like jumping off a cliff to avoid a tiger. You hesitate and hesitate. "Am I ready to do it? Well, I guess I have to."

Avoiding change has other bad consequences. When people don't make changes often they get rusty at it. Breaking down a big class into pieces can be pretty involved work unless you do it a couple of times a week. When you do, it becomes routine. You get better at figuring out what can break and what can't, and it is much easier to do.

The last consequence of avoiding change is fear. Unfortunately, many teams live with incredible fear of change and it gets worse every day. Often they aren't aware of how much fear they have until they learn better techniques and the fear starts to fade away.

We've talked about how avoiding change is a bad thing, but what is our alternative? One alternative is to just try harder. Maybe we can hire more people so that there is enough time for everyone to sit and analyze, to scrutinize all of the code and make changes the "right" way. Surely more time and scrutiny will make change safer. Or will it? After all of that scrutiny, will anyone know that they've gotten it right?

Chapter 2: Working with Feedback

Changes in a system can be made in two primary ways. I like to call them *Edit and Pray* and *Cover and Modify*. Unfortunately, *Edit and Pray* is pretty much the industry standard. When you use *Edit and Pray*, you carefully plan the changes you are going to make, you make sure that you understand the code you are going to modify, and then you start to make the changes. When you're done, you run the system to see if the change was enabled, and then you poke around further to make sure that you didn't break anything. The poking around is essential. When you make your changes, you are hoping and praying that you'll get them right, and you take extra time when you are done to make sure that you did.

Superficially, *Edit and Pray* seems like "working with care," a very professional thing to do. The "care" that you take is right there at the forefront, and you expend extra care when the changes are very invasive because much more can go wrong. But safety isn't solely a function of care. I don't think any of us would choose a surgeon who operated with a butter knife just because he worked with care. Effective software change, like effective surgery, really involves deeper skills. Working with care doesn't do much for you if you don't use the right tools and techniques.

Cover and Modify is a different way of making changes. The idea behind it is that it is possible to work with a *safety net* when we change software. The safety net we use isn't something that we put underneath our tables to catch us if we fall out of our chairs. Instead, it's kind of like a cloak that we put over code we are working on to make sure that bad changes don't leak out and infect the rest of our software. Covering software means covering it with tests. When we have a good set of tests around a piece of code, we can make changes and find out very quickly whether the effects were good or bad. We still apply the same care, but with the feedback we get, we are able to make changes more carefully.

If you are not familiar with this use of tests, all of this is bound to sound a little bit odd. Traditionally, tests are written and executed after development. A group of programmers writes code and a team of testers runs tests against the code afterward to see if it meets some specification. In some very traditional development shops, this is just the way that software is developed. The team can get feedback, but the feedback loop is large. Work for a few weeks or months, and then people in another group will tell you whether you've gotten it right.

Testing done this way is really "testing to attempt to show correctness." Although that is a good goal, tests can also be used in a very different way. We can do "testing to detect change."

In traditional terms, this is called regression testing. We periodically run tests that check for known good behavior to find out whether our software still works the way that it did in the past.

When you have tests around the areas in which you are going to make changes, they act as a software vise. You can keep most of the behavior fixed and know that you are changing only what you intend to.

Software Vise

vise (n.). A clamping device, usually consisting of two jaws closed or opened by a screw or lever, used in carpentry or metalworking to hold a piece in position. *The American Heritage Dictionary of the English Language, Fourth Edition*

When we have tests that detect change, it is like having a vise around our code. The behavior of the code is fixed in place. When we make changes, we can know that we are changing only one piece of behavior at a time. In short, we're in control of our work.

Regression testing is a great idea. Why don't people do it more often? There is this little problem with regression testing. Often when people practice it, they do it at the application interface. It doesn't matter whether it is a web application, a command-line application, or a GUI-based application; regression testing has traditionally been seen as an application-level testing style. But this is unfortunate. The feedback we can get from it is very useful. It pays to do it at a finer-grained level.

Let’s do a little thought experiment. We are stepping into a large function that contains a large amount of complicated logic. We analyze, we think, we talk to people who know more about that piece of code than we do, and then we make a change. We want to make sure that the change hasn’t broken anything, but how can we do it? Luckily, we have a quality group that has a set of regression tests that it can run overnight. We call and ask them to schedule a run, and they say that, yes, they can run the tests overnight, but it is a good thing that we called early. Other groups usually try to schedule regression runs in the middle of the week, and if we’d waited any longer, there might not be a timeslot and a machine available for us. We breathe a sigh of relief and then go back to work. We have about five more changes to make like the last one. All of them are in equally complicated areas. And we’re not alone. We know that several other people are making changes, too.

The next morning, we get a phone call. Daiva over in testing tells us that tests AE1021 and AE1029 failed overnight. She’s not sure whether it was our changes, but she is calling us because she knows we’ll take care of it for her. We’ll debug and see if the failures were because of one of our changes or someone else’s.

Does this sound real? Unfortunately, it is very real.

Let’s look at another scenario.

We need to make a change to a rather long, complicated function. Luckily, we find a set of unit tests in place for it. The last people who touched the code wrote a set of about 20 unit tests that thoroughly exercised it. We run them and discover that they all pass. Next we look through the tests to get a sense of what the code’s actual behavior is.

We get ready to make our change, but we realize that it is pretty hard to figure out how to change it. The code is unclear, and we’d really like to understand it better before making our change. The tests won’t catch everything, so we want to make the code very clear so that we can have more confidence in our change. Aside from that, we don’t want ourselves or anyone else to have to go through the work we are doing to try to understand it. What a waste of time!

We start to refactor the code a bit. We extract some methods and move some conditional logic. After every little change that we make, we run that little suite of unit tests. They pass almost every time that we run them. A few minutes ago, we made a mistake and inverted the logic on a condition, but a test failed and we recovered in about a minute. When we are done refactoring, the code is much clearer. We make the change we set out to make, and we are confident that it is right. We added some tests to verify the new behavior. The next programmers who work on this piece of code will have an easier time and will have tests that cover its functionality.

Do you want your feedback in a minute or overnight? Which scenario is more efficient?

Unit testing is one of the most important components in legacy code work. System-level regression tests are great, but small, localized tests are invaluable. They can give you feedback as you develop and allow you to refactor with much more safety.

What Is Unit Testing?

The term *unit test* has a long history in software development. Common to most conceptions of unit tests is the idea that they are tests in isolation of individual components of software. What are components? The definition varies, but in unit testing, we are usually concerned with the most atomic behavioral units of a system. In procedural code, the units are often functions. In object-oriented code, the units are classes.

Test Harnesses

In this book, I use the term *test harness* as a generic term for the testing code that we write to exercise some piece of software and the code that is needed to run it. We can use many different kinds of test harnesses to work with our code. In Chapter 5, *Tools*, I discuss the xUnit testing framework and the FIT framework. Both of them can be used to do the testing I describe in this book.

Can we ever test only one function or one class? In procedural systems, it is often hard to test functions in isolation. Top-level functions call other functions, which call other functions, all the way down to the machine level. In object-oriented systems, it is a little easier to test classes in isolation, but the fact is, classes don't generally live in isolation. Think about all of the classes you've ever written that don't use other classes. They are pretty rare, aren't they? Usually they are little data classes or data structure classes such as stacks and queues (and even these might use other classes).

Testing in isolation is an important part of the definition of a unit test, but why is it important? After all, many errors are possible when pieces of software are integrated. Shouldn't large tests that cover broad functional areas of code be more important? Well, they are important, I won't deny that, but there are a few problems with large tests:

- **Error localization**—As tests get further from what they test, it is harder to determine what a test failure means. Often it takes considerable work to pinpoint the source of a test failure. You have to look at the test inputs, look at the failure, and determine where along the path from inputs to outputs the failure occurred. Yes, we have to do that for unit tests also, but often the work is trivial.
- **Execution time**—Larger tests tend to take longer to execute. This tends to make test runs rather frustrating. Tests that take too long to run end up not being run.
- **Coverage**—It is hard to see the connection between a piece of code and the values that exercise it. We can usually find out whether a piece of code is exercised by a test using coverage tools, but when we add new code, we might have to do considerable work to create high-level tests that exercise the new code.

One of the most frustrating things about larger tests is that we can have error localization if we run our tests more often, but it is very hard to achieve. If we run our tests and they pass, and then we make a small change and they fail, we know precisely where the problem was triggered. It was something we did in that last small change. We can roll back the change and try again. But if our tests are large, execution time can be too long; our tendency will be to avoid running the tests often enough to really localize errors.

Unit tests fill in gaps that larger tests can't. We can test pieces of code independently; we can group tests so that we can run some under some conditions and others under other conditions. With them we can localize errors quickly. If we think there is an error in some particular piece of code and we can use it in a test harness, we can usually code up a test quickly to see if the error really is there.

Here are qualities of good unit tests:

1. They run fast.
2. They help us localize problems.

In the industry, people often go back and forth about whether particular tests are unit tests. Is a test really a unit test if it uses another production class? I go back to the two qualities: Does the test run fast? Can it help us localize errors quickly? Naturally, there is a continuum. Some tests are larger, and they use several classes together. In fact, they may seem to be little integration tests. By themselves, they might seem to run fast, but what happens when you run them all together? When you have a test that exercises a class along with several of its collaborators, it tends to grow. If you haven't taken the time to make a class separately instantiable in a test harness, how easy will it be when you add more code? It never gets easier. People put it off. Over time, the test might end up taking as long as 1/10th of a second to execute.

A unit test that takes 1/10th of a second to run is a slow unit test.

Yes, I'm serious. At the time that I'm writing this, 1/10th of a second is an eon for a unit test. Let's do the math. If you have a project with 3,000 classes and there are about 10 tests apiece, that is 30,000 tests. How long will it take to run all of the tests for that project if they take 1/10th of a second apiece? Close to an hour. That is a long time to wait for feedback. You don't have 3,000 classes? Cut it in half. That is still a half an hour. On the other hand, what if the tests take 1/100th of a second apiece? Now we are talking about 5 to 10 minutes. When they take that long, I make sure that I use a subset to work with, but I don't mind running them all every couple of hours.

With Moore's Law's help, I hope to see nearly instantaneous test feedback for even the largest systems in my lifetime. I suspect that working in those systems will be like working in code that can bite back. It will be capable of letting us know when it is being changed in a bad way.

Unit tests run fast. If they don't run fast, they aren't unit tests.

Other kinds of tests often masquerade as unit tests. A test is not a unit test if:

1. It talks to a database.
2. It communicates across a network.
3. It touches the file system.
4. You have to do special things to your environment (such as editing configuration files) to run it.

Tests that do these things aren't bad. Often they are worth writing, and you generally will write them in unit test harnesses. However, it is important to be able to separate them from true unit tests so that you can keep a set of tests that you can run *fast* whenever you make changes.

Higher-Level Testing

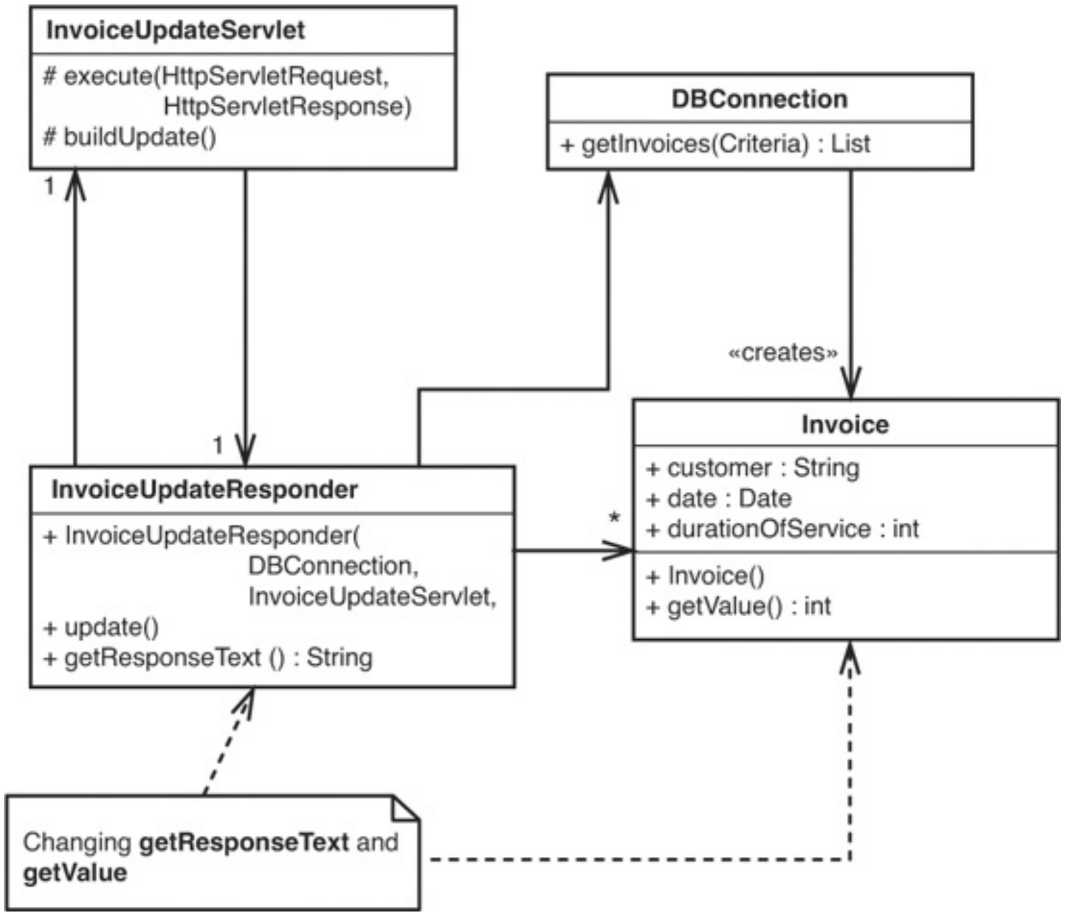
Unit tests are great, but there is a place for higher-level tests, tests that cover scenarios and interactions in an application. Higher-level tests can be used to pin down behavior for a set of classes at a time. When you are able to do that, often you can write tests for the individual classes more easily.

Test Coverings

So how do we start making changes in a legacy project? The first thing to notice is that, given a choice, it is always safer to have tests around the changes that we make. When we change code, we can introduce errors; after all, we're all human. But when we cover our code with tests before we change it, we're more likely to catch any mistakes that we make.

Figure 2.1 shows us a little set of classes. We want to make changes to the `getResponseText` method of `InvoiceUpdateResponder` and the `getValue` method of `Invoice`. Those methods are our change points. We can cover them by writing tests for the classes they reside in.

Figure 2.1 *Invoice update classes.*


InvoiceUpdateServlet
# execute(HttpServletRequest, HttpServletResponse)
# buildUpdate()
DBConnection
+ getInvoices(Criteria) : List
«creates»
Invoice
+ customer : String
+ date : Date
+ durationOfService : int
+ Invoice()
+ getValue() : int
1
1
*
InvoiceUpdateResponder
+ InvoiceUpdateResponder(DBConnection, InvoiceUpdateServlet,
+ update()
+ getResponseText () : String
Changing getResponseText and getValue

To write and run tests we have to be able to create instances of `InvoiceUpdateResponder` and `Invoice` in a testing harness. Can we do that? Well, it looks like it should be easy enough to create an `Invoice`; it has a constructor that doesn't accept any arguments. `InvoiceUpdateResponder` might be tricky, though. It accepts a `DBConnection`, a real connection to a live database. How are we going to handle that in a test? Do we have to set up a database with data for our tests? That's a lot of work. Won't testing through the database be slow? We don't particularly care about the database right now anyway; we just want to cover our changes in `InvoiceUpdateResponder` and `Invoice`. We also have a bigger problem. The constructor for `InvoiceUpdateResponder` needs an `InvoiceUpdateServlet` as an argument. How easy will it be to create one of those? We could change the code so that it doesn't take that servlet anymore. If the `InvoiceUpdateResponder` just needs a little bit of information from `InvoiceUpdateServlet`, we can pass it along instead of passing the whole servlet in, but shouldn't we have a test in place to make sure that we've made that change correctly?

All of these problems are dependency problems. When classes depend directly on things that are hard to use in a test, they are hard to modify and hard to work with.

Dependency is one of the most critical problems in software development. Much legacy code work involves breaking dependencies so that change can be easier.

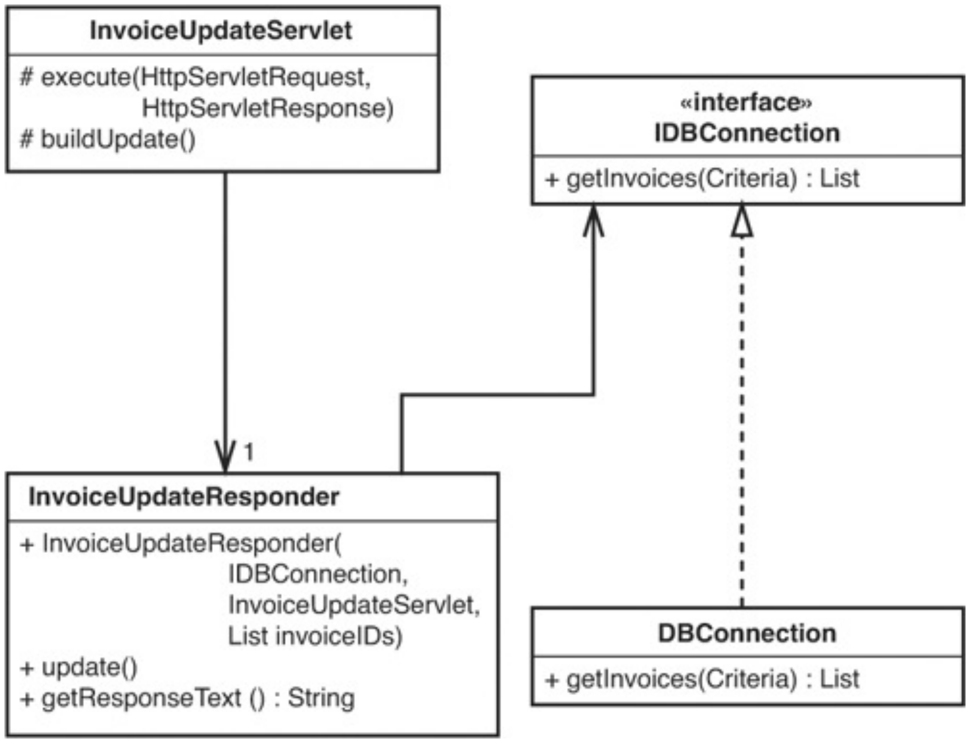
So, how do we do it? How do we get tests in place without changing code? The sad fact is that, in many cases, it isn't very practical. In some cases, it might even be impossible. In the example we just saw, we could attempt to get past the `DBConnection` issue by using a real database, but what about the servlet issue? Do we have to create a full servlet and pass it to the constructor of `InvoiceUpdateResponder`? Can we get it into the right state? It might be possible. What would we do if we were working in a GUI desktop application? We might not have any programmatic interface. The logic could be tied right into the GUI classes. What do we do then?

The Legacy Code Dilemma

When we change code, we should have tests in place. To put tests in place, we often have to change code.

In the Invoice example we can try to test at a higher level. If it is hard to write tests without changing a particular class, sometimes testing a class that uses it is easier; regardless, we usually have to break dependencies between classes someplace. In this case, we can break the dependency on `InvoiceUpdateServlet` by passing the one thing that `InvoiceUpdateResponder` really needs. It needs the collection of invoice IDs that the `InvoiceUpdateServlet` holds. We can also break the dependency that `InvoiceUpdateResponder` has on `DBConnection` by introducing an interface (`IDBConnection`) and changing the `InvoiceUpdateResponder` so that it uses the interface instead. Figure 2.2 shows the state of these classes after the changes.

Figure 2.2 *Invoice update classes with dependencies broken.*



InvoiceUpdateServlet
# execute(HttpServletRequest, HttpServletResponse)
# buildUpdate()
«interface» IDBConnection
+ getInvoices(Criteria) : List
1
InvoiceUpdateResponder
+ InvoiceUpdateResponder(IDBConnection, InvoiceUpdateServlet, List invoiceIDs)
+ update()
+ getResponseText () : String
DBConnection
+ getInvoices(Criteria) : List

Is this safe to do these refactorings without tests? It can be. These refactorings are named *Primitivize Parameter* (385) and *Extract Interface* (362), respectively. They are described in the dependency breaking techniques catalog at the end of the book. When we break dependencies, we can often write tests that make more invasive changes safer. The trick is to do these initial refactorings very conservatively.

Being conservative is the right thing to do when we can possibly introduce errors, but sometimes when we break dependencies to cover code, it doesn't turn out as nicely as what we did in the previous example. We might introduce parameters to methods that aren't strictly needed in production code, or we might break apart classes in odd ways just to be able to get tests in place. When we do that, we might end up making the code look a little poorer in that area. If we were being less conservative, we'd just fix it immediately. We can do that, but it depends upon how much risk is involved. When errors are a big deal, and they usually are, it pays to be conservative.

When you break dependencies in legacy code, you often have to suspend your sense of aesthetics a bit. Some dependencies break cleanly; others end up looking less than ideal from a design point of view. They are like the incision points in surgery: There might be a scar left in your code after your work, but everything beneath it can get better.

If later you can cover code around the point where you broke the dependencies, you can heal that scar, too.

The Legacy Code Change Algorithm

When you have to make a change in a legacy code base, here is an algorithm you can use.

1. Identify change points.
2. Find test points.
3. Break dependencies.
4. Write tests.
5. Make changes and refactor.

The day-to-day goal in legacy code is to make changes, but not just any changes. We want to make functional changes that deliver value while bringing more of the system under test. At the end of each programming episode, we should be able to point not only to code that provides some new feature, but also its tests. Over time, tested areas of the code base surface like islands rising out of the ocean. Work in these islands becomes much easier. Over time, the islands become large landmasses. Eventually, you'll be able to work in continents of test-covered code.

Let's look at each of these steps and how his book will help you with them.

Identify Change Points

The places where you need to make your changes depend sensitively on your architecture. If you don't know your design well enough to feel that you are making changes in the right place, take a look at Chapter 16, *I Don't Understand the Code Well Enough to Change It*, and Chapter 17, *My Application Has No Structure*.

Find Test Points

In some cases, finding places to write tests is easy, but in legacy code it can often be hard. Take a look at Chapter 11, *I Need to Make a Change. What Methods Should I Test?*, and Chapter 12, *I Need to Make Many Changes in One Area. Do I Have to Break Dependencies for All the Classes Involved?* These chapters offer techniques that you can use to determine where you need to write your tests for particular changes.

Break Dependencies

Dependencies are often the most obvious impediment to testing. The two ways this problem manifests itself are difficulty instantiating objects in test harnesses and difficulty running methods in test harnesses. Often in legacy code, you have to break dependencies to get tests in place. Ideally, we would have tests that tell us whether the things we do to break dependencies themselves caused problems, but often we don't. Take a look at Chapter 23, *How Do I Know That I'm Not Breaking Anything?*, to see some practices that can be used to make the first incisions in a system safer as you start to bring it under test. When you have done this, take a look at Chapter 9, *I Can't Get This Class into a Test Harness*, and Chapter 10, *I Can't Run This Method in a Test Harness*, for scenarios that show how to get past common dependency problems. These sections heavily reference the dependency breaking techniques catalog at the back of the book, but they don't cover all of the techniques. Take some time to look through the catalog for more ideas on how to break dependencies.

Dependencies also show up when we have an idea for a test but we can't write it easily. If you find that you can't write tests because of dependencies in large methods, see Chapter 22, *I Need to Change a Monster Method and I Can't Write Tests for It*. If you find that you can break dependencies, but it takes too long to build your tests, take a look at Chapter 7, *It Takes Forever to Make a Change*. That chapter describes additional dependency-breaking work that you can do to make your average build time faster.

Write Tests

I find that the tests I write in legacy code are somewhat different from the tests I write for new code. Take a look at Chapter 13, *I Need to Make a Change but I Don't Know What Tests to Write*, to learn more about the role of tests in legacy code work.

Make Changes and Refactor

I advocate using test-driven development (TDD) to add features in legacy code. There is a description of TDD and some other feature addition techniques in Chapter 8, *How Do I Add a Feature?* After making changes in legacy code, we often are better versed with its problems, and the tests we've written to add features often give us some cover to do some refactoring. Chapter 20, *This Class Is Too Big and I Don't Want It to Get Any Bigger*; Chapter 22, *I Need to Change a Monster Method and I Can't Write Tests for It*; and Chapter 21, *I'm Changing the Same Code All Over the Place* cover many of the techniques you can use to start to move your legacy code toward better structure. Remember that the things I describe in these chapters are "baby steps." They don't show you how to make your design ideal, clean, or pattern-enriched. Plenty of books show how to do those things, and when you have the opportunity to use those techniques, I encourage you to do so. These chapters show you how to make design better, where "better" is context dependent and often simply a few steps more maintainable than the design was before. But don't discount this work. Often the simplest things, such as breaking down a large class just to make it easier to work with, can make a significant difference in applications, despite being somewhat mechanical.

The Rest of This Book

The rest of this book shows you how to make changes in legacy code. The next two chapters contain some background material about three critical concepts in legacy work: sensing, separation, and seams.

Chapter 3: Sensing and Separation

Ideally, we wouldn't have to do anything special to a class to start working with it. In an ideal system, we'd be able to create objects of any class in a test harness and start working. We'd be able to create objects, write tests for them, and then move on to other things. If it were that easy, there wouldn't be a need to write about any of this, but unfortunately, it is often hard. Dependencies among classes can make it very difficult to get particular clusters of objects under test. We might want to create an object of one class and ask it questions, but to create it, we need objects of another class, and those objects need objects of another class, and so on. Eventually, you end up with nearly the whole system in a harness. In some languages, this isn't a very big deal. In others, most notably C++, link time alone can make rapid turnaround nearly impossible if you don't break dependencies.

In systems that weren't developed concurrently with unit tests, we often have to break dependencies to get classes into a test harness, but that isn't the only reason to break dependencies. Sometimes the class we want to test has effects on other classes, and our tests need to know about them. Sometimes we can sense those effects through the interface of the other class. At other times, we can't. The only choice we have is to impersonate the other class so that we can sense the effects directly.

Generally, when we want to get tests in place, there are two reasons to break dependencies: *sensing* and *separation*.

1. **Sensing**—We break dependencies to *sense* when we can't access values our code computes.
2. **Separation**—We break dependencies to *separate* when we can't even get a piece of code into a test harness to run.

Here is an example. We have a class named `NetworkBridge` in a network-management application:

```
public class NetworkBridge
{
    public NetworkBridge(EndPoint [] endpoints) {
        ...
    }

    public void formRouting(String sourceID, String destID) {
        ...
    }
    ...
}
```

`NetworkBridge` accepts an array of `EndPoints` and manages their configuration using some local hardware. Users of `NetworkBridge` can use its methods to route traffic from one endpoint to another. `NetworkBridge` does this work by changing settings on the `EndPoint` class. Each instance of the `EndPoint` class opens a socket and communicates across the network to a particular device.

That was just a short description of what `NetworkBridge` does. We could go into more detail, but from a testing perspective, there are already some evident problems. If we want to write tests for `NetworkBridge`, how do we do it? The class could very well make some calls to real hardware when it is constructed. Do we need to have the hardware available to create an instance of the class? Worse than that, how in the world do we know what the bridge is doing to that hardware or the endpoints? From our point of view, the class is a closed box.

It might not be too bad. Maybe we can write some code to sniff packets across the network. Maybe we can get some hardware for `NetworkBridge` to talk to so that at the very least it doesn't freeze when we try to make an instance of it. Maybe we can set up the wiring so that we can have a local cluster of endpoints and use them under test. Those solutions could work, but they are an awful lot of work. The logic that we want to change in `NetworkBridge` might not need any of those things; it's just that we can't get a hold of it. We can't run an object of that class and try it directly to see how it works.

This example illustrates both the sensing and separation problems. We can't sense the effect of our calls to methods on this class, and we can't run it separately from the rest of the application.

Which problem is tougher? Sensing or separation? There is no clear answer. Typically, we need them both, and they are both reasons why we break dependencies. One thing is clear, though: There are many ways to separate software. In fact, there is an entire catalog of those techniques in the back of this book on that topic, but there is one dominant technique for sensing.

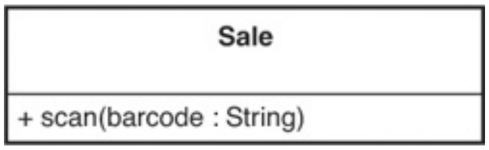
Faking Collaborators

One of the big problems that we confront in legacy code work is dependency. If we want to execute a piece of code by itself and see what it does, often we have to break dependencies on other code. But it's hardly ever that simple. Often that other code is the only place we can easily sense the effects of our actions. If we can put some other code in its place and test through it, we can write our tests. In object orientation, these other pieces of code are often called *fake objects*.

Fake Objects

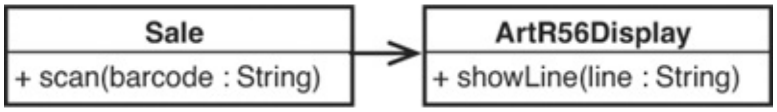
A *fake object* is an object that impersonates some collaborator of your class when it is being tested. Here is an example. In a point-of-sale system, we have a class called `Sale` (see Figure 3.1). It has a method called `scan()` that accepts a bar code for some item that a customer wants to buy. Whenever `scan()` is called, the `Sale` object needs to display the name of the item that was scanned, along with its price on a cash register display.

Figure 3.1 *Sale.*



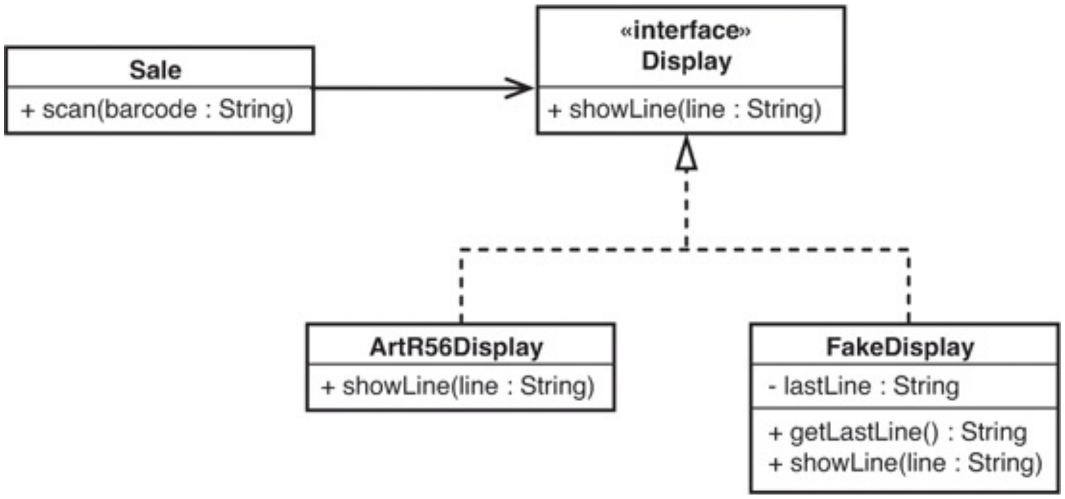
How can we test this to see if the right text shows up on the display? Well, if the calls to the cash register's display API are buried deep in the `Sale` class, it's going to be hard. It might not be easy to sense the effect on the display. But if we can find the place in the code where the display is updated, we can move to the design shown in Figure 3.2.

Figure 3.2 *Sale communicating with a display class.*



Here we've introduced a new class, `ArtR56Display`. That class contains all of the code needed to talk to the particular display device we're using. All we have to do is supply it with a line of text that contains what we want to display. We can move all of the display code in `Sale` over to `ArtR56Display` and have a system that does exactly the same thing that it did before. Does that get us anything? Well, once we've done that, we can move the a design shown in Figure 3.3.

Figure 3.3 *Sale with the display hierarchy.*



The `Sale` class can now hold on to either an `ArtR56Display` or something else, a `FakeDisplay`. The nice thing about having a fake display is that we can write tests against it to find out what the Sale does.

How does this work? Well, `Sale` accepts a display, and a display is an object of any class that implements the `Display` interface.

```
public interface Display
{
    void showLine(String line);
}
```

Both `ArtR56Display` and `FakeDisplay` implement `Display`.

A `Sale` object can accept a display through the constructor and hold on to it internally:

```
public class Sale
{
    private Display display;

    public Sale(Display display) {
        this.display = display;
    }

    public void scan(String barcode) {
        ...
        String itemLine = item.name()
                + " " + item.price().asDisplayText();
        display.showLine(itemLine);
        ...
    }
}
```

In the `scan` method, the code calls the `showLine` method on the `display` variable. But what happens depends upon what kind of a display we gave the `Sale` object when we created it. If we gave it an `ArtR56Display`, it attempts to display on the real cash register hardware. If we gave it a `FakeDisplay`, it won't, but we will be able to see what would've been displayed. Here is a test we can use to see that:

```
import junit.framework.*;

public class SaleTest extends TestCase
{
    public void testDisplayAnItem() {
        FakeDisplay display = new FakeDisplay();
        Sale sale = new Sale(display);

        sale.scan("1");
        assertEquals("Milk $3.99", display.getLastLine());
    }
}
```

The FakeDisplay class is a little peculiar. Let's look at it:

```
public class FakeDisplay implements Display
{
    private String lastLine = "";

    public void showLine(String line) {
        lastLine = line;
    }

    public String getLastLine() {
        return lastLine;
    }
}
```

The showLine method accepts a line of text and assigns it to the lastLine variable. The getLastLine method returns that line of text whenever it is called. This is pretty slim behavior, but it helps us a lot. With the test we've written, we can find out whether the right text will be sent to the display when the Sale class is used.

Fake Objects Support Real Tests

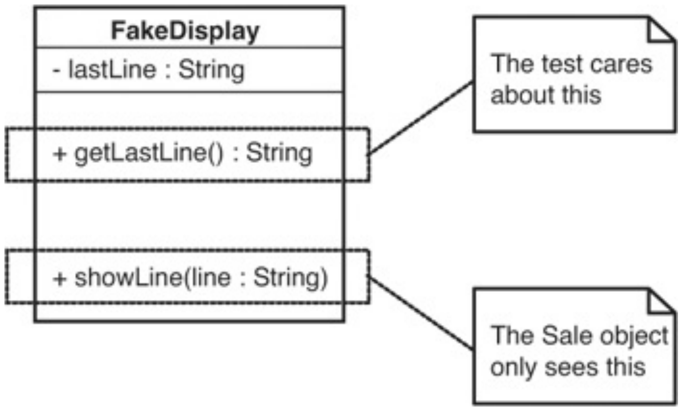
Sometimes when people see the use of fake objects, they say, "That's not really testing." After all, this test doesn't show us what really gets displayed on the real screen. Suppose that some part of the cash register display software isn't working properly; this test would never show it. Well, that's true, but that doesn't mean that this isn't a real test. Even if we could devise a test that really showed us exactly which pixels were set on a real cash register display, does that mean that the software would work with all hardware? No, it doesn't—but that doesn't mean that that isn't a test, either. When we write tests, we have to divide and conquer. This test tells us how Sale objects affect displays, that's all. But that isn't trivial. If we discover a bug, running this test might help us see that the problem isn't in Sale. If we can use information like that to help us localize errors, we can save an incredible amount of time.

When we write tests for individual units, we end up with small, well-understood pieces. This can make it easier to reason about our code.

The Two Sides of a Fake Object

Fake objects can be confusing when you first see them. One of the oddest things about them is that they have two "sides," in a way. Let's take a look at the FakeDisplay class again, in Figure 3.4.

Figure 3.4 *Two sides to a fake object.*


FakeDisplay
- lastLine : String
+ getLastLine() : String
+ showLine(line : String)
The test cares about this
The Sale object only sees this

The showLine method is needed on FakeDisplay because FakeDisplay implements Display. It is the only method on Display and the only one that Sale will see. The other method, getLastLine, is for the use of the test. That is why we declare display as a FakeDisplay, not a Display:

```
import junit.framework.*;

public class SaleTest extends TestCase
{
    public void testDisplayAnItem() {
        FakeDisplay display = new FakeDisplay();
        Sale sale = new Sale(display);
        sale.scan("1");
        assertEquals("Milk $3.99", display.getLastLine());
    }
}
```

The `Sale` class will see the fake display as `Display`, but in the test, we need to hold on to the object as `FakeDisplay`. If we don't, we won't be able to call `getLastLine()` to find out what the sale displays.

Fakes Distilled

The example I've shown in this section is very simple, but it shows the central idea behind fakes. They can be implemented in a wide variety of ways. In OO languages, they are often implemented as simple classes like the `FakeDisplay` class in the previous example. In non-OO languages, we can implement a fake by defining an alternative function, one which records values in some global data structure that we can access in tests. See Chapter 19, *My Project is Not Object-Oriented. How Do I Make Safe Changes?*, for details.

Mock Objects

Fakes are easy to write and are a very valuable tool for sensing. If you have to write a lot of them, you might want to consider a more advanced type of fake called a *mock object*. Mock objects are fakes that perform assertions internally. Here is an example of a test using a mock object:

```
import junit.framework.*;

public class SaleTest extends TestCase
{
    public void testDisplayAnItem() {
        MockDisplay display = new MockDisplay();
        display.setExpectation("showLine", "Milk $3.99");
        Sale sale = new Sale(display);
        sale.scan("1");
        display.verify();
    }
}
```

In this test, we create a mock display object. The nice thing about mocks is that we can tell them what calls to expect, and then we tell them to check and see if they received those calls. That is precisely what happens in this test case. We tell the display to expect its `showLine` method to be called with an argument of "`Milk $3.99`". After the expectation has been set, we just go ahead and use the object inside the test. In this case, we call the method `scan()`. Afterward, we call the `verify()` method, which checks to see if all of the expectations have been met. If they haven't, it makes the test fail.

Mocks are a powerful tool, and a wide variety of mock object frameworks are available. However, mock object frameworks are not available in all languages, and simple fake objects suffice in most situations.

Chapter 4: The Seam Model

One of the things that nearly everyone notices when they try to write tests for existing code is just how poorly suited code is to testing. It isn’t just particular programs or languages. In general, programming languages just don’t seem to support testing very well. It seems that the only ways to end up with an easily testable program are to write tests as you develop it or spend a bit of time trying to “design for testability.” There is a lot of hope for the former approach, but if much of the code in the field is evidence, the latter hasn’t been very successful.

One thing that I’ve noticed is that, in trying to get code under test, I’ve started to think about code in a rather different way. I could just consider this some private quirk, but I’ve found that this different way of looking at code helps me when I work in new and unfamiliar programming languages. Because I won’t be able to cover every programming language in this book, I’ve decided to outline this view here in the hope that it helps you as well as it helps me.

A Huge Sheet of Text

When I first started programming, I was lucky that I started late enough to have a machine of my own and a compiler to run on that machine; many of my friends starting programming in the punch-card days. When I decided to study programming in school, I started working on a terminal in a lab. We could compile our code remotely on a DEC VAX machine. There was a little accounting system in place. Each compile cost us money out of our account, and we had a fixed amount of machine time each term.

At that point in my life, a program was just a listing. Every couple of hours, I’d walk from the lab to the printer room, get a printout of my program and scrutinize it, trying to figure out what was right or wrong. I didn’t know enough to care much about modularity. We had to write modular code to show that we could do it, but at that point I really cared more about whether the code was going to produce the right answers. When I got around to writing object-oriented code, the modularity was rather academic. I wasn’t going to be swapping in one class for another in the course of a school assignment. When I got out in the industry, I started to care a lot about those things, but in school, a program was just a listing to me, a long set of functions that I had to write and understand one by one.

This view of a program as a listing seems accurate, at least if we look at how people behave in relation to programs that they write. If we knew nothing about what programming was and we saw a room full of programmers working, we might think that they were scholars inspecting and editing large important documents. A program can seem like a large sheet of text. Changing a little text can cause the meaning of the whole document to change, so people make those changes carefully to avoid mistakes.

Superficially, that is all true, but what about modularity? We are often told it is better to write programs that are made of small reusable pieces, but how often are small pieces reused independently? Not very often. Reuse is tough. Even when pieces of software look independent, they often depend upon each other in subtle ways.

Seams

When you start to try to pull out individual classes for unit testing, often you have to break a lot of dependencies. Interestingly enough, you often have a lot of work to do, regardless of how “good” the design is. Pulling classes out of existing projects for testing really changes your idea of what “good” is with regard to design. It also leads you to think of software in a completely different way. The idea of a program as a sheet of text just doesn’t cut it anymore. How should we look at it? Let’s take a look at an example, a function in C++.

```
bool CAsyncSslRec::Init()
{
    if (m_bSslInitialized) {
        return true;
    }
    m_smutex.Unlock();
    m_nSslRefCount++;

    m_bSslInitialized = true;

    FreeLibrary(m_hSslDll1);
    m_hSslDll1=0;
    FreeLibrary(m_hSslDll2);
    m_hSslDll2=0;

    if (!m_bFailureSent) {
        m_bFailureSent=TRUE;
        PostReceiveError(SOCKETCALLBACK, SSL_FAILURE);
    }

    CreateLibrary(m_hSslDll1,"syncesel1.dll");
    CreateLibrary(m_hSslDll2,"syncesel2.dll");

    m_hSslDll1->Init();
    m_hSslDll2->Init();

    return true;
}
```

It sure looks like just a sheet of text, doesn't it? Suppose that we want to run all of that method except for this line:

```
    PostReceiveError(SOCKETCALLBACK, SSL_FAILURE);
```

How would we do that?

It's easy, right? All we have to do is go into the code and delete that line.

Okay, let's constrain the problem a little more. We want to avoid executing that line of code because `PostReceiveError` is a global function that communicates with another subsystem, and that subsystem is a pain to work with under test. So the problem becomes, how do we execute the method without calling `PostReceiveError` under test? How do we do that and still allow the call to `PostReceiveError` in production?

To me, that is a question with many possible answers, and it leads to the idea of a seam.

Here's the definition of a seam. Let's take a look at it and then some examples.

Seam

A seam is a place where you can alter behavior in your program without editing in that place.

Is there a seam at the call to `PostReceiveError`? Yes. We can get rid of the behavior there in a couple of ways. Here is one of the most straightforward ones. `PostReceiveError` is a global function, it isn't part of the `CAsynchSslRec` class. What happens if we add a method with the exact same signature to the `CAsynchSslRec` class?

```
class CAsyncSslRec
{
    ...
    virtual void PostReceiveError(UINT type, UINT errorcode);
    ...
};
```

In the implementation file, we can add a body for it like this:

```
void CAsyncSslRec::PostReceiveError(UINT type, UINT errorcode)
{
    ::PostReceiveError(type, errorcode);
}
```

That change should preserve behavior. We are using this new method to delegate to the global `PostReceiveError` function using C++'s scoping operator (::). We have a little indirection there, but we end up calling the same global function.

Okay, now what if we subclass the `CAsyncSslRec` class and override the `PostReceiveError` method?

```
class TestingAsyncSslRec : public CAsyncSslRec
{
    virtual void PostReceiveError(UINT type, UINT errorcode)
    {
    }
};
```

If we do that and go back to where we are creating our `CAsyncSslRec` and create a `TestingAsyncSslRec` instead, we've effectively nulled out the behavior of the call to `PostReceiveError` in this code:

```
bool CAsyncSslRec::Init()
{
    if (m_bSslInitialized) {
        return true;
    }
    m_smutex.Unlock();
    m_nSslRefCount++;

    m_bSslInitialized = true;

    FreeLibrary(m_hSslDll1);
    m_hSslDll1=0;
    FreeLibrary(m_hSslDll2);
    m_hSslDll2=0;

    if (!m_bFailureSent) {
        m_bFailureSent=TRUE;
        PostReceiveError(SOCKETCALLBACK, SSL_FAILURE);
    }

    CreateLibrary(m_hSslDll1,"syncesel1.dll");
    CreateLibrary(m_hSslDll2,"syncesel2.dll");

    m_hSslDll1->Init();
    m_hSslDll2->Init();

    return true;
}
```

Now we can write tests for that code without the nasty side effect.

This seam is what I call an *object seam*. We were able to change the method that is called without changing the method that calls it. *Object seams* are available in object-oriented languages, and they are only one of many different kinds of seams.

Why seams? What is this concept good for?

One of the biggest challenges in getting legacy code under test is breaking dependencies. When we are lucky, the dependencies that we have are small and localized; but in pathological cases, they are numerous and spread out throughout a code base. The seam view of software helps us see the opportunities that are already in the code base. If we can replace behavior at seams, we can selectively exclude dependencies in our tests. We can also run other code where those dependencies were if we want to sense conditions in the code and write tests against those conditions. Often this work can help us get just enough tests in place to support more aggressive work.

Seam Types

The types of seams available to us vary among programming languages. The best way to explore them is to look at all of the steps involved in turning the text of a program into running code on a machine. Each identifiable step exposes different kinds of seams.

Preprocessing Seams

In most programming environments, program text is read by a compiler. The compiler then emits object code or bytecode instructions. Depending on the language, there can be later processing steps, but what about earlier steps?

Only a couple of languages have a build stage before compilation. C and C++ are the most common of them.

In C and C++, a macro preprocessor runs before the compiler. Over the years, the macro preprocessor has been cursed and derided incessantly. With it, we can take lines of text as innocuous looking as this:

```
TEST(getBalance,Account)
{
    Account account;
    LONGS_EQUAL(0, account.getBalance());
}
```

and have them appear like this to the compiler.

```
 class AccountgetBalanceTest : public Test
     { public: AccountgetBalanceTest () : Test ("getBalance" "Test") {}
           void run (TestResult& result_); }
   AccountgetBalanceInstance;
     void AccountgetBalanceTest::run (TestResult& result_)
{
     Account account;
{ result_.countCheck();
  long actualTemp = (account.getBalance());
  long expectedTemp = (0);
  if ((expectedTemp) != (actualTemp))
{ result_.addFailure (Failure (name_, "c:\\seamexample.cpp", 24,
StringFrom(expectedTemp),
StringFrom(actualTemp))); return; } }

}
```

We can also nest code in conditional compilation statements like this to support debugging and different platforms (aarrrgh!):

```
...
m_pRtg->Adj(2.0);

#ifdef DEBUG
#ifndef WINDOWS
    { FILE *fp = fopen(TGLOGNAME,"w");
    if (fp) { fprintf(fp,"%s", m_pRtg->pszState); fclose(fp); }}
#endif

m_pTSRTable->p_nFlush |= GF_FLOT;
#endif

...
```

It's not a good idea to use excessive preprocessing in production code because it tends to decrease code clarity. The conditional compilation directives (`#ifdef`, `#ifndef`, `#if`, and so on) pretty much force you to maintain several different programs in the same source code. Macros (defined with `#define`) can be used to do some very good things, but they just do simple text replacement. It is easy to create macros that hide terribly obscure bugs.

These considerations aside, I'm actually glad that C and C++ have a preprocessor because the preprocessor gives us more seams. Here is an example. In a C program, we have dependencies on a library routine named `db_update`. The `db_update` function talks directly to a database. Unless we can substitute in another implementation of the routine, we can't sense the behavior of the function.

```
#include <DFHLItem.h>
#include <DHLSRecord.h>
extern int db_update(int, struct DFHLItem *);

void account_update(
    int account_no, struct DHLSRecord *record, int activated)
{
    if (activated) {
        if (record->dateStamped && record->quantity > MAX_ITEMS) {
            db_update(account_no, record->item);
        } else {
            db_update(account_no, record->backup_item);
        }
    }
    db_update(MASTER_ACCOUNT, record->item);
}
```

We can use preprocessing seams to replace the calls to `db_update`. To do this, we can introduce a header file called `localdefs.h`.

```
#include <DFHLItem.h>
#include <DHLSRecord.h>

extern int db_update(int, struct DFHLItem *);

#include "localdefs.h"

void account_update(
    int account_no, struct DHLSRecord *record, int activated)
{
    if (activated) {
        if (record->dateStamped && record->quantity > MAX_ITEMS) {
            db_update(account_no, record->item);
        } else {
            db_update(account_no, record->backup_item);
        }
    }
    db_update(MASTER_ACCOUNT, record->item);
}
```

Within it, we can provide a definition for `db_update` and some variables that will be helpful for us:

```
#ifdef TESTING
...
struct DFHLItem *last_item = NULL;
int last_account_no = -1;

#define db_update(account_no,item)\
    {last_item = (item); last_account_no = (account_no);}
...
#endif
```

With this replacement of `db_update` in place, we can write tests to verify that `db_update` was called with the right parameters. We can do it because the `#include` directive of the C preprocessor gives us a seam that we can use to replace text before it is compiled.

Preprocessing seams are pretty powerful. I don't think I'd really want a preprocessor for Java and other more modern languages, but it is nice to have this tool in C and C++ as compensation for some of the other testing obstacles they present.

I didn't mention it earlier, but there is something else that is important to understand about seams: Every seam has an *enabling point*. Let's look at the definition of a seam again:

Seam

A seam is a place where you can alter behavior in your program without editing in that place.

When you have a seam, you have a place where behavior can change. We can't really go to that place and change the code just to test it. The source code should be the same in both production and test. In the previous example, we wanted to change the behavior at the text of the `db_update` call. To exploit that seam, you have to make a change someplace else. In this case, the enabling point is a preprocessor define named `TESTING`. When `TESTING` is defined, the `localdefs.h` file defines macros that replace calls to `db_update` in the source file.

Enabling Point

Every seam has an enabling point, a place where you can make the decision to use one behavior or another.

Link Seams

In many language systems, compilation isn't the last step of the build process. The compiler produces an intermediate representation of the code, and that representation contains calls to code in other files. Linkers combine these representations. They resolve each of the calls so that you can have a complete program at runtime.

In languages such as C and C++, there really is a separate linker that does the operation I just described. In Java and similar languages, the compiler does the linking process behind the scenes. When a source file contains an `import` statement, the compiler checks to see if the imported class really has been compiled. If the class hasn't been compiled, it compiles it, if necessary, and then checks to see if all of its calls will really resolve correctly at runtime.

Regardless of which scheme your language uses to resolve references, you can usually exploit it to substitute pieces of a program. Let's look at the Java case. Here is a little class called `FitFilter`:

```
package fitnesse;

import fit.Parse;
import fit.Fixture;

import java.io.*;
import java.util.Date;


import java.io.*;
import java.util.*;

public class FitFilter {

    public String input;
    public Parse tables;
    public Fixture fixture = new Fixture();
    public PrintWriter output;

    public static void main (String argv[]) {
        new FitFilter().run(argv);
    }

    public void run (String argv[]) {
        args(argv);
        process();
        exit();
    }

    public void process() {
        try {
            tables = new Parse(input);
            fixture.doTables(tables);
        } catch (Exception e) {
            exception(e);
        }
        tables.print(output);
    }
    ...
}
```

In this file, we import `fit.Parse` and `fit.Fixture`. How do the compiler and the JVM find those classes? In Java, you can use a classpath environment variable to determine where the Java system looks to find those classes. You can actually create classes with the same names, put them into a different directory, and alter the classpath to link to a different `fit.Parse` and `fit.Fixture`. Although it would be confusing to use this trick in production code, when you are testing, it can be a pretty handy way of breaking dependencies.

Suppose we wanted to supply a different version of the Parse class for testing. Where would the seam be?

The seam is the `new Parse` *call in the* `process` *method.*

Where is the enabling point?

The enabling point is the `classpath`.

This sort of dynamic linking can be done in many languages. In most, there is some way to exploit link seams. But not all linking is dynamic. In many older languages, nearly all linking is static; it happens once after compilation.

Many C and C++ build systems perform static linking to create executables. Often the easiest way to use the link seam is to create a separate library for any classes or functions you want to replace. When you do that, you can alter your build scripts to link to those rather than the production ones when you are testing. This can be a bit of work, but it can pay off if you have a code base that is littered with calls to a third-party library. For instance, imagine a CAD application that contains a lot of embedded calls to a graphics library. Here is an example of some typical code:

```
void CrossPlaneFigure::rerender()
{
    // draw the label
    drawText(m_nX, m_nY, m_pchLabel, getClipLen());
    drawLine(m_nX, m_nY, m_nX + getClipLen(), m_nY);
    drawLine(m_nX, m_nY, m_nX, m_nY + getDropLen());
    if (!m_bShadowBox) {
        drawLine(m_nX + getClipLen(), m_nY,
                    m_nX + getClipLen(), m_nY + getDropLen());
        drawLine(m_nX, m_nY + getDropLen(),
                    m_nX + getClipLen(), m_nY + getDropLen());
    }

    // draw the figure
    for (int n = 0; n < edges.size(); n++) {
        ...
    }

    ...
}
```

This code makes many direct calls to a graphics library. Unfortunately, the only way to really verify that this code is doing what you want it to do is to look at the computer screen when figures are redrawn. In complicated code, that is pretty error prone, not to mention tedious. An alternative is to use link seams. If all of the drawing functions are part of a particular library, you can create stub versions that link to the rest of the application. If you are interested in only separating out the dependency, they can be just empty functions:

```
void drawText(int x, int y, char *text, int textLength)
{
}

void drawLine(int firstX, int firstY, int secondX, int secondY)
{
}
```

If the functions return values, you have to return something. Often a code that indicates success or the default value of a type is a good choice:

```
int getStatus()
{
    return FLAG_OKAY;
}
```

The case of a graphics library is a little atypical. One reason that it is a good candidate for this technique is that it is almost a pure "tell" interface. You issue calls to functions to tell them to do something, and you aren't asking for much information back. Asking for information is difficult because the defaults often aren't the right thing to return when you are trying to exercise your code.

Separation is often a reason to use a link seam. You can do sensing also; it just requires a little more work. In the case of the graphics library we just faked, we could introduce some additional data structures to record calls:

```
std::queue<GraphicsAction> actions;

void drawLine(int firstX, int firstY, int secondX, int secondY)
{
    actions.push_back(GraphicsAction(LINE_DRAW,
            firstX, firstY, secondX, secondY);
}
```

With these data structures, we can sense the effects of a function in a test:

```
TEST(simpleRender,Figure)
{
    std::string text = "simple";
    Figure figure(text, 0, 0);

    figure.rerender();
    LONGS_EQUAL(5, actions.size());
    GraphicsAction action;
    action = actions.pop_front();
    LONGS_EQUAL(LABEL_DRAW, action.type);
    action = actions.pop_front();
    LONGS_EQUAL(0, action.firstX);
    LONGS_EQUAL(0, action.firstY);
    LONGS_EQUAL(text.size(), action.secondX);
}
```

The schemes that we can use to sense effects can grow rather complicated, but it is best to start with a very simple scheme and allow it to get only as complicated as it needs to be to solve the current sensing needs.

The enabling point for a link seam is always outside the program text. Sometimes it is in a build or a deployment script. This makes the use of link seams somewhat hard to notice.

Usage Tip

If you use link seams, make sure that the difference between test and production environments is obvious.

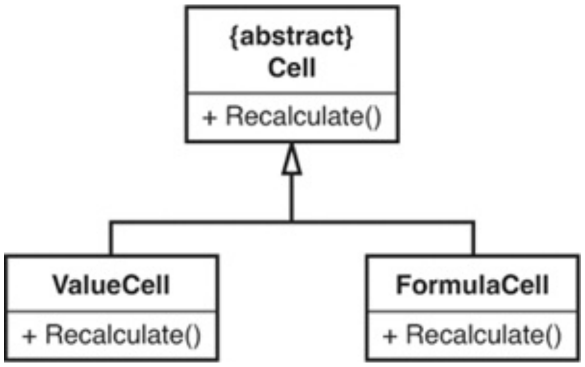
Object Seams

Object seams are pretty much the most useful seams available in object-oriented programming languages. The fundamental thing to recognize is that when we look at a call in an object-oriented program, it does not define which method will actually be executed. Let's look at a Java example:

```
cell.Recalculate();
```

When we look at this code, it seems that there has to be a method named `Recalculate` that will execute when we make that call. If the program is going to run, there has to be a method with that name; but the fact is, there can be more than one:

Figure 4.1 *Cell hierarchy.*



Which method will be called in this line of code?

```
cell.Recalculate();
```

Without knowing what object `cell` points to, we just don't know. It could be the `Recalculate` method of `ValueCell` or the `Recalculate` method of `FormulaCell`. It could even be the `Recalculate` method of some other class that doesn't inherit from `Cell` (if that's the case, `cell` was a particularly cruel name to use for that variable!). If we can change which `Recalculate` is called in that line of code without changing the code around it, that call is a seam.

In object-oriented languages, not all method calls are seams. Here is an example of a call that isn't a seam:

```
public class CustomSpreadsheet extends Spreadsheet
{
    public Spreadsheet buildMartSheet() {
        ...
        Cell cell = new FormulaCell(this, "A1", "=A2+A3");
        ...
        cell.Recalculate();
        ...
    }
    ...
}
```

In this code, we're creating a cell and then using it in the same method. Is the call to `Recalculate` an object seam? No. There is no enabling point. We can't change which `Recalculate` method is called because the choice depends on the class of the cell. The class of the cell is decided when the object is created, and we can't change it without modifying the method.

What if the code looked like this?

```
public class CustomSpreadsheet extends Spreadsheet
{
    public Spreadsheet buildMartSheet(Cell cell) {
        ...
        cell.Recalculate();
        ...
    }
    ...
}
```

Is the call to `cell.Recalculate` in `buildMartSheet` a seam now? Yes. We can create a `CustomSpreadsheet` in a test and call `buildMartSheet` with whatever kind of `Cell` we want to use. We'll have ended up varying what the call to `cell.Recalculate` does without changing the method that calls it.

Where is the enabling point?

In this example, the enabling point is the argument list of `buildMartSheet`. We can decide what kind of an object to pass and change the behavior of `Recalculate` any way that we want to for testing.

Okay, most object seams are pretty straightforward. Here is a tricky one. Is there an object seam at the call to `Recalculate` in this version of `buildMartSheet`?

```
public class CustomSpreadsheet extends Spreadsheet
{
    public Spreadsheet buildMartSheet(Cell cell) {
        ...
        Recalculate(cell);
        ...
    }

    private static void Recalculate(Cell cell) {
        ...
    }


    ...
}
```

The `Recalculate` method is a static method. Is the call to `Recalculate` in `buildMartSheet` a seam? Yes. We don't have to edit `buildMartSheet` to change behavior at that call. If we delete the keyword `static` on `Recalculate` and make it a protected method instead of a private method, we can subclass and override it during test:

```
public class CustomSpreadsheet extends Spreadsheet
{
    public Spreadsheet buildMartSheet(Cell cell) {
        ...
        Recalculate(cell);
        ...
    }
    protected void Recalculate(Cell cell) {
        ...
    }
    ...
}

public class TestingCustomSpreadsheet extends CustomSpreadsheet {
    protected void Recalculate(Cell cell) {
        ...
    }
}
```

Isn't this all rather indirect? If we don't like a dependency, why don't we just go into the code and change it? Sometimes that works, but in particularly nasty legacy code, often the best approach is to do what you can to modify the code as little as possible when you are getting tests in place. If you know the seams that your language offers and how to use them, you can often get tests in place more safely than you could otherwise.

The seams types I've shown are the major ones. You can find them in many programming languages. Let's take a look at the example that led off this chapter again and see what seams we can see:

```
bool CAsyncSslRec::Init()
{
    if (m_bSslInitialized) {
        return true;
    }
    m_smutex.Unlock();
    m_nSslRefCount++;

    m_bSslInitialized = true;

    FreeLibrary(m_hSslDll1);
    m_hSslDll1=0;
    FreeLibrary(m_hSslDll2);
    m_hSslDll2=0;

    if (!m_bFailureSent) {
        m_bFailureSent=TRUE;
        PostReceiveError(SOCKETCALLBACK, SSL_FAILURE);
    }

    CreateLibrary(m_hSslDll1,"syncesel1.dll");
    CreateLibrary(m_hSslDll2,"syncesel2.dll");

    m_hSslDll1->Init();
    m_hSslDll2->Init();
    return true;
}
```

What seams are available at the `PostReceiveError` call? Let's list them.

1. `PostReceiveError` is a global function, so we can easily use the *link seam* there. We can create a library with a stub function and link to it to get rid of the behavior. The enabling point would be our makefile or some setting in our IDE. We'd have to alter our build so that we would link to a testing library when we are testing and a production library when we want to build the real system.
2. We could add a `#include` statement to the code and use the preprocessor to define a macro named `PostReceiveError` when we are testing. So, we have a *preprocessing seam* there. Where is the *enabling point*? We can use a preprocessor define to turn the macro definition on or off.
3. We could also declare a virtual function for `PostRecieveError` like we did at the beginning of this chapter, so we have an *object seam* there also. Where is the enabling point? In this case, the enabling point is the place where we decide to create an object. We can create either an `CAsyncSslRec` object or an

object of some testing subclass that overrides `PostRecieveError`.

It is actually kind of amazing that there are so many ways to replace the behavior at this call without editing the method:

```
bool CAsyncSslRec::Init()
{
    ...
    if (!m_bFailureSent) {
        m_bFailureSent=TRUE;
        PostReceiveError(SOCKETCALLBACK, SSL_FAILURE);
    }
    ...

    return true;
}
```

It is important to choose the right type of seam when you want to get pieces of code under test. In general, *object seams* are the best choice in object-oriented languages. *Preprocessing seams* and *link seams* can be useful at times but they are not as explicit as *object seams*. In addition, tests that depend upon them can be hard to maintain. I like to reserve *preprocessing seams* and *link seams* for cases where dependencies are pervasive and there are no better alternatives.

When you get used to seeing code in terms of seams, it is easier to see how to test things and to see how to structure new code to make testing easier.

Chapter 5: Tools

What tools do you need when you work with legacy code? You need an editor (or an IDE) and your build system, but you also need a testing framework. If there are refactoring tools for your language, they can be very helpful as well.

In this chapter, I describe some of the tools that are currently available and the role that they can play in your legacy code work.

Automated Refactoring Tools

Refactoring by hand is fine, but when you have a tool that does some refactoring for you, you have a real time saver. In the 1990s, Bill Opdyke started work on a C++ refactoring tool as part of his thesis work on refactoring. Although it never became commercially available, to my knowledge, his work inspired many other efforts in other languages. One of the most significant was the Smalltalk refactoring browser developed by John Brant and Don Roberts at the University of Illinois. The Smalltalk refactoring browser supported a very large number of refactorings and has served as a state-of-the-art example of automated refactoring technology for a long while. Since then, there have been many attempts to add refactoring support to various languages in wider use. At the time of this writing, many Java refactoring tools are available; most are integrated into IDEs, but a few are not. There are also refactoring tools for Delphi and some relatively new ones for C++. Tools for C# refactoring are under active development at the time of this writing.

With all of these, tools it seems that refactoring should be much easier. It is, in some environments. Unfortunately, the refactoring support in many of these tools varies. Let's remember what refactoring is again. Here is Martin Fowler's definition from *Refactoring: Improving the Design of Existing Code* (Addison-Wesley 1999):

refactoring (n.). A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its existing behavior.

A change is a refactoring only if it doesn't change behavior. Refactoring tools should verify that a change does not change behavior, and many of them do. This was a cardinal rule in the Smalltalk refactoring browser, Bill Opdyke's work, and many of the early Java refactoring tools. At the fringes, however, some tools don't really check—and if they don't check, you could be introducing subtle bugs when you refactor.

It pays to choose your refactoring tools with care. Find out what the tool developers say about the safety of their tool. Run your own tests. When I encounter a new refactoring tool, I often run little sanity checks. When you attempt to extract a method and give it the name of a method that already exists in that class, does it flag that as an error? What if it is the name of a method in a base class—does the tool detect that? If it doesn't, you could mistakenly override a method and break code.

In this book, I discuss work with and without automated refactoring support. In the examples, I mention whether I am assuming the availability of a refactoring tool.

In all cases, I assume that the refactorings supplied by the tool preserve behavior. If you discover that the ones supplied by your tool don't preserve behavior, don't use the automated refactorings. Follow the advice for cases in which you don't have a refactoring tool—it will be safer.

Tests and Automated Refactoring

When you have a tool that does refactorings for you, it's tempting to believe that you don't have to write tests for the code you are about to refactor. In some cases, this is true. If your tool performs safe refactorings and you go from one automated refactoring to another without doing any other editing, you can assume that your edits haven't changed behavior. However, this isn't always the case.

Here is an example:

```
public class A {
    private int alpha = 0;
    private int getValue() {
        alpha++;
        return 12;
    }
    public void doSomething() {
        int v = getValue();
        int total = 0;
        for (int n = 0; n < 10; n++) {
             total += v;
        }
    }
}
```

In at least two Java refactoring tools, we can use a refactoring to remove the `v` variable from `doSomething`. After the refactoring, the code looks like this:

```
public class A {
    private int alpha = 0;
    private int getValue() {
        alpha++;
        return 12;
    }
    public void doSomething() {
       int total = 0;
       for (int n = 0; n < 10; n++) {
           total += getValue();
       }
    }
}
```

See the problem? The variable was removed, but now the value of `alpha` is incremented 10 times rather than 1. This change clearly didn't preserve behavior.

It is a good idea to have tests around your code before you start to use automated refactorings. You can do some automated refactoring without tests, but you have to know what the tool is checking and what it isn't. When I start to use a new tool, the first thing that I do is put its support for extracting methods through its paces. If I can trust it well enough to use it without tests, I can get the code into a much more testable state.

Mock Objects

One of the big problems that we confront in legacy code work is dependency. If we want to execute a piece of code by itself and see what it does, often we have to break dependencies on other code. But it's hardly ever that simple. If we remove the other code, we need to have something in its place that supplies the right values when we are testing so that we can exercise our piece of code thoroughly. In object-oriented code, these are often called mock objects.

Several mock object libraries are freely available. The web site www.mockobjects.com is a good place to find references for most of them.

Unit-Testing Harnesses

Testing tools have a long and varied history. Not a year goes by that I don't run into four or five teams that have bought some expensive license-per-seat testing tool that ends up not living up to its price. In fairness to tool vendors, testing is a tough problem, and people are often seduced by the idea that they can test through a GUI or web interface without having to do anything special to their application. It can be done, but it is usually more work than anyone on a team is prepared to admit. In addition, a user interface often isn't the best place to write tests. UIs are often volatile and too far from the functionality being tested. When UI-based tests fail, it can be hard to figure out why. Regardless, people often spend considerable money trying to do all of their testing with those sorts of tools.

The most effective testing tools I've run across have been free. The first one is the xUnit testing framework. Originally written in Smalltalk by Kent Beck and then ported to Java by Kent Beck and Erich Gamma, xUnit is a small, powerful design for a unit-testing framework. Here are its key features:

- It lets programmers write tests in the language they are developing in.
- All tests run in isolation.
- Tests can be grouped into suites so that they can be run and rerun on demand.

The xUnit framework has been ported to most major languages and quite a few small, quirky ones.

The most revolutionary thing about xUnit's design is its simplicity and focus. It allows us to write tests with little muss and fuss. Although it was originally designed for unit testing, you can use it to write larger tests because xUnit really doesn't care how large or small a test is. If the test can be written in the language you are using, xUnit can run it.

In this book, most of the examples are in Java and C++. In Java, JUnit is the preferred xUnit harness, and it looks very much like most of the other xUnits. In C++, I often use a testing harness I wrote named CppUnitLite. It looks quite a bit different, and I describe it in this chapter also. By the way, I'm not slighting the original author of CppUnit by using CppUnitLite. I was that guy a long time ago, and I discovered only after I released CppUnit that it could be quite a bit smaller, easier to use, and far more portable if it used some C idioms and only a bare subset of the C++ language.

JUnit

In JUnit, you write tests by subclassing a class named `TestCase`.

```
import junit.framework.*;

public class FormulaTest extends TestCase {
    public void testEmpty() {
        assertEquals(0, new Formula("").value());
    }

    public void testDigit() {
        assertEquals(1, new Formula("1").value());
    }
}
```

Each method in a test class defines a test if it has a signature of this form: `void testXXX()`, where `XXX` is the name you want to give the test. Each test method can contain code and assertions. In the previous `testEmpty` method, there is code to create a new `Formula` object and call its `value` method. There is also assertion code that checks to see if that value is equal to `0`. If it is, the test passes. If it isn't, the test fails.

In a nutshell, here is what happens when you run JUnit tests. The JUnit test runner loads a test class like the one shown previously, and then it uses reflection to find all of the test methods. What it does next is kind of sneaky. It creates a completely separate object for each one of those test methods. From the previous code, it creates two of them: an object whose only job is to run the `testEmpty` method, and an object whose only job is to run the `testDigit` object. If you are wondering what the classes of the objects are, in both cases, it is the same: `FormulaTest`. Each object is configured to run exactly one of the test methods on `FormulaTest`. The key thing is that we have a completely separate object for each method. There is no way that they can affect each other. Here is an example.

```
public class EmployeeTest extends TestCase {
    private Employee employee;

    protected void setUp() {
        employee = new Employee("Fred", 0, 10);
        TDate cardDate = new TDate(10, 10, 2000);
        employee.addTimeCard(new TimeCard(cardDate,40));
    }

    public void testOvertime() {
        TDate newCardDate = new TDate(11, 10, 2000);
        employee.addTimeCard(new TimeCard(newCardDate, 50));
        assertTrue(employee.hasOvertimeFor(newCardDate));
    }

    public void testNormalPay() {
        assertEquals(400, employee.getPay());
    }
}
```

In the `EmployeeTest` class, we have a special method named `setUp`. The `setUp` method is defined in `TestCase` and is run in each test object before the test method is run. The `setUp` method allows us to create a set of objects that we'll use in a test. That set of objects is created the same way before each test's execution. In the object that runs `testNormalPay`, an employee created in `setUp` is checked to see if it calculates pay correctly for one timecard, the one added in `setUp`. In the object that runs `testOvertime`, an employee created in `setUp` for that object gets an additional timecard, and there is a check to verify that the second timecard triggers an overtime condition. The `setUp` method is called for each object of the class `EmployeeTest`, and each of those objects gets its own set of objects created via `setUp`. If you need to do anything special after a test finishes executing, you can override another method named `tearDown`, defined in `TestCase`. It runs after the test method for each object

When you first see an xUnit harness, it is bound to look a little strange. Why do test-case classes have `setUp` and `tearDown` at all? Why can't we just create the objects we need in the constructor? Well, we could, but remember what the test runner does with test case classes. It goes to each test case class and creates a set of objects, one for each test method. That is a large set of objects, but it isn't so bad if those objects haven't allocated what they need yet. By placing code in `setUp` to create what we need just when we need it, we save quite a bit on resources. In addition, by delaying `setUp`, we can also run it at a time when we can detect and report any problems that might happen during setup.

CppUnitLite

When I did the initial port of CppUnit, I tried to keep it as close as I could to JUnit. I figured it would be easier for people who'd seen the xUnit architecture before, so it seemed to be the better thing to do. Almost immediately, I ran into a series of things that were hard or impossible to implement cleanly in C++ because of differences in C++ and Java's features. The primary issue was C++'s lack of reflection. In Java, you can hold on to a reference to a derived class's methods, find methods at runtime, and so on. In C++, you have to write code to register the method you want to access at runtime. As a result, CppUnit became a little bit harder to use and understand. You had to write your own suite function on a test class so that the test runner could run objects for individual methods.

```
Test *EmployeeTest::suite()
{
    TestSuite *suite = new TestSuite;
    suite.addTest(new TestCaller<EmployeeTest>("testNormalPay",
            testNormalPay));
    suite.addTest(new TestCaller<EmployeeTest>("testOvertime",
            testOvertime));
    return suite;
}
```

Needless to say, this gets pretty tedious. It is hard to maintain momentum writing tests when you have to declare test methods in a class header, define them in a source file, and register them in a suite method. A variety of macro schemes can be used to get past these issues, but I choose to start over. I ended up with a scheme in which someone could write a test just by writing this source file:

```
#include "testharness.h"
#include "employee.h"
#include <memory>

using namespace std;

TEST(testNormalPay,Employee)
{
    auto_ptr<Employee> employee(new Employee("Fred", 0, 10));
    LONGS_EQUALS(400, employee->getPay());
}
```

This test used a macro named `LONGS_EQUAL` that compares two long integers for equality. It behaves the same way that `assertEquals` does in JUnit, but it's tailored for `long`s.

The `TEST` macro does some nasty things behind the scenes. It creates a subclass of a testing class and names it by pasting the two arguments together (the name of the test and the name of the class being tested). Then it creates an instance of that subclass that is configured to run the code in braces. The instance is static; when the program loads, it adds itself to a static list of test objects. Later a test runner can rip through the list and run each of the tests.

After I wrote this little framework, I decided not to release it because the code in the macro wasn't terribly clear, and I spend a lot of time convincing people to write clearer code. A friend of mine, Mike Hill, ran into some of the same issues before we met and created a Microsoft-specific testing framework called TestKit that handled registration the same way. Emboldened by Mike, I started to reduce the number of late C++ features used in my little framework, and then I released it. (Those issues had been a big issue in CppUnit. Nearly every day I received e-mail from people who couldn't use templates or the standard library, or who had exceptions with their C++ compiler.)

Both CppUnit and CppUnitLite are adequate as testing harnesses. Tests written using CppUnitLite are a little briefer, so I use it for the C++ examples in this book.

NUnit

NUnit is a testing framework for the .NET languages. You can write tests for C# code, VB.NET code, or any other language that runs on the .NET platform. NUnit is very close in operation to JUnit. The one significant difference is that it uses attributes to mark test methods and test classes. The syntax of attributes depends upon the .NET language the tests are written in.

Here is an NUnit test written in VB.NET:

```
Imports NUnit.Framework

<TestFixture()> Public Class LogOnTest
    Inherits Assertion

    <Test()> Public Sub TestRunValid()
        Dim display As New MockDisplay()
        Dim reader As New MockATMReader()
        Dim logon As New LogOn(display, reader)
        logon.Run()
        AssertEquals("Please Enter Card", display.LastDisplayedText)
        AssertEquals("MainMenu",logon.GetNextTransaction().GetType.Name)
    End Sub
End Class
```

`<TestFixture()>` and `<Test()>` are attributes that mark `LogonTest` as a test class and `TestRunValid` as a test method, respectively.

Other xUnit Frameworks

There are many ports of xUnit to many different languages and platforms. In general, they support the specification, grouping, and running of unit tests. If you need to find an xUnit port for your platform or language, go to www.xprogramming.com and look in the Downloads section. This site is run by Ron Jeffries, and it is the de facto repository for all of the xUnit ports.

General Test Harnesses

The xUnit frameworks I described in the preceding section were designed to be used for unit testing. They can be used to test several classes at a time, but that sort of work is more properly the domain of FIT and Fitnesse.

Framework for Integrated Tests (FIT)

FIT is a concise and elegant testing framework that was developed by Ward Cunningham. The idea behind FIT is simple and powerful. If you can write documents about your system and embed tables within them that describe inputs and outputs for your system, and if those documents can be saved as HTML, the FIT framework can run them as tests.

FIT accepts HTML, runs tests defined in HTML tables in it, and produces HTML output. The output looks the same as the input, and all text and tables are preserved. However, the cells in the tables are colored green to indicate values that made a test pass and red to indicate values that caused a test to fail. You also can use options to have test summary information placed in the resulting HTML.

The only thing you have to do to make this work is to customize some table-handling code so that it knows how to run chunks of your code and retrieve results from them. Generally, this is rather easy because the framework provides code to support a number of different table types.

One of the very powerful things about FIT is its capability to foster communication between people who write software and people who need to specify what it should do. The people who specify can write documents and embed actual tests within them. The tests will run, but they won't pass. Later developers can add in the features, and the tests will pass. Both users and developers can have a common and up-to-date view of the capabilities of the system.

There is far more to FIT than I can describe here. There is more information about FIT at http://fit.c2.com.

Fitnesse

Fitnesse is essentially FIT hosted in a wiki. Most of it was developed by Robert Martin and Micah Martin. I worked on a little bit of it, but I dropped out to concentrate on this book. I'm looking forward to getting back to work on it soon.

Fitnesse supports hierarchical web pages that define FIT tests. Pages of test tables can be run individually or in suites, and a multitude of different options make collaboration easy across a team. Fitnesse is available at http://www.fitnesse.org. Like all of the other testing tools described in this chapter, it is free and supported by a community of developers.

Part II: Changing Software