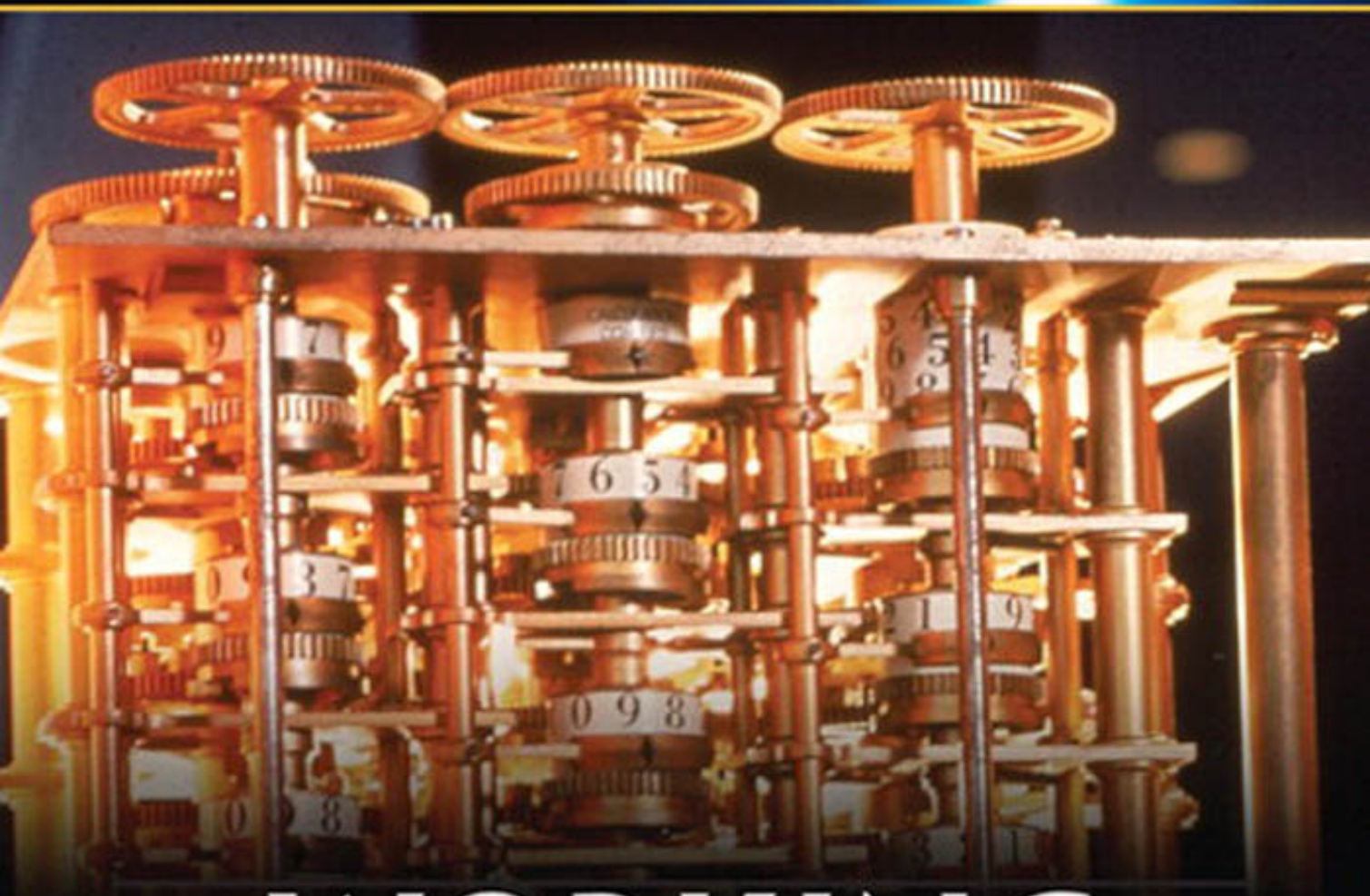
Robert C. Martin Series



WORKING EFFECTIVELY WITH LEGACY CODE

Michael C. Feathers

Working Effectively with Legacy Code

Robert C. Martin Series

This series is directed at software developers, team-leaders, business analysts, and managers who want to increase their skills and proficiency to the level of a Master Craftsman.

The series contains books that guide software professionals in the principles, patterns, and practices of programming, software project management, requirements gathering, design, analysis, testing, and others.

Working Effectively with Legacy Code

Michael C. Feathers

Prentice Hall Professional Technical Reference
Upper Saddle River, NJ 07458
www,phptr.com

Publisher: John Wait
Editor in Chief: Don O'Hagan
Acquisitions Editor: Paul Petralia
Editorial Assistant: Michelle Vincenti
Marketing Manager: Chris Guzikowski
Publicist: Kerry Guiliano
Cover Designer: Sandra Schroeder
Managing Editor: Gina Kanouse
Senior Project Editor: Lori Lyons
Copy Editor: Krista Hansing
Indexer: Lisa Stumpf
Compositor: Karen Kennedy
Proofreader: Debbie Williams
Manufacturing Buyer: Dan Uhrig

Library of Congress Cataloging-in-Publication Data: 2004108115

ISBN 0-13-117705-2

Text printed in the United States on recycled paper at Phoenix Book Tech.
First printing, September 2004

For Ann, Deborah, and Ryan,
the bright centers of my life.

— Michael

This page intentionally left blank

Contents

This page intentionally left blank

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.

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).

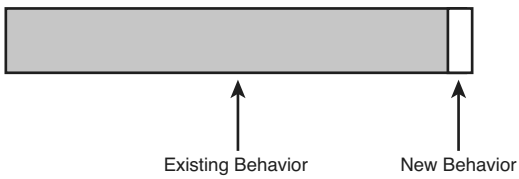


Figure 1.1 *Preserving behavior.*

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 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:

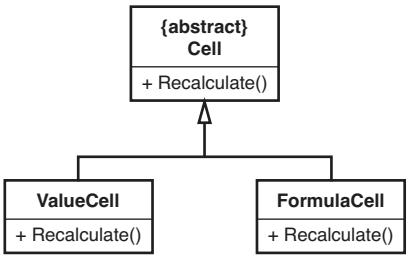


Figure 4.1 *Cell hierarchy.*

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.

Index