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.

This page intentionally left blank

Part I

The Mechanics of Change

This page intentionally left blank

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:

```java
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).

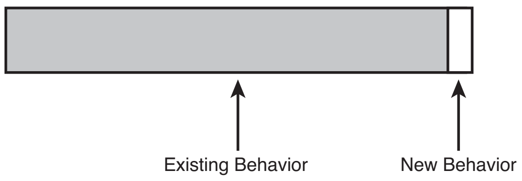

Figure 1.1 *Preserving behavior.*

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.