Extracted from:

Modern C++ Programming with Test-Driven Development

Code Better, Sleep Better

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Modern C++ Programming with Test-Driven Development

Code Better, Sleep Better

Jeff Langr

Edited by Michael Swaine

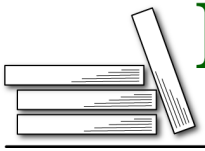
Modern C++ Programming with Test-Driven Development

Code Better, Sleep Better

Jeff Langr

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

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

The team that produced this book includes:

Michael Swaine (editor)
Potomac Indexing, LLC (indexer)
Kim Wimpsett (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-937785-48-2
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—October 2013

Introduction

Despite the current explosion in programming languages, C++ soldiers on. It is the fourth-most popular programming language, per the July 2013 Tiobe index. (You can find the latest index at http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.) The 2011 ISO standard (ISO/IEC 14822:2011, aka C++11) brings features to C++ that may increase its acceptance...or at least soften objections against its use.

C++ remains one of your best choices for building high-performance solutions. If your company's products integrate with hardware, chances are you have a sizeable existing system built in C++. If your company has been around since the 1990s or earlier, chances are you have a long-lived C++ system, and chances are also good that it's not disappearing anytime in the next several years.

Given that you're now working in C++, you might be thinking one of the following things:

- It's 2013. Why am I back in this difficult (but entertaining) language that I thought I'd abandoned years ago? How am I going to survive without shooting myself in the foot?
- I'm a seasoned C++ pro, I know this powerful language like the back of my hand, and I've been developing successfully for years. Why would I need to change how I work?
- Where's my paycheck?

My personal sentiment is that I first worked with C++ around the early 1990s, before the rise of things like templates (and template metaprogramming!), RTTI, STL, and Boost. Since then, I've had a few occasions where I've had to return to the powerful language and have done so with some dismay. Like any language, C++ allows you to shoot yourself in the foot—but with C++, you sometimes don't realize you shot yourself until it's too late. And you're probably missing more toes than folks working with other languages.

If you've been working with C++ for years, you likely have adopted many idioms and practices to help ensure your code remains of high quality. Die-hard C++ veterans are some of the more careful programmers across all languages because surviving in C++ for so long requires meticulous care and attention to the code you craft.

With all this care taken, one might think that the code quality on C++ systems should be high. Yet most C++ systems exhibit the same problems we all see time and time again, usually regardless of language.

- Monstrous source files with thousands of lines
- Member functions with hundreds or thousands of lines of inscrutable code
- Volumes of dead code
- Build times extending into several hours
- High numbers of defects
- Logic too convoluted by quick fixes to be safely managed
- Code duplicated across files, classes, and modules
- Code riddled with long-obsolete coding practices

Is this decay inevitable? No! Test-Driven Development is a tool you can master and wield in order to help stave off system entropy. It may even reinvigorate your passion for programming.

If you're simply seeking a paycheck, there are plenty of C++ jobs out there that will keep you employed. However, C++ is a highly technical and nuanced language. Wielding it carelessly will lead to defects, intermittent failures, and possibly multiday debugging sessions—factors that can put your paycheck at risk. TDD can help.

The effort required to add new functionality on such large, long-lived C++ systems usually disappoints and often is inestimable. Simply understanding a passage of code in order to change a few lines of code can take hours, even days. Productivity is further drained as developers wait hours to determine whether their changes compiled and wait even longer to see whether they integrated well with the remainder of the system.

It doesn't have to be this way. *Test-Driven Development* (TDD), a software design technique devised in the late 1990s, can help you wrestle your C++ system to the ground and keep it under control as you continue adding new features. (Notions of writing tests before code have been around for considerably longer. However, the TDD cycle in its formal, disciplined form was devised by Ward Cunningham and Kent Beck [*Test Driven Development: By Example [Bec02]*].)

The primary intent for this book is to teach you a disciplined approach for the practical application of TDD. You'll learn the following:

- The fundamental mechanics of TDD
- The potential benefits of TDD
- How TDD helps you address design deficiencies as they arise
- The challenges and costs of doing TDD
- How TDD can reduce or even eliminate debugging sessions
- How to sustain TDD over time

But Can It Work for Me on My System?

"What's all this fuss about unit testing? It doesn't seem to be helping me much."

You might have already tried unit testing. Perhaps you are currently struggling with writing unit tests against your legacy system. Maybe it seems like TDD is OK for those rare other souls fortunate enough to be working on a new system. But does it solve your day-to-day problems of working on a long-entrenched, challenging C++ system?

Indeed, TDD is a useful tool but is no silver bullet for dealing with legacy systems. While you can test-drive many of the new features you add to your system, you'll also need to begin chipping away at the years of accumulated cruft. You'll need additional strategies and tactics for a sustained cleanup approach. You'll need to learn about tactical dependency-breaking techniques and safe code change approaches that Michael Feathers presents in the essential book *Working Effectively with Legacy Code [Fea04]*. You'll need to understand how to approach large-scale refactorings without creating large-scale issues. For that, you'll learn about the *Mikado Method [BE12]*. This book will teach such supportive practices and more.

Simply adding unit tests for code you've already written (something I call *Test-After Development* [TAD]) usually has little impact as you struggle with "this is just how the system is." You might invest thousands of person-hours in writing tests with little measurable impact to system quality.

If you allow TDD to help you shape your systems, your designs will by definition be testable. They will also be different—in many ways, better—than if you did not use TDD. The more you understand what a good design should look like, the more TDD will help guide you there.

To aid you in shifting how you think about design, this book emphasizes principles that underlie good code management, such as the SOLID principles

of object-oriented design described in *Agile Software Development, Principles, Patterns, and Practices [Mar02]*. I discuss how this sense of good design supports ongoing development and productivity and how TDD can help a mindful developer achieve more consistent and reliable results.

Who This Book Is For

This book is written to help C++ programmers of all skill levels, from novices with a fundamental understanding of the language to old salts steeped in language esoterica. If you've been away from C++ for some time, you'll find that the rapid feedback cycles of TDD will help you rapidly reramp up on the language.

While the goal of this book is to teach TDD, you will find value in this book regardless of your TDD experience. If you are a complete novice to the concept of writing unit tests for your code, I'll take you small step by small step through the basics of TDD. If you are fairly new to TDD, you'll discover a wealth of expert advice throughout the book, all presented in simple fashion with straightforward examples. Even seasoned test-drivers should find some useful nuggets of wisdom, a stronger theoretical basis for the practice, and some new topics for exploration.

If you are a skeptic, you'll explore TDD from several angles. I'll inject my thoughts throughout about why I think TDD works well, and I'll also share experiences about when it didn't work so well and why. The book is not a sales brochure but an eyes-open exploration of a transformative technique.

Readers of all stripes will also find ideas for growing and sustaining TDD on their team. It's easy to get started with TDD, but your team will encounter many challenges along the way. How can you prevent these challenges from derailing your transition effort? How do you prevent such disasters? I present some ideas that I've seen work well in *Growing and Sustaining TDD*.

What You'll Need

To code any of the examples in this book, you'll need a compiler, of course, and a unit testing tool. Some of the examples also require third-party libraries. This section overviews these three elements. You'll want to refer to *Global Setup* for further details around what you'll need.

A Unit Testing Tool

Out of the dozens of available C++ unit testing tools, I chose Google Mock (which sits atop Google Test) for most of the examples in this book. It currently returns the most hits on a web search, but I primarily chose it because it

supports Hamcrest notation (a matcher-based assertion form designed to provide highly expressive tests). The information in *Global Setup* will help you come up to speed on Google Mock.

However, this book is neither a comprehensive treatise nor a sales brochure for Google Mock. It is instead a book that teaches the discipline of TDD. You'll learn enough Google Mock to practice TDD effectively.

You'll also use another unit testing tool named CppUTest for some of the examples. You'll find that it's fairly easy to learn another unit testing tool, which should help ease any concerns you might have if you're not using Google Mock or CppUTest.

If you are using a different unit testing tool such as CppUnit or Boost.Test, no worries! These other tools work much like Google Mock in concept and are often similar in implementation. You can easily follow along and do the TDD examples using virtually any of the other C++ unit testing tools available. See *Comparing Unit Testing Tools* for a discussion of what's important in choosing a unit testing tool.

Most examples in this book use Google Mock for mocking and stubbing (see *Test Doubles*). Of course, Google Mock and Google Test work together, but you might also be able to integrate Google Mock successfully with your unit testing tool of choice.

A Compiler

You'll need access to a C++ compiler with support for C++11. The book example code was originally built using gcc and works out of the box on Linux and Mac OS. See *Global Setup* for information about building the example code on Windows. All examples use the STL, an essential part of modern C++ development for many platforms.

Third-Party Libraries

Some of the examples use freely available third-party libraries. Refer to *Global Setup* for the specific list of libraries you'll need to download.

How to Use This Book

I designed the chapters in the book to function as stand-alone as possible. You should be able to pick up a random chapter and work through it without having to fully read any other chapters. I provide ample cross-references throughout to allow you to jump around easily if you're using an ereader.

Each chapter begins with a quick overview and ends with a chapter summary plus a preview of the next chapter. I chose the names of these brief sections to correspond cutely to the initialization and cleanup sections used by many unit test frameworks—"Setup" and "Teardown."

The Source

The book contains numerous code examples. Most of the code presented will reference a specific filename. You can find the complete set of example code for this book at http://pragprog.com/book/lotdd/modern-c-programming-with-test-driven-development and also at my GitHub page, http://github.com/jlangr.

Within the code distribution, examples are grouped by chapter. Within the directory for each chapter, you will find numbered directories, each number effectively referring to a version number (which allows the book to use and show examples of code changing as you progress through each chapter). As an example, the code with caption `c2/7/SoundexTest.cpp` refers to the file `SoundexTest.cpp` located in the seventh revision of the Chapter 2 (`c2`) code directory.

Book Discussion

Please join the discussion forum at https://groups.google.com/forum/?fromgroups#!forum/modern-cpp-with-tdd. The intent for the forum is to discuss the book as well as doing TDD in C++ in general. I will also post useful information regarding the book.

If You Are New to TDD: What's in the Book

While this book is geared to all, its primary focus is on programmers new to TDD, so its chapters are in a correspondingly sequential order. I highly recommend you work through the exercise in *Test-Driven Development: A First Example*. It will give you a strong feel for many of the ideas behind TDD as you work through a meaty example. Don't just read—type along and make sure your tests pass when they should!

The next two chapters, *Test-Driven Development Foundations* and *Test Construction*, are also essential reading. They cover core ideas about what TDD is (and is not) and how to construct your tests. Make sure you're comfortable with the material in these chapters before learning about mocks (see the *Test Doubles* chapter), a technique essential to building most production systems.

Don't skip the chapter on design and refactoring (*Incremental Design*) just because you think you know what that means. An essential reason to practice TDD is to enable you to evolve your design and keep your code clean continually through refactoring. Most systems exhibit poor design and difficult code, partly because developers aren't willing to refactor enough or don't know how.

You'll learn what's far enough and how to start reaping the potential benefits of a smaller, simpler system.

To wrap up core TDD techniques, *Quality Tests* takes a look at a number of ways to improve your return on investment in TDD. Learning some of these techniques can make the difference between surviving and thriving in TDD.

You'll of course be saddled with the struggles of an existing system that wasn't test-driven. You can get a jump start on some simple techniques to tackle your legacy code by reading *Legacy Challenges*.

Just past the legacy code material, you'll find a chapter dedicated to test-driving multithreaded code. The test-driven approach to TDD may surprise you.

The next chapter, *Additional TDD Concepts and Discussions*, dives deeper into fairly specific areas and concerns. You'll discover some up-to-date ideas around TDD, including some alternate approaches that differ from what you'll find elsewhere in this book.

Finally, you'll want to know what it takes to get TDD going in your team, and you'll of course want to make sure you're able to sustain your investment in TDD. The last chapter, *Growing and Sustaining TDD*, provides some ideas that you will want to incorporate into your shop.

If You Have Some Experience with TDD

You can probably get away with picking up chapters at random, but you'll find a lot of nuggets of hard-earned wisdom strewn throughout the book.

Conventions Used in This Book

Any sizeable code segment will appear separately from the text. When the text refers to code elements, the following conventions are used:

- A *ClassName* will appear in the same font as normal text ("text font") and will be UpperCamelCase.
- A *TestName* will also appear in the text font and be UpperCamelCase.
- All other code elements will appear in a code (nonproportional) font. Examples of these include the following:
  - functionName() (which will show an empty argument list, even if it refers to a function declaring one or more parameters). I will sometimes refer to member functions as *methods*.
  - variableName
  - keyword
  - All other code snippets

To keep things simple and waste fewer pages, code listings will often omit code irrelevant to the current discussion. A comment followed by ellipses represents obscured code. For example, the body of the for loop is replaced in this code snippet:

```
for (int i = 0; i < count; i++) {
   // ...
}
```

About "Us"

I wrote this book to be a dialogue between us. Generally I'm talking to you, as the reader. When I (ideally infrequently) refer to myself, it's usually to describe an experience-based opinion or preference. The implication is that it might not be a widely accepted concept (but it could be a fine idea!).

When it gets down to coding, I'd rather you didn't have to work alone, particularly since you're trying to learn. *We* will work through all of the coding exercises in the book together.

About Me

I've been programming since 1980, my junior year in high school, and professionally since 1982 (I worked at the University of Maryland while pursuing my BS in computer science). I transitioned from programmer to consultant in 2000, when I had the joy of working for Bob Martin and occasionally alongside some great folks at Object Mentor.

I started Langr Software Solutions in 2003 to provide consulting and training solutions related to Agile software development. Much of my work is either pairing with developers doing TDD or teaching it. I insist on alternating between consulting/training and being a "real" programmer on a real development team so that I stay up-to-date and relevant. Since 2002, I have been a full-time programmer in four different companies for significant durations.

I love writing about software development. It's part of how I learn things in depth, but I also enjoy helping others come up to speed on building quality code. This is my fourth book. I wrote *Essential Java Style: Patterns for Implementation [Lan99]* and *Agile Java: Crafting Code With Test-Driven Development [Lan05]*, and I co-wrote *Agile in a Flash [OL11]* with Tim Ottinger. I also contributed a couple chapters to Uncle Bob's *Clean Code: A Handbook of Agile Software Craftsmanship [Mar08]*. I've written more than a hundred articles published at sites other than mine. I write regularly for my own blog (at http://langrsoft.com/jeff) and have written or contributed to more than a hundred blog entries for the Agile in a Flash project at http://agileinaflash.com.

In addition to C++, I've programmed in several other languages extensively: Java, Smalltalk, C, C#, and Pascal, plus one other that shall remain unmentioned. I'm currently learning Erlang and can code enough Python and Ruby to survive. I've played with at least another dozen or so languages to see what they were like (or to support some short-lived effort).

About the C++ Style in This Book

While I have extensive experience on C++ systems of all sizes, ranging from small to extremely large, I don't consider myself a language expert. I've read the important books by Meyers and Sutter, plus a few more. I know how to make C++ work for me and how to make the resulting code expressive and maintainable. I'm aware of most of the esoteric corners of the language but purposefully avoid solutions requiring them. My definition for *clever* in the context of this book is "difficult to maintain." I'll steer you in a better direction.

My C++ style is very object-oriented (no doubt because of a lot of programming in Smalltalk, Java, and C#). I prefer that most code ends up scoped to a class. Most of the examples in this book fall in line with this style. For example, the Soundex code from the first example (see *Test-Driven Development: A First Example*) gets built as a class, but it doesn't need to be. I like it that way, but if it wads yer underwear, do it your way.

TDD can provide value regardless of your C++ style, so don't let my style turn you off to its potential. However, a heavier OO emphasis makes introducing test doubles (see *Test Doubles*) easier when you must break problematic dependencies. If you immerse yourself in TDD, you'll likely find that your style shifts more in this direction over time. It's not a bad thing!

I'm a little lazy. Given the relatively small scope of the examples, I chose to minimize the use of namespaces, though I would certainly incorporate them on any real production code effort.

I also prefer to keep my code as streamlined as possible and thus avoid what I sometimes view as visual clutter. In most implementation files, you'll find `using namespace std;` for this reason, although many consider that bad form. (Keeping your classes and functions small and focused makes this and other guidelines such as "All functions should have only one return" less useful.) No worries; TDD won't prevent you from sticking to your own standards, and neither will I.

A final word on C++: it's a big language. I'm certain there are better ways to code some of the examples in the book, and I would bet that there are library constructs I'm not taking advantage of. The beauty of TDD is that you'll be

able to rework an implementation a dozen different ways without fear of breaking something. Regardless, please send me your suggestions for improvement, but only if you're willing to test-drive them!

Acknowledgments

Thanks to my editor, Michael Swaine, and the great folks at PragProg for the guidance and resources needed to create this book.

Thanks, Uncle Bob, for the exuberant foreword!

Many thanks to Dale Stewart, my technical editor, for providing valuable assistance throughout the process, particularly feedback and help on the C++ code throughout the book.

I always ask for brutally honest feedback during the writing process, and Bas Vodde provided exactly that, supplying me with voluminous feedback on the entire book. He was the invisible pair partner I needed to keep the conversation honest.

Special thanks to Joe Miller, who painstakingly converted most of the examples so that they will build and run on Windows.

Many thanks to all the other folks who provided ideas or invaluable feedback: Steve Andrews, Kevin Brothaler, Marshall Clow, Chris Freeman, George Dinwiddie, James Grenning, Michael Hill, Jeff Hoffman, Ron Jeffries, Neil Johnson, Chisun Joung, Dale Keener, Bob Koss, Robert C. Martin, Paul Nelson, Ken Oden, Tim Ottinger, Dave Rooney, Tan Yeong Sheng, Peter Sommerlad, and Zhanyong Wan. My apologies if I missed anyone.

Thank you to those who supplied feedback on the PragProg errata page: Bradford Baker, Jim Barnett, Travis Beatty, Kevin Brown, Brett DiFrischia, Jared Grubb, David Pol, Bo Rydberg, Jon Seidel, Marton Suranyi, Curtis Zimmerman, and many others.

Thanks again to Tim Ottinger, who supplied some of the words in the introduction plus a few ideas for the book. I missed having you as a co-conspirator!

Thank you all for helping make this book better than I could ever hope to make it on my own!

Dedication

This book is dedicated to those who continue to support me in doing what I love, particularly my wife, Kathy.