Extracted from:

Your Code as a Crime Scene

Use Forensic Techniques to Arrest Defects, Bottlenecks, and Bad Design in Your Programs

The Pragmatic Bookshelf

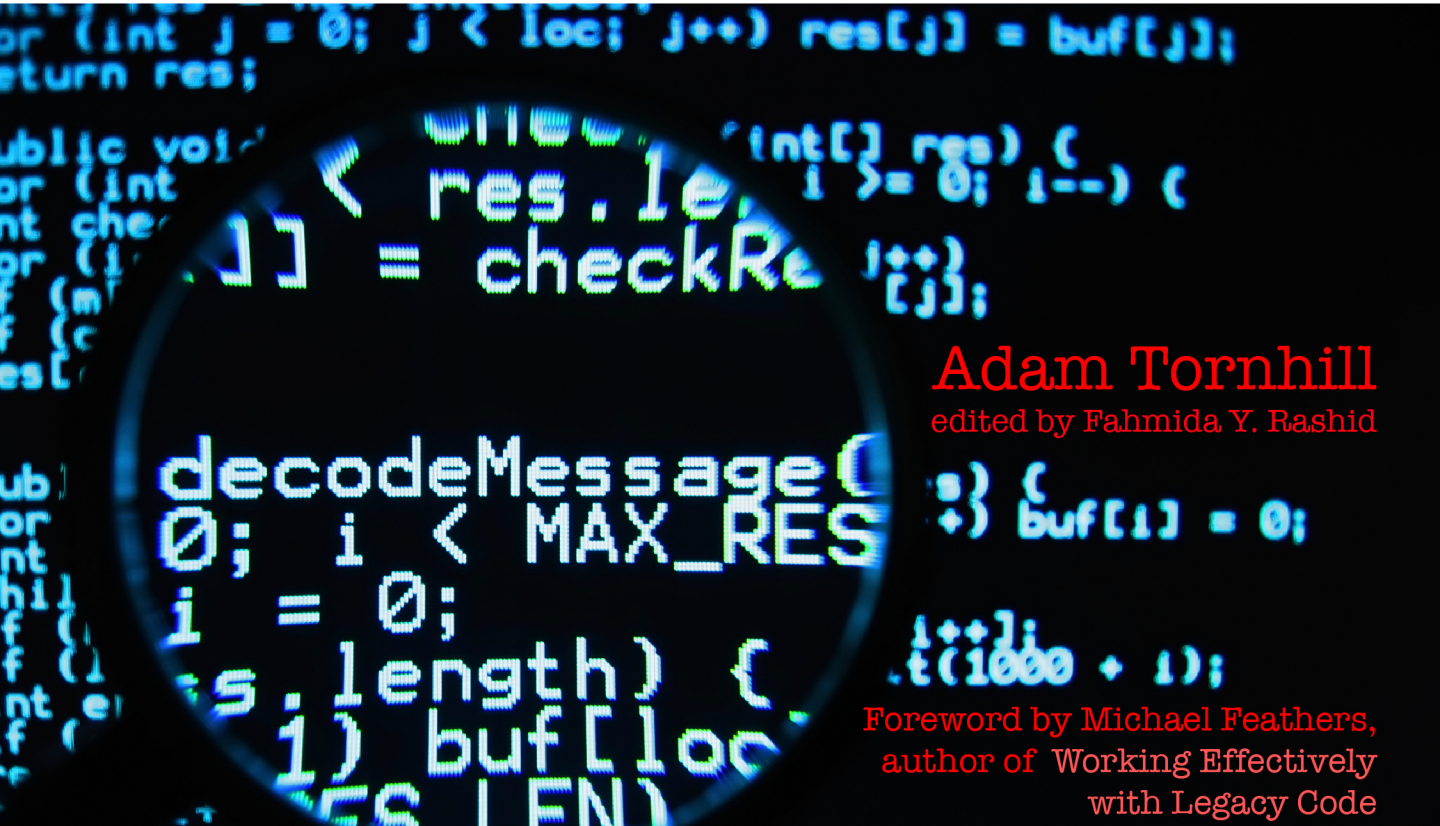
Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Investigator:
Date:
Case #:
Location:

Your Code As a Crime Scene

Use Forensic Techniques to Arrest Defects, Bottlenecks, and Bad Design in Your Programs



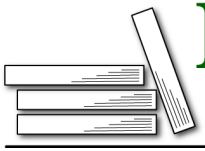
Your Code as a Crime Scene

Use Forensic Techniques to Arrest Defects, Bottlenecks, and Bad Design in Your Programs

Adam Tornhill

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 *https://pragprog.com*.

The team that produced this book includes:

Fahmida Y. Rashid (editor)
Potomac Indexing, LLC (indexer)
Cathleen Small (copyeditor)
Dave Thomas (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-038-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—March 2015

The central idea of *Your Code as a Crime Scene* is that we'll never be able to understand complex, large-scale systems just by looking at a single snapshot of the code. As you'll see, when we limit ourselves to what's visible in the code, we miss a lot of valuable information. Instead we need to understand both how the system came to be and how the people working on it interact with each other. In this book, you'll learn to mine that information from the evolution of your codebase.

Once you've worked through this book, you'll be able to examine a large system and immediately get a view of its health—that is, its health from both a technical perspective and from the development practices that led to the code you see today. You'll also be able to track the improvements made to the code and gather objective data on them.

About This Book

There are plenty of good books on software design for programmers. So why read another one? Well, unlike other books, *Your Code as a Crime Scene* focuses on your codebase. This book will help you identify potential problems in your code, find ways to improve it, and get rid of productivity bottlenecks.

Your Code as a Crime Scene blends forensic psychology and software evolution. Yes, it is a technical book, but programming isn't just about lines of code. We also need to focus on the psychological aspects of software development.

But *forensics*—isn't that about finding criminals? It sure is, but you'll also see that criminal investigators ask many of the same open-ended questions programmers ask while working through a codebase. By applying forensics concepts to software development, we gain valuable insights. And in our case, the offender is problematic code that we need to improve.

As you read along, you'll:

- Predict which sections of code have the most defects and the steepest learning curves.
- Use software evolution to find the code segment that matters most for maintenance.
- Understand how multiple developers and teams influence code quality.
- Learn how to track organizational problems in your code and get tips on how to fix them.
- Get a psychological perspective on your programs and learn how to make them easier to understand.

Who Should Read This Book?

This book is written for programmers, software architects, and technical leads. The techniques in the book are useful for both small and large systems. On small systems, you'll get new insights into your design and how well the actual code reflects your ideas. On large projects, you'll learn to find the code that matters most for your productivity and save maintenance costs, and you'll learn how to track down organizational problems in your codebase.

It doesn't matter what language you program in, as long as you are comfortable with the command prompt. The case studies in the book use Clojure, Java, and C#, but you don't need to know any of these languages to be able to follow along. Our discussions will focus on design principles, which are language-independent.

We'll also interact a lot with version-control systems. To get the most out of the book, you should know how to work with Subversion, Git, Mercurial, or a similar tool.

The strategies you'll learn will be useful regardless of the size of your codebase. But the more complex your codebase is, the more you'll need this book.

This book covers both technical and social issues in large-scale projects. If you're in a leadership position, use the strategies to maintain a high-level view of your system and development progress.

Optimize for Understanding

Most software development books focus on writing code. After all, that's what we programmers do: write code.

I thought that was our main job until I read *Facts and Fallacies of Software Engineering [Gla92]*. Its author, Robert Glass, convincingly argues that maintenance is the most important phase in the software development lifecycle. Somewhere between 40 and 80 percent of a typical project's total costs go toward maintenance. What do we get for all this money? Glass estimates that close to 60 percent of the changes are genuine enhancements, not just bug fixes.

These enhancements come about because we have a better understanding of the final product. Users spot areas that can be improved and make feature requests. Programmers make changes based on the feedback and modify the code to make it better. Software development is a learning activity, and maintenance reflects what we've learned about the project thus far.

Maintenance is expensive, but it isn't necessarily a problem. It can be a good sign, because only successful applications are maintained. The trick is to make maintenance effective. To do that, we need to know where we spend our time.

It turns out that understanding the existing product is the dominant maintenance activity (see *Facts and Fallacies of Software Engineering [Gla92]*). Once we know what we need to change, the modification itself may well be trivial. But the road to that enlightenment is often painful.

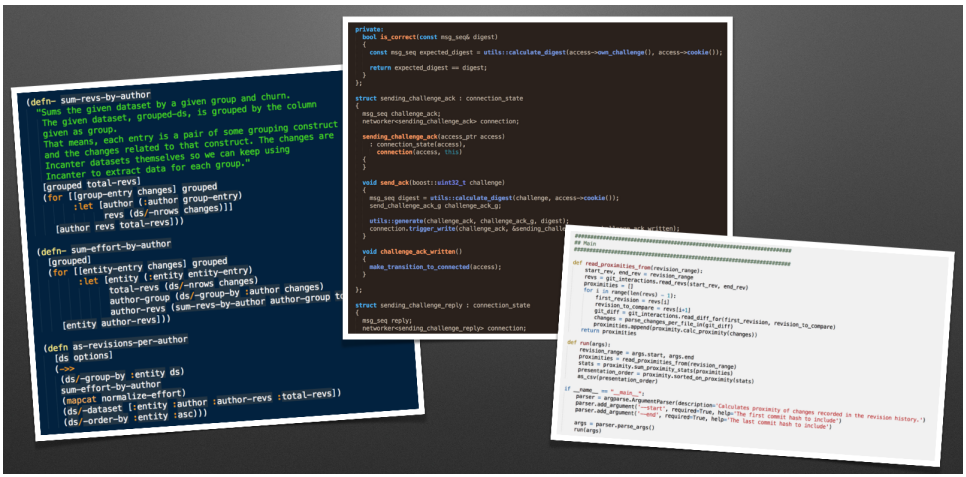
This means our primary task as programmers isn't to write code, but to understand it. The code we have to understand may have been written by our younger selves or by someone else. Either way, it's a challenging task.

This is just as important in today's Agile environments. With Agile, we enter maintenance mode immediately after the first iteration, because we modify existing code in later iterations. We spend the rest of the project in the most expensive phase of the development lifecycle. Let's ensure that it's time well-invested.

Know the Enemy of Change

To stay productive over time, we need to keep our programs' complexity in check. The human brain may be the most complex object in the known universe, but even our brain has limitations. As we program, we run into those limitations all the time. Our brain was never designed to deal with walls of conditional logic nested in explicit loops or to easily parse asynchronous events with implicit dependencies. Yet we face such challenges every day.

We can always write more tests, try to refactor, or even fire up a debugger to help us understand complex code constructs. As the system scales up, everything gets harder. Dealing with over-complicated architectures, inconsistent solutions, and changes that break seemingly unrelated features can kill both our productivity and our joy in programming. The code alone doesn't tell the whole story of a software system.



We need all the supporting techniques and strategies we can get. This book is here to provide that support.

How to Read This Book

This book is meant to be read from start to finish. Later parts build on techniques that I introduce gradually over the course of several chapters. Let's look at the big picture.

Part I Shows How You Detect Problematic Code

In Part I, you'll learn techniques to identify complex code that you also need to work with often. No matter how much we enjoy our work, when it comes to commercial products, time and money always matter. In this part, you'll learn methods to identify and prioritize the changes to the code that give you the most value.

We'll build the techniques on forensic methods used to predict and track down serial offenders. You'll see that each crime forms part of a larger pattern. Similarly, each change we make to our software leaves a trace. Each such trace is a clue to understanding the system we're building.

These modification patterns let you look beyond the current structure of the code to find out where it came from and how it evolved. By mining commit data and analyzing the history of your code, you learn to predict the code's future. This will allow you to start the fixes ahead of time.

Part II Shows How You Can Improve Your Architecture

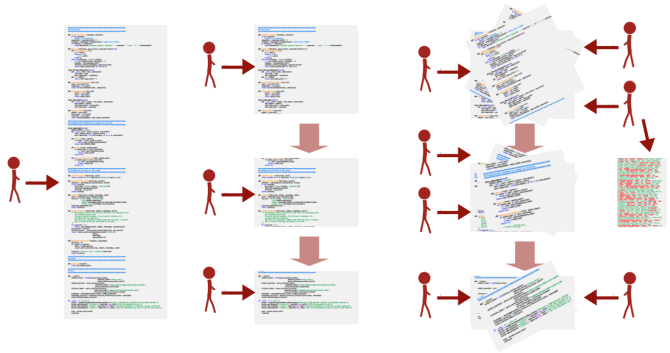
Once you know how to identify offending code in your system, you'll want to look at the bigger picture. You'll want to ensure that the high-level design of your system supports the evolution of your code.

In this part, we'll take inspiration from eyewitness testimonies to see how memory biases can frame both innocent bystanders and code. You'll then learn techniques to reduce memory biases and see how you can interview your own codebase. Your reward is information that you cannot deduce from the code alone.

After you've finished Part II, you'll know how to evaluate your software architecture against the modifications you make to your code. You'll also have techniques that let you identify signs of structural decay and expensive duplications of knowledge. In addition, you'll see how they provide you with refactoring directions and suggest new modular boundaries in your design.

Part III Shows How Your Organization Affects Your Code

Today's large software systems are developed by multiple teams. That intersection between people and code is an often overlooked aspect of software development. When there's a misalignment between how you're organized versus the work style your software architecture supports, code quality and communication suffers. As a result, we wind up with tricky workarounds and compromises to the design.



In Part III, you'll learn techniques to identify organizational problems that show up in your code. You'll see how to predict bugs from the way we work, understand how social biases influence software development, and uncover the distribution of knowledge among developers. As a bonus, you'll learn about group decisions, communication, false serial killers, and how they all relate to software development.

Because we base these techniques on version-control data as well, the methods are aligned with how you really work, instead of to any formal organizational chart. As you'll see, those two views often differ.

Toward a New Approach

Over the past decade, there's been some fascinating research on software evolution. Like most ideas and studies from academia, these findings have not crossed over into the industry. This book bridges that gap by translating academic research into examples for the practicing programmer.

You may be wondering how the strategies we cover in this book will relate to other software development practices. Let's sort that out so that you know how your new skills will complement your existing ones.

- *Tests*: The techniques you are about to learn let you identify the parts of your code most likely to contain defects. But they won't find the errors themselves. You still need to be testing the code. If you invest in automated tests, this book will give you tools to monitor the quality and evolution of those tests.
- *Static analysis*: Static analysis is a powerful technique for finding errors and dangerous coding constructs. Static analysis focuses on the impact your code has on the machine. In this book, we'll focus on how we humans look at the meaning of our code. Your code has to serve both audiences—machines and humans—so the techniques in this book complement static analysis rather than replace it.
- *Complexity metrics*: Complexity metrics have been around since the 1970s, but they're pretty bad at, well, spotting complexity. Metrics are language-specific, which means we cannot analyze a polyglot codebase. Another limitation of metrics is that they erase social information about how the code was developed. Instead of erasing that information, we'll learn to derive value from it. We'll complement metrics with more data.
- *Code reviews*: A manual process that is expensive to replicate, code reviews still have their place in software development. Done right, they're useful for both bug-hunting and knowledge-sharing. The techniques you'll learn in this book will help you prioritize the code you need to review.

As you see, the techniques you'll learn complement existing practices, rather than replacing them. I often use these techniques to identify parts of the code in need of manual inspection and review—or, as you'll see soon, to communicate with testers and other developers about the codebase.

Software Development is More Than a Technical Problem

In a surprisingly short time, we've moved from lighting fires in our caves to reasoning about multicores and CPU caches in cubicles. Yet we handle modern technology with the same biological tools as our prehistoric ancestors used for basic survival. That's why taming complexity in software has to start with how we think. Programming needs to be aligned with the way our brain works.

In this book, we'll take several opportunities to move beyond pure technical material. You'll learn why beauty is a fundamental quality of all good code, how individuals can bias group decisions, and how coding to music affects your problem-solving abilities.