The Pragmatic Programmers

Your Code as a Crime Scene

Second Edition

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

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

Welcome to the Crime Scene

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 and the code. In this book, you'll learn to mine that information from the evolution of your codebase.

Once you have worked through this book, you will be able to examine any system and immediately get a view of its health—both from 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.

Why You Should Read This Book

There are plenty of good books on software design and programming. So why read another one? Well, unlike other books, *Your Code as a Crime Scene* focuses on *your* codebase. This immediately helps you identify potential problems, find ways to fix them, and remove productivity bottlenecks one by one.

Your Code as a Crime Scene blends forensics and psychology with 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 forensic concepts to software development, we gain valuable insights. And in our case, the offender is problematic code that we need to improve.

As you progress through the book, you'll

- predict which sections of code have the most defects and the steepest learning curves;
- use behavioral code analysis to identify, prioritize, and remediate technical debt and maintenance issues;
- 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; and
- get a psychological perspective on your programs and learn how to make them easier to understand.

Who Should Read This Book?

To get the most out of this book, you're probably a programmer, software architect, or technical leader. Perhaps you're looking for effective ways to uncover the secrets of an existing codebase. Or, you might be embarking on a legacy migration project and looking for guidance. You might also strive to reduce defects, helping both yourself and your team to succeed. Maybe you're under pressure to deliver more code faster and want to figure out how to strike a balance between adding new features vs. improving existing code. No matter the scenario, you care about good code. Great—you're reading the right book.

It doesn't matter what language you program in. Our case studies mix Java, Go, JavaScript, Python, C++, Clojure, C#, and several other languages. However, the big advantage of crime-scene techniques is that you don't have to know any of these languages to follow along. All techniques are language-independent and will work no matter what technology you use. We also make sure to focus the discussions on principles rather than specifics.

The hands-on examples interact with version-control systems. To get the most out of the book, you should know the basics of Git, Subversion, Mercurial, or a similar tool.

Why Read This Book Now?

Never before has there been a larger shortage of software developers. Sure, the occasional economic downturn might slow things down, but at a macro level, this gap in supply and demand will continue to widen as society becomes more digitalized. Superficially, this might sound like good news for us: being

in demand does wonders for salaries. However, combine this shortage with an ever-increasing pressure to deliver on shorter cycles, and we can easily find ourselves mired in stress, unsustainable workloads, and software death marches.

A significant part of the problem is technical debt. The average software company wastes a large portion of developers' time dealing with the consequences of technical debt, bad code, and inadequate software architectures. It doesn't have to be that way.

Yet, technical debt is only part of the equation. A high staff turnover—which we've always had in the IT industry—means that companies continuously lose collective knowledge of their codebases. Unless we take measures to prevent it, our codebases end up as *terra nullius*—land belonging to nobody. This means it's more important than ever to pay attention to bad code so we can mitigate any offboarding impact. We also must be able to quickly orient ourselves in an unfamiliar codebase. Statistically, chances are we'll encounter this situation frequently.

Ultimately, it's all about freeing our time for more rewarding work to innovate interesting features and cool product ideas. Burning the midnight oil the day before a release, looking for that multithreaded bug in a file with 15,000 lines of opaque C++ code written by someone who quit last month after shunning documentation for years, is a miserable experience. Programming is supposed to be fun, and this book is here to help you reclaim that ideal.

How to Read This Book

This book is meant to be read from start to finish. Later parts build on techniques that you'll learn gradually over the course of several chapters. Let's look at the big picture so you know what lies ahead.

Part I: You'll Learn to Detect Problematic Code

You'll start by learning techniques for identifying complex code that you must work with often. No matter how much we enjoy our work, when it comes to commercial products, time and money always matter. That's why you'll explore methods for prioritizing refactoring candidates that give you the most value.

You'll build techniques based on forensic methods used to track down serial offenders. You'll see that each crime forms part of a larger pattern. Similarly, each change you make to your software leaves a trace. Analyzing those traces offers deep clues for understanding the system you're building. Analyzing the

history of your code also empowers you to predict the code's future. This helps you start making fixes ahead of time.

Part II: You'll Learn to Improve Software Architectures

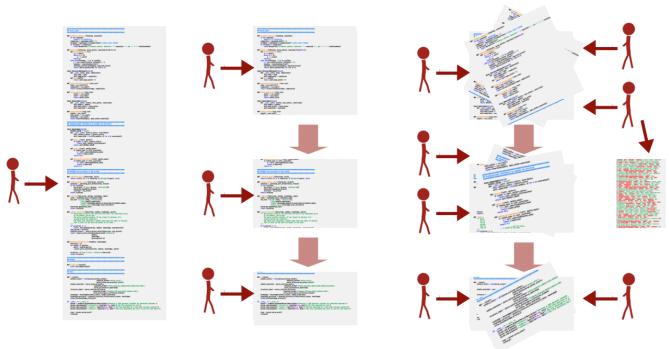
Once you know how to identify offending code in your system, you'll want to look at the bigger picture. That way, you can ensure that the high-level design of your system supports the features you implement and the way the codebase evolves.

Here, you'll take inspiration from eyewitness testimony to see how memory biases can frame both innocent bystanders and code. You'll use similar techniques to reduce memory biases and even interview your own codebase. The 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 done to the code, looking for signs of structural decay and expensive duplication of knowledge. In addition, you'll understand how the same techniques provide you with refactoring directions and potential new modular boundaries, which support important use cases such as breaking up monoliths or surviving legacy modernization projects.

Part III: You'll Learn How Your Organization Affects the Code

The majority of today's 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 vs. the work style your software architecture supports, code quality and communication suffer. As a result, you wind up with tricky workarounds and compromises to the design.



In Part III, you'll get to identify organizational problems in your code. You'll see how to predict bugs from the way you 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.

What's New in the Second Edition?

The core techniques in *Your Code as a Crime Scene* have stood the test of time because they focus on human behavior—people are a fairly stable construct, after all.

If you've read the first edition, you'll recognize most sections in the book. However, you'll still want to read those chapters because the case studies have been modernized and the text expanded with new insights, research findings, and actionable advice. This second edition brings extensive new content, reflecting all the lessons from applying crime-scene techniques at scale for a decade.

In addition, there are several new chapters which expand on the original work:

- Chapter 6, Remediate Complicated Code, on page ? explores a cognitive perspective on code complexity, which lets you focus on the code smells that actually matter.
- Chapter 7, Communicate the Business Impact of Technical Debt, on page ? makes the business case for paying down technical debt and refactoring in general. That way, you get all the data you need, so you can have conversations with non-technical stakeholders around something as deeply technical as code quality.
- Chapter 14, See How Technical Problems Cause Organizational Issues, on page ? flips the software organization on its head. Getting the "people side" of software development wrong will wreck any project, but here you learn why the reverse is true as well: how your code is written impacts the people and the organization.

Code as a Crime Scene Is a Metaphor

The crime scene metaphor helps to remind you that software design has social implications. Good code directs human behavior: in any well-designed system, there's one obvious place to touch when modifying code. The book's crime scene name is also an homage to the forensic techniques that inspired the analyses. At some point, however, it's useful to leave the metaphor and dive deeper into our core domain—programming. Hence, you'll examine knowledge drawn from many other sources, such as cognitive psychology, group theory, and software research.

Toward a New Approach

In recent decades, there's been some fascinating research on software evolution. Like most ideas and studies from academia, these findings have failed to cross over into the industry. This book bridges that gap by translating academic research into examples for the practicing programmer. That way, you know that the recommendations have a solid basis and actually work, instead of being mere opinions or personal preferences.

But, even if we stand on the shoulders of academia, this book isn't an academic text. *Your Code as a Crime Scene* is very much a book for the industry practitioner. As such, you may be wondering how the new strategies in this book relate to other software development practices. Let's sort that out:

- *Test automation*—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 decide what to automate first and to monitor the maintainability of the resulting 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, you'll focus on the other audience: how humans deduce meaning and intent from code. Hence, the techniques in this book complement static analysis by providing information you cannot get from code alone.
- *Code metrics*—Code complexity metrics have been around since the 1970s, but they're pretty bad at, well, spotting complexity. Metrics are language-specific, which means they 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, you'll learn to derive value from it. Only then can you take on the big problems like technical debt remediation or making sure your software architecture supports the way your organization works as a team.
- *Code reviews*—Being a manual process that is expensive to replicate, code reviews still have their place. Done right, they're useful for both bug hunting and knowledge sharing. The techniques you'll learn in this book help you prioritize the code you need to review.

To sum it up, *Your Code as a Crime Scene* is here to improve and enhance existing practices rather than replace them.

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 our prehistoric ancestors used for basic survival. That's why taming complexity in software must start with how we think. Programming needs to be aligned with the way our brain works.

Real-World Case Studies

Throughout the book, you will apply the crime-scene techniques to real-world codebases: prioritize technical debt in React, visualize growing code complexity in Kubernetes, get cognitive exercise when refactoring a cohesion problem in Hibernate, discover troublesome dependencies in Spring Boot, and determine the truck factor in a popular codebase from Facebook. And that's only part of it.

The case studies have been selected because they represent both popular applications and some of the best work we—as a software developer community—produce. This means that if the techniques you're about to learn can identify improvement opportunities in these codebases, chances are you'll be able to do the same in your own work.

Since these codebases are moving targets under constant development, we're going to use stable forks for this book. See the footnote[1] for all the relevant Git repositories.

Get Your Investigative Tools

To perform the behavioral code analyses, you obviously need to get your hands on behavioral data. That is, you need to trace how you and your team interact with the code. Fortunately, you're likely to already have all the data you need—although you might not be used to thinking about it as a data source. I'm referring to version control, which is a gold mine covering most of your needs.

To analyze version control, you need some tools to automate the mining and processing. When I wrote the first edition of this book, there weren't any tools available that could do the kind of analysis I wanted to share with you. So, I

1. https://github.com/code-as-a-crime-scene

had to write my own tools. The tool suite has evolved over the years and is capable of performing all the analyses in the book:

- *Code Maat*—Code Maat is a command-line tool used to mine and analyze data from version-control systems. It's completely free and open source, which means you can always dig in and inspect the details of various algorithms.
- *Git*—We focus our analysis on Git. However, you can still apply the techniques even if you use another version-control system, such as Perforce or Subversion. In that case, you'll need to perform a temporary migration to a read-only Git repository for the purpose of the analysis. The conversion is fully automated, as described in the excellent Pro Git book.[2]
- *Python*—The techniques don't depend on you knowing Python. We just include it here because Python is a convenient language for automating the occasional repetitive tasks. As such, the book will link to a Python script from time to time.

In addition, the case studies are available as interactive visualizations in the free community edition of *CodeScene*.[3] CodeScene is a SaaS tool, so you don't have to install it. Instead, we'll use it as an interactive gallery for our analyses. This saves you time as you can jump directly to the results instead of focusing on the mechanics of the analyses (unless you want to). And full disclosure: I work for CodeScene. I founded the company as a way of exploring the fascinating intersection of people and code. Hopefully, you'll find it useful, too.

Forget the Tools

Before you get to the installation of the tools, I want to mention that this book isn't about a particular tool, nor is it about version control. The tools are merely there for your convenience, allowing you to put the theories into practice. Instead, the crucial factor is you—when it comes to software design, there's no tool that replaces human expertise. What you'll learn goes beyond any tool.

Instead, the focus is on applying the techniques and interpreting and acting on the resulting data. That's the important part, and that's what we'll build on in the book.

2. https://git-scm.com/book/en/v2/Git-and-Other-Systems-Migrating-to-Git
3. https://codescene.com/

Install Your Tools

Code Maat comes packaged as an executable JAR file. You download the latest version from its release page on GitHub.[4] The GitHub README[5] contains detailed instructions. We'll cover the relevant options as we go along, but it's always good to have this information collected in one place.

You need a JVM such as OpenJDK to execute the Code Maat JAR. After installing that Java environment, make sure it functions properly by invoking it:

```
prompt> java -version
openjdk version "18.0.2"
```

After that, you're ready to launch Code Maat from the command line:

```
prompt> java -jar code-maat-1.0.4-standalone.jar
```

If everything was successfully installed, then the previous command will print out its usage description. Note that the version of Code Maat is likely to be different now—just grab the latest version.

To avoid excess typing, I recommend creating an alias for the command. Here's how it looks in a Bash shell:

```
prompt> alias maat='java -jar /adam/tools/code-maat-1.0.4-standalone.jar'
prompt> maat # now a valid shortcut for the full command
```

Use Git BASH on Windows

You can run Git in a DOS prompt on Windows. But, some of our commands will use special characters, such as backticks. Those characters have a different meaning in DOS. The simplest solution is to interact with Git through its *Git BASH shell* that emulates a Linux environment. The Git BASH shell is distributed together with Git itself.

This book also has its own web page.[6] Check it out—you'll find the book forum, where you can talk with other readers and with me. If you find any mistakes, please report them on the errata page.

4. https://github.com/adamtornhill/code-maat/releases
5. https://github.com/adamtornhill/code-maat
6. https://pragprog.com/titles/atcrime/your-code-as-a-crime-scene/

Know What's Expected

The strategies and tooling work on Mac-, Windows-, and Linux-based operating systems. As long as you use a version-control system sensibly, you'll find value in what you're about to learn.

You'll run the tools and scripts from a command prompt. That way, you'll truly understand the techniques and be able to extend and adapt them for your unique environment. Don't worry—I'll walk you through the commands.

As a convention, we'll use `prompt>` to indicate an operating system-independent, generic prompt. Whenever you see `prompt>`, mentally replace it with the prompt for the command line you're using. We'll also use the `maat` alias introduced in the previous section as a shorthand for the full command. So, now is a good time to add that alias. Optionally, whenever you see `maat` on a prompt, you'll need to type the actual command, for example, `java -jar code-maat-1.0.4-standalone.jar`.

Tools will come and go; details will change. The intent here is to go deeper and focus on timeless aspects of large-scale software development. (Yes, timeless sounds pretentious. It's because the techniques are about people and how we function—we humans change at a much more leisurely rate than the technology surrounding us.)

With that said, let's get started on the challenges of building software at scale.