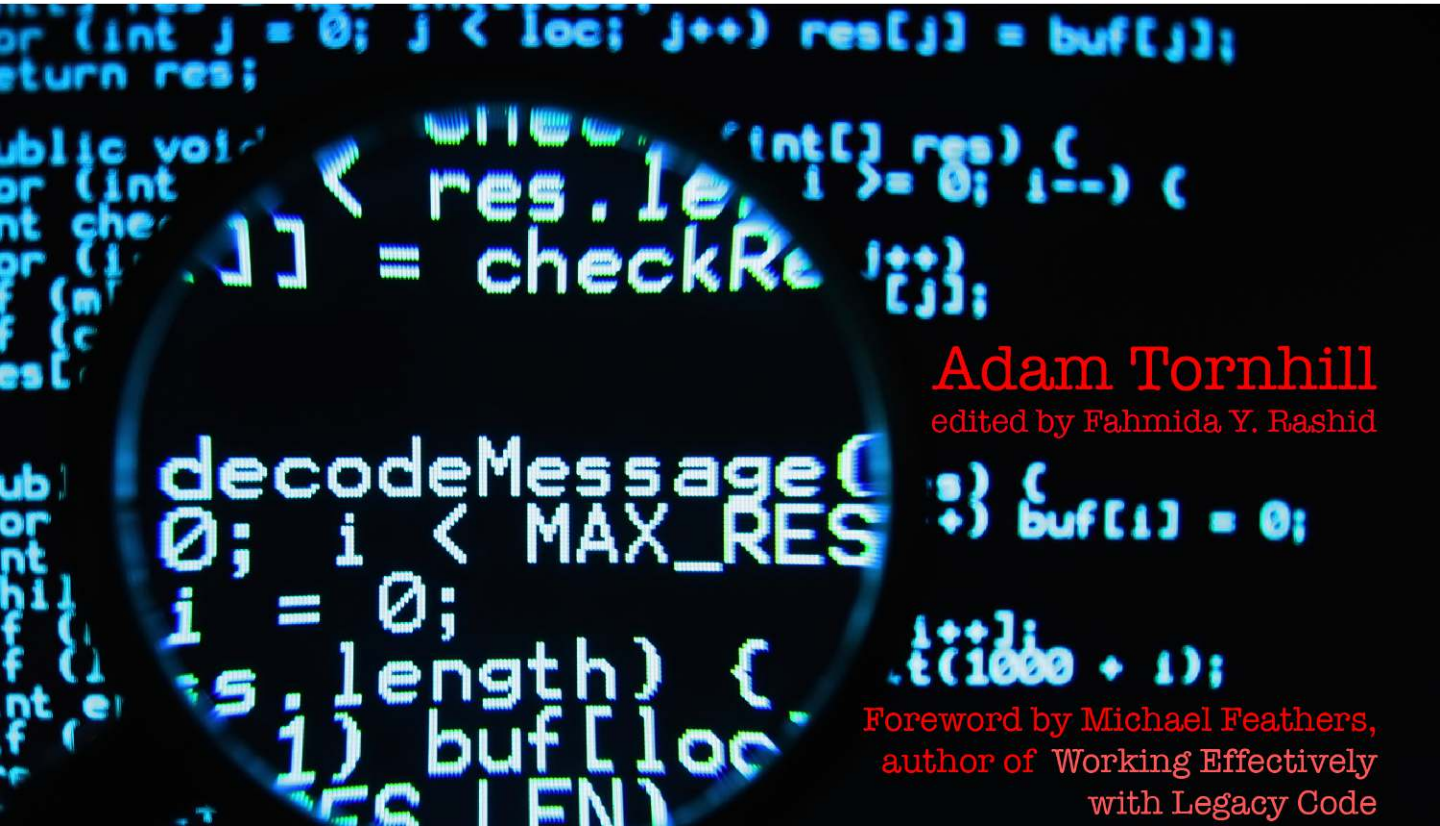
The Pragmatic Programmers

Investigator:
Date:
Case #:
Location:

Your Code As a Crime Scene

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



Early praise for Your Code as a Crime Scene

This book casts a surprising light on an unexpected place—my own code. I feel like I've found a secret treasure chest of completely unexpected methods. Useful for programmers, the book provides a powerful tool to smart testers, too.

➤ **James Bach**
Author, *Lessons Learned in Software Testing*

You think you know your code. After all, you and your fellow programmers have been sweating over it for years now. Adam Tornhill uses thinking in psychology together with hands-on tools to show you the bad parts. This book is a red pill. Are you ready?

➤ **Björn Granvik**
Competence manager

Adam Tornhill presents code as it exists in the real world—tightly coupled, unwieldy, and full of danger zones even when past developers had the best of intentions. His forensic techniques for analyzing and improving both the technical and the social aspects of a code base are a godsend for developers working with legacy systems. I found this book extremely useful to my own work and highly recommend it!

➤ **Nell Shamrell-Harrington**
Lead developer, PhishMe

By enlisting simple heuristics and data from everyday tools, Adam shows you how to fight bad code and its cohorts—all interleaved with intriguing anecdotes from forensic psychology. Highly recommended!

➤ **Jonas Lundberg**
Senior programmer and team leader, Netset AB

After reading this book, you will never look at code in the same way again!

➤ **Patrik Falk**
Agile developer and coach

Do you have a large pile of code, and mysterious and unpleasant bugs seem to appear out of nothing? This book lets you profile and find out the weak areas in your code and how to fight them. This is the essence of combining business with pleasure!

➤ **Jimmy Rosenskog**
Senior consultant, software developer

Adam manages to marry source code analysis with forensic psychology in a unique and structured way. The book presents tools and techniques to apply this concept to your own projects to predict the future of your codebase. Brilliant!

➤ **Mattias Larsson**
Senior software consultant, Webstep

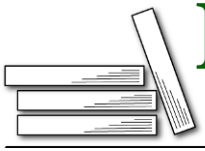
Your Code as a Crime Scene

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

Contents

Part II — Dissect Your Architecture

Part III — Master the Social Aspects of Code

Foreword by Michael Feathers

It's easy to look at code and think that there is nothing more than what we see. When we look at it, we see operators, identifiers, and other language structure, but that is all surface. We forget the depth. We spend so much time looking at the current state of the code that we forget its history and all of the forces that influenced it along its path toward the present.

We pay for this myopia. Many code changes are incredibly shortsighted, both in terms of our vision of what the code will be in the future, and the way that it got to be the way that it is.

Years ago, I was struck by the fact that we use version-control systems to keep track of our projects' histories, but we hardly ever revert to previous versions. Those versions exist as an insurance policy, and we're lucky when we never have to file a claim. It's easy, then, to look at that record of changes and see it as waste. Do we really need more than the last few versions? The naive answer is no, but we have a real opportunity when we enthusiastically say yes—we can mine our source code history to learn more about us, our environment, and how we work. That information is real power.

I think that we are at the beginning of a new era of awareness about how software changes. We're abandoning the static view of code and seeing it as a verb—a constantly changing medium that reacts to its immediate and extended environment. *Your Code as a Crime Scene* is the first book I've encountered that takes that view and runs with it. Adam presents tools, techniques, and insight that will change the way you develop software. You can't unread this information, and you will see software differently.

Dig in.

Michael Feathers

Acknowledgments

My writing experience has been fun, challenging, and rewarding. I owe that to all the amazing people who helped me out.

First of all, I'd like to thank the Pragmatic Bookshelf for this opportunity. In particular, I'd like to thank my editor, Fahmida Y. Rashid, who made this a much better book than what I could've done on my own. Thanks, Fahmida!

I'd also like to thank all my reviewers: John Cater, Nell Shamrell, Rod Hilton, Gunther Schmidl, Dan Shiovitz, Lief Eric Fredheim, Jeremy Frens, Kevin Beam, and Andy Lester. Thanks a lot for all your feedback and ideas!

Special thanks to Jonas Lundberg and James Bach for their deep insights and helpful suggestions.

As always, I could count on my great colleagues at Webstep to help out: Patrik Falk, Mattias Larsson, Jimmy Rosenskog, Björn Granvik, and Mikael Pahmp. Thanks for all your encouragement and technical expertise! I would also like to thank Martin Stenlund for being an amazing manager and a true leader.

I've always been a big fan of Michael Feathers' work. That's why I'm extra proud to include his foreword. Thanks, Michael—you're an inspiration!

The case studies in this book were possible due to skilled programmers who have made their code open source. So thanks to all contributors to Hibernate, Craft.Net, nopCommerce, and Scala. I have a lot of respect for your work.

My parents, Eva and Thorbjörn, have always supported me. You're the best—thanks for all you've done for me!

Finally, I'd like to thank my family for their endless support: Jenny, Morten, and Ebbe—thanks. I love you.

CHAPTER 1

Welcome!

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.

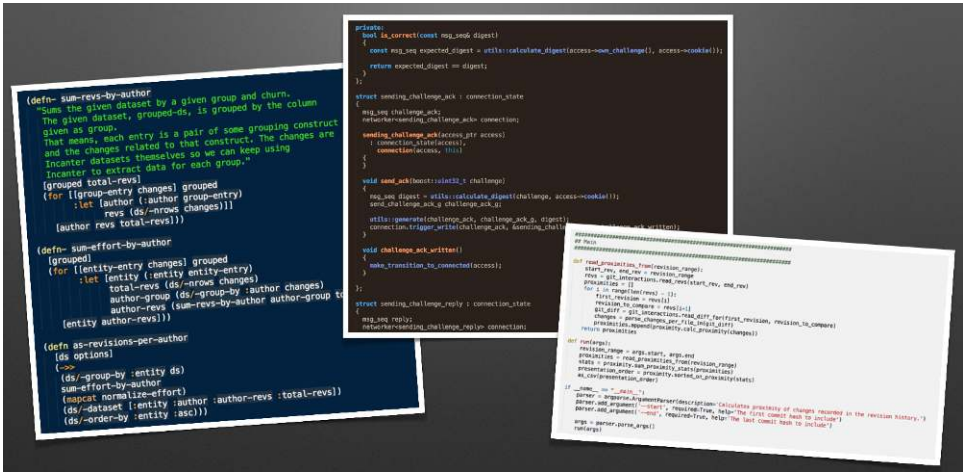
This means our primary task as programmers isn't to write code, but to understand it. The code we have to understand may have been written by our younger selves or by someone else. Either way, it's a challenging task.

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

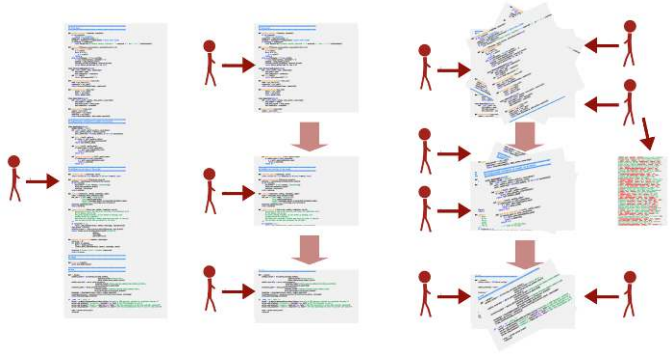
Once you know how to identify offending code in your system, you'll want to look at the bigger picture. You'll want to ensure that the high-level design of your system supports the evolution of your code.

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.

Get Your Investigative Tools

The techniques in this book are based on how you and your team interact with the code. Most of that information is stored within your version-control system. To analyze it, we need some automated tools to mine and process the data, but there aren't a lot of tools out there we can use.

To get you started, I've put together a suite of open-source tools capable of performing the analyses:

- *Code Maat*: Code Maat is a command-line tool used to mine and analyze data from version-control systems.
- *Git*: The techniques in this book would work with other types of version-control systems, but we'll use Git in our examples. You can refer to Code Maat's web page[1] to get an overview of mining data from Mercurial and Subversion.
- *Python*: The techniques don't depend on you knowing Python. We just include it here because Python is a convenient language for automating repetitive tasks.

1. https://github.com/adamtornhill/code-maat

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.

Forget the Tools

Before we get to the installation of the tools, I want to mention that even though we'll use Code Maat extensively, this book isn't about a particular tool. The tools here are prepared for your convenience as a way to put the theories into practice.

While Code Maat does help with the tasks ahead, the important factor here is you—when it comes to software design, there's no tool that replaces human expertise. What you'll learn goes beyond any tool; Code Maat just relieves you of repetitive calculations, input parsing, and result generation.

In each of the case studies you're about to read, the toolset is the least important part. The algorithms in Code Maat are fairly simple—you could implement them as plain scripts in a language of your choice.

That simplicity is a strength. It allows you to focus on the application of the techniques and how to interpret the resulting data. That's the important part, and that's what we'll build on in the book.

After you finish *Your Code as a Crime Scene,* you'll be in a position to move beyond Code Maat. There's even a closing chapter dedicated to that in Chapter 15, *Toward the Future*, on page 175.

Install Your Tools

You can get detailed setup instructions for all the tools we're using from the Code Maat distribution site.[2]

This book also has its own web page.[3] 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.

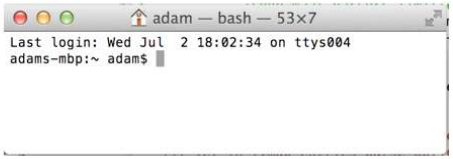
2. http://www.adamtornhill.com/code/crimescenetools.htm
3. https://pragprog.com/book/atcrime/code-as-a-crime-scene

Know What's Expected

I've applied the techniques in this book on a wide variety of programming languages, including Java, C#, Python, Clojure, C++, and Common Lisp. The techniques are language-independent and will work no matter what technology you use.

Similarly, I've used the strategies on both 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.

We'll run the tools and scripts from a command prompt. Sure, we *could* put a GUI on the tools to hide options and details. But I want you to truly understand the techniques we're discussing so you'll be able to extend and adapt them for your unique environment. Don't worry—I'll walk you through the commands. They're basic and will work on Windows, Mac, and Linux.



As a convention we'll use `prompt>` to indicate an operating system–independent, generic prompt. Whenever you see `prompt>`, replace it mentally with the prompt for the command line you're using.

The convention used to represent a prompt on your system

```
prompt> maat -h
```

You'll also see some Python, but we won't develop in it. We'll use Python to perform repetitive tasks so that we can focus on the fun stuff. If you haven't used Python before, it's fine; I'll provide the code you need and walk you through the algorithms so you can use them with a different language of your choice.

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.)

Let's get started with how forensic psychology helps you investigate your code.

Part I

Evolving Software

Let's start with the evolution of software designs. We'll discuss the consequences of scale and the challenges of legacy code. In this part, you'll learn novel techniques to assess and analyze your codebase.

By looking into the history of your system, you'll learn how to predict its future. This will allow you to act on time, should there be any signs of trouble.

CHAPTER 2

Code as a Crime Scene

We just discussed how software maintenance is both difficult and expensive. Our challenge is to reduce complexity and get code that is easy to modify. To succeed, we need to prioritize what improvements to make based on how we actually work with the system. That way, we get the most value in the least amount of time.

In this chapter, you'll learn novel strategies from forensic psychology that you can use to identify and prioritize software-design issues. It isn't enough to look at a static snapshot of the code; we also need to look at how the system evolved. We do that by treating version-control data as our evidence.

Along the way, you'll learn techniques to identify code that is hard to understand and tricky to modify. It's code that slows down your productivity and degrades the quality of your system. Knowing where the true problems are lets you make improvements where they are needed the most.

Meet the Problems of Scale

Think back to the last large project you worked on. If you could make any change to that codebase, what would it be? Since you spent a lot of time with the code, you can probably think quickly of several trouble spots. But do you know which of those changes would have the greatest impact on your productivity and make maintenance easier?

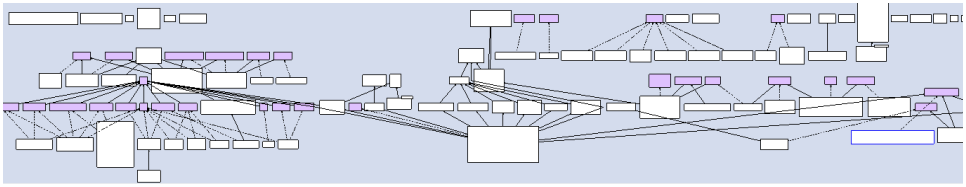
Your final choice has to balance several factors. Obviously, you want to simplify the tricky elements in the design. You may also want to address defect-prone modules. To get the most out of your redesign, you should improve the part of code you will most likely work with again in the future.

If your project is anything like typical large-scale projects, it will be hard to identify and prioritize those areas of improvement. Refactoring or redesigning can be risky, and even if you get it right, the actual payoff is uncertain.

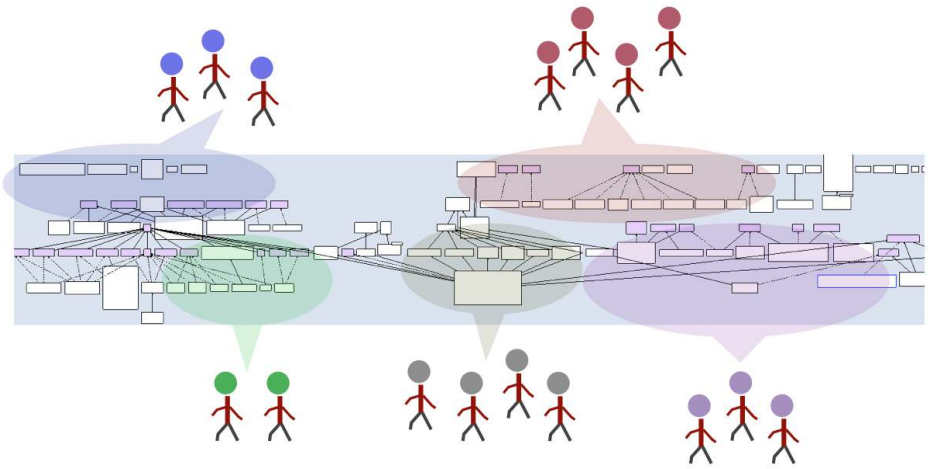
Like a hangover, it's a problem that gets worse the more you add to it.

Find the Code That Matters

Take a look at all the subsystem dependencies in the following figure. The system wasn't originally designed to be this complex, but this is where code frequently ends up. The true problems are hidden among all these interactions and aren't easy to find. Complexity obscures the parts that need attention.



We need help tackling these large-scale software applications. No matter how experienced you are, the human brain is not equipped to effectively step through a hundred thousand lines of entangled code. The problem also gets worse with the size of your development team, because everyone is working separately. Nobody has a holistic picture of the code, and you risk making biased decisions. In addition, all large-scale codebases have code that no one knows well or feels responsible for. Everyone gets a limited view of the system.



Decisions made based on incomplete information are troublesome. They may optimize one part of the code but push complexities and inconsistencies to

other areas maintained by other developers. You can address a part of this problem by rotating team members and sharing experiences. But you still need a way to aggregate the team's collective knowledge. We'll get there soon, but let's first discuss one of the traditional approaches—complexity metrics—and why they don't work on their own.

The Problem with Complexity Metrics

Complexity metrics are useful, but not in isolation. As we previously discussed in *Toward a New Approach*, on page 6, complexity metrics alone are not particularly good at identifying complexity. This also isn't an optimal approach.

Complexity is only a problem if we need to deal with it. If no one needs to read or modify a particular part of the code, does it really make a difference whether it's complex? Well, you may have a potential time bomb waiting to go off, but large-scale codebases are full of unwarranted complexity. It's unreasonable to address them all at once. Each improvement to a system is also a risk, as new problems and bugs may be introduced. We need strategies to identify and improve the parts that really matter. Let's uncover them by putting our forensic skills to work.

Get a Crash Course in Offender Profiling

You probably know a little bit about offender profiling already. Movies such as the 1990s hit *The Silence of the Lambs* popularized the profession, and that popularity exists even decades after the movie's theatrical release.

In *The Silence of the Lambs*, Dr. Hannibal Lecter, portrayed by Anthony Hopkins, is an imprisoned convicted killer. Throughout the movie, he is presented with details from different crime scenes. He takes the information and deduces the offender's personality and motivations for committing the crime. The police use his profile to identify and find the serial killer Buffalo Bill. (Sorry for the spoiler!) While impressive, Dr. Lecter's stunning forensics skills have a serious limitation: they only work in Hollywood movies.

Fortunately, we have scientifically valid profiling alternatives that extend to the real world. Read along, and I'll show you how the tools forensic psychologists use to attack these open-ended problems can also be useful for software developers.

Learn Geographical Profiling of Crimes

Geographical profiling has its scientific basis in statistics and environmental psychology. It's a complex subject with a fair share of controversies (just like programming!), but the basic principles are simple enough.

Criminals aren't that different from us: they go to work, visit restaurants and shops, and keep in touch with friends. They build mental maps of all the places they go. This is not specific to criminals—we all build mental maps. But an offender uses the mental map to decide where to commit a crime.

The locations where crimes occur are very rarely random—the geographical locations contain valuable information about the offender. Think about it for a moment: there must be an overlap in space and time between the offender and a victim, right?

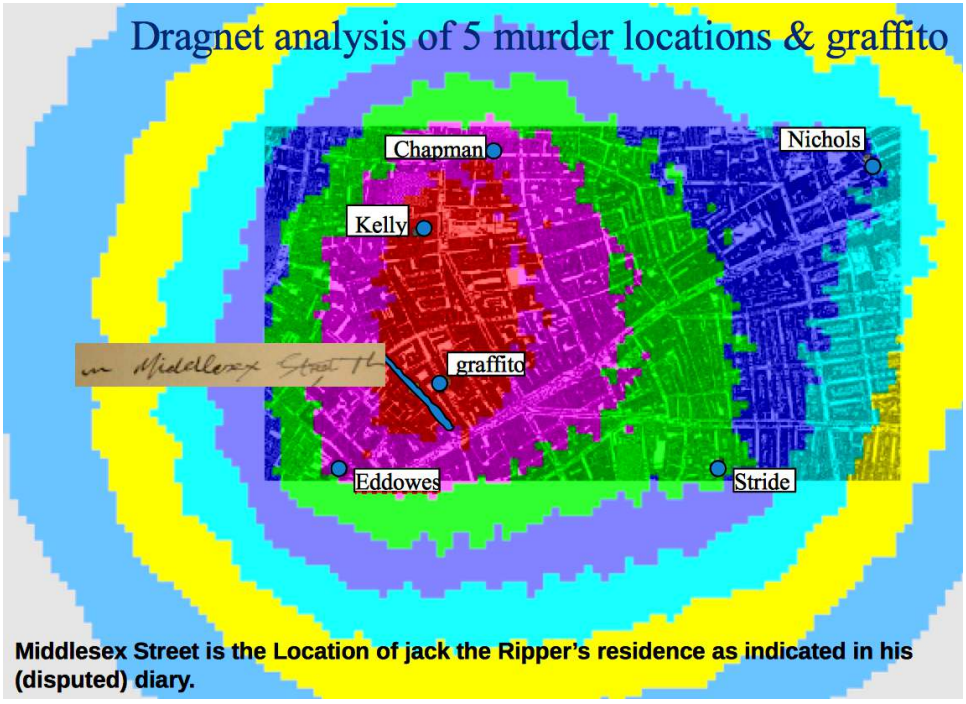
Once a crime has been committed, the offender realizes it would be too dangerous to return to that area. Typically, the location of the next crime is in the opposite direction from the first scene.

Over time, these crime scenes form a circle on a map. So while an offender's deeds may be bizarre, the rationale behind the processes is logical. (See *Principles of Geographical Offender Profiling [CY08a]* for an in-depth discussion.) We can look at the patterns and profile the person who committed the crimes. Let's take a look at how geographic profiling can help track down one of the most notorious serial killers in history, Jack the Ripper.

Profiling the Ripper

We can figure out where Jack the Ripper's home was with the following map. It was generated by Professor David Canter[1] using Dragnet, a software developed by The Center for Investigative Psychology.[2] Dragnet considers each crime location a center of gravity. It then combines the individual centers mathematically using one small twist; psychologically, all distances aren't equal. Thus, the crime locations are weighted depending on their relative distances. That weighted result points to the geographical area most likely to contain the home base of the offender, also known as our *hotspot*. The hotspot is indicated by the central red area on the map. The resulting hotspot is gold for investigators because they can focus their efforts on the smaller area instead of patrolling the entire city.

1. http://www.davidcanter.com/
2. http://www.i-psy.com/index.php



Middlesex Street is the Location of jack the Ripper's residence as indicated in his (disputed) diary.

Apply Geographical Offender Profiling to Code

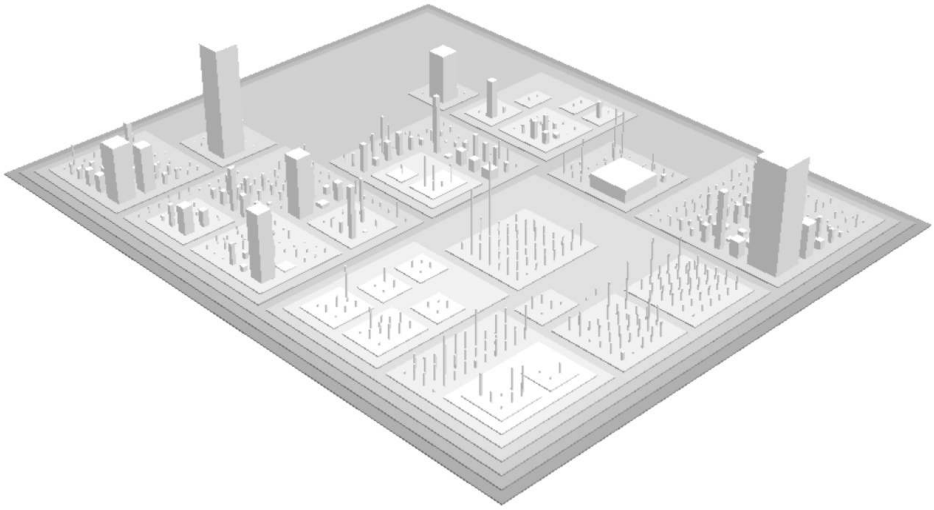
As I learned about geographical offender profiling in criminal psychology, I was struck by its possible applications to software. What if we could devise techniques that let us identify hotspots in large software systems? A hotspot analysis that could narrow down a large system to a few critical modules would be a big win in our profession.

Instead of speculating about potential design problems among million lines of code, geographical profiling would give us a prioritized lists of sections that need refactoring. It would also be dynamic, reflecting shifts in development focus over time.

Explore the Geography of Code

We need a geography of code. Despite its lack of physics, software is easy to visualize. My favorite tool is Code City.[3] It's fun to work with and matches the offender-profiling metaphor well. The following figure shows a sample city generated by the tool.

3. http://www.inf.usi.ch/phd/wettel/codecity.html



A city block represents a package, and each class is a building. The number of methods defines the height, and the number of attributes specifies the base of the building. Try out Code City, and you'll notice new patterns you didn't spot before in the code itself.

Code City is a nice starting point, but it limits us to looking at only object-oriented designs. Today's software world is increasingly polyglot. Even when you use the same language, you may have complex configurations in scripts, XML, and other markup formats. A geography must present a holistic picture, no matter what languages we choose. We'll soon explore other options, but before that we need to address a more serious limitation of our data.

Look at the large buildings in our city map again. If that information is all we have, those large buildings would be our hotspots. But there's nothing in the illustration to indicate on which building we should actually spend our efforts. Perhaps those large classes have been stable for years, are well-tested, and have little developer activity. It doesn't make sense to start there when other buildings may require immediate attention. In this case, the code doesn't tell the whole story.

Learn from the Spatial Movement of Programmers

Parts evolve at different rates in a codebase. As some modules stabilize, others become more fragile and volatile. When we profiled the Ripper, we used his spatial information to limit the search area. We pull off the same feat with code by focusing on areas with high developer activity.

Joe asks:
Who Was Jack?

Since Jack the Ripper was never caught, how do we know if the geographical offender profile is any good?

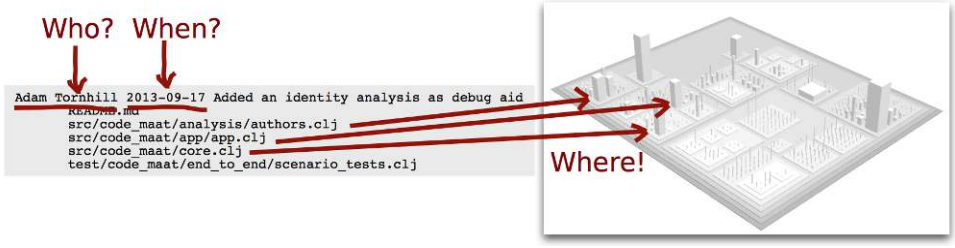
As of September 2014, there were reports of mitochondrial DNA evidence that presumably links one of the suspects, Aaron Kosminski, to a Jack the Ripper victim. There is a lot of controversy and debate around the claim, so let me introduce you to another likely suspect: James Maybrick.

In the early 1990s, a diary supposedly written by Liverpool cotton merchant James Maybrick surfaced. In this diary, Maybrick claimed to be the Ripper. Since its publication in *The Diary of Jack the Ripper [Har10]*, thousands of Ripperologists around the world have tried to expose the diary as a forgery using techniques such as handwriting analysis and chemical ink tests. No one has yet managed to prove the diary is fake, and its legitimacy is still under dispute.

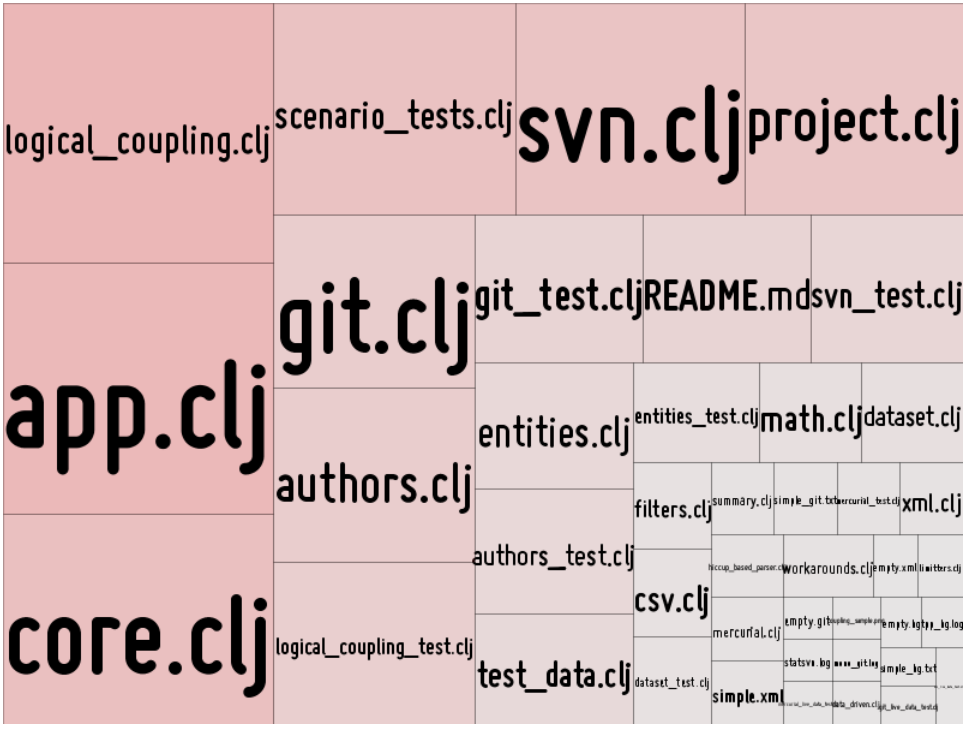
The interesting part about the diary for us is the fact that Maybrick wrote that he used to rent a room on Middlesex Street whenever he visited London. You can see Middlesex Street right inside our hotspot.

But what about Aaron Kosminiski's homebase? It, too, fits the profile, although not as well as Maybrick's does. Kosminski's probable home at the time of the murders is just a little bit east of the high-probability hotspot area.

Your development organization probably already applies tools that track your movements in code. Oh, no need to feel paranoid! It's not that bad—it's just that we rarely think about these tools this way. Their traditional purpose is something completely different. Yes, I'm talking about version-control systems.



The statistics from our version-control system can be an informational gold mine. Every modification to the system you've ever done is recorded, along with the related steps you took. It's more detailed than the geographical information you learned about in offender profiling. Let's see how version-control data enriches your understanding of the codebase and improves your map of the system. The following figure depicts the most basic version-control data using a tree-map algorithm.[4]



The size and color of each module is weighted based on how frequently it changes. The more recorded changes the module has, the larger its rectangle in the visualization. Volatile modules stand out and are easy to spot.

Measuring change frequencies is based on the idea that code that has changed in the past is likely to change again. Code changes for a reason. Perhaps the module has too many responsibilities or the feature area is poorly understood. We can identify the modules where the team has spent the most effort.

4. https://github.com/adamtornhill/MetricsTreeMap

Interpret Evolutionary Change Frequencies

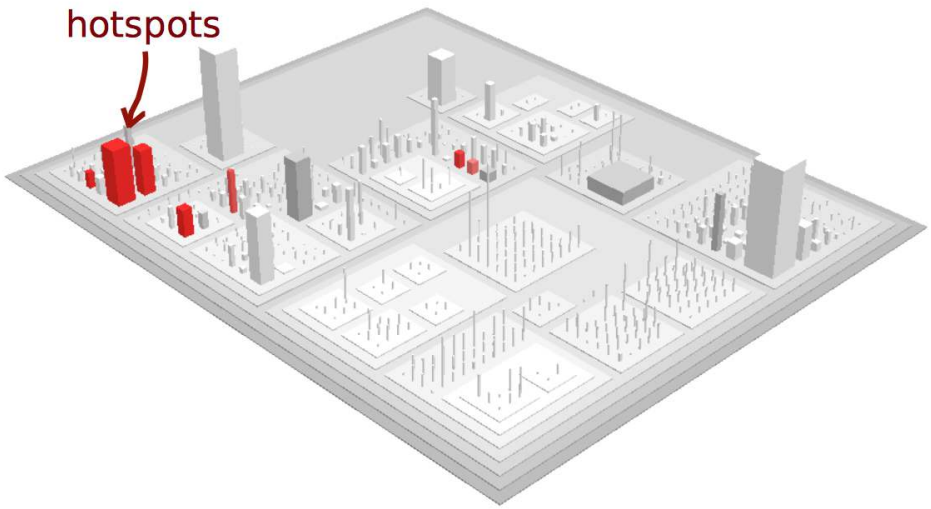
We are using change frequencies as a proxy for effort. Yes, it's a rough metric, but as you'll see soon it's a heuristic that works surprisingly well in practice.

In the earlier code visualization, we saw that most of the changes were in the logical_coupling.clj module, followed by app.clj. If those two modules turn out to be a mess of complicated code, redesigning them will have a significant impact on future work. After all, that's where we are currently spending most of our time.

While looking at effort is a step in the right direction, we need to also think about complexity. The temporal information is incomplete on its own because we don't know anything about the nature of the code. Sure, logical_coupling.clj changes often. But perhaps it is a perfectly structured, consistent, and clear solution. Or it may be a plain configuration file that we'd expect to change frequently anyway. Without information about the code itself, we don't know how important it is. Let's see how we can resolve that.

Find Hotspots by Merging Complexity and Effort

In the following illustration, we combine the two dimensions, complexity and effort. The interesting bit is in the overlap between them.



When put together, the overlap between complexity and effort signals a hotspot, an offender in code. Hotspots are your guide to improvements and refactorings. But there's more to them—hotspots are intimately tied to code quality, too. So before we move on, let's look at some research on the subject.

See That Hotspots Really Work

Hotspots represent complex parts of the codebase that have changed quickly. Research has shown that frequent changes to complex code generally indicate declining quality:

- After studying a long-lived system, a research team found that the number of times code changes is a better predictor of defects than pure size. (See *Predicting fault incidence using software change history [GKMS00]*.)
- In a study on code duplication, a subject we'll investigate in Chapter 8, *Detect Architectural Decay*, on page 77, modules that change frequently are linked to maintenance problems and low quality. (See *An Empirical Study on the Impact of Duplicate Code [HSSH12]*.)
- Several studies report a high overlap between these simple metrics and more complex measures. The importance of change to a module is so high that more elaborate metrics rarely provide any further predictive value. (See, for example, *Does Measuring Code Change Improve Fault Prediction? [BOW11]*.)
- Finally, a set of different predictor models was designed to detect quality issues in code. The number of lines of code and the modification status of modules turned out to be the two strongest individual predictors. (See *Where the bugs are [BOW04]*.)

When it comes to detecting quality problems, process metrics from version-control systems are far better than traditional code measurements.

Find Your Own Hotspots

In this chapter, you learned the theory behind hotspot analyses. You learned that to identify the parts that matter for your productivity, it isn't enough to look at a static snapshot of the code. Instead, you need to also look at where you spend most of your efforts.

We've now come full circle: we have a geography of code with a probability surface that lets you track down offending code. We'll explore this concept in more detail in the next chapter, and you'll learn to extract the information from version-control systems. It will be data that helps you find potential suspects, such as code smells and team-productivity bottlenecks.

Turn the page, and let's walk through your first offender profile.

CHAPTER 3

Creating an Offender Profile

We just learned how methods from forensic psychology let us find complex, buggy code and guide our refactoring efforts. Now it's time to put the theory into practice. Together, we're going to analyze a software system. In our case, the offender is problematic code, not criminals.

We'll start by mining evolutionary data from a version-control system. We'll then augment the change data with complexity estimates to identify hotspots. Along the way, we'll stop to look at existing research to see how the analysis helps us spot maintenance problems and quality issues.

When you finish this chapter, you'll be able to identify hotspots in your own code. As you'll see, hotspots point you to the parts of the code where you'll get the most from improvements in terms of time and money.

Finally, you'll get tips on how to use the resulting information to prioritize design problems, guide manual work, and communicate within a project. We'll also discuss the limitations of the hotspot concept.

Mining Evolutionary Data

Version-control data is our software-development equivalent to spatial movement in geographical profiling. A version-control system—such as Subversion, Git, or Mercurial—records the steps each developer took. I focus on Git in my examples, but you can find an overview of how to mine data from other systems in the Code Maat documentation.[1]

The first codebase we'll study is Code Maat. That's right—we'll use our analysis tool to analyze the tool itself. I'm a Lisp programmer—we love circular stuff like that.

1. https://github.com/adamtornhill/code-maat

To follow along with the examples, you need to clone the Code Maat repository[2] so that you have the complete source tree on your computer:

```
prompt> git clone https://github.com/adamtornhill/code-maat.git
Cloning into 'code-maat'...
remote: Reusing existing pack: 1092, done.
remote: Total 1092 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (1092/1092), 365.75 KiB | 271.00 KiB/s, done.
Resolving deltas: 100% (537/537), done.
Checking connectivity... done.
prompt>
```

Once the clone command completes, you'll find a local code-maat directory with the source code. Move into that directory:

```
prompt>cd code-maat
```

Turn Back Time

Code Maat is still under development, and some of the worst issues we spot here will probably be fixed by the time you read this. So we are going to pretend it's still 2013. That's fine—the digital world lets us easily travel back to less gracious times:

```
prompt> git checkout `git rev-list -n 1 --before="2013-11-01" master`
...
HEAD is now at d804759... Documented tree map visualizations
```

The git command is a bit tricky because it does two things: it fetches the revision on the specified date and checks out that revision. Other version-control systems provide similar rollback mechanisms.

Your local copy of Code Maat should now look as the code did back in 2013.

Investigate the Git Log

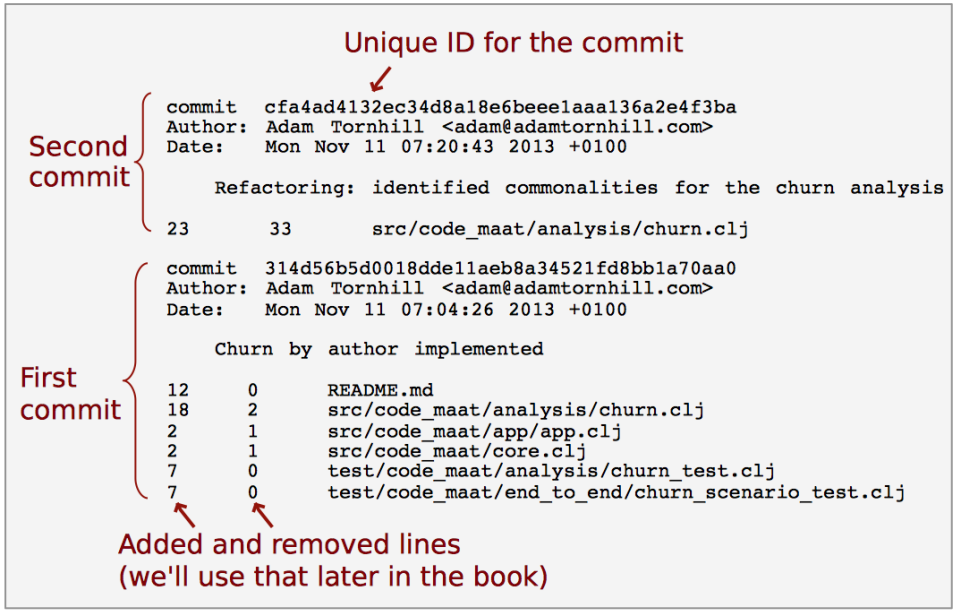
Git lets us inspect historical commits by its log command. To get the level of detail we need, we use the --numstat flag:

```
prompt> git log --numstat
```

This command will output a detailed log, as shown in the figure on page 25.

The sample output contains a lot of valuable information. In the following chapters, we'll get several opportunities to inspect it in depth. For now, we'll limit the analysis to the changed modules. We see that the oldest commit involved changes to six files, while the next one only modified churn.clj.

2. https://github.com/adamtornhill/code-maat



Verify Your Intuitions

Human intuition is wonderful for making quick decisions. The quality of those decisions, however, is not always wonderful.

Expert intuition can lead to high-quality decisions. The problem is that we don't know up front whether this time is one of those expert intuitions. Intuition is an automatic, unconscious process, and like all automatic mental processes, it's sensitive to cognitive and social biases. Factors in your surroundings or in the specific situation can influence your judgment. Most of the time you aren't aware of that influence. (You'll see some examples in Chapter 11, *Norms, Groups, and False Serial Killers*, on page 121.) For example, we may be notoriously bad at evaluating past decisions due to hindsight bias. That's why it's important to verify your intuitive ideas with supporting data.

From a practical perspective, we need guiding techniques like the ones presented here because intuition doesn't scale—especially not in complex areas, such as a large-scale software project that is constantly changing.

Automated Mining with Code Maat

In a large system under heavy development, hundreds of commits are made each day. Manually inspecting that data is error-prone and, more importantly, takes time away from all the fun programming. Let's automate this.

Calculating change frequencies is straightforward: parse the log file and summarize the number of times each module occurs. You could also add more complex processing to keep track of renamed or moved files.

You already know about Code Maat. Now we're going to use it to analyze change frequencies. The gitoutput on page 25 is fine for humans but too verbose for a tool. The following command generates a more compact version:

```
prompt> git log --pretty=format:'[%h] %an %ad %s' --date=short \
              --numstat --before=2013-11-01
```

Code Maat is strict about its input. (It doesn't have to be—it's just easier to write a parser if we can ignore special cases.) Here are the rules:

- Everything except --before is mandatory.
- Use the --before to get a reproducible, historical output in this example. Here we include all commits before that given date. It's our temporal period of interest for this analysis.
- If you want to analyze the complete evolution, just leave out the flag.
- Specify an optional start date through the --after flag.

As long as you keep the supported log format, you're free to vary and combine different filtering options.

To persist the log information, just redirect the git output to a file. For example:

```
prompt> git log --pretty=format:'[%h] %an %ad %s' --date=short \
              --numstat --before=2013-11-01 > maat_evo.log
```

This will result in a file maat_evo.log in your current directory. Before we feed this file to Code Maat, let's open it and take a look. You will see a logfile with the same type of information as shown in the earlier example.

Inspect the Data

Inspecting the input data is a good starting point. Code Maat provides a summary option that presents an overview of the information in the log. Once you've installed Code Maat as described on the distribution page,[3] fire up the tool by entering the following command—we'll discuss the options in just a minute:

```
prompt> maat -l maat_evo.log -c git -a summary
statistic,value
number-of-commits,88
number-of-entities,45
number-of-entities-changed,283
number-of-authors,2
```

The -a flag specifies the analysis we want. In this case, we're interested in a summary. In addition, we need to tell Code Maat where to find the logfile (-l maat_evo.log) and which version-control system we're using (-c git). That's it. These three options should cover most use cases.

The summary statistics displayed above are generated as comma-separated values (.CSV). The first line, statistic,value, specifies the heading of each column.

For our purposes, the row number-of-entities-changed holds the interesting data. During our specified development period, the different modules in the system have been changed 283 times. Let's see whether we can find any patterns in those changes.

Use CSV Output

Code Maat is designed to be minimalistic. It just collects the results. By generating output as .CSV, a well-supported text format, the output can be read by other programs. You can import the .CSV into a spreadsheet or, with a little scripting, populate a database with the data.

This model allows you to build more elaborate visualizations and analyses on top of Code Maat. Pure text is the universal interface.

Analyze Change Frequencies

Now that you have the modification data, the next step is to analyze the distribution of those changes across modules. To analyze change frequencies, specify the revisions analysis:

3. http://www.adamtornhill.com/code/crimescenetools.htm

```
prompt> maat -l maat_evo.log -c git -a revisions
entity,n-revs
src/code_maat/analysis/logical_coupling.clj,26
src/code_maat/app/app.clj,25
src/code_maat/core.clj,21
test/code_maat/end_to_end/scenario_tests.clj,20
project.clj,19
...
```

The revisions analysis results in two columns: an entity column specifying the name of a source code module, and n-revs, stating the number of revisions of that module.

The output is sorted on the number of revisions. That means our most frequently modified candidate is logical_coupling.clj with 26 changes, followed by 25 changes to the fuzzily named app.clj. I named it—I really should know better.

Thanks to the revisions analysis, you identified the parts of the code with most developer activity. Sure, the number of commits is a rough metric, but we'll meet more elaborate measures later. As you saw earlier in *See That Hotspots Really Work*, on page 22, the relative number of commits is a surprisingly good predictor of defects and design issues. Its simplicity makes it an attractive starting point.

Add the Complexity Dimension

You now have the data to trace the spatial movements of programmers within the code. In Chapter 2, *Code as a Crime Scene*, on page 13, we pointed out the importance of combining that data with a complexity dimension. Let's see where the complexity is hiding in this code.

Get Complexity by Lines of Code

We're going to use lines of code as a proxy for software complexity. Lines of code is a terrible metric, but the other ones are just as bad. (See the research by Herraiz and Hassan in *Making Software [OW10]* for a comparison of complexity metrics.) Using lines of code at least gives us some advantages:

- *It's fast and simple.* More elaborate metrics need to understand the language they're processing. That means they need to parse the code, which may take some time. Lines of code is a fast way to get the same approximation of complexity.
- *It's language-neutral.* Language neutrality is the main reason I prefer lines of code. In today's polyglot systems, sophisticated metrics lose their meaning. As we start to parse individual language constructs, we lose the

opportunity for cross-language comparisons. For example, web applications often combine HTML, CSS, and JavaScript in addition to a server-side technology, such as Java, C#, or Clojure. A language-neutral metric lets us get a holistic picture of all these parts, no matter what language they're written in.

We can always turn to language-specific techniques later to get more details on hotspots. Similarly, we can use any metric to represent complexity. For now, let's summarize the lines of code in our system.

Counting Lines with cloc

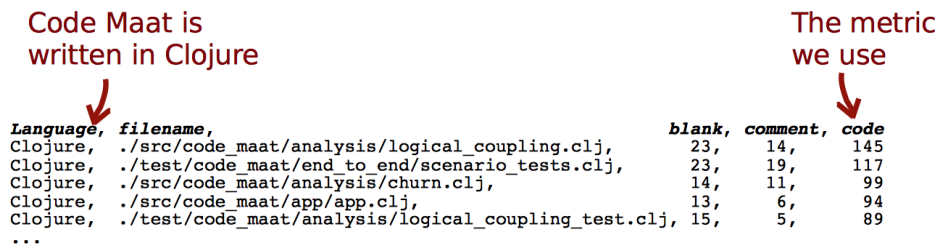
Many tools count lines of code. My favorite is cloc. It's free and easy to use. You can get a copy of cloc on SourceForge.[4]

With cloc installed, let's put it to work:

```
prompt> cloc ./ --by-file --csv --quiet

language,filename,blank,comment,code
Clojure,./src/code_maat/analysis/logical_coupling.clj,23,14,145
Clojure,./test/code_maat/end_to_end/scenario_tests.clj,23,19,117
Clojure,./src/code_maat/analysis/churn.clj,14,11,99
Clojure,./src/code_maat/app/app.clj,13,6,94
Clojure,./test/code_maat/analysis/logical_coupling_test.clj,15,5,89
...
```

Here we told cloc to count all files in the code-maat directory. We also specified that we want statistics --by-file (the alternative is a summary) and --csv output. As you can see in the following figure, cloc does a good job of detecting the programming language the code is written in.



Based on language, cloc separates lines containing comments from real code. We don't want blank lines or comments in our analysis.

4. http://cloc.sourceforge.net/

Merge Complexity and Effort

At this point, you have two different views of the codebase: one that tells you where the complexity is and one that shows individual change frequencies. We will find potential hotspots where the two views intersect.

To merge the two views, we first save the intermediate files to .CSV files:

```
prompt> maat -l maat_evo.log -c git -a revisions > maat_freqs.csv
prompt> cloc ./ --by-file --csv --quiet --report-file=maat_lines.csv
...
```

You should now have two .CSV files in your code-maat directory: maat_freqs.csv and maat_lines.csv. Because merging them would be tedious (but straightforward!), we'll use a Python script available from the Code Maat distribution page.[5]

After you've downloaded it, run the included scripts/merge_comp_freqs.py script with your two .CSV files as input. And remember to prefix the script with the path to your local copy of scripts/merge_comp_freqs.py:

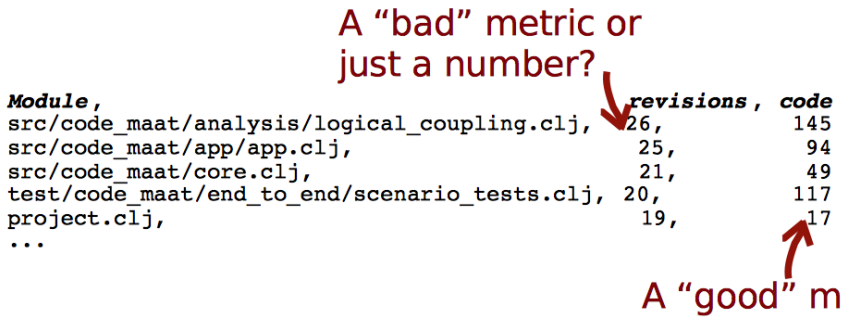
```
prompt> python scripts/merge_comp_freqs.py maat_freqs.csv maat_lines.csv
module,revisions,code
src/code_maat/analysis/logical_coupling.clj,26,145
src/code_maat/app/app.clj,25,94
src/code_maat/core.clj,21,49
test/code_maat/end_to_end/scenario_tests.clj,20,117
project.clj,19,17
...
```

The resulting output is sorted on change frequencies first—the best predictor of problems—and size second. That means we get a prioritized list of suspects immediately; our revisions analysis showed that the logical_coupling.clj module changed often. Now we see that it's complex, too. You've found your first hotspot!

Limitations of the Hotspot Criteria

There is a limitation when looking at hotspots—when is a particular module *hot*? We've just used whatever happened to be the maximum number of revisions in the specified temporal period. What if that number was 3? Or 845? Without any context, they're just numbers.

5. http://www.adamtornhill.com/code/crimescenetools.htm



In Part III, you'll learn about analyses that normalize the data. For now we're going to ignore that; in practice, the non-normalized data we're working on is generally good enough as a guide. We use the relative rank within a codebase to identify hotspots. Before we move on, I just wanted you to be aware of when this might not work.

The hotspots you'll find depend on the temporal period you defined when preparing the files. This can be tricky to get right. By using version-control data, we get a picture of how we actually interact with the system. Over time, our development focus shifts, and the hotspots will also shift. Similarly, as design issues get resolved, hotspots cool down. If they don't, that means the problem still exists.

Finally, individual commit styles may bias the data; some developers commit small, isolated changes, while others prefer a big-bang commit style. I've never experienced this as a problem myself, since features and changes tend to be developed on branches, and we perform our analyses on the master branch or trunk. That said, even if your analysis works fine, you have a lot to gain by getting your team to adopt a shared convention for all commits. It simplifies your workflow.

Use Hotspots as a Guide

Similar to forensic psychology methods, code offender profiling techniques help us narrow down the search area within a codebase. In the next chapter, you'll see the payoff on a large codebase where the hotspots make up a small percentage of the total code. That means we can focus our human expertise on smaller, more focused parts of the system. This opens up a number of possibilities:

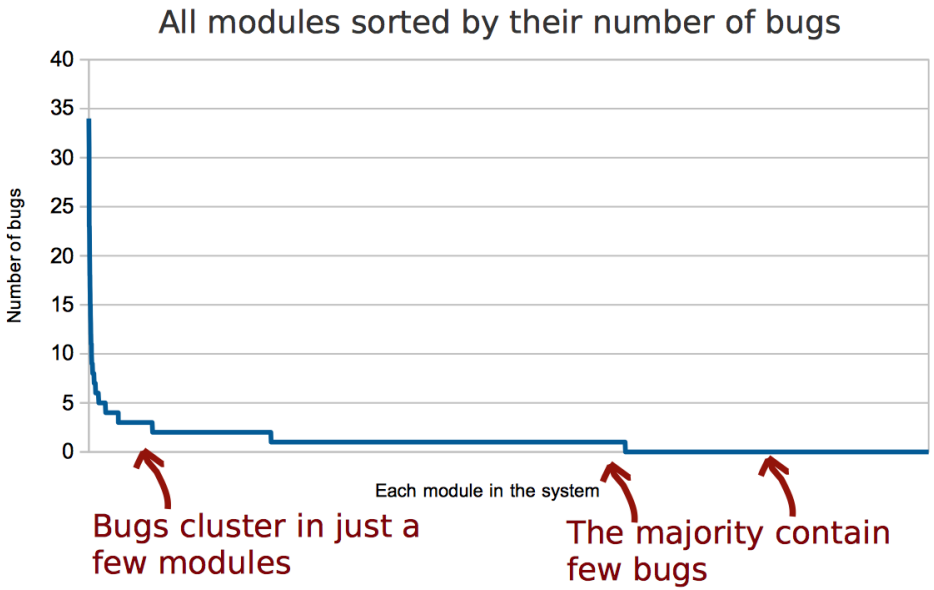
- *Prioritize design issues*: I started this investigation because I needed a way to plan and prioritize improvements to a legacy system. Some of the suggested improvements were redesigns that cost several weeks of intense

work. I had to ensure we spent time on improvements that actually helped future development efforts. Hotspots, indicating complex code that changes frequently, proved to be great candidates. With the material in this chapter, you have the basis to do the same analysis yourself.

- *Perform code reviews*: Code reviews have high defect-removal rates. A code review done right is also time-consuming. Because hotspots make good predictors of buggy code (see *The Relationship Between Hotspots and Defects*, on page 32), they identify the parts where reviews would be a good investment of time.
- *Improve communication*: On a recent project I used the results of a hotspot analysis to communicate with the test leader. That project had several skilled testers. They often spent the end of each iteration on exploratory testing. The test leader wanted a simple way to identify feature areas that could benefit from such additional testing. Hotspots make a perfect starting point for these kinds of tests.

The Relationship Between Hotspots and Defects

Hotspots make good predictors of defects. If your development team maintains a bug database, it's possible to map defects to modules in the code. For example, here's how it looked on a system I investigated:



As you see in the diagram, defects tend to cluster in a few problematic modules. This is a typical pattern found in many systems. In this particular system, the hotspots made up only 4 percent of the code, yet they contained seven of the eight modules with the greatest number of defects.

Guided by hotspots, you'll encircle those buggy areas with high precision. This allows you to improve the code that needs it the most.

Dig Deeper

Right now, we are at the central idea of the book. Subsequent chapters will expand on this concept. While we initially focused on code and design, we'll take the same basic techniques and apply them to organization, teamwork, and communication in Part III. But first let's dig deeper into the concept of hotspots.

The next chapter will discuss how you can investigate hotspots to see whether they're real problems or perhaps just false positives. In the latter case, we can relax, but more often than not we've identified some serious technical problems.

Finally, I'd like to emphasize that a geographic code profile isn't intended to provide an absolute truth about the system. Instead, it's a supporting tool, a guide for your expertise. And it's based on data from how we actually work with the code. How good is that?

CHAPTER 4

Analyze Hotspots in Large-Scale Systems

So far we've applied these concepts to small open-source projects. The real benefits come when we look at a codebase that has outgrown the head of a single developer.

In this chapter, we'll analyze a large open-source system. We'll visualize the analysis results, discuss their interpretations, and relate them to design principles.

When you've finished this chapter, you'll know how to identify weak spots in a large system.

Analyze a Large Codebase

When you start with a new project, how do you know which parts need extra attention? That kind of expertise takes time to build. You need to read a lot of code, talk to more experienced developers, and start small with your own changes. There's no way around this.

At the same time, it's important that you get a quick overview of where potential problems may be hiding. Those problems will influence how you approach design. If you have to add a feature in the middle of the worst spot, you want to know about it so that you can plan countermeasures, such as writing extra tests and setting aside time to refactor the code. You may decide to come up with a different design altogether.

A hotspot analysis gives you an overview of the good as well as the fragile areas of the codebase. The best part is that you get all that information faster than a CSI agent can hack together a Visual Basic GUI to track an IP address in real time.

As an example of in a large-scale system, let's investigate Hibernate[1]—a popular open-source Java library for object-relational mapping. We're using Hibernate because it's well known, has a rich history, and is under active development. If you've worked with a database in the Java ecosystem, chances are you've come across Hibernate.

Clone the Hibernate Repository

To get started, let's clone Hibernate's Git repository to your computer:

```
prompt> git clone https://github.com/hibernate/hibernate-orm.git
Cloning into 'hibernate-orm'...
...
Receiving objects: 100% (210129/210129), 127.83 MiB | 1.99 MiB/s, done.
Resolving deltas: 100% (118283/118283), done.
Checking connectivity... done.
```

Because Hibernate is under active development, we know things may have changed since I wrote this book. So let's roll back the code, as we learned in chapter *Turn Back Time*, on page 24, so that we all start with the same code:

```
prompt> git checkout `git rev-list -n 1 --before="2013-09-05" master`
Note: checking out '46c962e9b04a883e03137962b0bdb71fdcfa0c4e'.
...
HEAD is now at 46c962e... HHH-8468 cleanup and simplification
```

Now the Hibernate code on your computer looks as it did back in September of 2013. Let's generate a log, as we did in *Automated Mining with Code Maat*, on page 26.

Generate a Version-Control Log

We are going to limit our analysis to code changes made in the last year and a half. Here's how you specify that:

```
prompt> git log --pretty=format:'[%h] %an %ad %s' --date=short \
              --numstat --before=2013-09-05 --after=2012-01-01 > hib_evo.log
```

This generates a detailed hib_evo.log we can use with Code Maat. Let's explore the generated data:

```
prompt> maat -l hib_evo.log -c git -a summary
statistic,value
number-of-commits,1346
number-of-entities,10193
number-of-entities-changed,18258
number-of-authors,89
```

1. http://hibernate.org/

As you can see, there's been plenty of development activity over the last year and a half. Remember how we said earlier, in *Analyze a Large Codebase*, on page 35, that finding hotspots makes it easier to get started with a new project? This is a good example: you're starting out with Hibernate and are faced with 400,000 lines of unfamiliar code. Talking to the 89 different developers who've contributed to the project over the past year and a half is impractical (particularly since some of them may have left the project).

Follow along, and you'll see how a hotspot analysis can guide you through unfamiliar code territory.

Choose a Timespan for Your Analyses

First of all, it's important to limit the data you are analyzing to a shorter time period than the project's total lifetime. If you include too much historic data in the analysis, you skew the results and obscure important recent trends. You also risk flagging hotspots that no longer exist.

One approach is to include time in your analysis, by weighing individual measures by their relative age. The challenge if you choose that route is how to set up the algorithm. We go with an alternative approach in this book, which is to limit the period of time we look at. It's a more general approach, but it requires you to be familiar with the development history.

To select an appropriate analysis period, you have to know how you work on the project. You have to know the methodology you're using and the length of your release cycles. The period also depends on the questions you want answered. On my projects I choose the following timeframes:

- *Between releases:* Compare hotspots between releases to evaluate your long-term improvements.
- *Over iterations:* If you work iteratively, measure between each iteration. This lets you spot code that starts to grow into hotspots early.
- *Around significant events:* Define the temporal period around significant events, such as reorganizations of code or personnel. When you make large redesigns or change the way you work, it will reflect in the code. With this analysis method, you have a way to investigate both impact and outcome.

Start with a Long Period

As you start with your first analysis, go with a longer period, such as one or two years of historic data. That lets you explore the system and spot long-term trends. On projects with high development activity, select a shorter initial period, perhaps as little as one month.

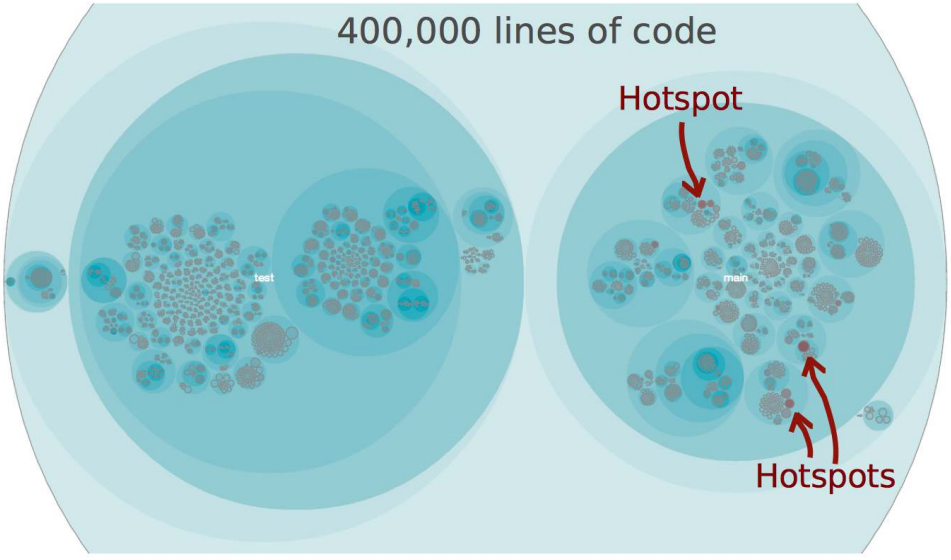
Visualize Hotspots

Large-scale systems will have massive amounts of analysis data. Even if Code Maat identifies the hotspots, it will still be hard to compare subsystems against each other or detect other trends, such as clusters of volatile modules. We need more help.

Visualizations are powerful when you have to make sense of large data sets. Our human brain is an amazing pattern-matching machine. The amount of visual information we're able to process is astonishing. Let's tap into all that brain power.

Use Circle Packing for Large Systems

We haven't identified the hotspots in Hibernate yet. But let's sneak ahead and see where we're heading. Here's how our Hibernate data looks in an *enclosure diagram* (a visualization form that works well for large systems):



Look at all those nested circles. Enclosure diagrams are based on a geometric layout algorithm called *circle packing*. Each circle represents a part of the system. The more complex a module, as measured by lines of code, the larger the circle. And the more effort we spend on a module, as measured by its number of revisions, the more intense its color.

Even if you don't know anything about Hibernate, the visualization gives you an entry point into understanding the system. In the preceding figure, you can see both the good and the fragile parts of the codebase. And that's even before you actually look at the code. Can you think of a better starting point as you enter a large-scale project? Let's see how you collect and interpret all that information.

Mining Hibernate

The steps used to mine Hibernate are identical to the ones you learned earlier in Chapter 3, *Creating an Offender Profile*, on page 23.

This time, we use the size of the codebase as a proxy for complexity. We determine the code size with cloc:

```
prompt> cloc ./ --by-file --csv --quiet --report-file=hib_lines.csv
```

The change frequencies of the modules are used to represent effort. These are calculated with Code Maat:

```
prompt> maat -l hib_evo.log -c git -a revisions > hib_freqs.csv
```

Combining the two views gives you the now-familiar overlap between complexity and effort—the hotspots:

```
prompt> python scripts/merge_comp_freqs.py hib_freqs.csv hib_lines.csv
module,revisions,code
build.gradle,79,402
hibernate-core/.../persister/entity/AbstractEntityPersister.java,44,3983
hibernate-core/.../cfg/Configuration.java,40,2673
hibernate-core/.../internal/SessionImpl.java,39,2097
hibernate-core/.../internal/SessionFactoryImpl.java,34,1384
...
```

The results we just got form the basis of the visualization in the preceding figure; it's just another view of the same data.

Explore the Visualization

The circle-packing visualization used earlier is based on an algorithm from D3.js,[2] a JavaScript library based on data-driven documents. We won't dig deeper into D3.js, as that is enough material for a book of its own. Instead, we'll explore a ready-made visualization.

Before we start, I want to remind you that the strategies we are learning in this book don't depend on specific tools. D3.js is just one of the many ways to visualize the code. Other options include:

- *Spreadsheets:* Since we're using .CSV as the output format, any spreadsheet application lets you visualize the results from Code Maat. Spreadsheet applications are great for processing your analysis results (for example, sorting and filtering the resulting data).
- *R programming language:*[3] The programming language R is a complete environment for both statistical computations and data visualizations. It has a steeper learning curve, but it pays off if you want to dive deeper into data analysis.

Launch the Hotspot Visualization

If you haven't done it already, download the samples from the Code Maat distribution[4] accompanying this book. Unpack the samples and open a command prompt in that directory.

The visualization is an .html file, so you can open it in any web browser. The .html file will load a JSON resource describing our hotspots. Modern browsers introduce a security restriction on that. To make it work flawlessly, launch Python's SimpleHTTPServer in the sample/hibernate directory:

```
prompt> python -m SimpleHTTPServer 8888
```

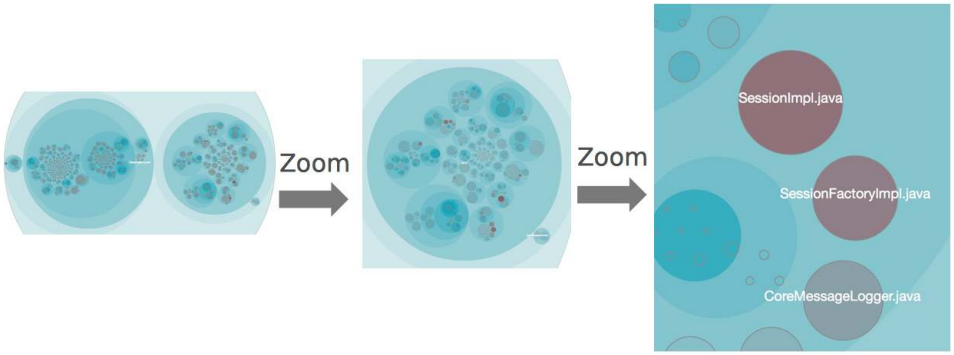
Point your browser to http://localhost:8888/hibzoomable.html. You should now see a familiar picture: the image we saw earlier on page 38.

Use Interactive Hotspot Visualizations

The resulting patterns do look cool—who said large-scale software isn't beautiful? But we're not here to marvel at its beauty. Instead, click on one of

2. http://d3js.org/
3. http://www.r-project.org/
4. http://www.adamtornhill.com/code/crimescenetools.htm

the circles. The first thing you notice is that the visualization is interactive, as shown in the following figure.



An interactive visualization lets you chose your own level of detail. If you see a hot area, zoom in to explore it further.

To get this visual advantage, we need to discuss one more trick to decide which modules to include.

Include All Modules in the Visualization

Look at the preceding figure again. See how the hotspots pop out? The reason is that the entire codebase is visualized, not just code we have recorded changes for. We get that for free, since cloc includes size metrics for all modules in the current snapshot. We then apply color only to the modules that changed within the period of interest.

This visualization makes it easy to toggle between analysis findings and the actual code they represent. Another win is that you get a quick overview of both volatile clusters and the stable parts of the system. Let's see what that distribution of hotspots tells us about our code.

Study the Distribution of Hotspots

Take another look at the the first hotspot picture on page 38. The cluster we see in the lower-right corner is something that frequently happens in software projects. The reason we see that cluster is because changes to one hotspot are intimately tied to changes in other areas.

Multiple hotspots that change together are signs of unstable code. In a well-designed system, we expect code to become more stable over time. Let's consider the underlying design principles to see why.

Individual Coding Styles Affect Hotspots

A project's success depends on the coding skills of the people involved. As humans we vary a lot in our capabilities. A cognitively demanding task like programming amplifies those variations. That's one reason why there's a large difference in quality between the code of different programmers, even on the same team. Given these differences, it is hardly surprising that the expertise and coding techniques of individual developers make up one reason for clusters of hotspots.

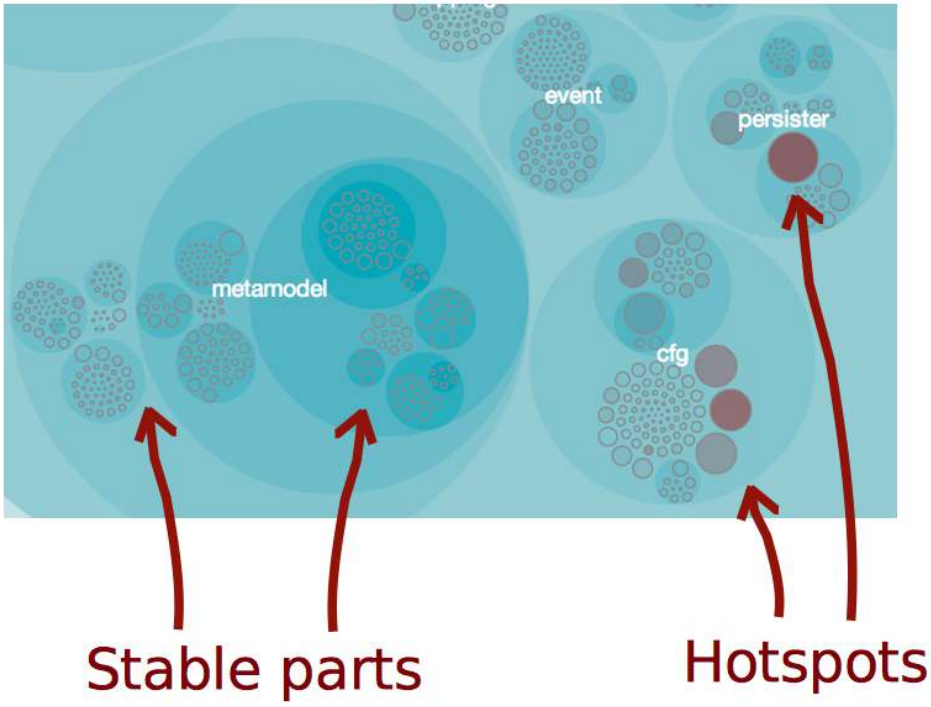
Specialization is very common in large software projects. Developers find their niche within a subsystem or feature area and stay there. If the same person is designing all the modules within that area, then the results are going to be similar. This may be one reason why hotspots attract each other.

We aren't talking about incompetency. When we look at the current hotspot, we don't have the original context, so mistakes look obvious in hindsight. But there are many reasons why the code looks as it does. Perhaps it was a strategic decision to build on top of a fragile base—a deliberate technical debt. Or the hotspot may be a clue that your developers don't know something. You can educate your team with better methods or even rearrange the teams to shift skills around. We'll devote Part III of this book to those areas.

Design to Isolate Change

As we discussed in *See That Hotspots Really Work*, on page 22, there's a strong correlation between the stability of code and its quality. If you look into research on the subject, you'll find support for the idea; the importance of change to a module is so high that more elaborate metrics rarely provide any further predictive value when it comes to defects. (See the research in *Does Measuring Code Change Improve Fault Prediction? [BOW11]*.)

In the following figure, metamodel represents a stable package. That's the kind of code you want. Contrast it with the subsystems cfg and persister, which both contain several hotspots that evolve together.



The importance of change isn't limited to individual classes. It's even more important on a system level. Here's why.

Stabilize by Extracting Cohesive Design Elements

Different features stabilize at different rates. You want to reflect that in your design such that modules in a specific subsystem evolve at similar rate.

When faced with clusters of hotspots, we need to investigate the kind of changes we are making. Because you're working with version-control data, you can look at each change made to the modules. You'll often find that the change patterns suggest new modular boundaries.

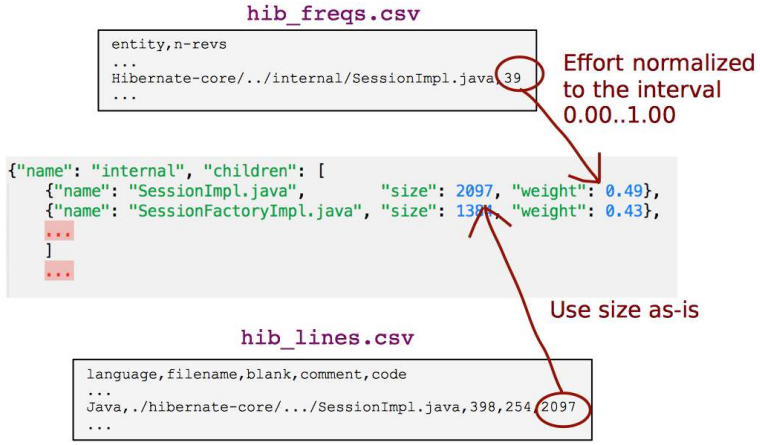
When modules aren't stable, we need to act. Often that failure stems from low *cohesion*; the subsystems keep changing because they have many reasons to do so.

The following figure shows a simplified example from Code Maat. Its `app.clj` module has changed several times, each time for different reasons. It has three responsibilites, and therefore three reasons to change. Refactoring this module to three independent ones would isolate the responsibilities and stabilize the code.

The more of your code you can protect from change, the better. Cohesion lets you isolate change. You have less code to modify in stable modules. This relieves your brain, as it means you have less knowledge to keep in your head. It's cognitive economy.

Create Your Own Visualizations

Before we move on, let's see how you can visualize your own code with a set of simple tools. The visualizations in this chapter all contain the same information as we have in the Code Maat output. Because that output is in `.CSV` format, it's easy to write scripts to transform it into other formats. The D3.js visualizations we've been using are based on the Zoomable Circle Packing algorithm.[5] Because D3.js is data-driven, you can ignore most details and treat the visualization as a black box. You'll find that it's surprisingly easy to experiment with.



5. http://bl.ocks.org/mbostock/7607535

The D3.js circle-packing algorithm expects a JSON document. That means you'll have to convert from .CSV to JSON and feed the resulting data into D3.js. I've included a script for the conversion in the Code Maat distribution you've downloaded. (It's the script used to create the visualizations in this chapter.) Run it with the -h flag as python csv_as_enclosure_json.py -h to get a usage description.

Try it out. Visualizing data is fun!

Differentiate Between True Problems and False Positives

We've covered a lot of ground in this chapter. You analyzed a large system, identified its hotspots, and learned techniques for investigating the visualizations.

You learned how a hotspot lets you focus on narrow areas of the code in need of attention. So you don't have to manually inspect hundreds of modules, the analysis gave you a prioritized list of problem modules. You can use that to guide your future work. If you're in a position to redesign the hotspots, then do it! Otherwise, you need to take defensive measures, such as writing additional tests or regularly inspecting code.

We also discussed how hotspots are distributed and how we can use this information to identify the stability of each subsystem or feature area. The key idea is that you want to evolve your modules into stability as soon as possible. (To read more on the subject, check out Michael Feathers's excellent blog on The Active Set of Classes,[6] which takes a slightly different view.)

You're now at a point where you can identify weak spots in a large system. To make efficient use of your new skills, you need strategies to sort out real problems from false positives. The good news is that you're just a chapter away from such techniques.

6. http://michaelfeathers.typepad.com/michael_feathers_blog/2012/12/the-active-set-of-classes.html

CHAPTER 5

Judge Hotspots with the Power of Names

If you profile offenders, you rely on hotspots in your hunt for criminals. However, while they are better than random guessing, they do not guarantee that you will find the offender. In code, hotspots can identify real problems, but you have to be prepared for false positives, too.

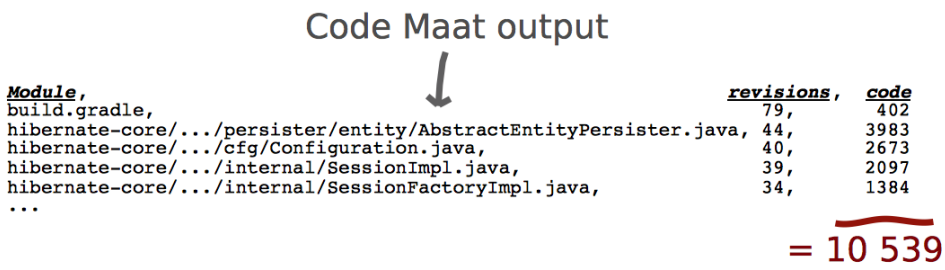
In this chapter, you'll learn heuristics to pass quick judgments on your hotspots. The goal is to differentiate between hotspots that are the result of complex application logic and those that are based on configuration files. The distinction is important. Hotspots give us a narrow view of the system, but we may still have thousands of lines of code to inspect. Sorting out the hotspots that really matter saves time and effort.

The first heuristic we'll discuss is naming. As you'll see, good names helps you understand your design elements. Here we'll use names as a heuristic on the quality of modules.

Know the Cognitive Advantages of Good Names

Back in *Mining Hibernate*, on page 39, you created a code offender profile of Hibernate. The resulting hotspots presented a different view of the system than what you normally see. Buried deep within 400,000 lines of code, the hotspot analysis flagged a number of potential design issues you needed to be aware of.

As you can see in the following figure, the top five hotspots still account for 10,000 lines of code. It's much better than 400,000, but it's still plenty of code.



This ratio between hotspots and total code size is quite typical across systems. Hotspots typically account for around 4 to 6 percent of the total codebase. Remember that hotspots reflect the *probability* of there being a problem, so false positives are possible. Any hotspot we can rule out is a win. We *could* look into the code to find out, but a faster way can do the trick: by looking at the name of the hotspot.

Names Make the Code Fit Your Head

When it comes to programming, the single most important thing we can do for our programs is to name their design elements. Put names on your concepts. A name is more than a description—it helps a program fit your head.

Our brain has several bottlenecks. To a programmer, the most painful bottleneck is working memory. Our brain's working memory is short term and serves as the mental workbench of the mind. This is where we integrate and manipulate information.

Working memory is also limited cognitively—there are only so many pieces of information we can hold in our head at once. Research indicates that we can keep three to seven items in memory simultaneously. Practically every programming task stretches our working memory to the max.

We can't expand the number of items we can keep in working memory, but we can make each item carry more information. We call this *chunking*. We create chunks as we take low-information items and group them together, such as when we group characters to form words. Similarly, we introduce chunks in our programs when we group computational expressions into named functions. Now each name serves as a chunk and makes the code easier to work with. Your brain will thank you for coming up with good names.

Recognize Bad Names

When we choose good names, we make our code cheaper to maintain. Remember back in *Optimize for Understanding*, on page 2, you learned that

we spend most of our time modifying existing code. Names guide us with this task. Research shows that we try to infer the purpose of the code and build a mental representation just by reading the name of a class or function. Names rule. (See *Software Design: Cognitive Aspects [DB13]* for the empirical findings.)

Top-level design elements, such as modules and classes, are always named. (This isn't true for concepts such as anonymous classes, but these are implementation details, not top-level elements.) We use those names to pass a quick judgment on the hotspots we find. The idea is to differentiate between hotspots due to complex application logic and plain configuration files. While we expect a configuration file to change frequently, hotspots in application logic signal serious design issues.

So, what's a bad name? To get an idea, let's take the guidelines for good naming and look for the complete opposite:

- A good name is descriptive and expresses intent. For example, `ConcurrentQueue` and `TcpListener`.
- Bad names carry little information and convey no hints to the purpose of the module. For example, `StateManager` (isn't state management what programming is about?) and `Helper` (a helper for what and whom?).
- A good name expresses a single concept that suggests cohesion. Remember, fewer responsibilities means fewer reasons to change. Again, `TcpListener` is a good example.
- A bad name is built with conjunctions, such as `and`, `or`, and so on. These are sure signs of low cohesion. Examples include `ConnectionAndSessionPool` (do connections and sessions express the same concept?) and `FrameAndToolbarController` (do the same rules really apply to both frames and toolbars?).

Bad names attract suffixes like lemonade draws wasps on a hot summer day. The immediate suspects are everything that ends with `Manager`, `Util`, or the dreaded `Impl`. Modules baptized like that are typically placeholders, but over time they end up housing core logic elements. You know they will hurt once you look inside.

The guidelines in this chapter apply to object-oriented inheritance hierarchies, too. Good interfaces express roles and communication protocols between objects. Their implementations specify both what's specific and what's different about the concrete instances.

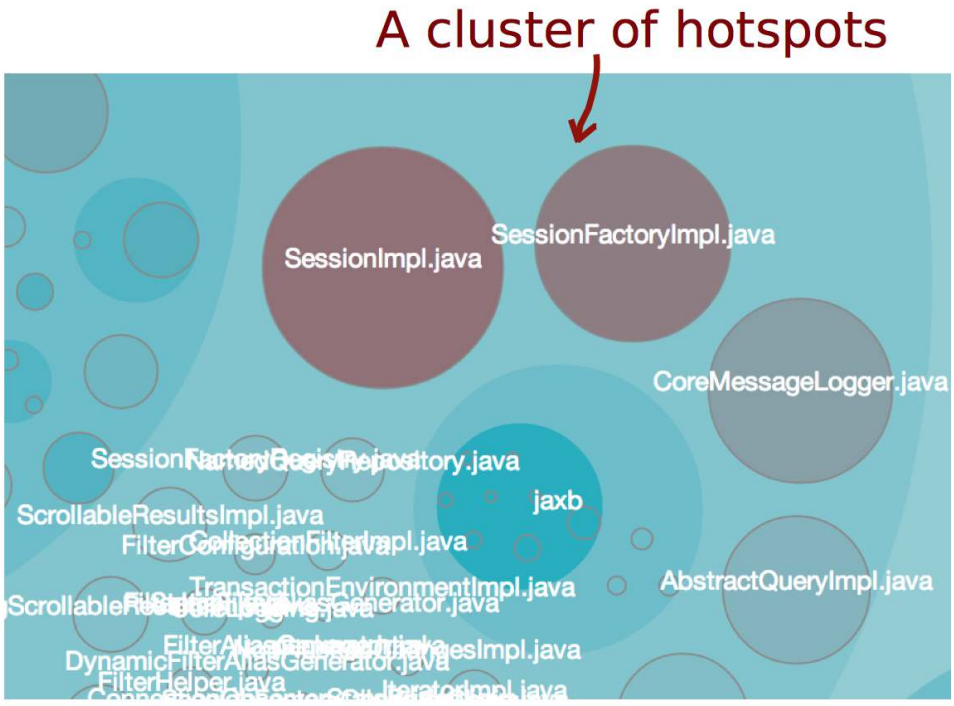
Bad interfaces suffer the same information drain as bad names. Their implementations fail to add specific info about the concrete instance. For example, the interface IState doesn't carry information (again, imperative programming is all about state) and its implementor State doesn't specify the context.

Express Intent and Suggest Usage

A good naming strategy for object-oriented hierarchies is to express intent and suggest usage. Say we create an intention-revealing interface: ChatConnection. (Yes, I did it—I dropped the cognitive distractor, the I prefix.) Let each implementation of this interface specify what makes it unique: SynchronousTcpChatConnection, AsynchronousTcpChatConnection, and so on.

Investigate a Hotspot by Its Name

Remember how we found our suspects in Hibernate back in Chapter 4, *Analyze Hotspots in Large-Scale Systems*, on page 35? Let's zoom in on our suspects, as we do in the following image. Now we can figure out what's going on by looking at their names.

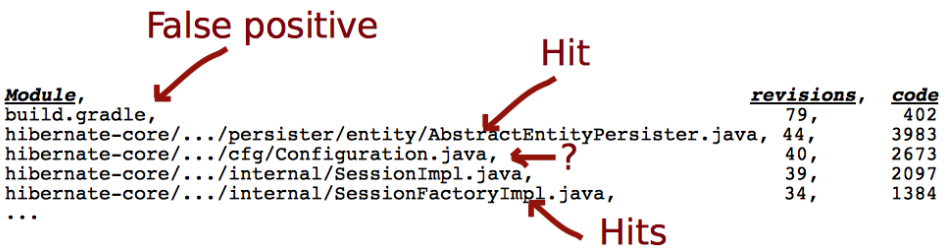


Both SessionImpl.java and SessionFactoryImpl.java have names that sound suspicious. Sessions are key concepts in ORM frameworks, such as Hibernate—so it's clear these files must have a lot of application logic inside.

Both modules are large. (Remember: the diameter of the circle represents the size dimension.) Combined with their very general names, we can assume that they are real problems and not false positives. If you were maintaining this system, I'd recommend you investigate these hotspots deeper.

Check Your Assumptions with Complexity

As you go through the top hotspots, you pass the same judgment on the AbstractEntityPersister. Persisting entities seems to pretty much nail what an ORM is about. Its prefix is a concern, too. To abstract means to take away—if the abstract representation of an entity persister still consists of 4,000 lines of code, you know you've found a candidate for refactoring.



The module build.gradle is another story. We can tell that it's part of the build system. Build files can still pose problems (I've spent years trying to debug legacy makefiles; that's time I will never get back), but its modest 402 lines of code suggests it's at least comprehensible.

Finally, we get to the trickiest module. Let's look at our analysis results again. By virtue of its name alone, Configuration.java would be considered a false positive because it's a configuration file. But its code size of 2,600 lines should make alarm bells go off. Perhaps there's more than plain configuration settings in there?

This is where our heuristics of using names and code size reach their limits. We either need to manually inspect Configuration.java or use more sophisticated methods to understand the module's complexity.

In the next chapter, you'll learn a quick way to estimate complexity. Until then, let's put Configuration.java on hold and go over what to do with the hotspots we've found so far.

Understand the Limitations of Heuristics

Heuristics are common in everyday life. We use heuristics all the time for our decisions and judgments. Real-life choices are a bit like modifying legacy code: we have to make decisions based on incomplete information and uncertain outcomes. In both situations, we aim for solutions that we believe have a high probability of leading to desirable outcomes.

Heuristics by definition are imprecise. A common source of error is to substitute a difficult question for a simple one. Because the mental processes are unconscious, we're not even aware that we answered the wrong question.

One example is *availability bias*: we base decisions on how easily examples come to mind. In a classic study by Paul Slovic in *Decision Making: Descriptive, Normative, and Prescriptive Interactions [SFL88]*, researchers asked people about the most likely causes of death. The participants could choose between pairs such as botulism or lightning, or murder or suicide. Respondents misjudged the probabilities in favor of the more dramatic and violent example—for example, choosing murder over suicide and lightning over botulism, although statistics show that the reverse is much more likely.

We're not immune to these biases during software development, either. Suppose you recently read a blog post describing a data access implementation. If you were asked where the problems are in your own system, the availability bias might well kick in, and you'd be predisposed to answer "data access." And that's even if you didn't recall that you had read that blog post.

Our constant reliance on heuristics is one reason why we need techniques like the ones in this book. These techniques support our decision-making and let us verify our assumptions. We humans are anything but rational.

Complement Your Heuristics with Data

When you started this chapter, you'd already identified some hotspots. Now you've learned about simple ways to classify them. By using the name of the potential offender, you can sort out true problems from false positives.

Heuristics are mental shortcuts. When we rely on them, we trade precision for simplicity. There's always a risk that we may draw incorrect conclusions. Remember how we saw a warning signal as we categorized `Configuration.java` in *Check Your Assumptions with Complexity*, on page 51? That's just a risk we have to take.

With hotspots such as `SessionImpl.java` and `SessionFactoryImpl.java`, we want to refactor these files. Such large-scale refactorings are challenging and require

more discipline than local changes. It's way too easy to code yourself into a corner. To support such refactorings, have a look at Appendix 1, *Refactoring Hotspots*, on page 183, which uses names as a guide during the initial refactoring effort once the offending code is found.

We also want to consider whether the hotspot code is deteriorating further or improving over time. Many teams actively refactor code, so perhaps the area flagged in the hotspot is actually in better shape now than it was a year ago. In that case, the code may be heading in the right direction. One way to investigate that is by looking at the complexity trends over time. In the next chapter, we'll investigate a fast, lightweight metric that lets us calculate and understand trends with a minimum of overhead.

CHAPTER 6

Calculate Complexity Trends from Your Code's Shape

You just learned how to use names when investigating design issues. Most hotspots you'll find are so complex that you need to make several passes through the code before you uncover and treat the root cause. During that time, you want to be sure the code is evolving in the right direction. Simpler *and* better. Here's how.

In this chapter, we'll use the shape of your code as a proxy for program complexity. We'll apply *indentation-based complexity measures* to the hotspot's revision history to calculate complexity trends. We'll be able to see whether the code is deteriorating or improving over time. This is additional information we can use to prioritize the hotspots we find.

On top of that, we'll discuss learning, encouraging design discussions, and measuring the modification effort. We'll do this by looking at what's not there—negative space is a new way to view your code. Let's see it in action.

Complexity by the Visual Shape of Programs

A few years ago, I used to commute to work by train. Since I went to the station at about the same time each day, I soon recognized my fellow commuters. One man used to code on his laptop during the ride. I don't know whether he wrote Java, C++, or C#, but even with just a hurried glance, I knew his software was complex.

You've probably done the same. Isn't it fascinating that we can form an impression of something as intricate as software by a quick look?

Humans are visual creatures. Our brain processes a tremendous amount of visual information in a single glance. As programmers, when we glimpse code, we automatically compare the code's shape—how the code looks visually—against other code we've seen. Even if we aren't consciously aware of it, after years of coding, we know what good code looks like. We can use this skill more deliberately.

Judge Complexity by Program Shape

Look at these modules, A and B. Which one would you rather maintain and extend?

Independent of the programming language used, the differences in complexity are striking. Module B looks pretty straightforward, while module A winds down a complex, conditional slope. You see how the shape of the code reveals its complexity?

Learn from the Shape of Code

I started to investigate the idea of using the shape of code while teaching test-driven development—TDD, a high-discipline methodology littered with pitfalls. (See http://www.adamtornhill.com/articles/codepatterns/codepatterns.htm for the original writeup.) The design context resulting from TDD is far from trivial. By creating compact overviews of the code's shape, similar to the image that follows on page 57, the programming team was able to:

- Compare differences in complexity between unit tests.
- Use shapes during code reviews as an effective way to encourage discussions about the high-level design.
- Identify parts of the design that diverge from the rest of the structure.
- Let the visual contrast serve as a tool to highlight basic design principles. For example, a solution using polymorphism looks quite different from one based on conditional logic.

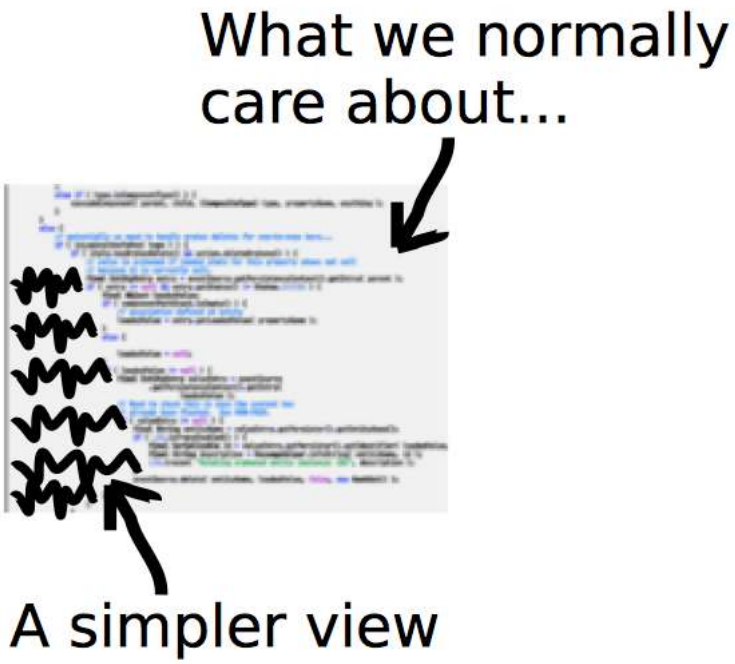
The reason this technique works is that visual shapes give you a compact high-level perspective without distracting with details.

Visual inspections are fine for analyzing individual units. But what happens when we want to scale up and look at complex systems? We automate the task.

This kind of image comparison is hard for a computer to do. For what your brain does by the flick of some neurons, a machine must use complex algorithms and have serious computing power. So we'll analyze the textual representation of the code. We'll also simplify the process by shifting our perspective: let's focus on the negative space.

Learn About the Negative Space in Code

Virtually all programming languages use whitespace as indentation to improve readability. (Even Brainf***[1] programs seem to use it, despite the goal implied by the language's name.) Indentation correlates with the code's shape. So instead of focusing on the code itself, we'll look at what's not there, the negative space. We'll use indentation as a proxy for complexity.



The idea of indentation as a proxy for complexity is backed by research. (See the research in *Reading Beside the Lines: Indentation as a Proxy for Complexity*

1. http://en.wikipedia.org/wiki/Brainfuck

Metric. Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on [HGH08].) It's a simple metric, yet it correlates with more elaborate metrics, such as McCabe cyclomatic complexity and Halstead complexity measures.

The main advantage to a whitespace analysis is that it's easy to automate. It's also fast and language-independent. Even though different languages result in different shapes, the concept works just as well on Java as it does on Clojure or C.

However, there is a cost: some constructs are nontrivial despite looking flat. (List comprehensions[2] come to mind.) But again, measuring software complexity from a static snapshot of the code is not supposed to produce absolute truths. We are looking for hints. Let's move ahead and see how useful these hints can be.

Whitespace Analysis of Complexity

Back in *Check Your Assumptions with Complexity*, on page 51, we identified the Configuration.java class in Hibernate as a potential hotspot. Its name indicates a plain configuration file, but its large size warns it is something more. A complexity measure gives you more clues.

Calculating indentation is trivial: just read a file line by line and count the number of leading spaces and tabs. Let's use the Python script complexity_analysis.py in the scripts folder of the code you downloaded from the Code Maat distribution page.[3]

The complexity_analysis.py script calculates logical indentation. Four spaces or one tab counts as one logical indentation. Empty and blank lines are ignored.

Open a command prompt in the Hibernate root directory and fire off the following command. Just remember to provide the real path to your own scripts directory:

```
prompt> python scripts/complexity_analysis.py \
 hibernate-core/src/main/java/org/hibernate/cfg/Configuration.java
n,total,mean,sd,max
3335,8072,2.42,1.63,14
```

Like an X-ray, these statistics give us a peek into a module to reveal its inner workings. The total column is the accumulated complexity. It's useful to

2. http://en.wikipedia.org/wiki/List_comprehension
3. http://www.adamtornhill.com/code/crimescenetools.htm

compare different revisions or modules against each other. (We'll build on that soon.) The rest of the statistics tell us how that complexity is distributed:

- The mean column tells us that there's plenty of complexity, on average 2.42 logical indentations. It's high but not too bad.
- The standard deviation sd specifies the variance of the complexity within the module. A low number like we got indicates that most lines have a complexity close to the mean. Again, not too bad.
- But the max complexity show signs of trouble. A maximum logical indentation level of 14 is high.

A large maximum indentation value means there is a lot of indenting, which essentially means nested conditions. We can expect islands of complexity. It looks as if we've found application logic hidden inside a configuration file.

Analyze Code Fragments

Another promising application is to analyze differences between code revisions. An indentation measure doesn't require a valid program—it works just fine on partial programs, too. That means we can analyze the complexity delta in each changed line of code. If we do that for each revision in our analysis period, we can detect trends in the modifications we make. This usage is a way to measure modification effort. A low effort is the essence of good design.

When you find excess complexity, you have a clear candiate for refactoring. Before you begin refactoring, you may want to check out the module's complexity trend. Let's apply our whitespace analysis to historical data and track trends in the hotspot.

Analyze Complexity Trends in Hotspots

In a healthy codebase, you can add new features with successively less effort. Unfortunately, the reverse is often true: new features add complexity to an already tricky design. Eventually, the system breaks down, and development slows to a crawl.

This phenomenon was identified and formalized by Manny Lehman[4] in a set of laws on software evolution. In his *law of increasing complexity*, Lehman states that "as an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it." (See *On Understanding Laws, Evolution, and Conservation in the Large-Program Life Cycle [Leh80]*.)

4. http://en.wikipedia.org/wiki/Manny_Lehman_%28computer_scientist%29

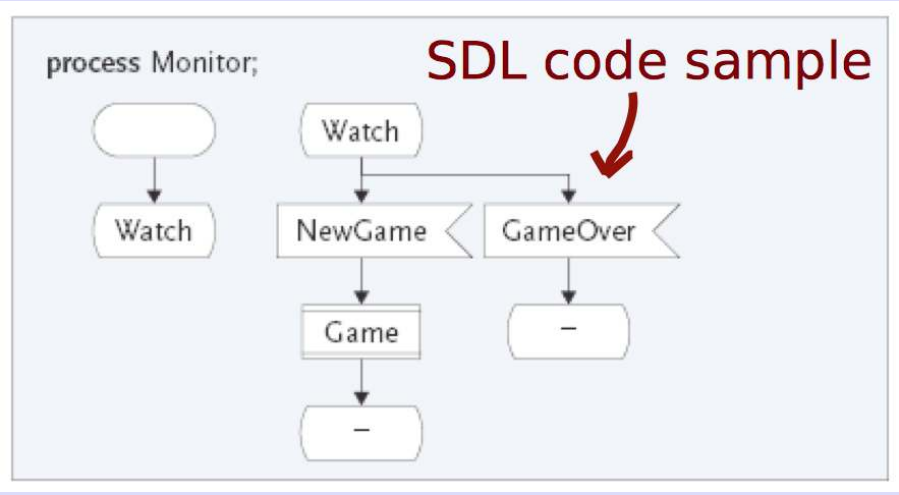
Joe asks:

With the Power of Shapes, Wouldn't Visual Programming Languages Give Us an Edge?

Since the dawn of computing, our industry has tried to simplify programming. Visual programming is one such approach. Instead of typing cryptic commands in text, what if we could just draw some shapes, press a button, and have the computer generate the program? Wouldn't that simplify programming? Indeed it would. But not in the way the idea is sold, nor in a way that matters.

Visual programming might make small tasks easier, but it breaks down quickly for larger problems. (*The Influence of the Psychology of Programming on a Language Design [PM00]* has a good overview of the research.) The thing is, it's the larger problems that would benefit from simplifying the process—small tasks generally aren't that complex. This is a strong argument against visual programming languages. It also explains why demonstrations of visual programming tools look so convincing—demo programs are small by nature.

Expressions also don't scale very well. A visual symbol represents one thing. We can assign more meanings to it by having the symbol depend on context. (Natural languages have tried this—hieroglyphs show the limitations of the system.) Contrast this with text where you're free to express virtually any concept.



I became painfully aware of the limitations of visual programming when I rewrote in C++ a system created in the graphical Specification and Description Language (SDL). What took four screens of SDL was transformed into just a few lines of high-level C++.

You already know about hotspot analyses to identify these "deteriorating structures" so that you can react and reduce complexity. But how do we know if we are improving the code over time or just contributing to the grand decline? Let's see how we uncover complexity trends in our programs.

Use Indentation to Analyze Complexity Trends

An indentation analysis is fast and simple. That means it scales to a range of revisions without eating up your precious time. Of course, you may well wonder if different indentation styles could affect the results. Let's look into that.

This chapter has its theoretical foundations in the study *Reading Beside the Lines: Indentation as a Proxy for Complexity Metric. Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on [HGH08]*. That research evaluated indentation-based complexity in 278 projects. They found that indentation is relatively uniform and regular. Their study also suggests that deviating indentations don't affect the results much.

The explanation is also the reason the technique works in the first place: indentation improves readability. It aligns closely with underlying coding constructs. We don't just indent random chunks of code (unless we're competing in the International Obfuscated C Code Contest).[5]

Similarly, it doesn't really matter if we indent two or four spaces. However, a change in indentation style midway through the analysis could disturb your results. For example, running an auto-indent program on your codebase would wreck its history and show an incorrect complexity trend. If you are in that situation, you can't compare revisions made before and after the change in indentation practices.

Even if individual indentation styles don't affect the analysis results as much as we'd think, it's still a good idea to keep a uniform style as it helps build consistency. With that sorted out, let's move on to an actual analysis.

Focus on a Range of Revisions

You've already seen how to analyze a single revision. Now we want to:

1. Take a range of revisions for a specific module.
2. Calculate the indentation complexity of the module as it occurred in each revision.
3. Output the results revision by revision for further analysis.

5. http://www.ioccc.org/

With version-control systems, we can roll back to historical versions of our code and run complexity analyses on them. For example, in git we look at historical versions with the show command.

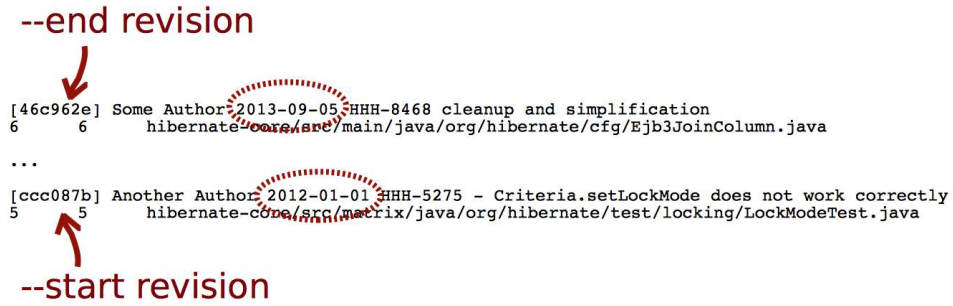
The receipe for a trend analysis is pretty straightforward, although it requires some interactions with the version-control system. Since this book isn't about git or even version-control systems, we're going to skip over the actual implementation details and just use the script already in your scripts directory. Don't worry, I'll walk you through the main steps to understand what's happening so that you can perform your own analysis on your code.

Discover the Trend

In your cloned Hibernate Git repository, type the following into the command prompt (and remember to reference your own scripts path) to run git_complexity_trend.py:

```
prompt> python scripts/git_complexity_trend.py \
 --start ccc087b --end 46c962e \
 --file hibernate-core/src/main/java/org/hibernate/cfg/Configuration.java
rev,n,total,mean,sd
e75b8a7,3080,7610,2.47,1.76
23a6280,3092,7649,2.47,1.76
8991100,3100,7658,2.47,1.76
8373871,3101,7658,2.47,1.76
...
```

This looks cryptic at first. What just happened is that we specified a range of revisions determined by the --start and --end flags. Their arguments represent our analysis period, as we see in the following image.



After that, you gave the name of the --file to analyze. In this case, we focus on our suspect, Configuration.java.

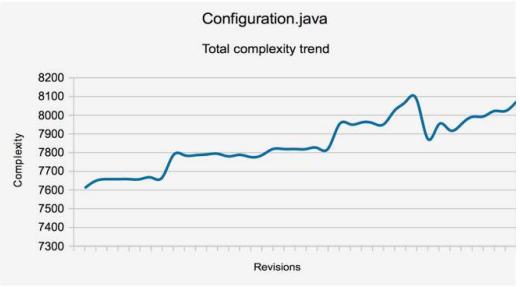
The analysis generates .CSV output similar to the file you got during the earlier single-module analysis. The difference here is that we get the complexity

statistics for each historical revision of the code. The first column specifies the commit hash from each revision's git code. Let's visualize the result to discover trends.

Visualize the Complexity Trend

Spreadsheets are excellent for visualizing .CSV files. Just save the .CSV output into a file and import it into Excel, OpenOffice, or a similar application of your choice.

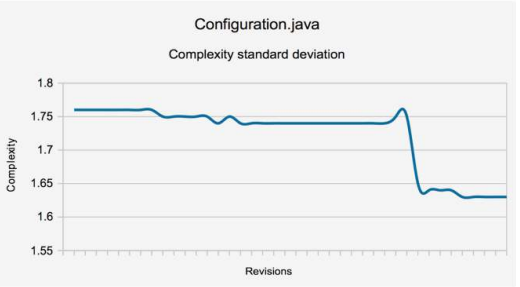
Let's look at the total complexity growth first. That would be the total column.



As you can see in the image, Configuration.java accumulated complexity over time.

This growth can occur in two basic ways:

1. New code is added to the module.
2. Existing code is replaced by more complex code.



Case 2 is particularly worrisome—that's the "deteriorating structure" Lehman's law warned us about. We calculated the standard deviation (in the sd column) to differentiate between these two cases. Let's see how it looks.

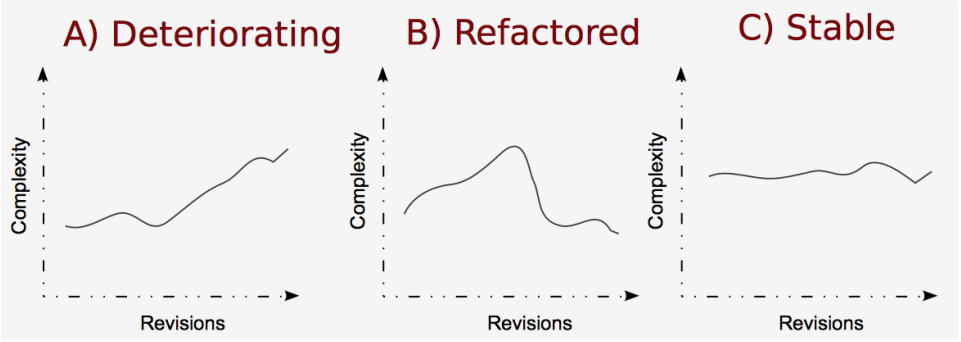
The standard deviation decreases. This means lines get more alike in terms of complexity, and it is probably a good thing. If you look at the mean, you see that it, too, decreases. Let's see what that means for your programs.

Evaluate the Growth Patterns

When you're analyzing complexity trends, the interesting thing isn't the numbers themselves but the shape of the evolutionary curve.

The following figure shows the shapes you're most likely to find in a typical codebase. The dip in the curve in Case B generally is a sign of a refactoring, because the code got simpler. Case C is also common because there's little

change in the file's complexity. We have a stable module where we make small tweaks and changes. I expect to see this pattern in a configuration file.



Case A is a warning sign. As the file gets more complex, it becomes harder to understand.

To know how bad it is, we look at descriptive statistics—such as standard deviation—as we did earlier. A high standard deviation points to many complex blocks of conditional logic, which is the kind of code that's hard to understand and maintain.

From Individual Hotspots to Architectures

In this chapter, we started to look inside modules. You learned that the visual shape of code tells us about its complexity. You saw how analyzing indentation provides fast and language-neutral results. We can use indentation to measure and compare complexity across our codebase.

By calculating complexity trends over a range of historical revisions, we get enough information to quickly judge the direction hotspots are going.

Now that we've finished this chapter, we've taken the concept of hotspot analysis in software development full circle. We learned how to analyze small systems, such as Code Maat, and large-scale codebases, such as Hibernate. We drilled into individual modules to reveal their internal complexity.

We learned that a hotspot analysis is an ideal first entry point into a new system. Now's the time to go from individual modules to high-level designs and architectures. In the next part of the book, you'll learn to find patterns in how multiple hotspots evolve together. This information lets you pass similar judgments on the architecture of your system, not just individual modules. Let's dissect our architectures!

Part II

Dissect Your Architecture

Part I showed you how to identify offending code in your system. Now we'll look at the bigger picture.

In this part, you'll learn to evaluate your software architecture against the modifications you make to your code. The techniques let you identify signs of structural decay, provide refactoring directions, and suggest new modular boundaries in your design. And it all starts with forensics. Let's see what an innocent robber can teach us about software design.

CHAPTER 7

Treat Your Code As a Cooperative Witness

In Part I, we looked at how to detect hotspots in code. You learned to pass quick judgments on the hotspots you found. You also learned how to measure their complexity trends to determine whether the code was improving or getting worse over time. This is the ideal starting point. The next step is to look at the bigger picture.

In this part of the book, we'll transition from looking at individual modules to analyzing complete software architectures. We'll evaluate how well the high-level design of our system supports the evolution of our code: is our architecture more of a hindrance than a help?

We'll still use hotspots, though. The techniques you're about to learn identify high-level structural problems, and we'll use hotspots to gain more information.

We'll start with another forensic psychology case study to learn about eyewitness interviews. This case study illustrates common memory biases and why we need to support our decisions with objective data. We'll then apply the concept to software development. You'll see how a change to one component leads to a cascade of complex changes in other parts of the code. Hotspots rarely walk alone.

This leads us to the concept of *temporal coupling*. Temporal coupling is a type of dependency you cannot deduce just by looking at the code. It's a powerful interview tool for our codebase and lets us identify improvements based on how we worked with the code in the past.

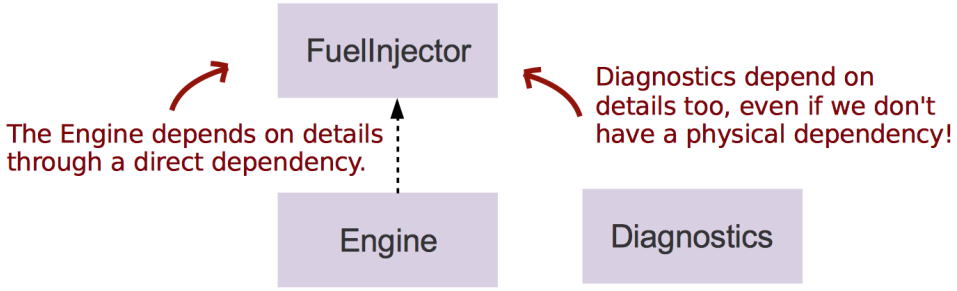
So let's move ahead and listen to the modification patterns in our system. They have a lot to tell us.

Know How Your Brain Deceives You

If you've worked in the software industry for some time, you're probably all too familiar with the following scenario. You are working on a product in which you or your company has invested money. This product will need to improve as users demand new features.

At first, the changes are small. For example, you tweak the FuelInjector algorithm. As you do, you realize that the Engine abstraction depends on details of the FuelInjector algorithm, so you modify the Engine implementation, too. But before you can ship the code, you discover by accident that the logging tool still shows the old values. You need to change the Diagnostics module, too. Phew—you almost missed that one.

If you had run a hotspot analysis on this fictional codebase, Engine and Diagnostics probably would've popped up as hotspots. But what the analysis would've failed to tell you is that they have an implicit dependency on each other. Changes to one of them means changes in the other. They're entangled.



The problem gets worse if there isn't any explicit dependency between them, as you can see in our example. Perhaps the modules use an intermediate format to communicate over a network or message bus. Or perhaps it's just copy-paste code that's been tweaked. In both cases, there's nothing in the structure of your code that points at the problem. In this scenario, dependency graphs or static-analysis tools won't help you.

If you spend a lot of time with the system, you'll eventually find out about these issues. Perhaps you'll even remember them when you need to, even when you're under a time crunch, stressed, and not really at your best. Most of us fail sometimes. Our human memory is everything but precise. Follow along to see how it deceives us.

The Paradox of False Memories

But if I'm confident in a memory, it must be correct, right?

Sorry to disappoint you, but no, confidence doesn't guarantee a correct memory. To see what I mean, let's look into the scary field of false memories.

A false memory sounds like a paradox. False memories happen when we remember a situation or an event differently from how it actually looked or occurred. It's a common phenomenon and usually harmless. Perhaps you remember rain on your first day of school, while in fact the sun shone. But sometimes, particularly in criminal investigations, false memories can have serious consequences. Innocent people have gone to jail.

There are multiple reasons why we have false memories. First of all, our memory is constructive, meaning the information we get *after* an event can shape how we recall the original situation. Our memory organizes the new information as part of the old information, and we forget when we learned each piece. This is what happened in the Father Pagano case we'll work on in this chapter.

Our memory is also sensitive to suggestibility. In witness interviews, leading questions can alter how the person recalls the original event. Worse, we may trust false memories even when we are explicitly warned about potential misinformation. And if we get positive feedback on our false recall, our future confidence in the (false) memory increases.

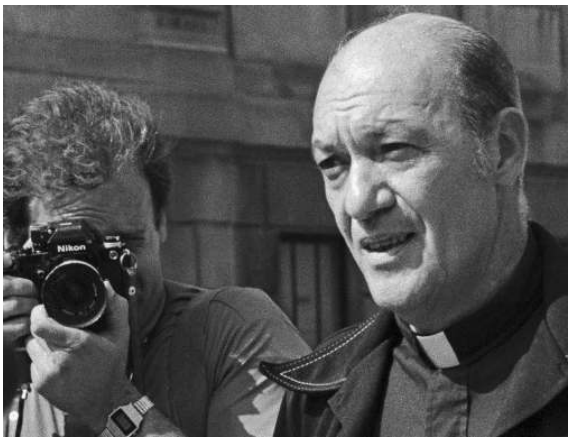
Keep a Decision Log

In software, we can always look back at the code and verify our assumptions. But the code doesn't record the whole story. Your recollection of why you did something or chose a particular solution is sensitive to bias and misinformation, too. That's why I recommend keeping a decision log to record the rationale behind larger design decisions. The mind is a strange place.

Meet the Innocent Robber

Our human memory is a constructive process. That means our memories are often sketchy, and we fill out the details ourselves as we recall the memory. This process makes memories sensitive to biases. This is something Father Pagano learned the hard way.

Back in 1979, several towns in Delaware and Pennsylvania were struck by a series of robberies. The salient characteristic of these robberies was the perpetrator's polite manners. Several witnesses identified a priest named Father Pagano as the robber. Case solved, right?



Father Pagano probably would have gone to jail if it hadn't been for the true robber, Roland Clouser, and his dramatic confession. Clouser showed up during the trial, and Father Pagano walked free.

Let's see why all those witnesses were wrong andsee if that tells us something about programming.

Verify Your Intuitions

Roland Clouser and Father Pagano looked nothing alike. So what led the witnesses to make their erroneous statements?

First of all, politeness is a trait many people associate with a priest. This came up because the police mentioned that the suspect might be a priest. To make things worse, of all the suspects the police had, Father Pagano was the only one wearing a clerical collar (see *A reconciliation of the evidence on eyewitness testimony: Comments on McCloskey and Zaragoza [TT89]*).

The witnesses weren't necessarily to blame, either. The procedures in how eyewitness testimony was collected were also flawed. As *Forensic Psychology [FW08]* points out, the "police receive surprisingly little instruction on how to interview cooperative witnesses."

Law-enforcement agencies in many countries have learned from and improved their techniques thanks to case studies like this. New interview procedures focus on tape-recording conversations, comparing interview information with other pieces of evidence, and avoiding leading questions. These are things we could use as we look at our code, too.

In programming, our code is also cooperative—it's there to solve our problems. It doesn't try to hide or deceive. It does what we told it to do. So how do we treat our code as a cooperative witness while avoiding our own memory's traps?

Reduce Memory Biases with Supporting Evidence

A common memory bias is *misattribution*. Our memories are often sketchy. We may remember a particularly tricky design change well, but misremember when it occurred or even in what codebase. And as we move on, we forget the problem, and it comes back to haunt us or another developer later.

You need supporting data in other situations, too. On larger projects, you can't see the whole picture by yourself. The temporal coupling analysis we go over in this chapter lets you collect data across teams, subsystems, and related programs, such as automated system tests. Software is so complex that we need all the support we can get.

Learn the Modus Operandi of a Code Change

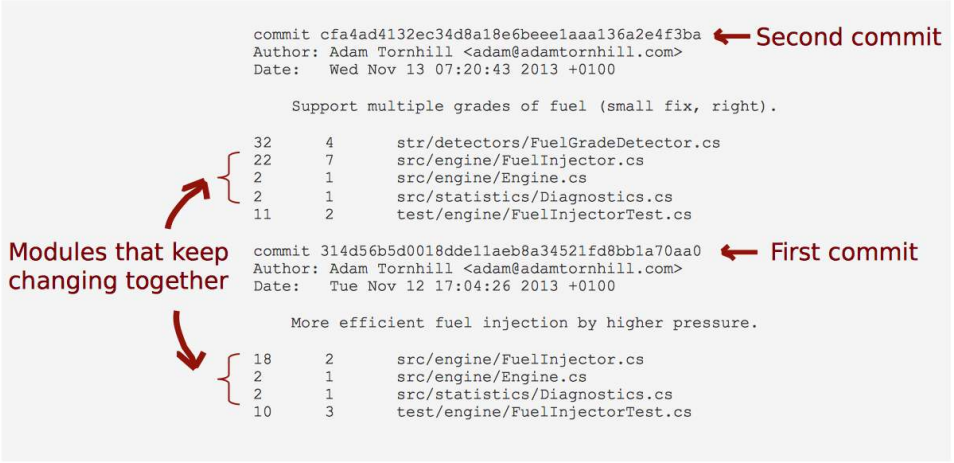
When we program, we're stuck with the same brain with its same tendencies to make mistakes. Sure, in programming we can go back and recheck the code. The problem is that we have to do that repeatedly—the sheer complexity of software makes it impossible to hold all of the details in our heads. That means our brain works with a simplified view. As soon as we drop details, we run the risk of missing something.

This is why we need to guide our decisions by objective data. The good part is that this data is based on how we actually work and interact with the system. Let's see how.

Link Commits to Detect Temporal Coupling

Remember the crash course in geographical profiling of crimes, back in *Learn Geographical Profiling of Crimes*, on page 16? We learned that linking related crimes allows us to make predictions and take possible counter-steps. We can do the same with code.

In programming, our version-control data allows us to trace changes over series of commits to spot patterns in the modifications. One prominent pattern is called temporal coupling, and you can see an example in the following figure.



Modules change together in temporal coupling. This is different from traditional coupling in that there may not be any explicit software dependencies between modules. There is a hidden, implicit dependency, which means a change in one predictably results in a change in the coupled module. Remember what happened in our initial fictional example with FuelInjector? Let's see how temporal coupling can help us detect and fix this problem.

Use Temporal Coupling to Reduce Bias

Law enforcement improved their interview processes. We use similar techniques: avoid leading questions, play back tape-recorded conversations, and compare interview information with other information. We just do it with code and not people.

Now, let me show you how temporal coupling looks. You'll get a high-level view of the concept, which makes it easier to apply in practice later.

See Temporal Coupling in a System

It's difficult to show temporal coupling with a single illustration. What would be best is video. Despite the advances in ebook technology, we're not quite there, so bear with me as I walk you through the following images.

Joe asks:

If I Know Where the Problems Are, Does Temporal Coupling Really Add Value?

A temporal coupling analysis complements but does not replace your expertise. This analysis is great for large codebases with multiple developers. I once analyzed a large project that I was involved with and found several unexpected cases of hidden dependencies. These dependencies had been costing time and effort as well as introducing bugs. Once they were uncovered, we redesigned the project.

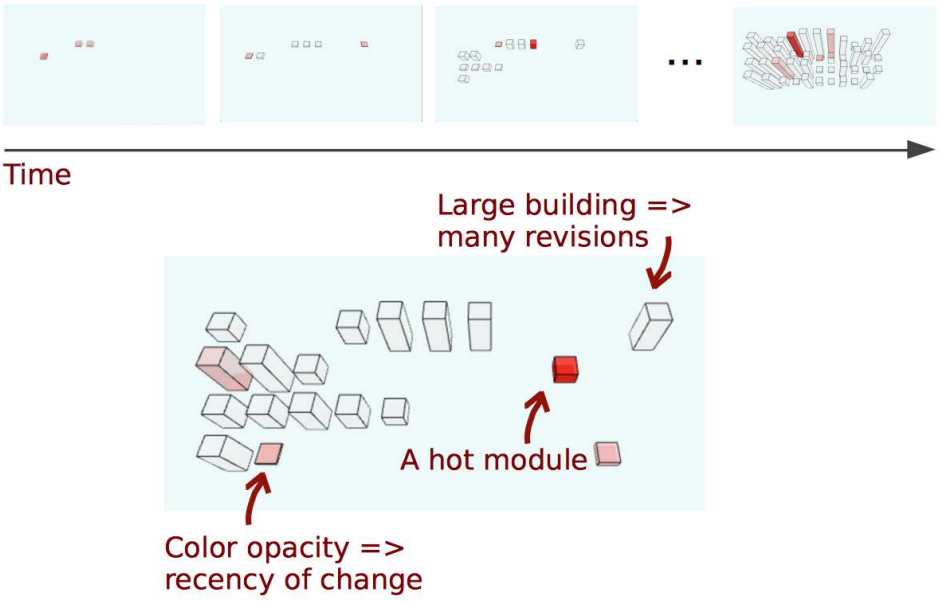
It's also a technique I've found useful in my own private projects. The analysis gives me a different view into the design and reveals things I missed.

It's also a helpful tool when considering design changes. By analyzing historical change patterns, we get an idea of how deep and far-reaching our proposed change will go.

If we look into the research on temporal coupling (or its synonyms, *change coupling* and *logical coupling*), we find several applications:

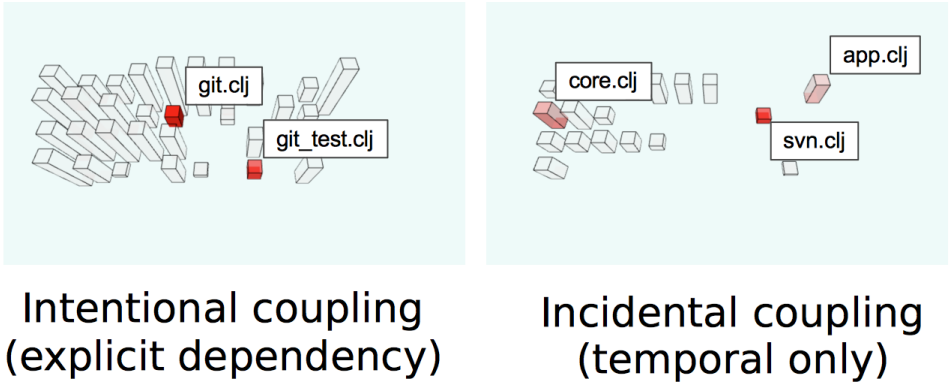
- One research team used visualization techniques to compare coupled modules against the specified software architecture. This allowed them to identify signs of structural decay (see *Animated Visualization of Software History using Evolution Storyboards [BH06]*).
- Another research team used temporal coupling as a code-recommendation guide. Using the results of such an analysis, they could recommend relevant source code for specific modification tasks (see *Predicting source code changes by mining change history [YMNC04]*).
- A study of an object-oriented system used temporal coupling to detect architectural weaknesses, such as poorly designed inheritance hierarchies (see *CVS release history data for detecting logical couplings [GK03]*).

In the chapters to come, you'll learn about these applications. The idea is that it's just impossible to keep track of everything that's happening in a codebase under heavy development. In a temporal coupling analysis, we have a tool that lets us monitor and react to costly problems early.



I replayed version-control data and animated the growing system to illustrate how Code Maat evolved. Each time a module changed, the size of its building grew a little. Tall buildings in the illustration have high change frequencies. To make it easier to spot patterns, I increased the opacity of the building's color every time it changed. As the hotspot cooled down, I decreased the opacity.

Looking long enough at this animation could drive you crazy, but you would spot some patterns. In the following figure, I highlight two patterns showing how modules are changing.



- *Explicit coupling*: git.clj and git_test.clj tend to change together. This is hardly surprising since the latter is a unit test on the former. In fact, we'd be surprised if the pattern wasn't there. A unit test always has a strong degree of direct coupling to the code under testing.
- *Temporal coupling*: The right-hand snapshot is more interesting: core.clj and svn.clj change together. It's interesting because there isn't any explicit dependency between them. You have to dig into the source code to figure out they are related. Congratulations, you've just detected a case of temporal coupling in Code Maat.

Temporal coupling can point to either expected co-changes, such as a module and its unit tests, or serious problems in design. Let's see what they are.

Understand the Reasons Behind Temporal Dependencies

Temporal coupling is a powerful interview tool for your codebase. It lets you identify design issues you cannot spot in the code alone. Once you've found them, the reasons behind the temporal coupling often suggest places to refactor, too:

- *Copy-paste*: The most common case of temporal coupling is copy-paste code. This one is straightforward to address; extract and encapsulate the common functionality.
- *Inadequate encapsulation*: Temporal coupling is related to encapsulation and cohesion. In the next chapter, you'll see temporal coupling that was the result of not isolating program arguments from application logic. Encapsulating the concept that varies would improve the design.
- *Producer-consumer*: Finally, temporal coupling may reflect different roles, such as a producer and consumer of specific information. In that case it's not obvious what to do, and it might not be a good idea to change the structure. In situations like this, we rely on our expertise to make an informed decision.

Findings like these are the main strengths of a temporal coupling analysis. They give us objective data on how our changes interact with the codebase and suggest new modular boundaries.

As we move on to perform an analysis in the next chapter, we'll see further refactoring support: a temporal coupling analysis also shows how severe and deep the necessary changes will go. This guides our design and reasoning upfront, too.

Prepare to Analyze Temporal Coupling

We started this chapter by discussing modules that keep changing together. We learned that they make maintenance harder, cost more, and open us up to more mistakes.

After that, we saw how our brain does its best to deceive us when it comes to remembering such complex, detailed information. Just as crime investigators have techniques to reduce bias, so do we. That's why we introduced the concept of temporal coupling as a way to interview our codebase about its past.

You learned that temporal coupling lets you detect hidden, implicit dependencies in your system and got ideas on why those dependencies might show up. You can use that information as objective data to guide your refactorings and redesigns.

With the theory fresh in our minds, let's move on and perform a temporal coupling analysis on our code.

CHAPTER 8

Detect Architectural Decay

In the previous chapter, you learned how temporal coupling detects hidden dependencies in your system. Now it's time to learn how to perform an analysis of temporal coupling on your code.

In this chapter, we'll analyze two systems of different sizes. The smaller project shows how temporal coupling can still give us fresh insights into the design even when we're very familiar with the code. The larger project shows how to detect architectural decay so that we can make improvements early in the process. You'll also see that the structures you're working with aren't always aligned with the official architecture.

Let's see how information-rich the change patterns in a system can be for our analysis.

Support Your Redesigns with Data

I once worked on a project with severe problems in its database access. Changes were awkward, they took longer than they should, and bugs swarmed like mosquitoes at a Swedish barbecue.

Learning from mistakes is important, so I decided to redesign the worst parts of the database layer. Something interesting happened. Even though the database layer was in better shape, developers still complained about how fragile and unstable it was. Changes still broke unrelated features. What went wrong? Did I mess up?

While the database improved, it turned out that wasn't where the true problems were. The database was just the messenger subtly warning us about temporal coupling (and we shot the messenger).

Other parts of the system unexpectedly depended on the data storage. The true problem was in automatic system tests. A minor change to the data format

triggered a cascade of changes to the test scripts. This wasn't obvious because the scripts didn't explicitly call the database code.

After reading the previous chapter, you now can see how a temporal coupling analysis could've helped us find this problem earlier. Redesigns are about minimizing risk and prioritizing areas of code that have the largest impact on the work we're doing now. Get it wrong like we did, and you will miss an opportunity to make genuine improvements to your code. Let's see how we can use temporal coupling to avoid these mistakes.

Analyze Temporal Coupling

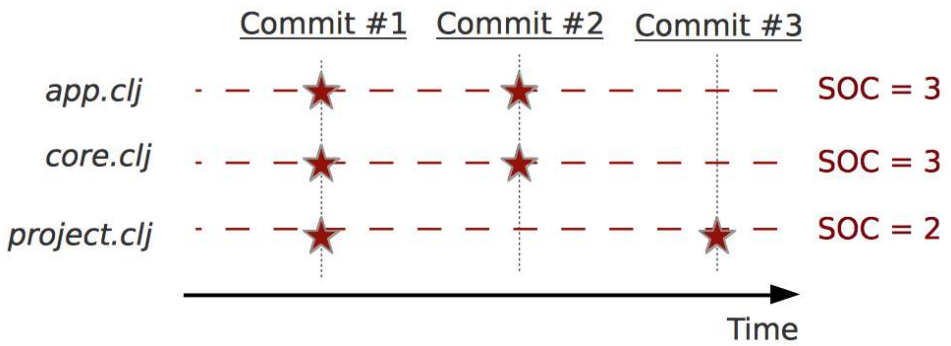
In Chapter 7, *Treat Your Code As a Cooperative Witness*, on page 67, we said that temporal coupling can be an interview tool for your codebase. The first step in an interview is to know who you should talk to.

Let's use *sum of coupling* analysis to find our first code witness.

Use Sum of Coupling to Identify the Modules to Inspect

You've already seen that there are different reasons for modules to be coupled. Some couples, such as a unit and its unit test, are valid. So modules with the highest degree of coupling may not be the most interesting to us. Instead, we want modules that are architecturally significant. A sum of coupling analysis finds those modules.

Sum of coupling looks at how many times each module has been coupled to another one in a commit and sums it up. For example, in the following figure, you see that module app.clj changed with both core.clj and project.clj in Commit #1, but just with core.clj in Commit #2. Its sum of coupling is three.



The module that changes most frequently together with others must be important and is a good starting point for an investigation. Let's try it out on

Code Maat by reusing the the logfile we mined in Chapter 3, *Creating an Offender Profile*, on page 23.

Move into the top-level directory in your Code Maat repository and type the following command:

```
prompt> maat -l maat_evo.log -c git -a soc
entity,soc
src/code_maat/app/app.clj,105
test/code_maat/end_to_end/scenario_tests.clj,97
src/code_maat/core.clj,93
project.clj,74
...
```

You can see that this command uses the same format we saw in the earlier hotspot analysis. The only difference is that we're requesting `-a soc` (sum of coupling) instead.

We see that `app.clj` changes the most with other modules. Let's keep an eye on `app.clj` as we dive deeper.

Measure Temporal Coupling

At this point you know that `app.clj` is the module with the most temporal coupling. The next step is to find out which modules it's coupled to. We use Code Maat for this analysis:

```
prompt> maat -l maat_evo.log -c git -a coupling
entity,coupled,degree,average-revs
src/code_maat/parsers/git.clj,test/code_maat/parsers/git_test.clj,83,12
src/code_maat/analysis/entities.clj,test/code_maat/analysis/entities_test.clj,76,7
src/code_maat/analysis/authors.clj,test/code_maat/analysis/authors_test.clj,72,11
...
```

The command line is identical to the one you just used, with the exception that we're requesting `-a coupling` instead. The resulting `.CSV` output contains plenty of information:

1. *entity*: This is the name of one of the involved modules. Code Maat always calculates pairs.
2. *coupled*: This is the coupled counterpart to the *entity*.
3. *degree*: The degree specifies the percent of shared commits. The higher the number, the stronger the coupling. For example, `git.clj` and `git_test.clj` change together in 83 percent of all commits.
4. *average-revs*: Finally, we get a weighted number of total revisions for the involved modules. The idea here is that we can filter out modules with too few revisions to avoid bias.

You see a typical pattern in the output: each unit changes together with its unit test (e.g. git.clj and git_test.clj, entities.clj and entities_test.clj).

This kind of temporal coupling is expected and not a problem. Code Maat was developed with test-driven development, so I'd say that getting any other result would've been a problem. Just plain old physical coupling—nothing too exciting here.

Things get interesting a bit farther down:

```
prompt> maat -l maat_evo.log -c git -a coupling
...
src/code_maat/app/app.clj,src/code_maat/core.clj,60,23
src/code_maat/app/app.clj,test/code_maat/end_to_end/scenario_tests.clj,57,23
...
```

We see that app.clj changed with core.clj 60 percent of the time and with scenario_tests.clj 57 percent of the time. There's no way to tell why just from the names alone, but 60 percent is a high degree of coupling. We are talking about every second (or so), change in app.clj triggering a change in two other modules. That can't be good. Let's investigate why.

Check Out the Evolution Radar

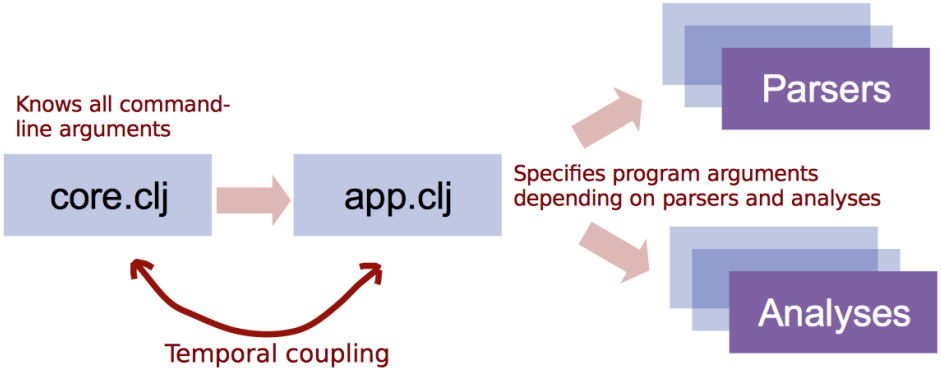
In a large codebase, a temporal coupling analysis sparks an explosion of data. Code Maat resolves that by allowing us to specify optional thresholds. The research tool *Evolution Radar*[1] takes a different approach and lets us zoom in and out to the level of detail we're interested in. So check out the tool and take inspiration.

Investigate Temporal Couples

Once we make such a finding, we need to drill down into the code. Because all changes are recorded in our version-control system, we can perform a diff on the modules. I'd recommend focusing on the shared commits and look for recurring modification patterns within those commits.

Code Maat is written in Clojure. Although an exciting language, it's far outside the scope of this book. So let's stay with temporal coupling, and allow me to walk you through the design to spot the problems.

1. http://www.inf.usi.ch/phd/dambros/tools/evoradar.php



I'm a bit ashamed to admit that core.clj is the command-line interface of Code Maat. (I changed it later to a better name.) It parses the arguments you give it, converts them to a Clojure representation, and forwards them to app.clj.

app.clj glues the program together by mapping the given arguments to the correct invocations of parsers, analyses, and output formats. As you can see, the program arguments cause the coupling; every time a new argument is added, two distinct modules have to evolve to know about it.

So, your first takeaway is actually a reminder about the power of names that you learned about in Chapter 5, *Judge Hotspots with the Power of Names*, on page 47. With proper naming, we'd have a better entry point for our manual code inspection. Second, we failed to encapsulate a concept that varies. If we extract the knowledge of all command-line arguments from app.clj, we break the coupling and make the code easier to evolve and maintain.

Use Temporal Coupling for Design Insights

The analysis on Code Maat illustrates how we can use temporal coupling analysis on small projects. Code Maat (which I wrote to learn Clojure during my daily commute) is a single-developer project with less than 2,000 lines of code.

Such small projects don't need a hotspot analysis. We already know which modules are hard to change. Temporal coupling is different because it provides insights into our design. We get active feedback on our work so that we can spot improvements we hadn't even thought of.

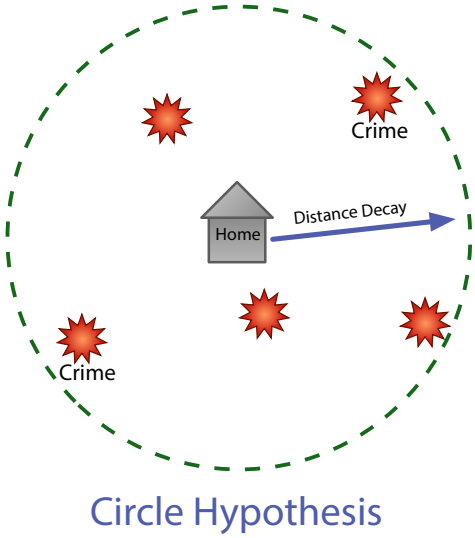
Keep Your Temporal Coupling Algorithms Simple

The algorithm we've used so far isn't the only kid in town. Temporal coupling means that some entities change together over time. But there isn't any formal

definition of what *change together* means. In research papers, you'll find several alternative measures.

One typical alternative adds the notion of time to the algorithm; the degree of coupling is weighted by the age of the commits. The idea is to prioritize recent changes over changes in the more distant past. A relationship thus gets weaker with the passage of time. However, as you'll see soon when we discuss software defects, a time parameter doesn't necessarily improve the metric.

The algorithm that Code Maat implements, the percent of shared commits, is chosen because when faced with several alternatives that seem equally good, simplicity tends to win. The Code Maat measure is straightforward to implement and, more importantly, intuitive to reason about and verify.



Interestingly enough, simplicity may win in criminal investigations, too. In a fascinating study, researchers trained people on two simple heuristics for predicting the home location of criminals:

- *Distance decay*: Criminals do not travel far from their homes to offend. Thus, crimes are more likely closer to an offender's home and less likely farther away.
- *Circle hypothesis*: Many serial offenders live within a circle defined by the criminals' two farthest crime locations.

Using these simple principles allowed the participants to predict the likely home location of serial killers with the same accuracy as a sophisticated geographical profiling system. (See *Applications of Geographical Offender Profiling [CY08]*.) We build the techniques in this book on the same kind of simplicity.

Know the Limitations of Temporal Coupling

Our simple definition of temporal coupling as modules that change in the same commit works well. Often, that definition takes us far enough to identify unexpected relationships in our system. But in larger organizations, our

measure is too narrow. When multiple teams are responsible for different parts of the system, the temporal period of interest is probably counted in days or even weeks. We'll address this problem in Chapter 12, *Discover Organizational Metrics in Your Codebase*, on page 133, where you'll learn to group multiple commits into a logical change set based on a custom timespan.

Another problem with the measure is that we're limited to the information contained in commits. We may miss important coupling relationships that occur *between* commits. The solution to this problem requires hooks into our text editors and our IDE to record precise information on our code interactions. Tools like that are under active research.

Yet another bias is moving and renaming modules. While version-control systems track renames, Code Maat does not. (If I ever turn Code Maat into a commercial product, that's a feature I'd add.) It sounds more limiting than it actually is: problematic modules tend to remain where they are. The good thing is that because we lose some of the supporting information, the results we get are more likely to point to true problems. Consider renaming the module as a reset switch triggered by refactoring.

Catch Architectural Decay

Temporal coupling has a lot of potential in software development. We can spot unexpected dependencies and suggest areas for refactoring.

Temporal coupling is also related to software defects. There are multiple reasons for that. For example, a developer may forget to update one of the (implicitly) coupled modules. Another explanation is that when you have multiple modules whose evolutionary lifelines are intimately tied, you run the risk of unexpected feature interactions. You'll also soon see that temporal coupling often indicates architectural decay. Given these reasons, it's not surprising that a high degree of temporal coupling goes with high defect rates.

Temporal Coupling and Software Defects

Researchers found that different measures of temporal coupling outperformed traditional complexity metrics when it came to identifying the most defect-prone modules (see *On the Relationship Between Change Coupling and Software Defects [DLR09]*). What's surprising is that temporal coupling seems to be particularly good at spotting more severe bugs (major/high-priority bugs).

The researchers made another interesting finding when they compared the bug-detection rate of different coupling measures.

Temporal Coupling and Software Defects

Some measures included time awareness, effectively down-prioritizing older commits and giving more weight to recent changes. The results were counterintuitive: the simpler sum of coupling algorithm that you learned about in this chapter performed better than the more sophisticated time-based algorithms.

My guess is that the time-based algorithms performed worse because they're based on an assumption that isn't always valid. They assume code gets better over time by refactorings and focused improvements. In large systems with multiple developers, those refactorings may never happen, and the code keeps on accumulating responsibilities and coupling. Using the techniques in this chapter, we have a way to detect and avoid that trap. And now we know how good the techniques are in practice.

Enable Continuing Change

Back in Chapter 6, *Calculate Complexity Trends from Your Code's Shape*, on page 55, we learned about Lehman's law of increasing complexity. His law states that we must continuously work to prevent a "deteriorating structure" of our programs as they evolve. This is vital because every successful software product will accumulate more features.

Lehman has another law, the *law of continuing change*, which states a program that is used "undergoes continual change or becomes progressively less useful" (see *On Understanding Laws, Evolution, and Conservation in the Large-Program Life Cycle [Leh80]*).

There's tension between these two laws. On one hand, we need to evolve our systems to make them better and keep them relevant to our users. At the same time, we don't want to increase the complexity of the system.

One risk with increased complexity is features interacting unexpectedly. We make a change to one feature, and an unrelated one breaks. Such bugs are notoriously hard to track down. Worse, without an extensive regression test suite, we may not even notice the problem until later, when it's much more expensive to fix.

To prevent horrors like that from happening in our system, let's see how we can use temporal coupling to track architectural problems and stop them from spreading in our code.

Identify Architecturally Significant Modules

In the following example, we're going to analyze a new codebase. Craft.Net[2] is a set of Minecraft-related .NET libraries. We're analyzing this project because it's a fairly new and cool project of suitable size with multiple active developers.

To get a local copy of Craft.Net, clone its repository:

```
prompt> git clone https://github.com/SirCmpwn/Craft.Net.git
```

Let's perform the trend analysis step by step so that we can understand what's happening. Each step is nearly identical; the time period is the only thing that changes. We can automate this with a script later. Let's find the first module to focus on.

Move into the Craft.Net directory and perform a sum of coupling analysis:

```
prompt> git log --pretty=format:'[%h] %an %ad %s' --date=short --numstat \
 --before=2014-08-08 > craft_evo_complete.log
prompt> maat -l craft_evo_complete.log -c git -a soc
entity,soc
Craft.Net.Server/Craft.Net.Server.csproj,685
Craft.Net.Server/MinecraftServer.cs,635
Craft.Net.Data/Craft.Net.Data.csproj,521
Craft.Net.Server/MinecraftClient.cs,464
...
```

Notice how we first generate a Git log and then feed that to Code Maat. Sure, there's a bit of Git magic here, but nothing you haven't seen in earlier chapters. You can always refer back to Chapter 3, *Creating an Offender Profile*, on page 23, if you need a refresher on the details.

When you look for modules of architecural significance in the results, ignore the C# project files (.csproj). The first real code module is MinecraftServer.cs. As you see, that class has the most cases of temporal coupling to other modules. Looks like a hit.

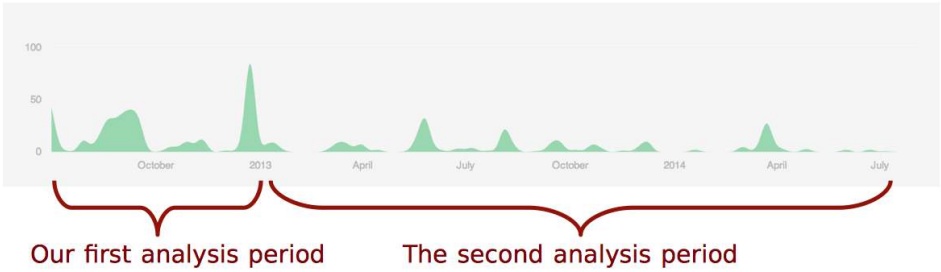
The name of our code witness, MinecraftServer, is also an indication that we've found the right module; a MinecraftServer sounds like a central architectural part of any, well, Minecraft server. We want to ensure that the module stays on track over time. Here's how we do that.

2. https://github.com/SirCmpwn/Craft.Net

Perform Trend Analyses of Temporal Coupling

To track the architectural evolution of the MinecraftServer, we're going to perform a trend analysis. The first step is to identify the periods of time that we want to compare.

The development history of Craft.Net goes back to 2012. There was a burst of activity that year. Let's consider that our first development period.



To perform the coupling analysis, let's start with a version-control log for the initial period:

```
prompt> git log --pretty=format:'[%h] %an %ad %s' --date=short --numstat \
  --before=2013-01-01 > craft_evo_130101.log
```

We now have the evolutionary data in craft_evo_130101.log. We use the file for coupling analysis, just as we did earlier in this chapter:

```
prompt> maat -l craft_evo_130101.log -c git -a coupling > craft_coupling_130101.csv
```

The result is stored in craft_coupling_130101.csv. That's all we need for our first analysis period. We'll look at it in a moment. But to spot trends we need more sample points.

In this example, we'll define the second analysis period as the development activity in 2013 until 2014. Of course, we could use multiple, shorter periods, but the GitHub activity shows that period contains roughly the same amount of activity. So for brevity, let's limit the trend analysis to just two sample points.

The steps for the second analysis are identical to the first. We just have to change the filenames and exclude commit activity before 2013. We can do both in one sweep:

```
prompt> git log --pretty=format:'[%h] %an %ad %s' --date=short --numstat \
  --after=2013-01-01 --before=2014-08-08 > craft_evo_140808.log
prompt> maat -l craft_evo_140808.log -c git -a coupling > craft_coupling_140808.csv
```

We now have two sampling points at different stages in the development history. Let's investigate them.

Investigate the Trends

When we perform an analysis of our codebase, we want to track the evolution of all interesting modules. To keep this example short, we'll focus on one main suspect as identified in the sum of coupling analysis: the MinecraftServer module. So let's filter the data to inspect its trend.

I opened the result files, craft_coupling_130101.csv and craft_coupling_140808.csv, in a spreadsheet application and removed everything but the modules coupled to MinecraftServer to get the filtered analysis results.

Temporal coupling to MinecraftServer in 2012

entity	degree	average-revs
Craft.Net.Server/MinecraftClient.cs	38	89
Craft.Net.Server/Craft.Net.Server.csproj	41	91

Temporal coupling to MinecraftServer in 2014

entity	degree	average-revs
source/Test/TestServer/Program.cs	39	21
source/Craft.Net.Server/RemoteClient.cs	68	22
source/Craft.Net.Server/Handlers/PacketHandlers.cs	48	17
source/Craft.Net.Server/Craft.Net.Server.csproj	47	19
source/Craft.Net.Server/Handlers/LoginHandlers.cs	42	17
source/Craft.Net.Server/EntityManager.cs	41	17
source/Craft.Net.Anvil/Level.cs	36	17
Craft.Net/Packets.cs	37	14

Time

There's one interesting finding in 2012: the MinecraftServer.cs is coupled to MinecraftClient.cs. This seems to be a classic case of temporal coupling between a producer and a consumer of information, just as we discussed in *Understand the Reasons Behind Temporal Dependencies*, on page 75. When we notice a case like that, we want to track it.

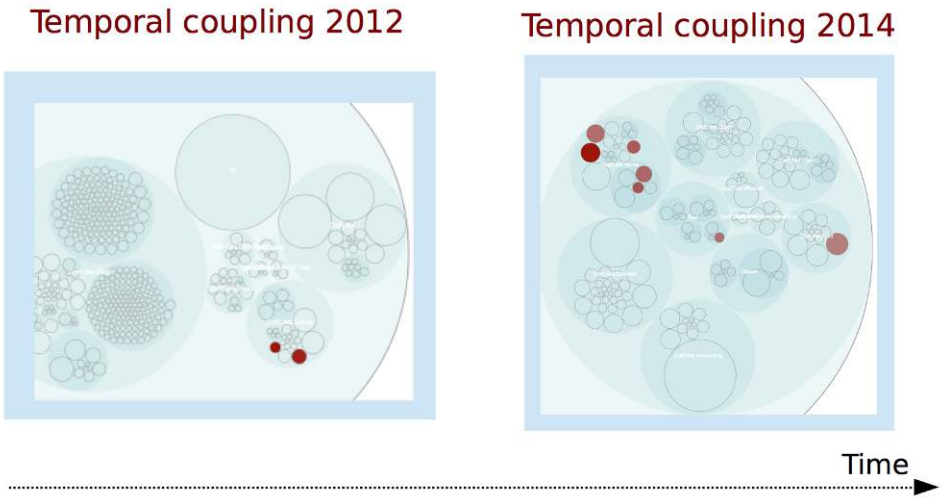
Forward to 2014. The coupling between server and client isn't present a year and a half later, but we have other problems. As you can see, the MinecraftServer has accumulated several heavy temporal dependencies compared to its cleaner start in the initial development period.

When that happens, we want to understand why and look for places to improve. Let's see how.

React to Structural Trends

The following figure presents a visual view of the architectural decay we just spotted. It's the same enclosure diagrams we used back in Chapter 4, *Analyze*

Hotspots in Large-Scale Systems, on page 35, but now they're illustrating the modules coupled to MinecraftServer at two different points in time.



The obvious increase in temporal coupling says there are more modules that have to change with the MinecraftServer in 2014 than earlier in the development history. Note that the number of coupled modules isn't a problem in itself. To classify a temporal coupling, you need to look at the architectural boundaries of the coupled modules.

When the coupled modules are located in entirely different parts of the system, that's structural decay. Our data in the trend table on page 87 shows one obvious case in 2014: Craft.Net.Anvil/Level.cs.

That coupling, together with the growing trend, suggests that our MinecraftServer has been accumulating responsibilities.

Remember how we initially discussed code changes that seem to break unrelated features? The risk with the trend we see here is that it leaves the system vulnerable to such unexpected feature interactions.

If allowed to grow, increased temporal coupling leads to fragile systems. As you saw earlier, temporal coupling has a high correlation with defects. That's why we want to integrate the analysis into a team's workflow. Let's see how.

Use a Storyboard to Track Evolution

The trend analysis we just performed is reactive. It's an after-the-fact analysis. The results are useful because they help us improve, but we can do even better.

With more activity, you want more sample points. So why not make it a habit to perform regular analyses on the projects you work on?

If you work iteratively, perform the analyses in each iteration. This approach has several advantages:

- You spot structural decay immediately.
- You see the structural impact of each feature as you work with it.
- You make your evolving architecture visible to everyone on the team.

I recommend that you visualize the result of each analysis, perhaps as in Figure , , on page 88, print them all out, and put them on a storyboard for each iteration.

Think back to our initial example on automated tests with nasty implicit couplings to a database. With an evolutionary storyboard, we'd spot the decay as soon as we noticed the pattern—a few iterations at most, and that's it.

An iterative trend analysis of temporal coupling is a low-tech approach that helps us improve. It also has the notable advantage of putting focus on the right parts of the system. As such, an evolutionary storyboard is invaluable to complement and stimulate design discussions with peers.

If you find as much promise in this approach as I do, check out the article *Animated Visualization of Software History using Evolution Storyboards [BH06]*. The authors are the pioneers of the storyboard idea, and their paper shows some cool animations of growing systems.

Scale to System Architectures

This chapter started with a sum of coupling analysis. With that analysis, we identified the architecturally significant modules. We also noted that those modules aren't necessarily the ones we'd expect from our formal specification or design.

After that, we saw how a temporal coupling analysis gives us information we cannot extract from the code alone. It's information that gives us design insights and refactoring directions. When used as a refactoring guide, we can assume that modules that have changed together in the past are likely to

continue to change together. We looked at that in our second analysis of Craft.Net.

You then learned to spot architectural decay by applying trend analyses to the coupling. Finally, you learned how to track potential decay with an evolutionary storyboard.

With temporal coupling behind us, we've completed our initial set of analysis methods. Before we move on to discuss teams and social dynamics, we're going to build on what we've learned so far. Until now, we have limited the analyses to individual files. But now you'll see how temporal coupling scales to system architecture, too. That will be exciting!

CHAPTER 9

Build a Safety Net for Your Architecture

You just learned about temporal coupling of modules. What you may not have realized is that this analysis scales to architectures, too. In this chapter, you'll learn to analyze architecture with respect to your team's modification patterns.

We'll start with an analysis of automated system tests before we extend the concept to other parts of your architecture in the next chapter. We start with tests because they're frequently added as an afterthought, almost like a hidden architectural layer. You'll also see how you can set up early warning systems to detect when automated tests go wrong. In the process, you'll learn about problem-solving and how it relates to programming. Let's dive in and see what hidden secrets we can cover in our architecture.

Know What's in an Architecture

If someone approaches you on a dark street corner and asks if you're interested in software architecture, chances are he'll pull out a diagram. It will probably look UML-like, with a cylinder for the database and lots of boxes connected by lines. It's a structure—a static snapshot of an ideal system.

But architecture goes beyond structure, and just a blueprint isn't enough. We should treat architecture as a set of principles rather than as a specific collection of modules. Let's think of architecture as principles that help us reason and navigate large-scale systems. Breaking principles is expensive. Let me illustrate with a short story.

View Your Automated Tests as Architecture

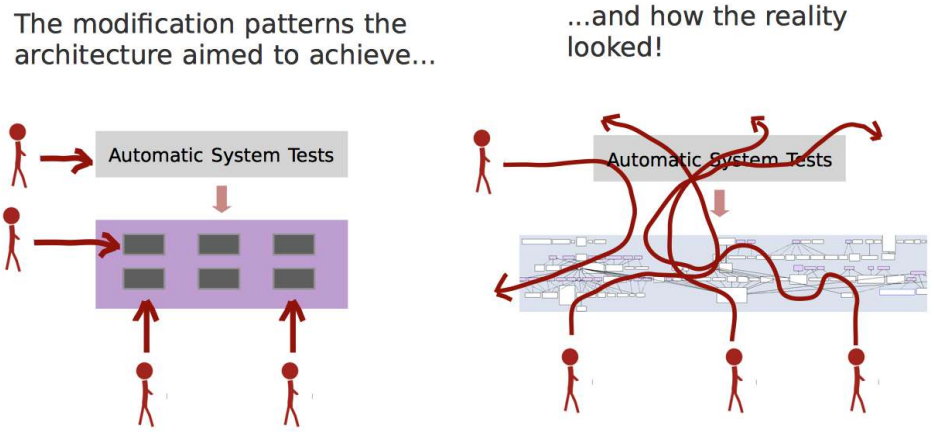
Do you remember my war story in the previous chapter? The one about automated system tests that depended upon the data storage? Like so many other failed designs, this one started with the best of intentions.

The first iterations went fine. But we soon noticed that new features started to become expensive to implement. What ought to be a simple change suddenly involved updating *multiple* high-level system tests. Such a test suite is counterproductive because it makes change harder. We found out about these problems by performing the same kind of analysis you'll learn about in this chapter. We also made sure to build a safety net around our tests to prevent similar problems in the future. Let's see why it's needed.

Automated tests becoming mainstream is a promising trend. When we automate the mundane tasks, we humans can focus on real testing, where we explore and evaluate the system. Test automation also makes changes to the system more predictable. We get a safety net when modifying software, and we use the scripts to communicate knowledge and drive additional development. While we all know these benefits, we rarely talk about the risks and costs of test automation. Automated tests, particularly on the system level, are hard to get right. And when we fail, these tests create a time sink, halting all progress.

Test scripts are architecture, too—albeit an often neglected aspect. Like any architectural boundary, a good test system should encapsulate details and avoid depending on the internals of the code being tested. We want to be able to refactor the implementation without affecting how the tests run. If we get this wrong, we lose the predictability advantage that a good test suite provides when we're modifying code.

In addition to the technical maintenance challenge, as the following figure shows, such tests lead to a significant communication and coordination overhead. We developers now risk breaking each other's changes.



Automated tests are no different from any other subsystem. The architecture we choose must support the kind of changes we make. This is why you want to track your modification patterns and ensure that they are supported by your design. Here's how.

Analyze the Evolution on a System Level

You've already learned to analyze temporal coupling between individual modules. Now we're raising the abstraction level to focus on system boundaries. We start with just two boundaries: the production code and the test code.

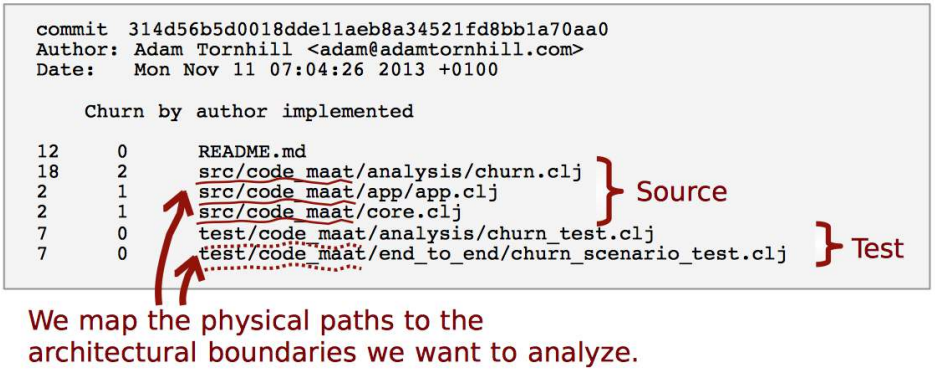
Specify Your Architectural Boundaries

The first step is to define application code and test code. In Code Maat, which we're returning to for this analysis, the definition is simple: everything under the src/code_maat directory is application code, and everything located in test/code_maat is test code.

Once we've located the architectural boundaries, we need to tell Code Maat about them. We do that by specifying a *transformation*. Open a text editor and type in the following text:

```
src/code_maat => Code
test/code_maat => Test
```

The text specifies how Code Maat translates files within physical directories to logical names. You can see an example of how individual modifications get grouped in the following figure.



Save your transformations in a file named maat_src_test_boundaries.txt and store it in your Code Maat repository root. You're now ready to analyze.

We perform an architectural analysis with the same set of commands we've been using all along. The only difference is that we must specify the transformation file to use. We do that with the -g flag:

```
prompt> maat -l maat_evo.log -c git -a coupling -g maat_src_test_boundaries.txt
entity,coupled,degree,average-revs
Code,Test,80,65
```

The analysis results are delivered in the same format used in the previous chapter. But this time Code Maat categorizes every modified file into either Code or Test before it performs the analysis.

The results indicate that our logical parts Code and Test have a high degree of temporal coupling. This might be a concern. Are we getting ourselves into an automated-test death march where we spend more time keeping tests up to date than evolving the system? We cannot tell from the numbers alone. So let's look at the factors we need to consider to interpret the analysis result.

Interpret the Analysis Result

Our analysis results tells us that in 80 percent of all code changes we make, we need to modify some test code as well. The results don't tell us how much we have to change, how deep those changes go, or what kind of changes we need. Instead, we get the overall change pattern. To interpret it, we need to know the context of our product:

- What's the test strategy?
- Which type of tests are automated?
- On what level do we automate tests?

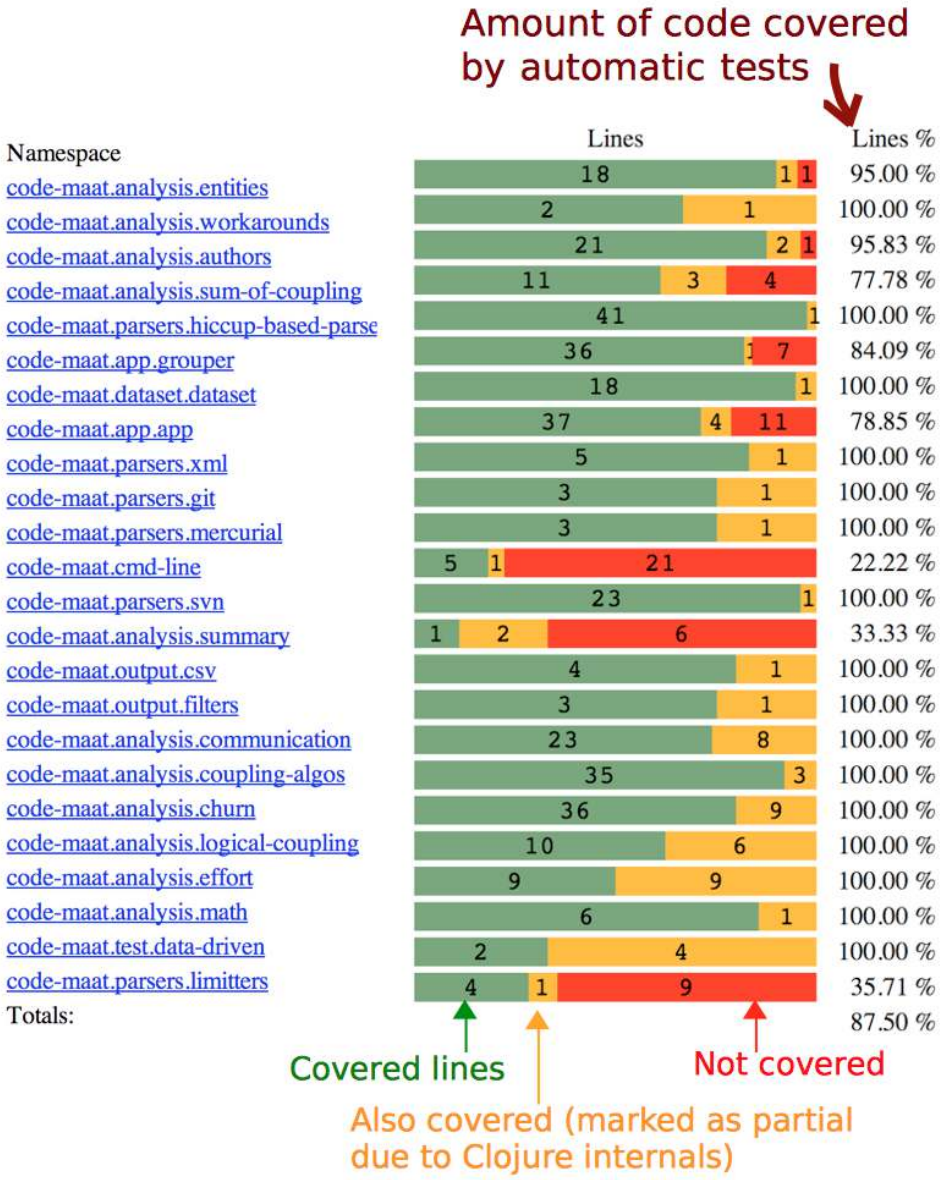
Let's see how Code Maat answers those questions.

As you can see in the test coverage figure on page 95, we try to automate as much as we can in Code Maat.

Code Maat has a fairly high code coverage (that is, if we ignore the embarrassing, low-coverage modules such as code-maat.cmd-line and code-maat.analysis.summary that I wish I'd written tests for *before* I published this data). That coverage has a price. It means our tests have many reasons to change. Here's why.

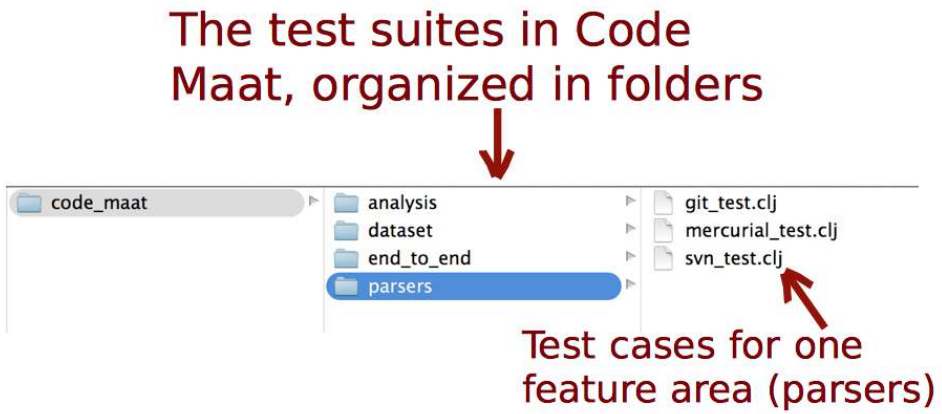
Differentiate Between the Level of Tests

In Code Maat, the partitioning between tests and application code isn't a stable architectural boundary; you identified a temporal coupling of 80 percent. That means they'll change together most of the time.



But our current analysis has a limitation. Code Maat uses both unit tests and system-level tests. In our analysis, we grouped them all together. Let's see what happens when we separate the different types of tests.

If you look into the folder test/code_maat of your Code Maat repository, you'll find four folders, shown in the following figure. Each of them contains a particular suite of test cases. Let's analyze them by their individual boundaries.



Open a text editor, enter the following mapping, and save it as maat_src_test_boundaries.txt:

```
src/code_maat => Code
test/code_maat/analysis => Analysis Test
test/code_maat/dataset => Dataset Test
test/code_maat/end_to_end => End to end Tests
test/code_maat/parsers => Parsers Test
```

With the individual test groups defined, launch a coupling analysis:

```
prompt> maat -l maat_evo.log -c git -a coupling \
 -g maat_src_detailed_test_boundaries.txt
entity,coupled,degree,average-revs
Code,End to end Tests,42,50
Analysis Test,Code,42,49
Code,Parsers Test,41,49
```

These results give us a more detailed view:

- Analysis Test and Parsers Test contain unit tests. These tests change together with the application code in about 40 percent of all commits. That's a reasonable number. Together with the coverage results we saw earlier, it means we keep the tests alive, yet manage to avoid having them change too frequently. A higher coupling would be a warning sign that the tests depend on implementation details, and that we're testing the code on the wrong level. Again, there are no right or wrong numbers; it all depends on your test strategy. For example, if you use test-driven development, you should expect a higher degree of coupling to your unit tests.

Joe asks:

Code Coverage? Seriously, Is It Any Good?

Code coverage is a simple technique to gain feedback. However, I don't bother with analyzing coverage until I've finished the initial version of a module. But then it gets interesting. The feedback you get is based on your understanding of the application code you just wrote. Perhaps there's a function that isn't covered or a branch in the logic that's never taken?

To get the most out of this measure, try to analyze the cause behind low coverage. Sometimes it's okay to leave it as is, but more often you'll find that you've overlooked some aspect of the solution.

The specific coverage figure you get is secondary; while it's possible to write large programs with full coverage, it's not an end in itself, nor is it meaningful as a general recommendation. It's just a number. The value you get from code coverage is by the implicit code review you perform when you study uncovered lines.

Finally—and this is a double-edged sword—code coverage can be used for gamification. I've seen teams and developers compete with code coverage high scores. To a certain degree this is good. I found it useful when introducing test automation and getting people on a team to pay attention to tests. Who knew automated tests could bring out the competitiveness in us?

- `Dataset Test` was excluded by Code Maat because its coupling result was below the default threshold of interest. (You can fine-tune these parameters—look at Code Maat's documentation.[1])
- `End to end Tests` define system-level tests. These change together with the application code in 40 percent of all commits. This is a fairly high number compared to the unit tests—we'd expect the higher-level tests to be more stable and have fewer reasons to change. Our data indicate otherwise. Is there a problem?

Encapsulate Test Data

It turns out there's a reason that almost every second change to the application code affects the system-level tests, too. And, unfortunately for me as the programmer responsible, it's not a good reason. So, let me point this out so you can avoid the same problem in your own codebase.

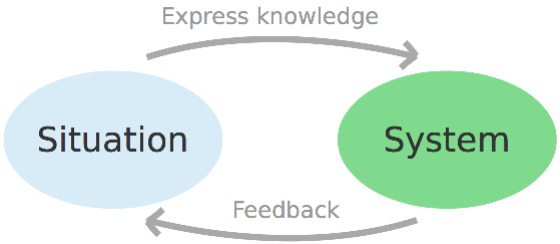
The system tests in Code Maat are based on detailed test data. Most of that data is collected from real-world systems. I did a lot of experimentation with

1. https://github.com/adamtornhill/code-maat

different data formats during the early development of Code Maat. Each time I changed my mind about the data format, the system tests had to be modified, too.

So, why not choose the right test data from the beginning?

That would be great, wouldn't it? Unfortunately, you're not likely to get there. To a large degree, programming is problem-solving. And as the following figure illustrates, human problem-solving requires a certain degree of experimentation.



The preceding figure presents a model from educational psychology. (See *Understanding and solving word arithmetic problems [KG85]*.) We programmers face much the same challenges as educators: we have to communicate with programmers who come to the code after we've left. That knowledge is built by an iterative process between two mental models:

- The *situation model* contains everything you know about the problem, together with your existing knowledge and problem-solving strategies.
- The *system model* is a precise specification of the solution—in this case, your code.

You start with an incomplete understanding of the problem. As you express that knowledge in code, you get feedback. That feedback grows your situation model, which in turn makes you improve the system model. It means that human problem-solving is inherently iterative. You learn by doing. It also means that you don't know up front where your code ends up.

Remember those architectural principles we talked about earlier in this chapter? This is where they help. Different parts of software perform different tasks, but we need consistency to efficiently understand the code. Architecture specifies that consistency.

This model of problem-solving above lets us define what makes a good design: one where your two mental models are closely aligned. That kind of design is easier to understand because you can easily switch between the problem and the solution.

The take-away is that your test data has to be encapsulated just like any other implementation detail. Test data is knowledge, and we know that in a well-designed system, we don't repeat ourselves.

Violating the Don't Repeat Yourself (DRY) principle with respect to test data is a common source of failure in test-automation projects. The problem is sneaky because it manifests itself slowly over time. We can prevent this, though. Let's see how.

Create a Safety Net for Your Automated Tests

Remember how we monitored structural decay back in *Use a Storyboard to Track Evolution*, on page 89? We're going to set up a similar safety net for automated tests.

Our safety net is based on the change ratio between the application code and the test code. We get that metric from an analysis of change frequencies, just like the analyses you did back in Part I.

Monitor Tests in Every Iteration

To make it into a trend analysis, we need to define our sampling intervals. I recommend that you obtain a sample point in each iteration or at least once a month. In case you're entering an intense period of development (for example, around deadlines—they do bring out the worst in people), perform the analysis more frequently.

To acquire a sample point, just specify the same transformations you used earlier in this chapter:

```
src/code_maat => Code
test/code_maat => Test
```

Ensure that your transformation is saved in the file maat_src_test_boundaries.txt in your Code Maat repository.

Now you just have to instruct Code Maat to use your transformations in the revisions analysis:

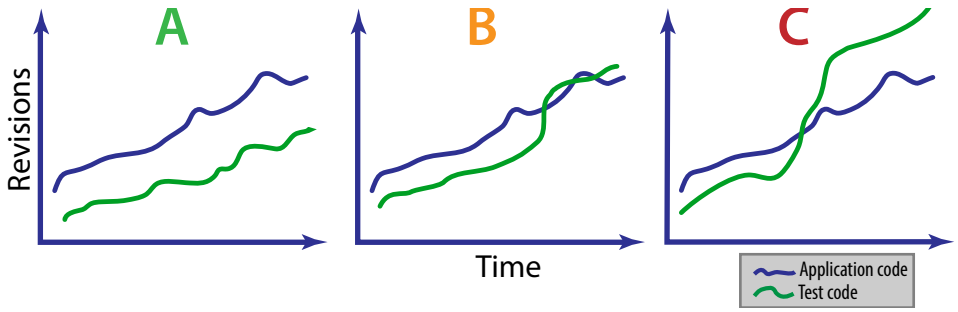
```
prompt> maat -l maat_evo.log -c git -a revisions -g maat_src_test_boundaries.txt
entity,n-revs
Code,153
Test,91
```

The analysis results show that in this development period, we've modified application code in 153 commits and test code in 91. Let's see why that's interesting.

Date	Code Growth	Test Growth
2013-09-01	153	91
2013-10-01	183	122
2013-11-01	227	137

Track Your Modification Patterns

If we continue to collect sample points, we're soon able to spot trends. It's the patterns that are interesting, particularly with respect to the relationship between application code and test code growth. Let's look at some trends to see what they tell us. You can see the typical patterns you can expect (although I do hope you never meet alternative C) in the following figure. Each case shows how fast the test code evolves compared to the application code. Note that we're talking system-level tests now.



In Case A, you see an ideal change ratio. The test code is kept alive and in sync with the application. Most of the effort is spent in the application code.

Case B is a warning sign. The test code suddenly got more reasons to change. When you see this pattern, you need to investigate. There may be legitimate reasons: perhaps you're focusing refactoring efforts on your test code. That's fine, and the pattern is expected. But if you don't find any obvious reason, you risk of having your development efforts drown in test-script maintenance.

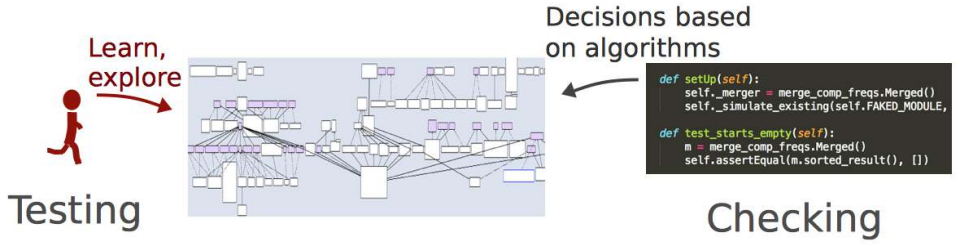
Case C means horror. The team spends too much effort on the tests compared to the application code. You recognize this scenario when you make what should be a local change to a module, and suddenly your build breaks with several failing test cases. These scenarios seem to go together with long build times (counted in hours or even days). That means you get the feedback spread out over a long period, which makes the problem even more expensive to address. At the end, the quality, predictability, and productivity of your work suffers.

Avoid the Automated-Test Death March

If you run into the warning signs—or in the worst case, the death march—make sure to run a coupling analysis on your test code as recommended in *Supervise the Evolution of the Test Scripts Themselves*, on page 102. Combine it with a hotspot analysis, as you saw in Part I. Together, the two analyses help you uncover the true problems.

But, don't wait for warning signs. There's much you can do up front.

First of all, it's important to realize that automated scripts don't replace testing. Skilled testers will find different kinds of bugs compared to what automated tests will find. In fact, as James Bach and Michael Bolton have pointed out,[2] we shouldn't even use the phrase "test automation," since there's a fundamental difference between what humans can do and what our automated scripts do. As the following figure shows, *checking* is a better word for the tasks we're able to automate.



So, if you're investing in automated checks, make sure you read *Test Automation Snake Oil [Bac96]* for some interesting insights into these issues.

Automation also raises questions about roles in an organization. A tragedy like Case C on page 100, happens when your organization makes a mental divide between test and application code. As we know, they need to evolve together and adhere to the same quality standards. To get there, we need to have developers responsible for writing and maintaining the test infrastructure and frameworks.

What about the test cases themselves? Well, they are best written by testers and developers in close collaboration. The tester has the expertise to decide on *what* to test, while the developer knows *how* to express it in code. That moves the traditional role of the tester to serve as a communication bridge between the business side, with its requirements, and the developers that make them happen with code.

2. http://www.satisfice.com/blog/archives/856

You get additional benefits with a close collaboration:

- Well-written test scripts capture a lot of knowledge about the system and its behavior. We developers get to verify our understanding of the requirements and get feedback on how well the test automation infrastructure works.
- Testers pick up programming knowledge from the developers and learn to structure the test scripts properly. This helps you put the necessary focus on the test-automation challenges.
- As a side effect, collaboration tends to motivate the people on your team, since everyone gets to see how the pieces fit together. In the end, your product wins.

Supervise the Evolution of the Test Scripts Themselves

From a productivity perspective, the test scripts you create are just as important as the application code you write. That's why I recommend that you track the evolution of test scripts with an analysis of temporal coupling, the analysis you learned about in Chapter 8, *Detect Architectural Decay*, on page 77.

If you identify clusters of test scripts that change together, my bet is that there's some serious copy-paste code to be found. You can simplify your search for it by using tools for copy-paste detection. Just be aware that these tools don't tell the whole story. Let's see why.

We programmers have become conditioned to despise copy-paste code. For good reasons, obviously. We all know that copy-paste makes code harder to modify. There's always the risk of forgetting to update one or all duplicated versions of the copied code. But there's a bit more to the story.

Perhaps I'm a slow learner, since it took me years to understand that no design is exclusively good. Design always involves tradeoffs. When we ruthlessly refactor away all signs of duplication, we raise the abstraction level in our code. And abstracting means taking away. In this case, we're trading ease of understanding for locality of change.

Just in case, I'm not advocating copy-paste; I just want you to be aware that reading code and writing code put different requirements on our designs. It's a fine but important difference; just because two code snippets look similar does not mean they should share the same abstraction. (Remember, DRY is about knowledge, not code.)

The Distinction Between Code and Knowledge Duplication

Consider the distinction between concepts from the problem domain and the solution domain. If two pieces of code look similar but express different domain-level concepts, we should probably live with the code duplication. Because the two pieces of code capture different business rules, they're likely to evolve at different rates and in divergent directions. On the other hand, we don't want any duplicates when it comes to code that makes up our technical solution.

Tests balance these two worlds. They capture a lot of implicit requirements that express concepts in the problem domain. If we want our tests to also communicate that to the reader, the tests need to provide enough context. Perhaps we should accept some duplicated lines of code and be better off in the process?

Know the Costs of Automation Gone Wrong

The most obvious problem with failed test automation is that teams spend more time keeping tests up and running than on developing new features. That's not where you want to be. Remember Lehman's law of continuing change? Without new features, your software becomes less useful.

Another less obvious cost is psychological. Consider an ideal system. There, a broken test would be a loud cry to pause other tasks and focus on finding the bug causing the test to fail.

But when we start to accept failing tests as the normal state of affairs, we've lost. A failing test is no longer a warning signal, but a potential false positive. Over time, you lose faith in the tests and blame them. As such, a failed test-automation project costs more than just the time spent on maintaining test environments and scripts.

This is why you learned to set up a safety net to catch such potential problems early. The next natural step is to generalize that safety net. Let's have a look at how we can do the same for different software architectures as well.

CHAPTER 10

Use Beauty as a Guiding Principle

By now you know how to patrol your architetural boundaries with respect to automatic tests. You've also learned to supervise the evolution of your tests, which provides you with an early warning system when things start to go downhill.

In this final chapter of Part II, we'll apply these techniques to architecture in general. We'll base the discussions around common architectural patterns and see how we can analyze their effectiveness with respect to the way we work with the code.

We'll start with an analysis of Code Maat's architecture. Once you're comfortable with the analysis of a small system, we'll move on to investigate a large web-based application built on multiple technologies like .Net, JavaScript, and CSS. Finally, we'll discuss how you can analyze *micro-service* architectures.

By focusing on architectures built on radically different patterns, you'll learn the general principles behind the analysis methods. That will give you the tools you need to apply the techniques on your own system, no matter what architectural style you use.

This chapter takes a different starting point from what you'll otherwise meet in programming books. Instead of focusing on design principles, we'll use beauty as a reasoning tool. You'll see that beauty is a fundamental quality of all good code. Here you'll learn what beautiful code is, why it matters, and how your brain loves it. Let's dive into attractiveness.

Learn Why Attractiveness Matters

Think about your daily work and the kinds of changes you make to your programs. Truth be told, how often do you get something wrong because your

conceptual model of what the code does didn't match up with the program's real behavior? Perhaps that query method you called had a side effect that you rightfully didn't expect. Or perhaps there's a feature that breaks sporadically due to an unknown timing bug, particularly when it's the full moon and, of course, just before that critical deadline.

Programming is hard enough without having to guess a program's intent. As we get experience with a codebase, we build a mental model of how it works. When that code then fails to meet our expectations, bad things are bound to happen. Those are the moments that trigger hours of desperate debugging, introduce brittle workarounds, and kill the joy of programming faster than you can say "null pointer exception."

These problems are hard because they hit us with the full force of surprise. And surprise is something that's expensive to our human brain. To avoid those horrors, we need to write beautiful code. Let's see what that is.

Define Beauty

Beauty is a fundamental quality of all good code. But what exactly is beauty? To find out, let's look at beauty in the physical world.

At the end of the 1980s, scientist Judith Langlois performed an interesting experiment. (See *Attractive faces are only average [LR90]*.) Aided by computers, she developed composite pictures by morphing photos of individual faces. As she tested the attractiveness of all these photos on a group, the results turned out to be both controversial and fascinating. Graded on physical attractiveness, the composite pictures won. And they won big.

The reason for the controversy comes from the process that produced the apparently attractive faces. When you morph photos of faces, individual differences disappear. The more photos you merge, the more average the end result. That would mean that beauty is nothing more than average!

The idea of beauty as averageness seems counterintuitive. In our field of programming, I'd be surprised if the average enterprise codebase would receive praise for its astonishing beauty. But beauty is not average in the sense of ordinary, common, or typical. Rather, beauty lies in the mathematical sense of averageness found in the composite faces.

The reason the composite pictures won is that individual imperfections were also evened out with each additional morphed photo. This is surprising since it makes beauty a negative concept, defined by what's absent rather than

Pictures with kind permission by: The Face research Lab, University of Glasgow

what's there. Beauty is the absence of ugliness. Let's look at the background to see how it relates to programming.

Our preference for beauty is shaped by evolution to guide us away from bad genes. This makes sense since our main evolutionary task was to find a partner with good genes. And back in the Stone Age, DNA tests weren't easy to come by. (In our time the technology is there, but trust me, a date will not end well if you ask your potential partner for a DNA sample.)

Instead, we tacitly came to use beauty as a proxy for good genes. The theory is that natural selection operates against extremes. This process works to the advantage of the composite pictures that are as average as it gets.

Now, let's see what a program with such good genes would look like.

Write Beautiful Code

Translated to our world of software, the good genes theory means the absence of special cases. Beautiful code has a consistent level of expression. Just as deviations from the mathematical averageness makes a face less attractive, so does any coding construct that deviates from the main style of your application.

These constructs signal bad genes in our programs because they make it harder to form a mental model of the program. That's just the way your brain works; when presented with inconsistencies and conflicting representations, your brain selects one of the stimuli at the price of the other. You can switch between them, but it's going to be expensive.

That means as soon as you break the expectations of someone reading your code, you've introduced a cognitive cost. This is a cost that makes your programs harder to understand and riskier to modify. Broken expectations are to blame for many bugs.

Attractive Criminals

Criminals benefit from beauty, too. There's a growing body of research on the topic. The research indicates that attractive defendants are perceived as less guilty and, should they be convicted, receive a more lenient sentence than unattractive offenders. And it's a finding that seems to hold for both mock jurors, used during experiments, and real-life judges (source: *The Psychology of Physical Attraction [SF08]*).

These findings are, of course, worrisome. But sometimes the attractiveness of offenders works against them. A good-looking criminal may receive a more lenient sentence for a burglary. But if the criminal used his good looks to swindle his victims, we penalize his success.

If you've ever doubted the importance of beautiful code, you now see how profound attractiveness is in our lives.

Use Beauty in Your Architecture

The beauty principle applies to software architectures, too. Since an architectural decision is by definition more important than a local coding construct, breaking beauty in a high-level design is even worse.

Consider a codebase that has multiple ways to do interprocess communication, differs in its error-handling policies, or uses several conflicting mechanisms for data access without any obvious benefit. Such a system is hard to learn and work with—particularly since the knowledge you build up when working with one part of the codebase doesn't necessarily transfer to others. Here's what we can do to prevent it.

Avoid Surprises in Your Architecture

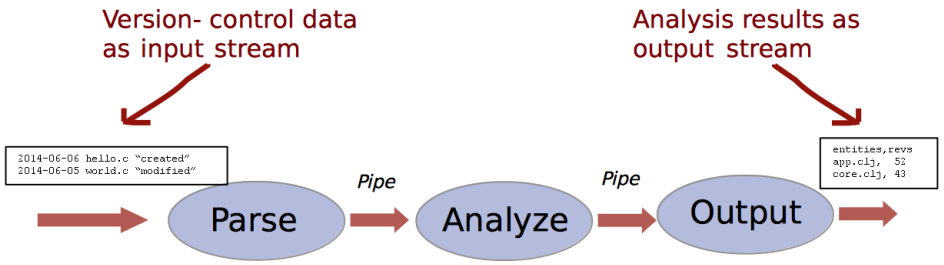
So beauty is about consistency and avoiding surprises. Fine. But what you consider a surprise depends on context. In the real world, you won't be surprised to see an elephant at the zoo, but you'd probably rub your eyes if you saw one in your front yard (at least here in Sweden, where I live). Context matters in software, too (elephants less).

When you use beauty as a reasoning tool, you need principles to measure against. This is where patterns help. Let's see how we can use them to detect nasty surprises in our designs.

Measure Against Your Patterns

We've already performed a few analyses on Code Maat. Now we'll look at its overall architecture. Let's start by defining its architectural boundaries.

Code Maat is built on the architectural pattern *Pipes and Filters*. Pipes and Filters is used to process a stream of input—in this case, the version-control data—and transform it into a stream of analysis results.



The idea behind Pipes and Filters is to "divide the application's task into several self-contained data processing steps" (qoutation from *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing [BHS07]*). That means any Pipes and Filters implementation with temporal coupling between its processing steps would be a surprise to a maintenance programmer. A sure sign of ugliness.

So this looks like a good principle against which to evaluate the architecture. Let's do a temporal coupling analysis across Code Maat's data-processing steps.

Specify the Architecturally Significant Components

Remember how you specified a transformation to evaluate automatic tests in Chapter 9, *Build a Safety Net for Your Architecture*, on page 91? You use the same strategy to analyze any software architecture. Just open a text editor and specify the following transformations:

```
src/code_maat/parsers   => Parse
src/code_maat/analysis  => Analyze
src/code_maat/output    => Output
src/code_maat/app       => Application
```

Compare this transformation to the architecture in the preceding figure. As you see, each logical name in the transformation corresponds to one Filter in Code Maat. In addition, we include an Application component. Application serves as the entry point for Code Maat.

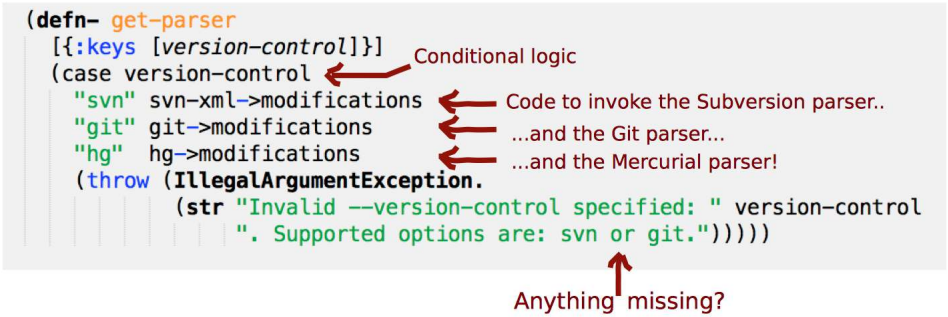
This transformation allows you to detect surprising modification patterns that break the architectural principle. Just save the text you just entered as maat_pipes_filter_boundaries.txt and run the following analysis:

```
prompt> maat -l maat_evo.log -c git -a coupling -g maat_pipes_filter_boundaries.txt
entity,coupled,degree,average-revs
Analyze,Application,37,32
Application,Parse,31,29
```

Hmm, the results don't show any violation of the Pipes and Filters principle. That's reassuring. However, there seems to be something strange going on with the top-level Application component—it's coupled to two filters. That may be bad enough. Let's see why.

Identify the Offending Code

Since Code Maat is a small codebase, we can go directly to the source code. To find the offending code, you'd need to compare the revisions of the code where any module in Application was changed together with Parse or Analyze.



If you follow that track, you'll soon find the code above. As you see, the piece of Clojure code determines the version-control system to use. It then returns a function—for example, svn-xml->modifications—that knows how to invoke a parser for that system.

This explains the coupling between the logical parts Application and Parse. When a parser component changes, those functions have to change as well. In a small codebase like Code Maat, this isn't a severe problem. But the general design is questionable because it encourages coupling between parts that

should be independent. Now, would you be surprised if I told you that a similar type of coding construct is used to select the analysis to run?

As you see in the analysis results, the Analyze and Application components change together as well. Since Code Maat mainly grows by new analysis components, this becomes a more severe problem than the coupling to the parsers. It's the kind of design that becomes an evolutionary hurdle for the program. If we break that change coupling, we remove a surprise and make our software easier to evolve in the process. That's a big win.

Spot the Uncovered Bug

Before we move on, did you spot the other surprise in the code above? Hint: have a look at the last line.

The code supports three parsers: svn, hg, and git. Now, have a look at the error message we throw as default. The message says "Supported options are: svn or git." Oops—we missed the hg option there!

This kind of bug is typical for code constructs built on conditional logic and far from our beauty ideal. You'll probably make similar findings yourself; when you investigate analysis results, you get a different view of your code. That change in perspective opens your eyes to those obvious mistakes that you'll otherwise just skim over. (See *Code Coverage? Seriously, Is It Any Good?*, on page 97, for a related discussion.)

Now that you've seen how to analyze one type of architecture, let's scale up to a more complex system.

Analyze Layered Architectures

Code Maat is a small system with a simple architecture. More elaborate architectures require more sophisticated transformation specifications:

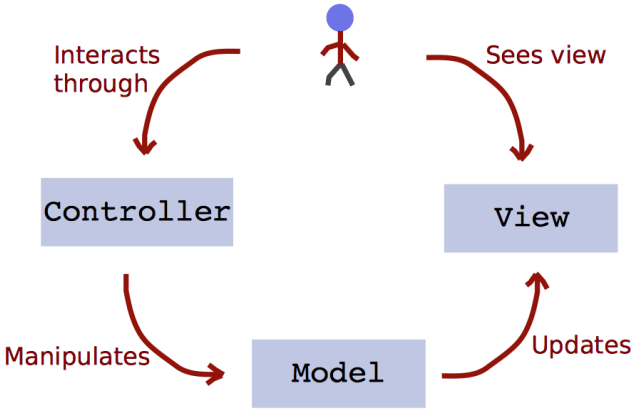
- Don't specify all components. For example, ignore small utility components.
- Differentiate how the code is laid out in the file system against its logical structure. Your analysis transformations don't have to mirror the code structure the transformations operate on.
- Use beauty as a guide. You know how you want the ideal system to look. Now think of all the ugly ways those expectations could be broken. In particular, look for beauty breakers that would be expensive.

Let's see it in practice on a large system.

Identify Significant Layers

Our next case study uses nopCommerce.[1] nopCommerce is an open-source product used to build e-commerce sites. It's a competent piece of code consisting of 200,000 lines of C# and JavaScript together with a bunch of SQL scripts and CSS files—a perfect opportunity to see how the analysis method works across multiple languages.

The first step is to identify the architectural principles of the system. nopCommerce is a web application built around the *Model-View-Controller* pattern.



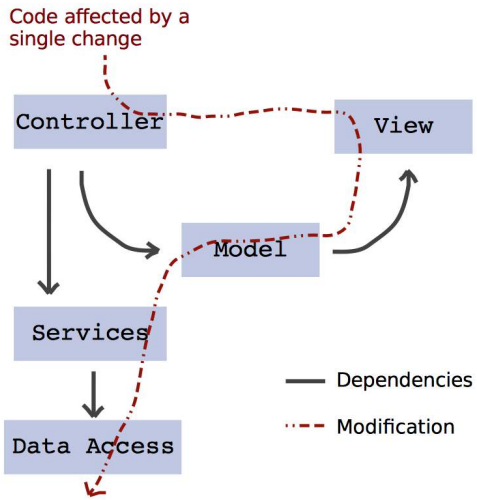
Model-View-Controller (MVC) is a pattern for implementing user-interface applications. Like all patterns, each implementation looks different.

One variation is to introduce a service layer encapsulating the business logic and have the Controller delegate to that layer. This is how it's done in nopCommerce. Often, additional layers, such as data access, follow.

There's a simple idea behind all those layers. That idea is to separate concerns. In theory, that allows us to change our mind and swap a layer for one with another implementation. This potential flexibility comes at a price.

In practice, layered architectures rarely deliver upon their promise. Instead, you'll often find that each modification you make to the code ripples through multiple layers. Such modification patterns are an indication that the layers aren't worth the price you pay. Perhaps they even make your code *harder* to change. Let's see whether that's the case.

1. http://www.nopcommerce.com/



Find Surprising Change Patterns

Start by cloning the nopCommerce git repository:

```
git clone https://git01.codeplex.com/nopcommerce
```

Move into your repository and generate a version-control log. Let's look at all changes to the code from the start of 2014 until the present day (defined as 2014-09-25—the day I'm writing this):

```
prompt> git log --pretty=format:'[%h] %an %ad %s' --date=short --numstat \
 --after 2014-01-01 --before 2014-09-25 > nop_evo_2014.log
```

Your git log is now stored in the file nop_evo_2014.log. Let's define a transformation to go with it.

Define Each Layer as an Architectural Boundary

Just as we did for our Pipes and Filters analysis, we map each architectural part to a logical name. Here's the transformation for nopCommerce:

```
src/Presentation/Nop.Web/Administration/Models      => Admin Models
src/Presentation/Nop.Web/Administration/Views       => Admin Views
src/Presentation/Nop.Web/Administration/Controllers => Admin Controllers
src/Libraries/Nop.Services                          => Services
src/Libraries/Nop.Core                              => Core
src/Libraries/Nop.Data                              => Data Access
src/Libraries/Nop.Services                          => Business Access Layer
src/Presentation/Nop.Web/Models                     => Models
src/Presentation/Nop.Web/Views                      => Views
src/Presentation/Nop.Web/Controllers                => Controllers
```

I derived the transformation from the nopCommerce documentation.[2] I also had a look at the source code to identify the Model-View-Controller layers that you see below the src/Presentation/Nop.Web and src/Presentation/Nop.Web/Administration folders. (When you analyze your own system, you're probably already familiar with its high-level design.)

Before we run the analysis, note that nopCommerce consists of two applications: one administration application and one application for the actual store. We specify both in our transformation, since they're logical parts of the same system and have to be maintained together.

Now, store the transformation specification in the file arch_boundaries.txt, and you're set for the analysis:

```
prompt> maat -c git -l nop_evo_2014.log -g arch_boundaries.txt -a coupling
entity,coupled,degree,average-revs
Admin Models,Admin Views,75,74
Admin Controllers,Admin Models,68,73
Admin Controllers,Admin Views,66,89
Admin Models,Core,54,76
Core,Services,46,130
Models,Views,46,47
Admin Views,Core,44,92
Admin Controllers,Core,39,91
Controllers,Models,38,60
Admin Controllers,Services,35,128
Admin Models,Services,35,113
Admin Views,Services,34,129
Controllers,Views,34,83
Controllers,Services,33,132
Admin Controllers,Controllers,31,92
```

The results reveal several cases of temporal coupling. The Admin parts of the system exhibit the strongest coupling. Let's focus on them.

Identify Expensive Change Patterns

Remember that one idea behind the MVC pattern is to allow us to swap in different views. Since the Views change together with the Models in 75 percent of all cases, that idea is probably not fulfilled; if we do add a different set of views, those will have to change frequently as well, which will slow us down.

The following figure also shows that all components in the MVC part have a temporal dependency upon the Core and Services layers. Let's get support from a hotspot analysis to find out how serious that is.

2. http://docs.nopcommerce.com/display/nc/nopCommerce+Documentation

Component	Coupled Component	Coupling (%)
Admin Models	Admin Views	75
Admin Models	Admin Controllers	68
Admin Controllers	Admin Views	66
Admin Models	Core	54
Admin Views	Core	44
Admin Controllers	Core	39
Admin Controllers	Services	35
Admin Models	Services	35
Admin Views	Services	34
Admin Controllers	Controllers	31

The MVC parts change together.

The MVC parts change with the Core component as well as...

...with the Services layer.

Temporal coupling across application boundaries.

Use Hotspots to Assess the Severity

Instead of identifying individual modules as hotspots, we'll reuse the transformation. That allows you to find hotspots on the level of your architecture. That is, a hotspot in this analysis corresponds to a whole layer:

```
prompt> maat -c git -l nop_evo_2014.log -g arch_boundaries.txt -a revisions
entity,n-revs
Services,393
Views,388
Admin Controllers,257
Admin Views,253
Controllers,181
Core,169
Data Access,122
Admin Models,76
Models,36
```

As you see, the Services layer has the most volatile code. That means any temporal coupling that involves Services is, on average, a more serious concern than the others. This is information we use to reason about the cost of changes—for example, when exploring design alternatives.

However, a temporal coupling analysis can't tell us the direction of the change dependency; we don't know whether changes to the Services lead to predictable changes in the MVC parts or whether it is the other way around. But we do know there's a 35 percent risk that our change will affect multiple layers.

Finally, note the strange change dependency between the Admin Controllers and the Controllers that we see at the bottom of the preceding figure. The controllers in two different packages change together 31 percent of the time.

When we find cross-cutting temporal coupling like that, we should investigate the cause. Often, coupled components share a common abstraction, such as a base class that accumulates too many responsibilities. In other cases, we'll find the opposite problem, and we have a shared abstraction waiting to be discovered. To find it, we want to apply a temporal coupling analysis, as we did back in Chapter 8, *Detect Architectural Decay*, on page 77.

Treat Patterns as Helpful Friends

Before we move on, note that these results aren't presented to show that design patterns don't work—quite to the contrary. Patterns are context-dependent and do not, by some work of magic, provide universally good designs. You can't take the human out of the design loop. That said, patterns have a lot to offer:

- *Patterns are a guide*. Our architectural principles are likely to evolve together with our system. Remember, problem-solving is an iterative process. Agreeing on the right set of initial principles is challenging, and this is where the knowledge captured in patterns helps.
- *Patterns share knowledge*. Patterns come from existing solutions and experiences. Since few designs are really novel, we'll often find patterns that apply to our new problem as well.
- *Patterns have social value*. As the architect Christopher Alexander formalized patterns, the intent was to enable collaborative construction using a shared vocabulary. As such, patterns are more of a communication tool than a technical solution.
- *Patterns are reasoning tools*. You learned about chunking back in *Names Make the Code Fit Your Head*, on page 48. Patterns are a sophisticated form of chunking. Their names serve as handles to knowledge stored in our long-term memory. Patterns optimize our working memory and guide us as we evolve mental models of the problem and solution space.

Expand Your Analyses

When we uncover problems in our analyses, we want to react. We typically reconsider some architectural principles, perhaps even the overall patterns we built on. As a result, we evolve parts of our system into a new direction. As we do this, we want to be able to track that as well.

The techniques you've learned will be there to support you, since the analyses aren't limited to the patterns we've discussed in this chapter. Understanding

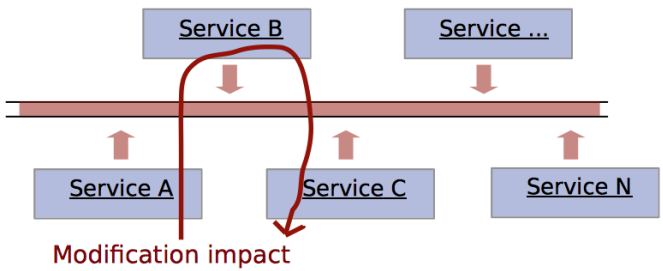
the underlying ideas lets you apply the analyses to new situations. So before we move on to the final part of the book, let's have a quick look at some different architectural styles you may encounter.

Analyze Microservices

At the time of this writing, *microservices* are gaining rapid popularity. That means many of tomorrow's legacy systems are likely to be microservice architectures. Let's stay a step ahead and see what we would want to analyze when we come across such systems.

Microservices are based on an old idea where you organize your code by feature. You keep each part small and orthogonal to others, and use a simple mechanism to glue everything together (for example, a message bus or an HTTP API). In fact, these are the same principles on which UNIX has built since the dawn of curly braces in code.

A microservice architecture attempts to encapsulate each fine-grained responsibility in a service. This principle implies that a microservice architecture is attractive when it allows us to modify and replace individual services without affecting other services. The warning sign in a microservices architecture is a commit that affects multiple services:



When we analyze microservices, we want to consider each service an architectural boundary. That's what we specify in our transformations. As soon as we see changes that ripple through multiple services, we know that ugliness is creeping into our system, and we can react immediately.

Reverse-Engineer Your Principles from Code

As you saw in the microservice example, we use the same techniques to analyze all kinds of architectures. But what if we don't have any existing principles on which we can base our reasonings? What if we just inherited a monster codebase without any obvious structure or style? Well, our focus changes. Let's see how.

All codebases, even the worst spaghetti monsters, have some principles. All programmers have their own style. It may change over time, but we can find and build upon consistencies.

When you find yourself wading through legacy code, take the time to step back. Look at the records in your version-control system. Often, you can spot patterns. Complement that information with what you learn as you make changes to the code. Perhaps most of the database access is located in an inaptly named `utility` module. Maybe each subscreen in the GUI is backed by its own class. Fine—you just uncovered your first principles.

As you start to reverse-engineer more principles, tailor the analyses in this chapter accordingly. Look for changes that break the principles. The principles may not be ideal, and the system may not be what you want. But at least this strategy will give you an opportunity to assess how consistent the system is. Used that way, the analyses will help you improve the situation and make code changes more predictable over time.

Use Your Suite of Analysis Techniques

Now you have a set of new skills that allow you to analyze everything from individual design elements all the way up to architectures and automated tests. With these techniques, you'll be able to detect when your programs start to evolve in a direction your architecture cannot support.

The key to these high-level analyses is to formulate simple rules based on your architectural principles. We introduced beauty as a supporting tool, and you set up your analyses to capture the cases where we break it.

Once you've formulated your rules, run the analyses frequently. Use the results as an early warning system and as the basis for design discussions. You also want to complement the temporal coupling results with a hotspot analysis. Hotspots help you assess the severity of your temporal couples.

Throughout Part II, we have focused on how to interview our codebase and evaluate the code's health. But the challenges of large-scale software go beyond technology. Many of the problems you'll find in a forensic code analysis have social roots.

In Part III, we'll move into this fascinating area. You'll meet new techniques to identify the organizational problems that creep into your code. You'll also learn about social biases that influence your development team and how to avoid classic pitfalls when scaling your development efforts. Of course, we'll mine supporting evidence for all claims. Let's move on to people!

Part III

Master the Social Aspects of Code

Here we expand our vision into the fascinating field of social psychology. Software development is prone to many of the social fallacies and biases we meet in our everyday life. They just manifest themselves in a different setting.

In this part, you'll learn to analyze the communication and interaction between developers. We'll also cover the perils of multiple authors working on the same module and introduce techniques to predict post-release defects. Best of all, we'll pull it off from the perspective of your code. This is your codebase like you've never seen it before!

CHAPTER 11

Norms, Groups, and False Serial Killers

Part II showed you how to analyze high-level designs and architectures. We based the techniques on the concept of temporal coupling. You learned to use temporal coupling to evaluate how well your software architecture supports the modifications you make to the code.

We also discussed how you can use that information to detect structural decay, supervise test-automation efforts, and guide your design discussions. You also saw how the hotspot analyses from Part I complement your new forensic code skills. As far as technology goes, you're set with what you need to uncover the mysteries of your codebase.

But large-scale software projects are more than technical problems. Software development is also a social activity. Programming involves social interactions with our peers, customers, and managers. Those interactions range from individual conversations to important decisions made in large groups.

Just as you want to ensure that your software architecture supports the way you evolve your system, you also want to ensure that the way you organize your work aligns with how the system is structured.

How well you and your team fare on these social aspects influences how your system looks. That's why social psychology is just as important to master as any programming language. In this final part of the book, you'll learn about social biases, how you can predict bugs from the way we work, and how to build a knowledge map of your codebase. And just as before, we'll mine supporting data from our version-control systems.

We'll start with the social biases. These biases may lead to disastrous decisions, which is why you want to be aware of them and recognize them. We'll then to see how we gather objective data on the social aspects of software

development to inform our decisions about team organization, responsibilities, and our process. Let's start in the middle of a criminal investigation.

Learn Why the Right People Don't Speak Up

In the early 1990s, Sweden had its first serial killer. The case led to an unprecedented manhunt. Not for an offender—he was already locked up—but for the victims. There were no bodies.

A year earlier, Thomas Quick, incarcerated in a mental institution, started confessing to one brutal murder after another. The killings Quick confessed to were all well-known unsolved cases.

Over the course of some hectic years, Swedish and Norwegian law enforcement dug around in forests and traveled all across the country in search of hard evidence. At the height of the craze, they even emptied a lake. Yet not a single bone was found.

This striking lack of evidence didn't prevent the courts from sentencing Quick to eight of the murders. His story was judged as plausible because he knew detailed facts that only the true killer could've known. Except Quick was innocent. He fell prey to powerful cognitive and social biases.

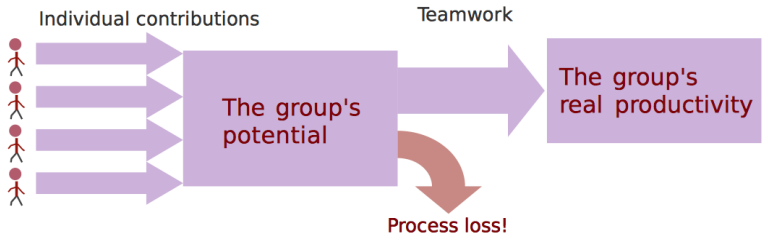
The story about Thomas Quick is a case study in the dangers of social biases in groups. The setting is much different from what we encounter in our daily lives, but the biases aren't. The social forces that led to the Thomas Quick disaster are present in any software project.

See How We Influence Each Other

We'll get back to the resolution of the Quick story soon. But let's first understand the social biases so we can prevent our own group disasters.

When we work together in a group to accomplish something—for example, to design that amazing web application that will knock Google Search down—we influence each other. Together, we turn seemingly impossible things into reality. Other times, the group fails miserably. In both cases, the group exhibits what social scientists call *process loss*.

Process loss is the theory that groups, just as machines, cannot operate at 100 percent efficiency. The act of working together has several costs that we need to keep in check. These costs are losses in coordination and motivation. In fact, most studies on groups find that they perform below their potential.



So why do we choose to work in groups when it's obviously inefficient? Well, often the task itself is too big for a single individual. Today's software products are so large and complex that we have no other choice than to build an organization around them. We just have to remember that as we move to teams and hierarchies, we pay a price: process loss.

When we pay for something, we expect a return. We know we'll lose a little efficiency in all team efforts; it's inevitable. (You'll learn more about coordination and communication in subsequent chapters.) What's worse is that social forces may rip your group's efforts into shreds and leave nothing but broken designs and bug-ridden code behind. Let's see what we can do to avoid that.

Learn About Social Biases

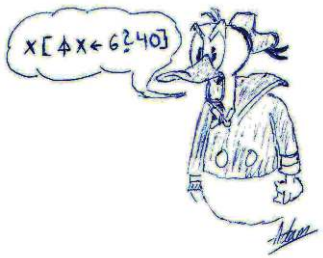
Pretend for a moment that you've joined a new team. On your first day, the team gathers to discuss two design alternatives. You get a short overview before the team leader suggests that you all vote for the best alternative.

It probably sounds a little odd to you. You don't know enough about the initial problem, and you'd rather see a simple prototype of each suggested design to make an informed decision. So, what do you do?

If you're like most of us, you start to look around. You look at how your colleagues react. Since they all seem comfortable and accepting of the proposed decision procedure, you choose to go along with the group. After all, you're fresh on the team, and you don't want to start by rejecting something everyone else believes in. As in Hans Christian Andersen's fairy tale, no one mentions that the emperor is naked. Let's see why. But before we do, we have to address an important question about the role of the overall culture.

Isn't All This Group Stuff Culture-Dependent?

Sure, different cultures vary in how sensitive they are to certain biases. Most research on the topic has focused on East-West differences. But we don't need to look that far. To understand how profoundly culture affects us, let's look at different programming communities.



Take a look at the code in the speech balloon. It's a piece of APL code. APL is part of the family of array programming languages. The first time you see APL code, it will probably look just like this figure: a cursing cartoon character or plain line noise. But there's a strong logic to it that results in compact programs. This compactness leads to a different mindset.

The APL code calculates six lottery numbers, guaranteed to be unique, and returns them sorted in ascending order.[1] As you see in the code, there are no intermediate variables to reveal the code's intent. Contrast this with how a corresponding Java solution would look.

Object-oriented programmers value descriptive names such as randomLotteryNumberGenerator. To an APL programmer, *that's* line noise that obscures the real intent of the code. The reason we need more names in Java, C#, or C++ is that our logic—the stuff that really does something—is spread out across multiple functions and classes. When our language allows us to express all of that functionality in a one-liner, our context is different, and it affects the way we and our community think.

Different cultures have different values that affect how their members behave. Just remember that when you choose a technology, you also choose a culture.

Understand Pluralistic Ignorance

What just happened in our fictional example is that you fell prey to *pluralistic ignorance*. Pluralistic ignorance happens in situations where everyone privately rejects a norm but thinks that everyone else in the group supports it. Over time, pluralistic ignorance can lead to situations where a group follows rules that all of its members reject in private.

We fall in this social trap when we conclude that the behavior of our peers depends on beliefs that are different from our own, even if we behave in an identical way ourselves. That's what happened around Andersen's naked emperor. Because everyone praised the emperor's new clothes, each individual thought they missed something obvious. That's why they chose to conform to the group behavior and play along with the praise of the wonderful clothes they couldn't see.

1. http://en.wikipedia.org/wiki/APL_(programming_language)

Another common social bias is to mistake a familiar opinion for a widespread one. If we hear the same option repeatedly, we come to think of that opinion as more prevalent than it really is. As if that wasn't bad enough, we fall for the bias even if it's the *same* person who keeps expressing that opinion (source: *Inferring the popularity of an opinion from its familiarity: A repetitive voice can sound like a chorus [WMGS07]*).

This means it's enough with one individual, constantly expressing a strong opinion, to bias your whole software development project. It may be about technology choices, methodologies, or programming languages. Let's see what you can do about it.

Challenge with Questions and Data

Most people don't like to express deviating opinions, but there are exceptions. One case is when our minority opinion is aligned with the group ideal. That is, we have a minority opinion, but it deviates from the group norm in a positive way; the group has some quality it values, and we take a more extreme position and value it even more. In that setting, we're more inclined to speak up, and we'll feel good about it when we do.

Within our world of programming, such "good" minority opinions may include desired attributes such as automatic tests and code quality. For example, if tests are good, then testing everything must be even better (even if it forces us to slice our designs in unfathomable pieces). And since code quality matters, we must write code of the highest possible quality all the time (even when prototyping throwaway code).

Given what we know about pluralistic ignorance and our tendency to mistake familiar opinions for common ones, it's easy to see how these strong, deviating opinions may move a team in a more extreme direction.

Social biases are hard to avoid. When you suspect them in your team, try one of the following approaches:

- *Ask questions*: By asking a question, you make others aware that the proposed views aren't shared by everyone.
- *Talk to people*: Decision biases like pluralistic ignorance often grow from our fears of rejection and criticism. So if you think a decision is wrong but everyone else seems fine with it, talk to your peers. Ask them what they like about the decision.
- *Support decisions with data*: We cannot avoid social and cognitive biases. What we can do is to check our assumptions with data that either supports or challenges the decision. The rest of this book will arm you with several analyses for this purpose.

If you're in a leadership position, you have additional possibilities to guide your group toward good decisions:

- Use outside experts to review your decisions.
- Let subgroups work independently on the same problem.
- Avoid advocating a specific solution early in the discussions.
- Discuss worst-case scenarios to make the group risk-aware.
- Plan a second meeting upfront to reconsider the decisions of the first one.

These strategies are useful to avoid *groupthink* (source: *Group Process, Group Decision, Group Action [BK03]*). Groupthink is a disastrous consequence of social biases where the group ends up supressing all forms of internal dissent. The result is group decisions that ignore alternatives and the risk of failure, and that give a false sense of consensus.

As you've seen, pluralistic ignorance often leads to groupthink. This seems to be what happened in the Thomas Quick case.

Witness Groupthink in Action

Let's get back to our story of Thomas Quick. Quick was sentenced for eight murders before he stopped cooperating in 2001. Without Quick's confessions, there was little to do—remember, there was no hard evidence in any of the murder cases. It took almost ten years for the true story to unfold.

What had happened was that Thomas Quick was treated with a pseudoscientific version of psychotherapy back in the 1990s. The therapists managed to restore what they thought were recovered memories. (Note that the scientific support for such memories is weak at best.) The methods they used are almost identical to how you implant false memories. (See *The Paradox of False Memories*, on page 68.) Quick also received heavy dozes of benzodiazepines, drugs that may make their users more suggestible.

The murder investigation started when the therapists told the police about Quick's confessions. Convinced by the therapists' authority that repressed memories were a valid scientific theory, the lead investigators started to interrogate Quick.

These interrogations were, well, peculiar. When Quick gave the wrong answers, he got help from the chief detective. After all, Quick was fighting with repressed memories and needed all the support he could get. Eventually, Quick got enough clues to the case that he could put together a coherent story. That was how he was convicted.

By now, you can probably see where the Thomas Quick story is heading. Do you recognize any social biases in it? To us in the software world, the most interesting aspects of this tragic story are in the periphery. Let's look at them.

Know the Role of Authorities

Once the Quick scandal with its false confessions was made public, many people started to speak up. These people, involved in the original police investigations, now told the press about the serious doubts they'd had from the very start. Yet few of them had spoken up ten years earlier, when Quick was originally convicted.

The social setting was ideal for pluralistic ignorance—particularly since the main prosecutor was a man of authority and was convinced of Quick's guilt. He frequently expressed that opinion and contributed to the groupthink.

From what you now know about social biases, it's no wonder that a lot of smart people decided to keep their opinions to themselves and play along. Luckily, you've also got some ideas for how you can avoid having similar sit-

uations unfold in your own teams. Let's add one more item to that list by discussing a popular method that often does more harm than good—brainstorming.

Move Away from Traditional Brainstorming

If you want to watch process loss in full bloom, check out any brainstorming session. It's like a best-of collection of social and cognitive biases. That said, you can be productive with brainstorming, but you need to change the format drastically. Here's why and how.

The original purpose of brainstorming was to facilitate creative thinking. The premise is that a group can generate more ideas than its individuals can on their own. Unfortunately, research on the topic doesn't support that claim. On the contrary, research has found that brainstorming produces *fewer* ideas than expected and that the quality of the produced ideas may suffer as well.

The are several reasons for the dramatic process loss. For example, in brainstorming we're told not to criticize ideas. In reality, everyone knows they're being evaluated anyway, and they behave accordingly. Further, the format of brainstorming allows only one person at a time to speak. That makes it hard to follow up on ideas, since we need to wait for our time to talk. In the meantime, it's easy to be distracted by other ideas and discussions.

To reduce the process loss, you need to move away from the traditional brainstorming format. Studies suggest that a well-trained group leader may help you eliminate process loss. Another promising alternative is to move to computers instead of face-to-face communication. In that setting, where social biases are minimized, electronic brainstorming may actually deliver on its promise. (See *Idea Generation in Computer-Based Groups: A New Ending to an Old Story [VDC94]* for a good overview of the research.)

Now you know what to avoid and watch out for. Before we move on, take a look at some more tools you can use to reduce bias.

Discover Your Team's Modus Operandi

Remember the geographical offender-profiling techniques you learned back in *Learn Geographical Profiling of Crimes*, on page 16? One of the challenges with profiling is linking a series of crimes to the same offender. Sometimes there's DNA evidence or witnesses. When there's not, the police have to rely on the offender's *modus operandi*.

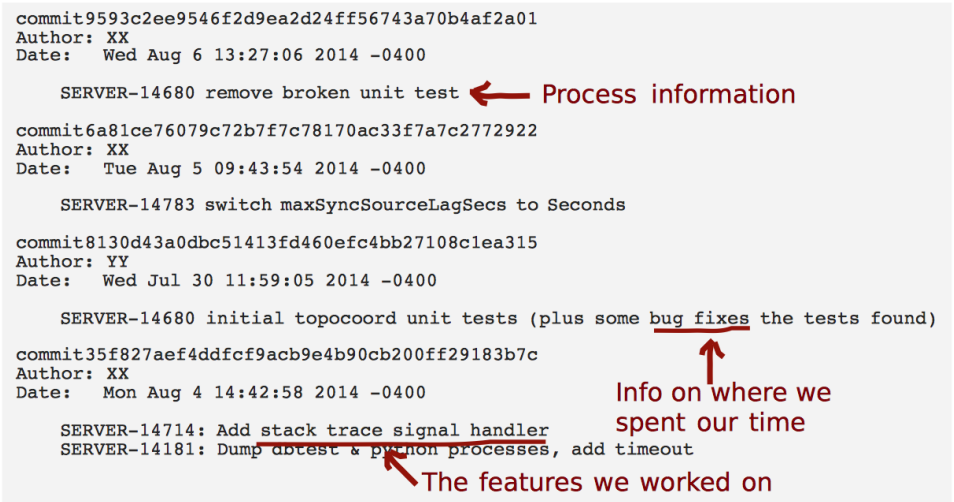
A modus operandi is like a criminal signature. For example, the gentleman bandit you read about in *Meet the Innocent Robber*, on page 69, was characterized by his polite manners and concern for his victims.

Software teams have their unique modus operandi, too. If you manage to uncover it, it will help you understand how the team works. It will not be perfect and precise information, but it can guide your discussions and decisions by opening new perspectives. Here's one trick for that.

Use Commit Messages as a Discussion Basis

Some years ago, I worked on a project that was running late. On the surface, everything looked fine. We were four teams, and everyone was kept busy. Yet the project didn't make any real progress in terms of completed features. Soon, the overtime bell began to ring.

Luckily, there was a skilled leader on one of teams. He decided to find out the root cause of what was holding the developers back. I opted in to provide some data as a basis for the discussions. Here's the type of data we used:



Until now, we have focused our techniques around the code you're changing. But a version-control log has more information. Every time you commit a change, you provide social information.

Have a look at the *word cloud*. It's created from the commit messages in the Craft.Net repository by the following command:

```
prompt> git log --pretty=format:'%s'
Merge pull request #218 from NSDex/master
Don't add empty 'extra' fields to chat msg JSON
Fix Program.cs
Revert "Merge pull request #215 from JBou/master"
...
```

The command extracts all commit messages. You have several simple alternatives to visualize them. The one was created by pasting the messages into Wordle.[2]

If we look at the commit cloud, we see that certain terms dominate. What you'll learn right now is by no means scientific, but it's a useful heuristic: the words that stand out tell you where you spend your time. For the Craft.Net team, it seems that they get a lot of features in, as indicated by the word "Added," but they also spend time on "Fixing" code.

On the project I told you about—the one that was running late and no one knew why—the word cloud had two prominent words. One of them highlighted a supporting feature of less importance where we surprisingly spent a lot of time. The second one pointed to the automated tests. It turned out the teams

2. http://www.wordle.net

were spending a significant portion of their workdays maintaining and updating tests. This finding was verified by the techniques you learned in Chapter 9, *Build a Safety Net for Your Architecture*, on page 91. We could then focus improvements on dealing with the situation.

What story does your own version-control log tell?

Commit Messages Tell a Story

Commit clouds are a good basis for discussion around our process and daily work. The clouds present a distilled version of our team's daily code-centered activities. What we get is a different perspective on our development that stimulates discussions.

What we *want* to see in a commit cloud is words from our domain. What we *don't want* to see is words that indicate quality problems in code or in our process. When you find those indications, you want to drill deeper.

But commit messages have even more to offer; A new line of research proposes that commit messages tell something about the team itself. A team of researchers found this out by analyzing commit messages in different open-source projects with respect to their emotional content. The study compared the expressed emotions to factors such as the programming language used, the team location, and the day of the week. (See *Sentiment analysis of commit comments in GitHub [GAL14]*.)

Among other findings, the results of the study point to Java programmers expressing the most negative feelings, and distributed teams the most positive.

The study is a fun read. But there's a serious topic underpinning it. Emotions play a large role in our daily lives. They're strong motivators that influence our behavior on a profound level, often without making us consciously aware of why we react the way we do. Our emotions mediate our creativity, teamwork, and productivity. As such, it's surprising that we don't pay more attention to them. Studies like this are a step in an important direction.

Data Doesn't Replace Communication

Given all fascinating analyses, it's easy to drown in technical solutions to social problems. Just remember that no matter how many innovative data analyses we have, there's no replacement for actually talking to the rest of the team and taking an active role in the daily work. The methods in this chapter just help you ask the right questions.

Mine Organizational Metrics from Code

In this chapter, you learned about process loss and that groups never perform at their maximum potential. As such, teamwork and organizations are investments we pay for, and they should be considered as such.

You also learned that groups are sensitive to social biases. You saw that there are biases in all kinds of groups—software development included—and you need to be aware of the risks.

That leads us to the challenges of scaling software development. As we go from a small group of programmers to interdependent teams, we increase the coordination and communication overhead, which in turn increases the risk for biased decisions. As such, the relative success of any large-scale programming effort depends more on the people on the project than it does on any single technology.

Over the next chapters, you'll learn about fascinating research findings that support this view. As you'll see, if you want to know about the quality of a piece of software, look at the organization that built it. You'll also learn how to mine and analyze organizational data from your version-control system.

So please keep the social biases in the back of your head as you read along; by using the techniques you're about to learn, you'll get information to help you make informed decisions and challenge groupthink. Let's start with how the number of programmers affects code quality.

CHAPTER 12

Discover Organizational Metrics in Your Codebase

In the previous chapter, you learned about social biases and how they affect group decisions and interactions with other developers. We also discussed the idea that the social aspects of software development are just as important as the technical ones. Now you'll see what that means in practice.

We'll start by revisiting the classic software "laws" of Brooks and Conway to see how they fare against modern research. Based on those findings, we'll introduce organizational metrics that let us analyze the quality of our code from the team's perspective. You'll also learn to use hotspots and temporal coupling to evaluate how well your team organization aligns with the way you work with the code. Let's start with a story of a doomed software project.

Let's Work in the Communication Business

I once joined a project that was doomed from the very beginning. The stakeholders wanted the project completed within a timeframe that was virtually impossible to meet. Of course, if you've spent some time in the software business, you know that's standard practice. What made this case different was that the company had detailed data on an almost identical project. And they tried to game it.

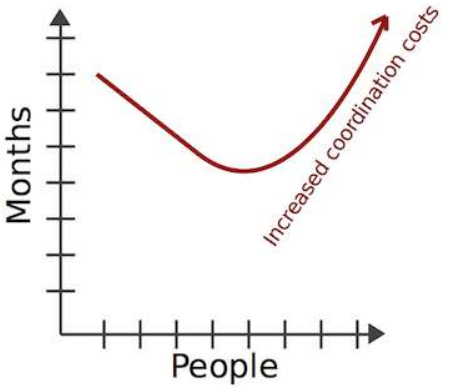
The historical data indicated that the project would need at least a year, but this time they wanted it done in three months. That's a quarter of the time. The stakeholders did realize that; their solution was to throw four times as many developers on the project. Easy math, right? There was just one slight problem: this tactic doesn't work and never has. Let's see why.

See That a Man-Month Is Still Mythical

If you pick up a forty-year-old programming book, you expect it to be dated. Our technical field has changed a lot over the past decades. But the people side of software development is different. The best book on the subject, *The Mythical Man-Month: Essays on Software Engineering [Bro95]*, was published in the 1970s and describes lessons from a development project in the 1960s. Yet the book hasn't lost its relevance.

In a way that's depressing for our industry, as it signals a failure to learn. But it goes deeper than that. Although our technology has advanced dramatically, people haven't really changed. We still walk around with brains that are biologically identical to the ones of our ancestral, noncoding cavemen. That's why we keep falling prey to the same social and cognitive biases over and over again, with failed projects covering our tracks.

You saw this in the story of the doomed project at the beginning of this chapter. It's a story that captures Brooks's law from *The Mythical Man-Month: Essays on Software Engineering [Bro95]*: "Adding manpower to a late software project makes it later." Let's see why.



Move Beyond Brooks's Law

The rationale behind Brooks's law is that intellectual work is hard to parallelize. While the total amount of work that gets done increases, the additional communication effort increases at a more rapid rate. At the extreme, we get a project that gets little done besides administrating itself (aka the Kafka management style).

Social psychologists have known this for years. Group size alone has a strong negative impact on communication. With increased group size, a smaller percentage of group members takes part in discussions, process loss accelerates, and the larger anonymity leads to less responsibility for the overall goals.

There's a famous and tragic criminal case that illustrates how group size impacts our sense of responsibility. Back in 1964, Kitty Genovese, a young woman, was assaulted and killed on her way home in New York City. The attack lasted for 30 minutes. Her screams for help were heard through the

windows by at least a dozen neighbors. Yet no one came to help, and not one called the police.

The tragedy led to a burst of research on responsibility. Why didn't anyone at least call the police? Were people that apathetic?

The researchers that studied the Kitty Genovese case focused on our social environment. Often, the situation itself has a stronger influence upon our behavior than personality factors do. In this case, each of Kitty Genovese's neighbors assumed that someone else had already called the police. This psychological state is now known as *diffusion of responsibility*, and the effect has been confirmed in experiments. (See the original research in *Bystander intervention in emergencies: diffusion of responsibility [DL68]*.)

Software development teams aren't immune to the diffusion of responsibility either. You'll see that with increased group size, more quality problems and code smells will be left unattended.

Skilled people can reduce these problems but can never eliminate them. The only winning move is not to scale—at least not beyond the point your codebase can sustain. Let's look at what that means and how you can measure it.

Remember the Diffusion of Responsibility

Once you know about diffusion of responsibility, it gets easier to avoid. When you witness someone potentially in need of help, just ask if he or she needs any—don't assume someone else will take that responsibility. The same principle holds in the software world: if you see something that looks wrong, be it a quality problem or organizational trouble, just bring it up. Chances are that the larger your group, the fewer people who will react, and you can make a difference.

Find the Social Problems of Scale

In the first parts of this book, we discussed large codebases and how we fail to get a holistic view of them. We just can't keep it all in a single brain. We recognize when we suffer from quality problems or when the work takes longer than we'd expect it to, but we don't know why.

The reasons go beyond technical difficulties and include an organizational component as well. On many projects, the organizational aspects alone determine success or failure. Let's understand them better.

Know the Difference Between Open-Source and Proprietary Software

So far we've used real-world examples for all our analyses. The problems we've uncovered are all genuine. But when it comes to the people side, it gets harder to rely on open-source examples because the projects don't have a traditional corporate organization.

Open-source projects are self-selected communities, which creates different motivational forces for the developers. In addition, open-source projects tend to have relatively flat and simple communication models. As a result, research on the subject has found that Brooks's law doesn't hold up as well: the more developers involved in an open-source project, the more likely that the project will succeed (source: *Brooks' versus Linus' law: an empirical test of open source projects [SEKH09]*).

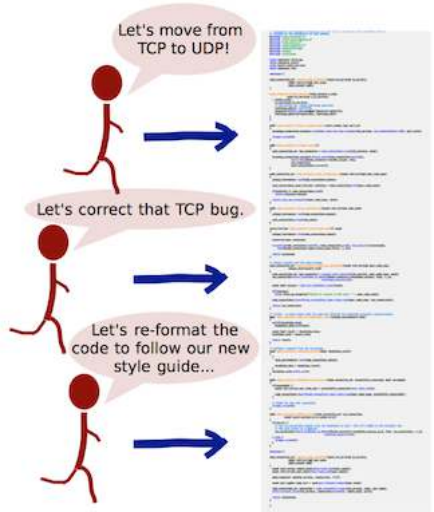
However, there are other aspects to consider. In a study on Linux, researchers found that "many developers changing code may have a detrimental effect on the system's security" (source: *Secure open source collaboration: an empirical study of Linus' law [MW09]*). More specifically, with more than nine developers, the modules are sixteen times more likely to contain security flaws. The result just means that open source cannot evade human nature; we pay a price for parallel development in that setting, too.

Anyway, we'll need to pretend a little in the following case studies. We need to pretend that the open-source projects are developed by a traditional organization. The intent is to show you how the analyses work so that you can use them on your own systems. Proprietary or not, the analyses are the same, but the conclusions may vary. With that in mind, let's get started!

Understand How Hotspots Attract Multiple Authors

Adding more people to a project isn't necessarily bad as long as we can divide our work in a meaningful way. The problems start when our architecture fails to sustain all developers.

We touched on the problem as we investigated hotspots. Hotspots frequently arise from code that accumulates responsibilities. That means programmers working on independent features are forced into the same part of the code. (Hotspots are the traffic jams of the software world.)

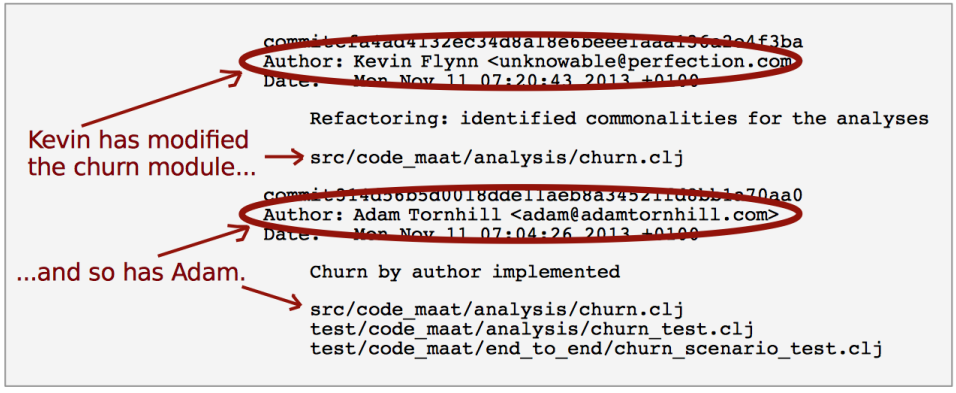


When multiple programmers make changes in parallel to the same piece of code, things often go wrong. We risk conflicting changes and inconsistencies, and we fail to build mental models of the volatile code.

If we want to work effectively on a larger scale, we need to ensure a certain isolation. Here's how you find that information.

Analyze Your Code for Multiple Authors

As you can see in the following figure, each commit contains information about the programmer who made the change. Just as we calculated modification frequencies to determine hotspots, let's now calculate author frequencies of our modules.



In this case study, we'll move back to Hibernate because the project has many active contributors. You can reuse your hib_evo.log log file if you still have it.

Otherwise, just create a new one, as we did back in *Generate a Version-Control Log*, on page 36.

Use the authors analysis to discover the modules that are shared between multiple programmers:

```
prompt> maat -l hib_evo.log -c git -a authors
entity,n-authors,n-revs
../persister/entity/AbstractEntityPersister.java,14,44
libraries.gradle,11,28
../internal/SessionImpl.java,10,39
../loader/Loader.java,10,23
../mapping/Table.java,9,28
...
```

The results show all modules in Hibernate, sorted by their number of authors. The interesting information is in the n-authors column, which shows the number of programmers who have committed changes to the module.

As you see, the AbstractEntityPersister.java class is shared between fourteen different authors. That may be a problem. Let's see why.

Learn the Value of Organizational Metrics

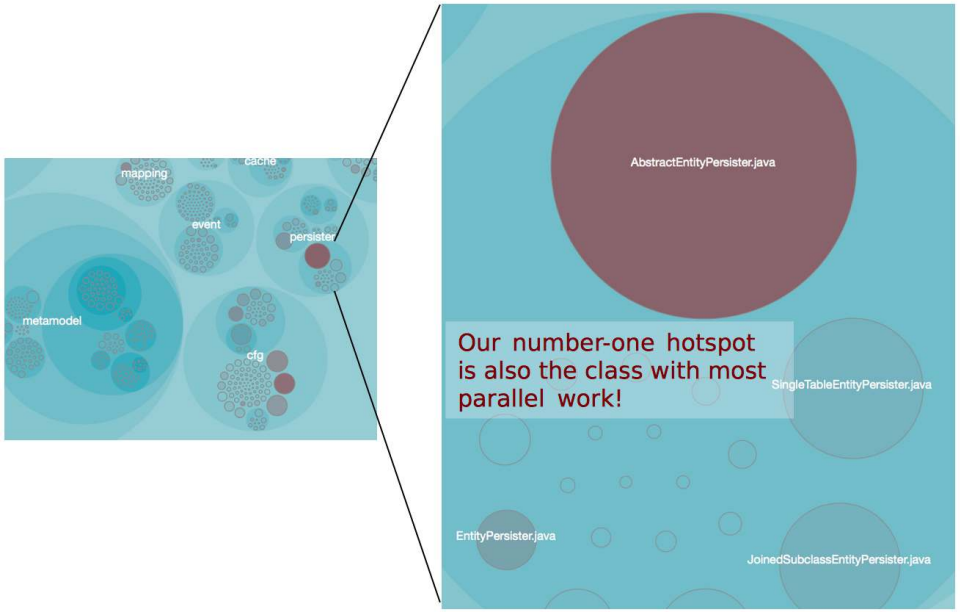
In an impressive research effort, a team of researchers investigated one of the largest pieces of software ever written: Windows Vista. The project was investigated for the links between product quality and organizational structure. (Read about the research in *The Influence of Organizational Structure on Software Quality [NMB08]*.) The researchers found that organizational metrics outperform traditional measures, such as code complexity or code coverage. In fact, the organizational structure of the programmers that create the software is a better predictor of defects than any property of the code itself!

One of these super-metrics was the number of programmers who worked on each component. The more parallel work, the more defects in that code. This is similar to the analysis you just performed on Hibernate. Let's dig deeper.

Measure Temporal Coupling over Organizational Boundaries

The research findings from the Windows Vista study suggest that quality decreases with the number of programmers. It's easy to see the link to Brooks's law: more programmers implies more coordination overhead, which translates to more opportunities for misunderstandings and errors.

One way to highlight the severity of parallel work is by comparing the modules with most authors to the hotspots you identify. So let's look back at the hotspot analysis we did in Chapter 4, *Analyze Hotspots in Large-Scale Systems*, on page 35:



As you can see, the AbstractEntityPersister—the class with the most programmers—is also our number-one hotspot. That means the trickiest part of the code affects the most programmers. That can't be good. Let's see why.

Interpret Conway's Law

Brooks wasn't the first to point out the link between organization and software design. A decade earlier, Melvin Conway published his classic paper that included the thesis we now recognize as *Conway's Law* (see *How do committees invent? [Con68]*):

> Any organization that designs a system (defined more broadly here than just information systems) will inevitably produce a design whose structure is a copy of the organization's communication structure.

Conway's law has received a lot of attention over the years, so let's keep this brief. Basically, we can interpret Conway's law in two ways. First, we can interpret it in the cynical (and fun) way, as in the The Jargon File:[1] "If you have four groups working on a compiler, you'll get a 4-pass compiler."

1. http://catb.org/~esr/jargon/html/C/Conways-Law.html

The other interpretation starts from the system we're building: given a proposed software architecture, how should the optimal organization look to make it happen? When interpreted in reverse like this, Conway's law becomes a useful organizational tool. Let's see how you can use it on existing systems.

Use Conway's Law on Legacy Systems

As you learned in *Optimize for Understanding*, on page 2, we spend most of our time modifying existing code. Even though Conway formulated his law around the initial design of a system, the law has important implications for legacy code as well.

There's a big difference when you need to cooperate with a programmer sitting next to you, versus someone you've never met who is located in a different time zone. So let's find out where your communication dependencies are.

Start your analysis from the hotspots in the system, since these are central to your daily work. From there, identify other modules that have a temporal coupling to the hotspots. Once you know the modules that evolve together, look for the main developers of those modules. From there, we can start to reason about ease of communication. Here's how you do it.

Calculate Temporal Coupling over a Day

To analyze temporal coupling over organizational boundaries, we need to consider all commits during the same day as parts of a logical change set. Different authors will by definition commit their work independently, so we can't limit ourselves to modules in the same commit. We focus on a single day as a heuristic; modules that keep changing together that often over a period of time are probably related.

In the `authors` analysis, we identified `AbstractEntityPersister` as the module with most contributors. Because it's also a hotspot, we'll zoom in on it. Specify the `--temporal-period 1` option to make Code Maat treat all commits within the same day as a single, logical change set:

```
prompt> maat -c git -l hib_evo.log -a coupling --temporal-period 1
entity,coupled,degree,average-revs
..
../AbstractEntityPersister.java, ../CustomPersister.java,45,11
../AbstractEntityPersister.java, ../EntityPersister.java,45,11
../AbstractEntityPersister.java, ../GoofyPersisterClassProvider.java,43,12
...
```

The analysis results show that `AbstractEntityPersister` tends to change together with a bunch of other modules. Every time you make a change to the

AbstractEntityPersister, there's a 45 percent chance that three different classes will change during that same day.

The next step is to find out the main developers of the coupled modules. Once we have that information, we can compare it to the formal organization of developers and reason about ease of communication. Here's the analysis.

Evaluate Communication Costs

To reason about communication costs, we need to know who's communicating with whom. The analysis model we'll use is based on the idea that we can identify a *main developer* of each module.

We'll define the main developer as the programmer who's likely to know the most about the specific code. Because code knowledge isn't easy to measure, we'll use the number of contributed lines of code instead.

Like all heuristics, our metric has its flaws—in particular, since we measure something as multifaceted as programmer contributions. That doesn't mean the results are useless; the metrics are there to support your decisions, not to make them for you. Your knowledge and expertise cannot be replaced by data.

So sure, using the number of added lines of code is a rough approximation, but the overall results tend to be good enough. Let's see the metric in action.

Identify Main Developers by Removed Code

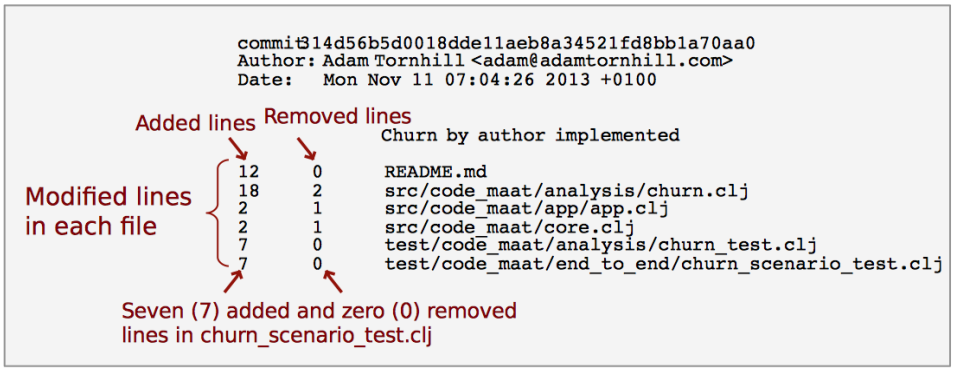
Since we used the number of added lines to identify main developers, this means that a copy-paste cowboy could easily conquer parts of the codebase. So, let's turn it around and find an alternative.

Good programmers take pride in doing more with less. That means you could use the number of *removed* lines of code instead. That tweak to the algorithm would identify developers who actively refactor the code. Since Code Maat implements the analysis, refactoring-main-dev, go ahead and try it yourself.

In practice, you'll often find that in projects that care about code quality, like Hibernate, the two algorithms identify the same people. This is why we used the conceptually simpler metric of added lines in our case study.

Identify Main Developers

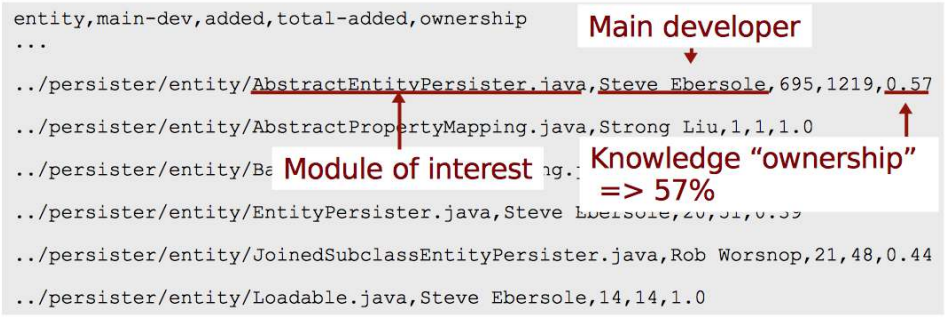
As the following figure shows, the contribution information is recorded in every commit. We just need to instruct Code Maat to sum it up and calculate a degree of ownership for each entity. The author with most added lines is considered the main developer, the knowledge owner, of that module.



You perform a main developer analysis with the main-dev option. On Hibernate, this analysis will deliver a long list of results. (Remember, Hibernate is a large codebase.) So let's save the results to a file for further inspection:

```
prompt> maat -c git -l hib_evo.log -a main-dev > main_devs.csv
```

Let's look inside main_devs.csv to find the main developer of AbstractEntityPersister:



The results identify Mr. Ebersole, the productive project lead on Hibernate, as the main developer. In our analysis period, he contributed 695 of the 1,219 lines that have been added to AbstractEntityPersister, an ownership of 57%.

Remember, we're after expensive communication paths. So who does Mr. Ebersole have to communicate with? To find out, we need to identify the main developers of the modules that are temporally coupled to AbstractEntityPersister. Let's look at that.

This Only Works on Git

The main-dev analysis we ran only works on Git. The reason is that neither Subversion nor Mercurial includes the number of modified lines of code in its log files. Fortunately, there's a workaround that's almost as good.

If you're on another version-control system—for example, Subversion—then run the main-dev-by-revs analysis instead. That analysis classifies the programmer who has contributed the most commits to a specific module as its main developer.

Analyze Contributions to Coupled Modules

When we analyzed the temporal coupling to AbstractEntityPersister in the code on page 140, we identified three dependent modules. Let's extract the main developers of those from our main_devs.csv analysis results:

Coupled Module	Main Developer	Ownership (%)
CustomPersister.java	Steve Ebersole	58
EntityPersister.java	Steve Ebersole	39
GoofyPersisterClassProvider.java	Steve Ebersole	54

As you can see in the preceding figure, all entities that have a temporal coupling to the AbstractEntityPersister are within the mind of the same developer. Looks good—or does it? The low ownership degree, 39 percent, of EntityPersister.java indicates that the code is shared between several authors. Let's see how much each programmer contributed before we can feel safe.

Calculate Individual Contributions

The contributions of each developer are available from the same version-control information. We just need to use an entity-ownership analysis instead. Here's how it looks, filtered for EntityPersister:

```
prompt> maat -c git -l hib_evo.log -a entity-ownership
entity,author,added,deleted
...
../EntityPersister.java,Gail Badner,1,0
../EntityPersister.java,Steve Ebersole,20,9
../EntityPersister.java,Rob Worsnop,3,0
../EntityPersister.java,Eric Dalquist,19,8
../EntityPersister.java,edalquist,8,2
...
```

Oops—one of the programmers, Eric Dalquist, uses two committer names. We see it immediately in the output above, but Code Maat had no way to know. That means we've run into our first analysis bias!

The problem is easy to fix once we've identified the authors with multiple aliases. On your own projects, you want to investigate and clean the log *before* any analyses. Once we've done a quick search-and-replace on our data, we rerun the analysis on the cleaned log:

A) Original Data

Coupled Module	Main Developer	Ownership (%)
CustomPersister.java	Steve Ebersole	58
EntityPersister.java	Steve Ebersole	39
GoofyPersisterClassProvider.java	Steve Ebersole	54

Flawed data

B) Cleaned Data

Coupled Module	Main Developer	Ownership (%)
CustomPersister.java	Steve Ebersole	58
EntityPersister.java	Eric Dalquist	53
GoofyPersisterClassProvider.java	Steve Ebersole	54

Corrected!

The algorithm now identifies the correct main developer. If we put our results together, we can start to reason about communication:

1. We have a temporal coupling between EntityPersister and AbstractEntityPersister.
2. Since AbstractEntityPersister is a hotspot, we know we need to modify the code frequently.
3. That means its coupled part, EntityPersister, will need to change often as well, but the two modules have different main developers!

Let's look at the consequences.

Check Communication Dependencies Against Your Organization

Hibernate is open source with a development process that's different from what most companies in the industry use. Without more context and insight, it's hard to reason about the consequences of our findings.

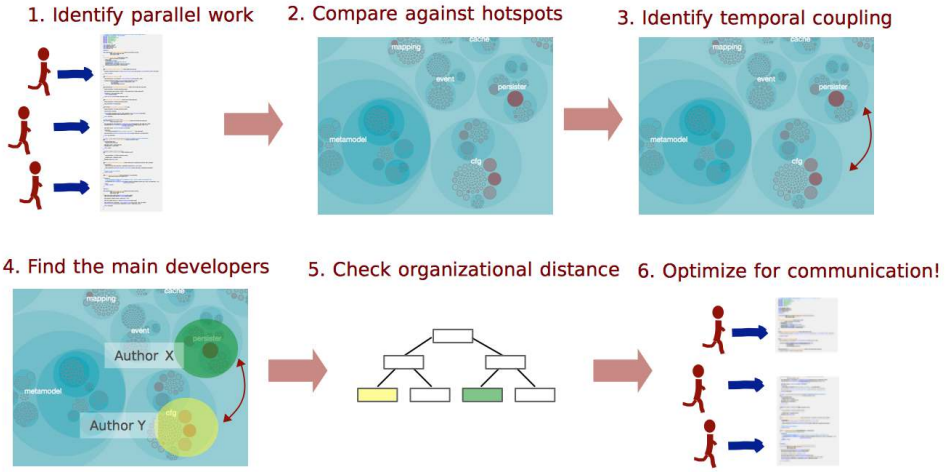
What we do know is that communication costs are likely to increase with organizational distance. So when you identify a case like this in your own projects, you want to check the information against your organization. Are the two programmers on the same team? Are they located at the same site? If not, it may be a concern.

When we work together, we develop informal communication channels. We meet in the hallway, grab a coffee together in the morning, or chat about our work during lunch breaks. If we lose those opportunities for informal talks, our products suffer.

In this chapter, you got the basic tools to start analyzing how well your own development work aligns with those communication channels. Let's sum up.

Take It Step by Step

The dependencies between organizational aspects and system design are complex. As a consequence, the analysis in this chapter contains multiple steps. Let's recap them:



Using the steps above, you're now able to perform a basic analysis of communication problems and team productivity bottlenecks. When you find them, you need to react. Unfortunately, that's the hard part because it depends on your specific organization and system.

One typical action is to rearrange the teams according to communication needs. Because those needs will change over the course of a longer project, you probably need to revisit these topics at regular intervals. Done right, such a rebalancing of project teams has been found to minimize communication overhead. (See *The Effect of Communication Overhead on Software Maintenance Project Staffing [DHAQ07]*.)

This is what happened on the project I told you about at the beginning of the chapter. The project was supposed to deliver in three months. After one year of expensive and intense parallel development, the project was put on hold and analyzed, and the teams were reorganized. After that, we had fewer developers working on the code, and yet you won't be surprised to learn that we got a productivity boost that lasted.

Sometimes it's easier—and indeed more appropriate—to redesign the shared parts of the system to be more inline with the structure of the organization. This alternative is close to what Conway himself suggests in his classic paper as he concludes that "a design effort should be organized according to the

need for communication" (source: *How do committees invent? [Con68]*). The techniques in this chapter are there to help you with this challenging task.

Toward a Communication Map

In this chapter, you learned that the number of authors behind a module correlates to the number of post-release defects in that code. We discussed how this is a consequence of the increased communication overhead that comes with parallel work. You also learned to investigate it yourself through an `authors` analysis.

We then looked at Conway's law and how the way we organize our work impacts the code. You learned that your project organizations must align with the way the system is structured. You also got the basic tools to perform that analysis. During these analyses, we found a way to identify how much each developer has contributed in terms of code. We used this metric as a crude device to find the main developers.

With the analyses in this chapter, you'll be able to spot many organizational problems that creep into your code. But we can do even better. In the next chapter, we'll look at individual developer patterns. As you'll see, those patterns make good predictors of defects. As a bonus, you'll learn to build a complete knowledge map of your system. It's a map that helps you plan, communicate, and estimate knowledge loss in case a core developer leaves. Let's move on!

CHAPTER 13

Build a Knowledge Map of Your System

In the previous chapter, you learned to analyze how well your team structure fits the way you work with the code. We also looked at how the number of authors of a piece of code affects its quality. Now we'll dive deeper into different author patterns to see how your organization affects your codebase.

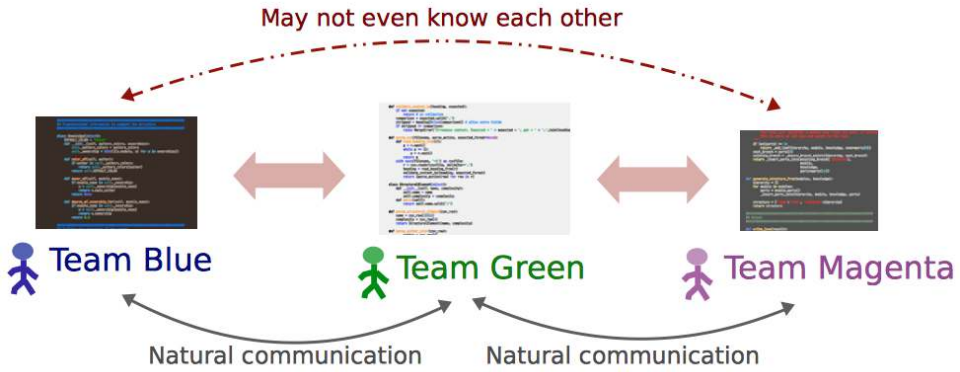
Coordination is vital to all large-scale software development. It's also important to find out what and with whom we need to coordinate. In this chapter, we'll take a look at supporting techniques to map out the knowledge distribution in our codebase. As a bonus, you'll see how the information lets you predict defects, support planning, and assess knowledge drain in case a developer leaves your project.

At the end of this chapter, you'll have a radically different view of your system. Let's get started!

Know Your Knowledge Distribution

A while ago, I worked with a large development organization. We were hundreds of developers organized in multiple divisions and teams. To make it work, each team was responsible for one subsystem. The subsystems all had well-documented interfaces. However, sometimes the API had to change to support a new feature. That's where things started to get expensive.

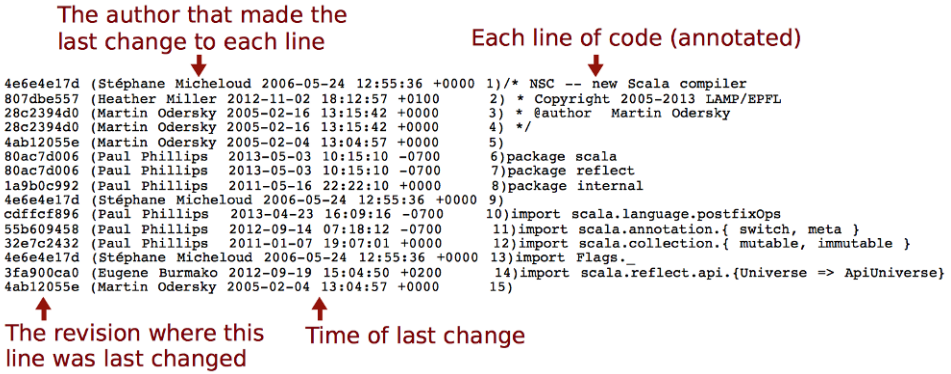
You're probably reminded of Conway's law from the previous chapter. The team organization was closely aligned to the system architecture. As a consequence, the communication worked well—at least on the surface. When we needed to change an API, we knew which team we should talk to. The problem was that these changes often rippled through the system; a change to our API meant another subsystem had to change as well. And to change that subsystem, our peers had to request a change to yet another API, and so on.



As the figure shows, when an organization creates isolated teams, collaboration across system boundaries suffers. While we may know the expertise on the team responsible for the interfaces we use, we rarely know what happens behind them. This is a problem when it comes to design, code reviews, and debugging. Ideally, you'd like to get input from everyone who's affected by a change. So let's find out who they are.

Find the Author to Praise

Modern version-control systems all provide a `blame` command. (I love that Subversion aliases the command as `praise`!) `blame` is useful if you know exactly which module you need to change. As you can see in the following figure, `blame` shows the author who last modified each line in a given file.



The information from `blame` is useful, but it doesn't take us far enough. If we don't know that part of the system, which is probably why we want to talk to someone else in the first place, then we don't know which file to inspect. And

even when we do, the information from blame is low-level. What we'd need is a summary, a high-level overview.

blame is also sensitive to superficial changes. So if we want information that reflects knowledge, we need to look deeper at the contributions that led up to the current code. Let's see how to do that.

It's Not About Blame or Praise

The analyses in this part of the book are easy to misuse by applying them as some kind of evaluation of programmer performance or productivity. There are several reasons why that's a bad idea. Let's focus on the one that social psychologists call a *fundamental attribution error*.

The fundamental attribution error describes our tendency to overestimate personality factors when we explain other people's behavior. For example, when you see that I committed a bunch of buggy spaghetti last week, you know it's because I'm a bad programmer, irresponsible, and perhaps even a tad stupid. When you, on the other hand, deliver scrappy code (yes, I know—it's a hypothetical scenario), you know it's because you were close to a deadline, had to save the project, or just intended it to be a prototype. As you see, we attribute the same observable behavior to different factors depending on whether it concerns us or someone else.

There's also a group aspect to this bias. When we judge the behavior of someone closer to us, such as a team member, we're more likely to understand the situational influence. That means the fundamental attribution error is a bias that we can learn to avoid. We just need to remind ourselves that the power of the situation is strong and often a better predictor of behavior than a person's personality.

Dig Below the Surface with Developer Patterns

Chapter 12, *Discover Organizational Metrics in Your Codebase*, on page 133, used version-control data to identify the number of developers behind each module. While the measure is correlated with bugs, the number itself doesn't reveal much. Even with many contributors, a module may still have one main developer who maintains overall consistency while other programmers contribute minor fixes to the code. Or, it could indeed be a shared effort where many different programmers contribute significant chunks of the total code.

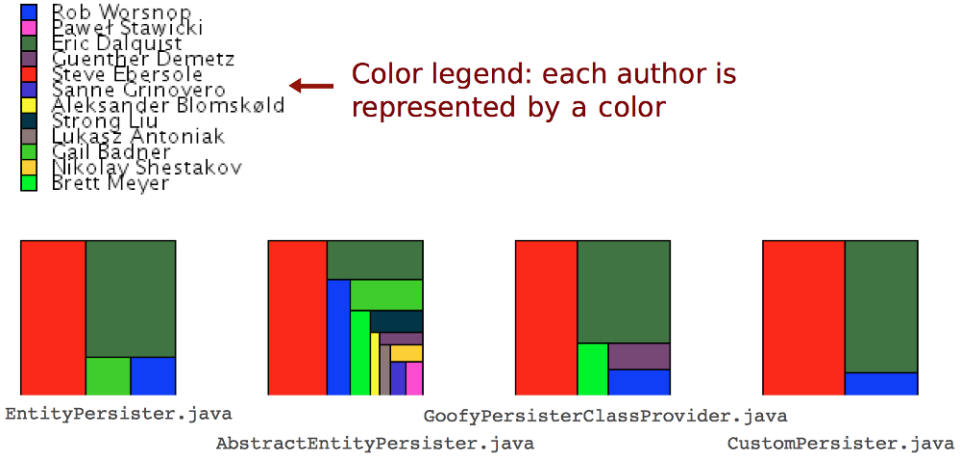
To dig deeper, we need to get a summary of individual contributions. The algorithm we use summarizes the number of commits for each developer and presents it together with the total number of revisions for that module. Here's how the metric looks on Hibernate:

```
prompt> maat -c git -l hib_evo.log -a entity-effort
entity,author,author-revs,total-revs
...
 AbstractEntityPersister.java,Steve Ebersole,17,44
 AbstractEntityPersister.java,Brett Meyer,3,44
 AbstractEntityPersister.java,Rob Worsnop,5,44
...
 AbstractEntityPersister.java,Gail Badner,4,44
 AbstractEntityPersister.java,Paweł Stawicki,1,44
 AbstractEntityPersister.java,Strong Liu,2,44
...
```

The results above are filtered on the AbstractEntityPersister module that we identified as a potential problem back in *Evaluate Communication Costs*, on page 141. While these analysis results let you reason about how fragmented the development effort is, the raw text output soon becomes daunting; it's hard to get the overall picture. So let's turn to a more brain-friendly approach.

Visualize Developer Effort with Fractal Figures

Take a look at the following figure. In contrast to the raw analysis results, the *fractal figures* visualization immediately provides you with a view of how the programming effort was shared—how fragmented the developer effort is for each module in your system.



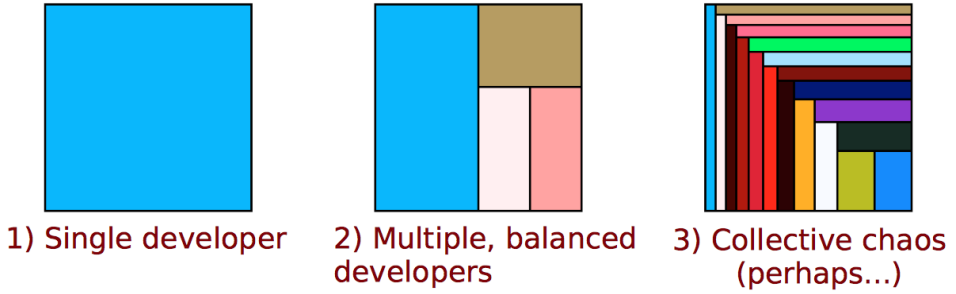
The fractal figures algorithm is simple: represent each programmer with a color and draw a rectangle whose area is proportional to the percentage of commits by that programmer. You can also see that the rectangles are rendered in alternating directions to increase the visual contrast between different parts. (You'll find more details in the original research paper *Fractal Figures: Visualizing Development Effort for CVS Entities [DLG05]*.)

If you want to try fractal figures on your own system—and you really should—check out the implementation and documentation on GitHub.[1] All you need is a result file from a Code Maat `entity-effort` analysis.

Now, let's see what the different patterns tell us about the codebase.

Distinguish the Ownership Models

Three basic patterns keep showing up when you visualize development effort, and these patterns can be used to predict code quality. You can see the patterns in the following figure:



From a communication point of view, a single developer provides the simplest communication structure; there's just one person to talk to. It's also likely that the code within that module is consistent. The quality of the code depends, to a large extent, on the expertise of one single developer.

The second case with multiple, balanced developers is more interesting. Often, such code has one developer who contributed the most code. It turns out that the ownership proportion of that main developer is a good predictor of the quality of the code! The higher the ownership proportion of the main developer, the fewer defects in the code. (See *Don't Touch My Code! Examining the Effects of Ownership on Software Quality [BNMG11]*.)

1. https://github.com/adamtornhill/FractalFigures

An even stronger predictor of defects is the number of minor contributors. You see an example of that in case 3 in the preceding figure. When we make a change to a module where we are the main developer, we sometimes need to change a related piece of code that we haven't worked on before. As such, we don't know the background and thinking that went into the original code. It's in that role as minor contributors that we're more likely to introduce defects.

The fractal figures give you another investigative tool to uncover expensive development practices. Once you've identified one of the warning signs, such as many minor contributors, you react by performing code reviews, running a hotspot analysis, and talking to the contributing programmers to see whether they experience any problems.

Do More with Fractal Figures

Fractal figures work well on an architectural level, too. On this level, you use them to visualize the fragmentation of subsystems or even whole systems. You generate architectural fractal figures by specifying a transformation as we did back in *Specify Your Architectural Boundaries*, on page 93. Then you just run an entity-effort analysis and tell Code Maat to use your transformation.

Another interesting variation on fractal figures is to use their size dimension to express other properties of the code. For example, you can use the size of each figure to visualize complexity or the number of historic bugs in each module. Used that way, fractal figures allow you to present a lot of information in a compact and brain-friendly way.

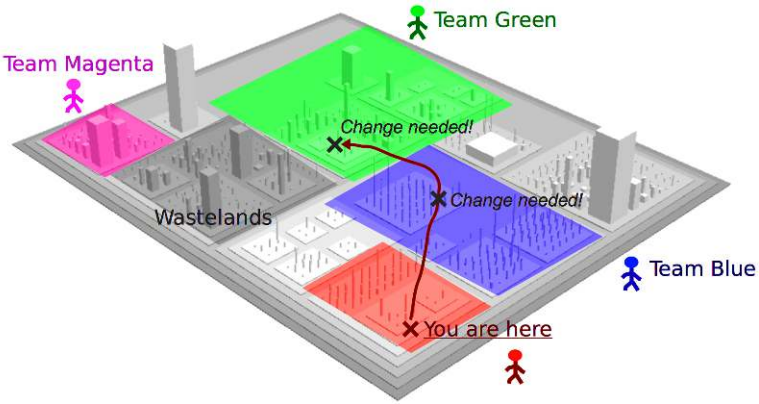
Grow Your Mental Maps

Remember how we discussed geographical offender profiling back in *Learn Geographical Profiling of Crimes*, on page 16? We built our hotspot analyses based on the idea that just as we spot patterns in the movement of criminals, our version-control data lets us identify patterns in the changes we make to the codebase. What I didn't mention back then is that the movement of offenders is constrained by a concept called *mental maps*.

A mental map is our subjective view of a specific geographic area. Our mental maps deviate from how a real map would look. For example, geographical hindrances such as highways and rivers often constrain and skew our perception of an area. In the small town where I grew up, it took me years to venture across the heavily trafficked road that cut through the city. As a consequence,

my mental map ended at that the street. It was the edge of the world. Similarly, the mental maps of criminals shape where their crimes take place.

We programmers have mental maps, too. Over time, at least when we work with others, we tend to specialize and get to know some parts of the system better than others. These knowledge barriers shape our perception of the system—our mental maps constrain our view of the system to the parts we know. Let's see how we can tear them down.



Explore Your Knowledge Map

Imagine for a moment that you had a map of the individual knowledge distribution in your organization. No, no—not some out-of-date Excel file that's stashed away on the intranet. To be useful, the information has to be based on how we actually work. In reality, in code.

The concept we'll develop is a *knowledge map*. A knowledge map lets you find the right people to discuss a piece of code, fix hotspots, and help out with debugging. Let's see how the end result looks in the figure on page 154, so we know where we're heading.

This knowledge map of the programming language *Scala* is based on the same concept as the fractal figures, as each programmer is assigned a color. This lets us reason about knowledge distribution on a system level. For example, the map shows that components such as `scaladoc` (generates API documentation), `asm` (Java bytecode manipulation), and `reflect` (dynamic type inspection and manipulation) seem to be in the hands of a single developer—there's little knowledge distribution. In contrast, other components, such as the `compiler`, exhibit a shared effort, with contributions from multiple developers.

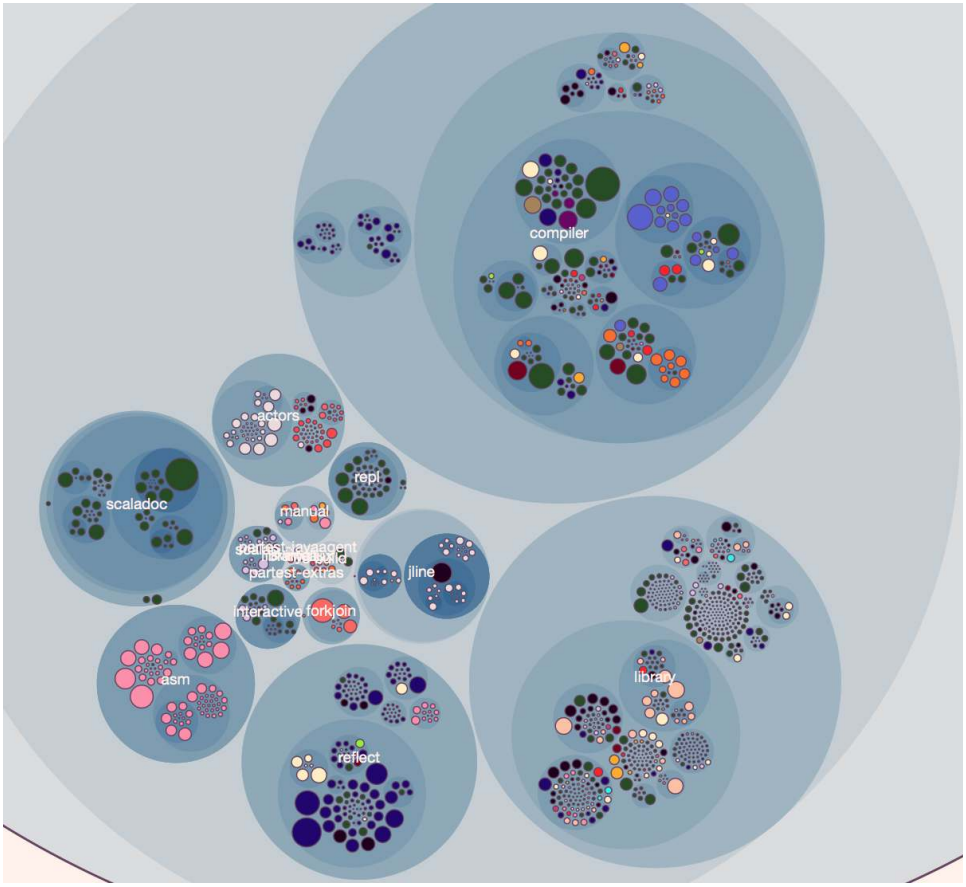


Figure 1—Knowledge map showing the main developer (indicated by color) of each module.

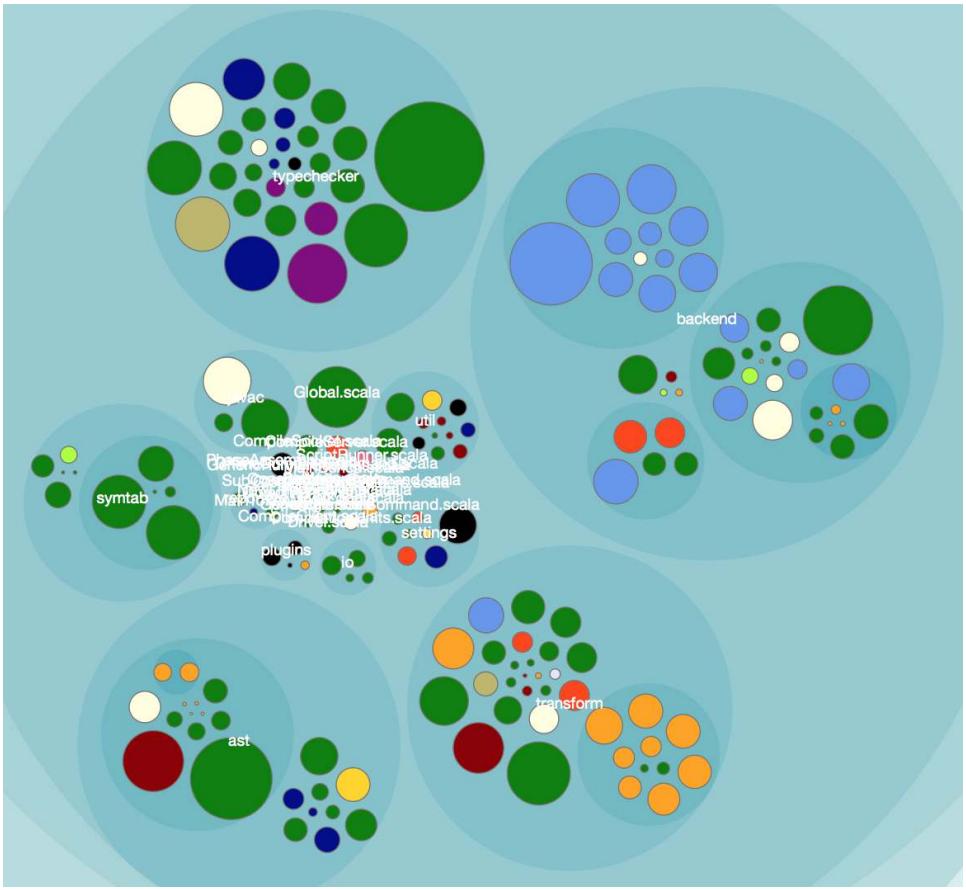
The visualization is based on interactive enclosure diagrams, just like we used back in *Visualize Hotspots, on page 38*. Have a look at the visualization samples you downloaded from the Code Maat distribution site.[2] There's a scala directory inside that bundle. Open a command prompt in that directory and launch Python's SimpleHTTPServer:

```
prompt> python -m SimpleHTTPServer 8888
```

Now you can point your browser to http://localhost:8888/scala_knowledge.html to view the Scala knowledge map. The interactive visualization lets you zoom in on

2. http://www.adamtornhill.com/code/crimescenetools.htm

individual components for a detailed examination. Let's see what the map tells us about the Scala compiler.



Pretend for a moment that we join this project and need to make a change to the typechecker. Our map immediately points us to the green developer as the correct person to discuss our proposed design with. If we come across some task in the backend instead, green may still be able to help us out, but we're more likely to get the right expertise if we involve the light-blue developer, too. All right, let's see how you can create this map.

Investigate Knowledge in the Scala Repository

The Scala codebase is fairly large, with approximately 300,000 lines of code. The project flourishes with developer activity; over the past two years, more than 150 developers have contributed to the project. That scale of development

makes it virtually impossible for any single developer to keep it all in his or her head. So, let's put our map together to guide us.

Start by cloning Scala's git repository:

```
prompt> git clone https://github.com/scala/scala.git
Cloning into 'scala'...
...
```

To get reproducible results, we need to go back in time to when this book was written. We do that with the piece of git magic we learned in *Turn Back Time*, on page 24. But we need to be careful. Because Scala uses different branches, we need to know where we are before we travel in time. We do that with the git status:

```
prompt> git status
On branch 2.11.x
Your branch is up-to-date with 'origin/2.11.x'.
```

You use the name of the branch—in this case, origin/2.11.x—as the final argument to the command that rolls back the codebase:

```
prompt> git checkout `git rev-list -n 1 --before="2013-12-31" origin/2.11.x`
...
HEAD is now at 969a269..
```

Now your local Scala repository should look just like it did at the end of 2013, and you're ready to analyze.

Analyze the Knowledge Distribution

Because we want to use the information to find the right people to talk to, we have to identify the developers who know the different parts of the system. This is a similar problem to the one we solved back in *Evaluate Communication Costs*, on page 141, where we used a main developer analysis to identify the top contributor to each module.

We start by generating a version-control log:

```
prompt> git log --pretty=format:'[%h] %an %ad %s' --date=short \
 --numstat --before=2013-12-31 --after=2011-12-31 > scala_evo.log
```

The command limits our analysis period to the last two years. That's because knowledge is fragile and dissolves over time; if we haven't touched a piece of code for a couple of years, it's unlikely that we remember much about it. In addition, code that's been stable for that long is rarely the focus of our daily activities. But remember that these time periods are all heuristics that you may have to fine-tune in your own projects.

Now that we have a version-control log, let's see what happens when we aggregate all that information to identify the main developers:

```
prompt> maat -c git -l scala_evo.log -a main-dev > scala_main_dev.csv
```

The command saves the analysis results to the file scala_main_dev.csv for further processing. If you look inside it, you should see the main developer of each module:

```
entity,main-dev,added,total-added,ownership
..
GenICode.scala,Paul Phillips,584,1579,0.37
ICodeCheckers.scala,Jason Zaugg,19,44,0.43
ICodes.scala,Grzegorz Kossakowski,16,32,0.5
...
```

The main developer information serves as the basis of our knowledge map. We just need to project the information on the geography of our system. Let's do it.

Project the Main Developers onto a Map

Back in Chapter 3, *Creating an Offender Profile*, on page 23, we used lines of code as a proxy for complexity as we hunted hotspots. It makes sense to use the same metric and visualization here, since it allows you to compare hotspots against the knowledge map.

We collect the information with cloc:

```
prompt> cloc ./ --by-file --csv --quiet --report-file=scala_lines.csv
```

Now we have the elements we need. We have the structure in scala_lines.csv and the presumed knowledge owners in scala_main_dev.csv. Let's combine those with a unique color for each individual developer.

Specify the Color of Each Developer

We humans can distinguish between hundreds of thousands of different color variations. However, in your visualization, you want to keep a larger distinction between each color. Several tools can help you select good color schemes. (See, for example, ColorBrewer.[3])

The colors we use in Figure 1, *Knowledge map showing the main developer (indicated by color) of each module*, on page 154, are specified as HTML5 color names.[4] Take a look in the visualization samples we downloaded from the

3. http://colorbrewer2.org/
4. http://www.w3schools.com/html/html_colornames.asp

Code Maat distribution site.[5] Inside that bundle, there's a scala folder with a scala_author_colors.csv. That file specifies the mapping from author to color for the top contributors. Here's a sample:

```
author,color
Martin Odersky,darkred
Adriaan Moors,orange
Paul Phillips,green
...
```

Now that we've assigned a color to each developer, we can put it all together.

Generate Your Own Map

As we discussed in *Visualize Hotspots*, on page 38, the enclosure diagram is built on D3.js.[6] Since D3.js is data-driven, we need to serve it a JSON document that specifies the content to visualize.

That JSON document is generated by a Python script included on the Code Maat distribution page. Before you run it, just remember to specify your local path to the Python scripts:

```
prompt> python scripts/csv_main_dev_as_knowledge_json.py \
 --structure scala_lines.csv --owners scala_main_dev.csv \
 --authors scala_author_colors.csv > scala_knowledge_131231.json
```

Now you should have a scala_knowledge_131231.json file in your local directory. The JSON inside that file should be identical to the one that was used to create Figure 1, *Knowledge map showing the main developer (indicated by color) of each module*, on page 154.

Once you have the JSON document, you can reuse the d3.js code that's included in the Code Maat sample visualization of Scala. Just open the scala_knowledge.html file and replace the included scala_knowledge_131231.json with a reference to your own content. Explore, experiment, and automate from there.

Visualize Knowledge Loss

Think back to the last project you worked on. What if one of the core developers suddenly left? Literally just walked out the door. What parts of the code would now be left in the wild? And what parts should the next developer start to look at? Most of the time, we don't know the answers. Let's see how our knowledge map puts us in a better position.

5. http://www.adamtornhill.com/code/crimescenetools.htm
6. http://d3js.org/

Learn the Predictive Power of Abandoned Code

Practices such as good documentation, close collaboration, and code reviews help to spread the knowledge of the codebase. But even under ideal conditions, practices can never replace the intricate knowledge that comes from working with a piece of code over time. That's one reason why the number of ex-developers who have worked on a component is a good predictor of the number of post-release defects the code will have. (See *The Influence of Organizational Structure on Software Quality [NMB08]* for the original research.)

In early 2014, the Scala project faced that challenge. Paul Phillips, who'd worked on the codebase for five years, left the project–you can watch him tell the story here.[7] Let's see if we can find the resulting knowledge gap.

Identify Abandoned Code

You've already seen how the knowledge map lets you identify the main contributors for each module. When it comes to identifying abandoned code—that's code written by a programmer who's no longer in the company—we can simplify it. The only thing we actually need is a color to identify the ex-programmers.

In this case, we just assign a color to Paul Phillips:

```
author,color
Paul Phillips,green
```

Save the CSV as scala_ex_programmers.csv and generate a JSON document for our new visualization:

```
prompt> python scripts/csv_main_dev_as_knowledge_json.py \
  --structure scala_lines.csv --owners scala_main_dev.csv \
  --authors scala_ex_programmers.csv > scala_knowledge_loss.json
```

You should now have a scala_knowledge_loss.json ready to visualize the knowledge drain in the Scala project. All we need to do is open the scala_knowledge.html file and point to our own JSON file. The figure on page 160 shows the resulting knowledge loss.

A good programmer like Paul Phillips is, of course, impossible to replace. What we can do, however, is to use our knowledge of where the abandoned code is as an input to planning and risk assessments. Since we now know where our blind spots are, we need to allocate extra time in case we plan modifications to them. It's still hard, but at least we know that up front.

7. https://www.youtube.com/watch?v=uiJycy6dFSQ

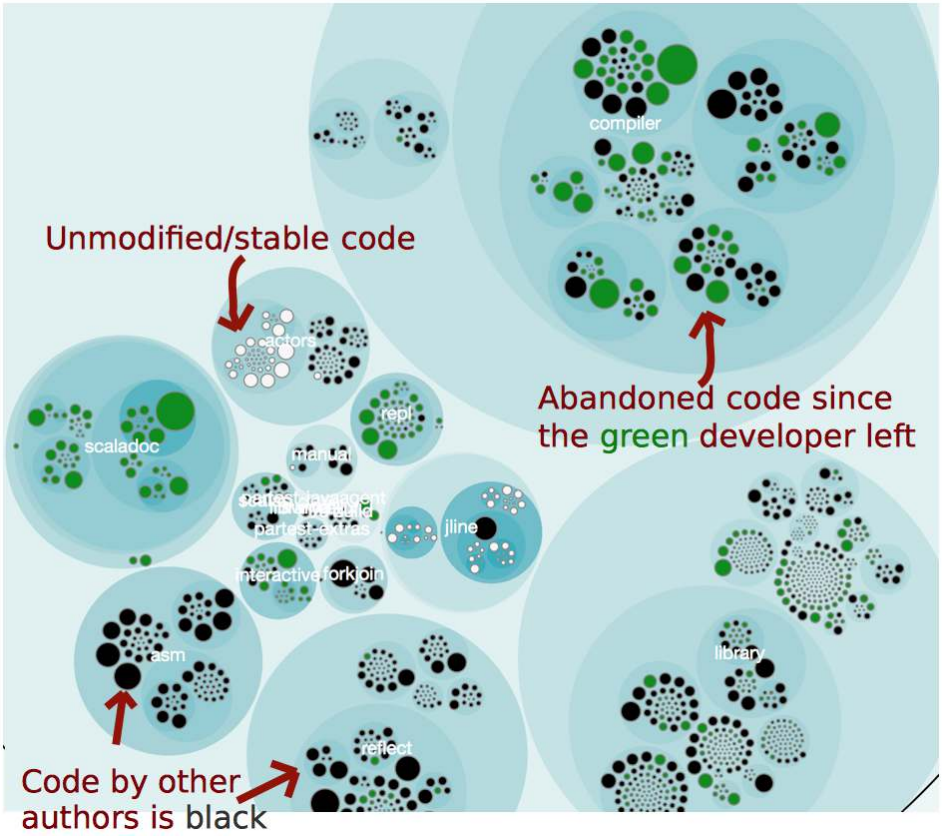


Figure 2—The green color marks code written by a programmer who's no longer with the company.

Know the Uses and Misuses

The knowledge map is useful to everyone on a project:

- We developers use it to identify peers who can help out with code reviews, design discussions, and debugging tasks.
- New project members use the knowledge as a communication aid.
- Testers grab a digital copy of the map to find the developer who's most likely to know about a particular feature.
- Finally, technical leaders use the information to evaluate how well the system structure fits the team structure, identify knowledge loss, and ensure that we get the natural informal communication channels we need to write great code.

A knowledge map is also a great supplement to a temporal coupling analysis. When you identify components that are temporally coupled, you want to break that dependency. In the meantime, you want to ensure that the main developers of the coupled components work closely together.

Unfortunately, it's easy to misuse the knowledge map. It's not a summary of individual productivity, nor is it a way to evaluate people. Used that way, the information does more harm than good. Eventually, we developers learn to game the metric, and the quality of the code and the work environment suffers in the process. Don't go there.

Get More Details with Code Churn

We covered a lot of ground in this chapter. First, we talked about the perils of fragmented development efforts. You learned about its link to post-release defects. You also learned to analyze the problem and visualize the result with fractal figures. With fractal figures, you got an overview of how the development was shared.

We then took the idea to the system level. You've already learned that much of our time as software developers is spent communicating. Now you've seen how to build a knowledge map of your codebase to help you find the correct person to discuss a particular feature or piece of code with. We also saw that these techniques go beyond programming and provide information useful to technical managers as well.

Our fractal figures and maps are based on how we as developers have worked so far. The measure we used builds on a concept called *code churn*. Code churn measures the rate at which our code evolves. By digging deeper into the subject, you'll be able to predict defects, identify unstable parts in your system, and even gain important information about the real process your team follows. Let's see how it works!

CHAPTER 14

Dive Deeper with Code Churn

In the previous chapter, we looked at the risks and costs of fragmented development efforts. We also created a knowledge map to serve as a communication and planning aid. In this chapter, we'll build on that information by introducing the concept of code churn.

Code churn is a measure that tells us the rate at which our code evolves. By analyzing code churn, we can identify fluctuating parts of our codebase. This lets us predict both post-release defects and unstable feature areas.

You'll also see how code churn lets you reverse-engineer your team's coding process. That information lets you evaluate your development practices and investigate your potential process loss. As always, let's start with a war story.

Cure the Disease, Not the Symptoms

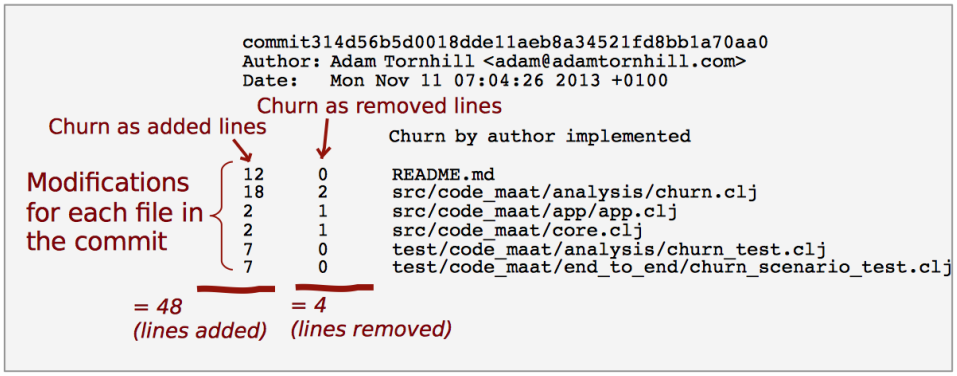
So, how much do you enjoy merging code from different development branches? Not at all? Neither do I. If you're like me, you've probably had your fair share of merge nightmares. Remember the project I talked about in Chapter 12, *Discover Organizational Metrics in Your Codebase*, on page 133? You know, the story of the project that got stacked with four times as many developers as needed to get it done "faster"?

In that project, one of the major bottlenecks turned out to be parallel work. We often spent a few hours adding a feature, only to find out that the relevant code had been rewritten in another branch. Not only is that way of working frustrating, it's also expensive and a sure way to breed defects. We don't want that.

Better tools like Git and Mercurial have improved the situation, but we cannot expect our tools to cure dysfunctional practices. Let's see how code churn can uncover them.

Meet Code Churn

Code churn refers to a family of measures that reflect how rapidly your codebase evolves. Each time you commit a change to your source code, your version-control system records the lines you've added, modified, and deleted. The following image shows how it looks in Git.



Over time, we get thousands of such small churn contributions in our repositories. Each one of them provides a small hint about how we work. Let's see how we can use that to discover process loss in our organization.

Discover Your Process Loss from Code

By using code churn, we can detect problems in our process. Again, we're not referring to the official processes our companies use to get a sense of predictiveness in software development. Instead, we're referring to the actual process you and your team use. Formal or not, chaotic or ordered—there's always a process.

Now, remember how we discussed process loss back in Chapter 11, *Norms, Groups, and False Serial Killers*, on page 121? Process loss means that we, as a team, will never operate at 100 percent efficiency. But what we can do is minimize the loss. To succeed, we need to know where we stand. So let's see how code churn lets us trace process loss in the history of our codebase.

Measure the Churn Trend

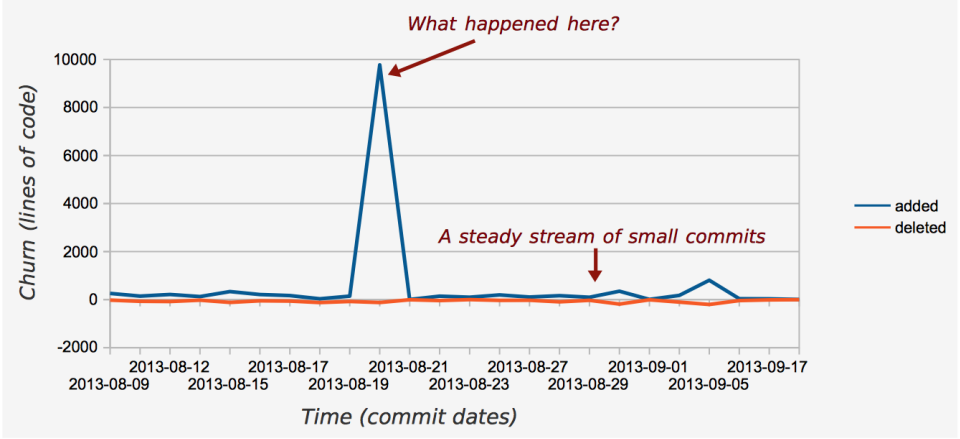
Let's start with a code churn analysis on Code Maat to reverse-engineer its coding process. Move into your Code Maat repository and request an `abs-churn` analysis:

```
prompt> maat -c git -l maat_evo.log -a abs-churn
date,added,deleted
2013-08-09,259,20
2013-08-11,146,70
2013-08-12,213,79
2013-08-13,126,23
2013-08-15,334,118
...
```

The abs-churn analysis calculates the *absolute code churn* in the project. That means we get to see the total number of added and deleted lines of code grouped by each commit date.

When we re-engineer our process from code, the churn numbers themselves aren't that interesting. What matters is the overall pattern. Let's look at it.

A simple way to investigate churn patterns is to visualize the analysis results in a line diagram. Save the analysis results to a file and import the data into a spreadsheet application of your choice. As the following figure shows, the overall trend is a steady stream of commits. That's a good trend because it means we're building the software in small increments. Small increments make our progress more predictive. It's a coding process that makes it easier to track down errors, since we can roll back the specific and isolated changes until we reach a stable state.



But all is not well. There's a suspiciously high peak in the middle of the churn diagram. What happened there? According to the results from our abs-churn analysis, that spike occurred on the 2013-08-20. At that day, the Code Maat codebase grew by a factor of 60 compared to the normal daily churn. If that

growth occurred due to new application logic, we may have a problem, because high code churn predicts defects.

Our first investigative step is to look at the version-control log for the date of interest. In this case, we find that the spike is due to the addition of static test data. As you can see in the following image, a complete Subversion log was committed to the repository.

```
[67a4379] Adam Tornhill 2013-08-20 End-to-end test with live data from statsvn
1       1       src/code_maat/app/app.clj
17      6       test/code_maat/end_to_end/scenario_tests.clj
9629    0       test/code_maat/end_to_end/statsvn.log
```

Here's the churn spike

The spike is due to the addition of static test data

So in this case, our spike was a false positive (although test data may be fragile, too, as we discussed in *Encapsulate Test Data*, on page 97). But you need to be aware of other churn patterns. Let's look at them.

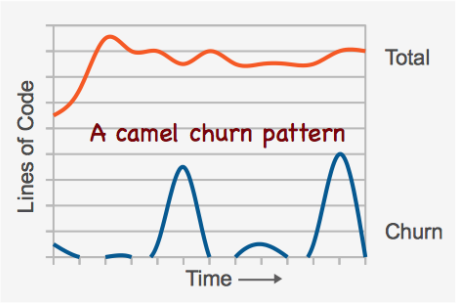
Know the Common Churn Patterns

When we investigate churn trends, we'll typically find one of the patterns illustrated in the figure below. As we discuss these patterns, imagine a fictional deadline approaching. Now, let's look at each pattern.

The first case shows a decreasing churn trend. That means we're able to stabilize more and more of our code bases as time goes on. This is the pattern we want to see.

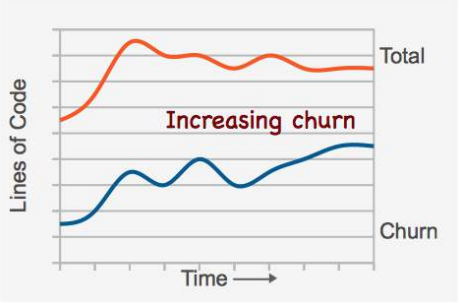
The second pattern is harder to interpret. The first time I saw it, it didn't seem to make sense. Take a look at the following figure. What happens is that you have a period of virtually no activity, and then you get a sudden churn spike.

Then there's no activity again before another large spike occurs. What situational forces can bring forth a trend like this?



Remember the project I told you about at the start of this chapter? The project where we spent hours on complicated merges? That project exhibited this pattern. Once we looked into it, we found that there were exactly two weeks between each spike. Of course, the team used an iterative development process. And, you guessed it, each iteration was two weeks. At the start of each iteration, the developers got a feature assigned to them. The developers then branched out and coded along. As the deadline approached—in this case, the end of the iteration marked by a demo—each developer hurried to get his or her feature merged into the main branch.

The takeaway is that deadlines bring out the worst in us. No one wanted to miss the demo. As a consequence, code was often rushed to completion so that it could be merged. That in itself is a problem manifested in this pattern. But there's more to it. Even when each feature is well-tested in isolation on its respective branch, we don't know how the different features are going to work together. That puts us at risk for unexpected feature interactions, which are some of the trickiest bugs to track down.



Finally, our last churn pattern shows a scary place to be. As you see in the following figure, that project approaches a deadline but keeps changing progressively more code. Since there's a positive correlation between code churn and defects, this patterns means we put the quality of our code at risk.

This churn pattern means that the project won't hit the deadline. There's a lot more work before we have anything like a stable codebase. Running regular analyses of hotspots and temporal coupling lets you uncover more about these problems. Another useful strategy is to analyze what kind of growth you have by applying the tools from Chapter 6, *Calculate Complexity Trends from Your Code's Shape*, on page 55. If the

new code is more complex than the previous code, the project is probably patching conditional statements into a design that cannot sustain them.

You Need Other Tools for SVN and Mercurial

Code Maat only supports code churn measures for Git. The reason is that Git makes the raw data available directly in the version-control log on which Code Maat operates. That doesn't mean you're out of luck, though.

If you use Subversion, you can still calculate churn metrics. You need to write a script that iterates through all revisions in your repository and performs an `svn diff` for each revision. The output is fairly straightforward to parse in order to collect the churn values. I'd also recommend that you check out StatSVN,[1] which is a tool that calculates a churn trend for you.

Mercurial users can apply the strategy to extract raw churn values. In addition, Mercurial comes with a churn extension that provides useful statistics.[2]

Investigate the Disposal Sites of Killers and Code

As we introduced hotspots in Chapter 2, *Code as a Crime Scene*, on page 13, we based our hotspots on a core idea from geographical profiling: the spatial movement of criminals helps us identify and catch them. Similarly, we've been able to identify patterns in our spatial movement in code. And these patterns let us identify maintenance problems and react to them.

Over the years, forensic psychologists have looked at other behavioral patterns as well. One recent study investigated the location of disposal sites used by serial killers. It sure is a macabre research subject, but the information gained is valuable. Let's look into it.

The deeds of a serial killer are bizarre. There's not much to understand there. But although the deeds are irrational, there is a certain logic to the places where serial killers choose to dispose of their victims. One driving force is minimizing the risk of detection. That means the disposal sites are carefully chosen. Often, the geographical distribution of these locations overlaps with the offender's other noncriminal activities. (See *Principles of Geographical Offender Profiling [CY08a]*.) As a consequence, the location of disposal sites contains additional information that points to the offender.

1. http://www.statsvn.org/
2. http://mercurial.selenic.com/wiki/ChurnExtension

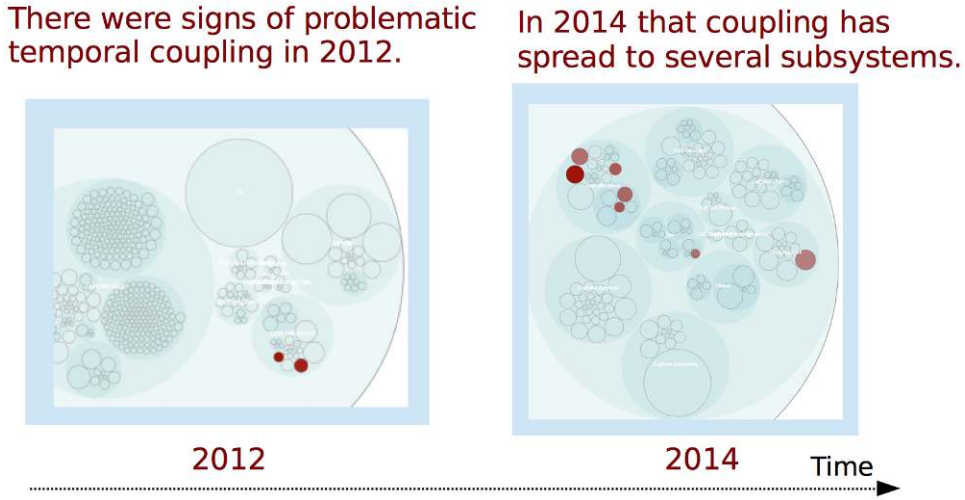
Our programing activities are nowhere near as gruesome, but our codebases do have disposal sites. Disposal sites of code that shouldn't be there are also hard to find. Just as criminal investigators improve their models by looking for additional data, so should we. Let's see how code churn provides that information.

Link Code Churn to Temporal Coupling

Our early design decisions frequently lead to problems as our code evolves. Because programming is a learning activity, it's ironic that we have to make so many fundamental design choices early, at the point where we know the least about the system. That's why we need to revisit and improve those choices. We need to reflect our increased understanding in the system we're building.

The analyses we've learned aim to let us pick up the signs when things start to evolve in the wrong direction. One typical sign is when our software exhibits unexpected modification patterns. In Part II, you learned to catch that problem with temporal coupling analyses. Let's return to one of those case studies and supplement it with code churn data.

We'll reuse the version-control log from Craft.Net that we investigated in *Catch Architectural Decay*, on page 83. In that chapter, we found that the central MinecraftServer module kept accumulating temporal dependencies. We interpreted this trend as a sign of structural decay.



Let's revisit the results from that temporal coupling analysis. You can reuse the version-control log we generated back then. (If you don't have one, follow

the steps in *Catch Architectural Decay*, on page 83.) As you can see in the following figure, the dependencies go across multiple packages:

Coupled to a test program

Coupled to modules in the same package

Coupled to other packages!

entity	degree	average-revs
source/Test/TestServer/Program.cs	39	21
source/Craft.Net.Server/RemoteClient.cs	68	22
source/Craft.Net.Server/Handlers/PacketHandlers.cs	48	17
source/Craft.Net.Server/Craft.Net.Server.csproj	47	19
source/Craft.Net.Server/Handlers/LoginHandlers.cs	42	17
source/Craft.Net.Server/EntityManager.cs	41	17
source/Craft.Net.Anvil/Level.cs	36	17
Craft.Net/Packets.cs	37	14

Figure 3—Modules that have temporal coupling to MinecraftServer

The structural decay in the preceding figure is a reason for concern. We have a cluster of 7 modules with strong temporal dependencies on the MinecraftServer. Trying to break all of these dependencies at once would be a high-risk operation. Instead, we'd like to prioritize the problems. Are some dependencies worse than others? A code churn analysis cannot tell for sure, but it gives us enough hints. Let's try it out.

Link Code Churn to Temporal Coupling

In our first churn analysis, we calculated a trend for the complete codebase. Now we want to focus on individual modules instead and see how the churn is distributed across the system. We do that by an entity-churn analysis. Here's how it looks in the Craft.Net repository:

```
prompt> maat -c git -l craft_evo_140808.log -a entity-churn
entity,added,deleted
...
Craft.Net.Server/MinecraftServer.cs,1315,786
Craft.Net.Server/EntityManager.cs,775,562
Craft.Net.Client/Session.cs,678,499
Craft.Net/Packets.cs,676,3245
...
```

The results show the amount of churned code in each module. For example, you see that we added 1,315 lines of code to the MinecraftServer.cs, but we also deleted 786 lines. Let's combine this information with our temporal coupling results:

Module	Coupling (%)	Added Lines	Deleted Lines
Test/TestServer/Program.cs	39	88	28
Server/RemoteClient.cs	68	313	45
Server/Handlers/PacketHandlers.cs	48	224	136
Client/Handlers/LoginHandlers.cs	42	179	99
Server/EntityManager.cs	41	65	569
Anvil/Level.cs	36	411	11
Packets.cs	37	676	3245

Table 1—Code churn for temporal coupling with the MinecraftServer

The churn metrics give us a more refined picture of the structural problems. Let's interpret our findings.

Interpret Temporal Coupling with Churn

In the preceding table, we can see that the Level.cs module has increased significantly in size. As part of this growth, it got coupled to the MinecraftServer. That's the kind of dependency I'd recommend you break soon.

Our churn dimensions also tell us that TestServer/Program.cs and Handlers/LoginHandlers.cs only contain small modifications. That means they get low priority until our more serious problems have been addressed.

Finally, the EntityManager.cs presents an interesting case. Given what you learned in Chapter 5, *Judge Hotspots with the Power of Names*, on page 47, the name of the module makes an alarm go off. But our metrics show that the module shrank by 500 lines during our analysis period. Since code is like body fat after the holiday season—it's good to get rid of some—this decrease is a promising sign. You see, code churn can be used to track improvements, too.

As you see, adding churn metrics to your other analyses lets you prioritize the improvements. Code churn also helps to track your progress. Used this way, code churn becomes a tool to focus refactoring efforts where they are likely to pay off quickly.

Predict Defects

A high degree of code churn isn't a problem in and of itself. It's more of a symptom, because code changes for a reason. Perhaps we have a feature area that's poorly understood. Or maybe we just have a module with a low-quality implementation.

Given these reasons, it's hardly surprising that code churn is a good predictor of defects. Let's see how we can use that in our hotspot analyses.

Analyze Churn on an Architectural Level

In Chapter 10, *Use Beauty as a Guiding Principle*, on page 105, we used temporal coupling to identify expensive change patterns in different architectures. We used the analysis results to detect modification patterns that violated architectural principles. Code churn measures supplement such analyses as well. Let's see how.

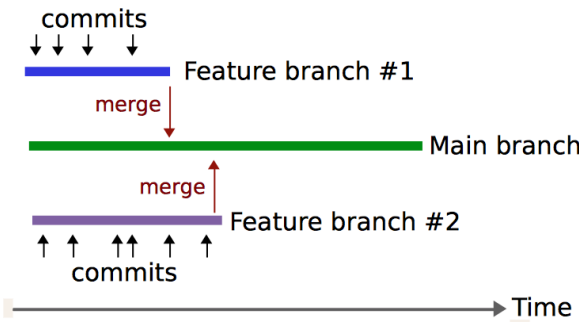
In the architectural analyses, we specify a transformation file. This file defines our architecturally significant components. To run a churn analysis on that level, we just specify the same transformation when we request an entity-churn analysis. When combined with temporal coupling, code churn provides additional insights on how serious the identified dependencies are.

Detect Hotspots by Churn

In this book, we used the number of revisions of each module to detect hotspots. It's a simple metric that works surprisingly well. But it sure has its limitations. (We discussed them back in *Limitations of the Hotspot Criteria*, on page 30.)

Code churn gives you an alternative metric that avoids some of these biases. Here are the typical cases where you should consider code churn:

- *Differences in individual commit styles*: Some developers keep their commits small and cohesive; others stick to big-bang commits.
- *Long-lived feature branches*: If we develop code on branches that live for weeks without being merged, as you see in the following figure, we may lose important history with regard to the original change frequencies on the branch.



While both scenarios indicate symptoms of deeper problems, sometimes you'll find yourself in one of them. In that case, code churn provides a more accurate metric than raw change frequencies.

To use code churn in a hotspot analysis, you combine the results from an `entity-churn` analysis with a complexity metric—for example, lines of code. The overlap between these two dimensions lets you identify the hotspots.

Consider Relative Code Churn

The code churn measures we've used so far are based on absolute churn values. That means code churn erases the differences between commit styles; it no longer matters if someone puts a day's work into a single commit or if you commit often. All that matters is the amount of code that was affected.

However, it's worthwhile to investigate an alternative measure. In *Use of relative code churn measures to predict system defect density [NB05]*, a research team found that code churn was highly predictive of bugs. The twist is that the researchers used a different measure than we do. They measured *relative code churn*.

Relative code churn means that the absolute churn values are adjusted by the size of each file. And according to that research paper, the relative churn values outperform measures of absolute churn. So, have I wasted your time with almost a whole chapter devoted to absolute churn? I certainly hope not. Let's see why.

First of all, a subsequent research paper found no difference between the effectiveness of absolute and relative churn measures. In fact, absolute values proved to be slightly better at predicting defects. (See *Does Measuring Code Change Improve Fault Prediction? [BOW11]*.) Further, relative churn values are more expensive to calculate. You need to iterate over past revisions of each file and calculate the total amount of code. Compare that to just parsing a version-control log, as we do to get absolute churn values.

The conclusion is that we just cannot tell for sure whether one measure is better than the other. It may well turn out that different development styles and organizations lend themselves better to different measures. In the meantime, I recommend that you start with absolute churn values. Simplicity tends to win in the long run.

Know the Limitations of Code Churn

Like all metrics, code churn has its limitations, too. You saw one such case in *Measure the Churn Trend*, on page 164, where a commit of static test data biased the results. Thus, you should be aware of the following pitfalls:

- *Generated code*: This problem is quite easy to solve by filtering out generated code from the analysis results.
- *Refactoring*: Refactorings are done in small, predictable increments. As a side effect, code that undergoes refactorings may be flagged as high churn even though we're making it better.
- *Superficial changes*: Code churn is sensitive to superficial changes, such as renaming the instance variables in a class or rearranging the functions in a module.

In this chapter, we've used code churn to complement other analyses. We used the combined results to support and guide us. In my experience, that's where churn measures are the most valuable. This strategy also lets you minimize the impact of code churn's limitations.

Time to Move On

In this chapter, you learned about the concept of code churn. We used code churn to re-engineer your coding process from version-control data. That technique lets you investigate potential process loss from how you work as a team.

We then transitioned to calculate code churn for individual modules. You learned to use this information to supplement temporal coupling analyses. You also saw how we can measure churn on an architectural level. We then looked into code churn as an alternative to change frequencies in hotspot analyses. You also learned about the differences between absolute and relative churn.

This chapter completes our tour of software evolution and forensic code investigations. You've come a long way since you identified your first hotspot. But before I leave you on your own, we'll take a step back and reflect on how all these techniques fit together. We'll also share a look into the crystal ball to see what the future may look like. So, let's turn the page and find out what the next steps are!

CHAPTER 15

Toward the Future

You're almost through with the book now, and I hope you've enjoyed it! In this chapter, we'll consider the various analytic tools together to see how they complement each other.

We'll also talk about where you go from here by discussing other types of analyses. You'll see how you can analyze developer networks, craft analyses to detect violations of design principles, and apply the techniques you already know to new areas. To do that, you may need to build your own custom tools. You'll get some tips here, including a brief introduction to the *Moose* platform, which can assist you with the task.

Finally, you'll learn about the limitations in today's technologies. You'll learn how the next generation of tools will have to go beyond version-control systems and track what's happening in between commits. All right, I can't wait, so let's jump right into our final chapter!

Let Your Questions Guide Your Analysis

The techniques in this book came about because we needed to understand large software systems. Because, let's face it, software development is hard—we programmers need all the help we can get. Our collection of analysis and heuristics provides such support. We just need to apply it wisely. Let's discuss how.

Start Simple and Then Elaborate as Needed

The hotspot analysis from Part I is as close as we get to a silver bullet. Sure, you've learned about the limitations of hotspots; you've seen false positives and biased data. Yet a hotspot analysis often manages to provide you with a high-level view of the codebase's condition. A hotspot analysis is an ideal first step.

In addition to hotspot analysis, I always check temporal coupling. Start with an analysis of individual modules, as we did back in Chapter 8, *Detect Architectural Decay*, on page 77. Look for surprising modification patterns and patterns that cross architectural boundaries.

If you know the codebase well, I also recommend that you specify its architecturally significant boundaries in a transformation file and perform an analysis on that level, as we did in Chapter 10, *Use Beauty as a Guiding Principle*, on page 105.

When you need more supporting data, either to understand the problems or to prioritize improvements, look to supplement your results with the code churn measures we learned about in Chapter 14, *Dive Deeper with Code Churn*, on page 163.

Finally, you need to consider the social environment where your system evolves. Let's recap that part.

Support Collaboration

We started Part III with an overview of how we work in groups. You learned about social biases and saw how they can turn group decisions into disasters. These biases are hard to avoid, and you should keep in mind that we need to challenge them, as we saw in *Challenge with Questions and Data*, on page 125.

In small organizations, we all know each other and how we work. But as soon as an organization grows, even for a group of seven to ten people, things change for the worse, and you need communication aids. The knowledge map that we discussed in Chapter 13, *Build a Knowledge Map of Your System*, on page 147, is a powerful concept to guide you in such settings.

If you work with multiple teams, I recommend that you keep track of parallel work in your codebase. As you learned in Chapter 12, *Discover Organizational Metrics in Your Codebase*, on page 133, parallel work leads to lower-quality code and more defects. When you identify modules that suffer from parallel work, you investigate them further with fractal figures, as we did in *Visualize Developer Effort with Fractal Figures*, on page 150.

Changing the way you work will never be easy. The techniques in this book can only help you make more informed decisions that let you move closer to your team's potential productivity.

There's much more to be said about the social influences on software design. For example, we haven't talked much about how our physical workplace affects our ability to code.

Joe asks:

Great, So How Does the Physical Workplace Affect Our Ability to Code?

Our office space is an important determiner of job performance. As *Peopleware: Productive Projects and Teams [DL99]* reports, individuals in quiet working conditions are one-third more likely to deliver zero-defect work than their peers in noisy environments. And it gets worse with increased levels of noise.

Studies like this should be an alarming message to any company that depends upon the creativity and problem-solving skills of its employees. In reality, our office environment is often neglected. Many programmers, myself included, fall back on earphones and music to shield us from the noise. It's important to understand the tradeoffs here: when we choose a soundtrack to our code, the effect varies with the task.

Music is an excellent choice when you need a distraction to help you get through a repetitive, routine task. It may get you to perform slightly better and may make the task more enjoyable in the process. On the other hand, music will hurt your performance when working on novel and cognitively demanding tasks, which include programming. However, a noisy work environment is even worse. If you have to code under noisy conditions, music is a decent alternative. Just remember to select music with the following qualities:

1. Avoid music that affects you emotionally. Choose something that you neither strongly like nor strongly dislike.
2. Avoid music with lyrics, because words will compete with the code for your attention.
3. Pick *white noise* if you prefer it over music. White noise works well as a noise-cancellation technique, but just like music, it cannot compete with quiet working conditions.

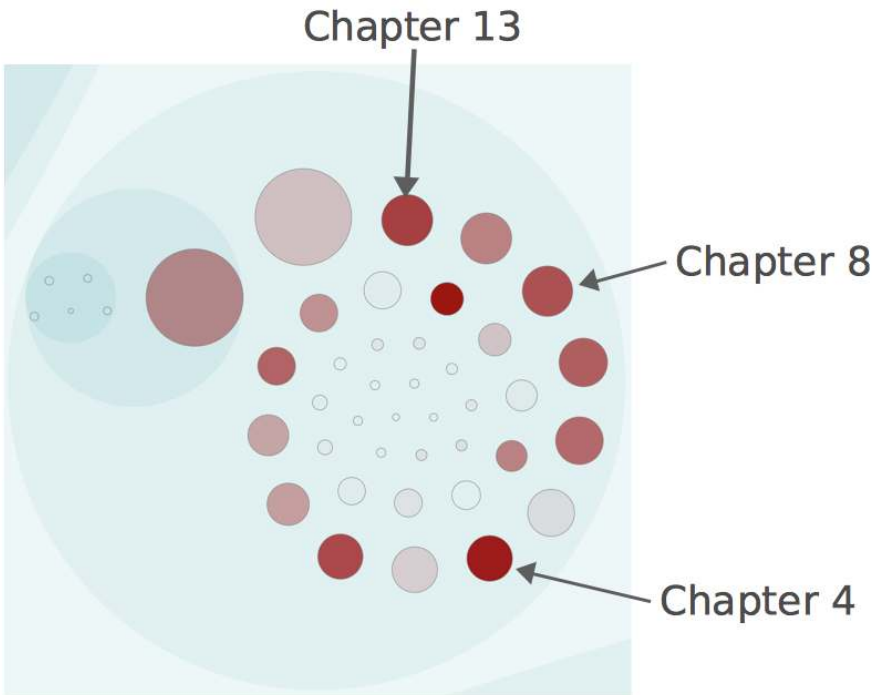
Take Other Approaches

The techniques in this book are a starting point. There's much more information in our code repositories. So before we leave, let's look at some other approaches. What follows are strategies that might give you even more information. These ideas may also serve as an inspiration once you choose to explore our topic in more depth.

Investigate More Than Source Code

If you have other artifacts stored under version control, you can include them in the analyses as well. Some examples include documents, requirement specifications, or manuals for your product. Perhaps you'll even look for temporal dependencies between your requirements and code.

Have a look at the following figure as an example of a non-code analysis. This picture shows the hotspots in the book you're reading right now. (If you're looking to contribute to the errata, Chapter 13 seems like a chance to score big.)



Find Classes That Violate Design Principles

Our version-control systems record changes on a much more granular level than the file. This information is available to you through a simple diff between two revisions. The details that diff gives you let you reason about how the interior of a class evolves.

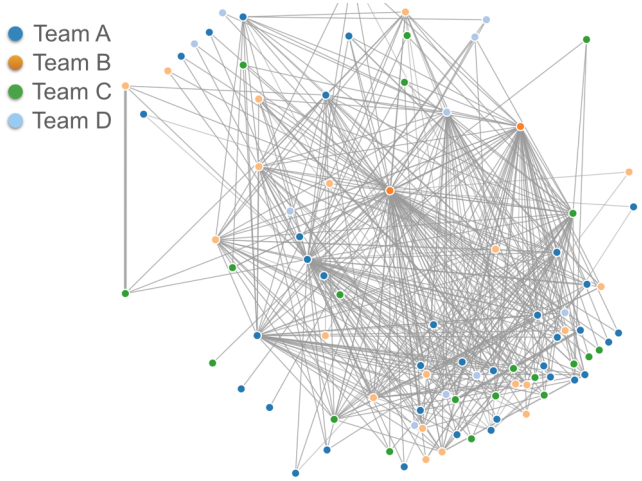
One of the more interesting approaches in this area is Michael Feathers's[1] use of version-control data to ferret out violations of the *Single Responsibility Principle*. His technique uses the *added, changed, and deleted* lines of code to identify clusters of methods within a class. For example, you might find that some methods tend to change together within the same day. When you spot a trend, it might mean you've detected a responsibility that you can express in a new class.

Michael's technique is basically a temporal coupling analysis between methods. The analysis is harder to implement because our tools need to be language-aware. The payoff is a tool that provides us with refactoring support based on what our code actually needs. (See Appendix 1, *Refactoring Hotspots*, on page 183, for a heuristic based on the same idea.)

Analyze Your Developer Networks

Social factors play an important role in how our code evolves. We've talked about communication and knowledge distribution. Let's take that a step further by analyzing developer networks.

The following figure shows the relationship between different programmers based on their interactions in code. All programmers are represented by nodes colored with their team affiliation. Each time we touch the same piece of code as another developer, we get a link between us. The more often we work with the same code, the stronger the link. This information allows us to detect social dependencies across team boundaries.



1. https://michaelfeathers.silvrback.com/using-repository-analysis-to-find-single-responsibility-violations

The network information is mined by Code Maat's communication analysis. An interesting approach is to analyze the data as a graph to extract complex relationships we cannot spot in the visualization alone. For example, a graph lets us find all programmers that we depend on together with all the programmers they depend on themselves.

Remember *Conway's law*—our designs work best when we align them with our organization. A developer network lets you evaluate your design from that perspective.

Craft Your Own Tools

So far we've discussed general analysis tools that we can apply to most codebases. Once you've gained experience and are comfortable with them, you'll decide you need to run more analyses. You need to craft your own tools. The advantage of building custom tools is that you can tailor the analysis to specific content. Let's look into that a bit.

The version-control analyses we've performed previously aren't difficult from a mathematical point of view. The trickiest part—which wasn't that difficult either—was to parse the input data into a format on which you can perform calculations. You can simplify a lot here. Version-control systems like Git support a tremendous number of options to tailor the log output. Use those options to derive a log format containing the minimum of what's needed to answer the analysis questions you have.

Another tool-building approach is to leverage existing expertise. Moose[2] is an open-source platform for building custom analyses quickly. (You've already seen a piece of Moose work in Code City that we looked at in Chapter 2, *Code as a Crime Scene*, on page 13.) Moose helps with tasks such as parsing, visualizations, and more. I recommend you check it out.

Customize Your Analyses for Pair Programming

Most of the analyses we've done ignore the author field in the commit info. However, in case you want to build a knowledge map or analyze parallel work between teams, you need that information. And it may be a little bit harder to obtain if you pair program. Let's see what you can do.

When you pair program, you can identify the involved programmers through the commit message. Some pairs already put their initials at the start of the commit message. If you make that a practice, you have all the data you need.

2. http://www.moosetechnology.org/

You just need to tailor your tools to pick that information instead of the author field in the log.

Let's Look into the Future

Remember the old saying that what happens in a commit stays in a commit? Well, probably not, since I just made it up. But it's nonetheless true, and that's a problem.

Today's tooling limits us to a commit as the smallest cohesive unit. If we knew what happened within a commit, we could take our analyses and predictions to a new level. Consider a temporal coupling analysis as an example. The analysis lets us identify modules that change together. But we cannot tell anything about the direction. Perhaps a change to module `A` is always followed by predictable modifications to `B` and `C`, never the other way around. That would be valuable information to have.

The next generation of tools has to go beyond version-control systems. We need tools that integrate with the rest of our development environment, and tools that record our every interaction with the code. Once we get there, we'll be able to support the most important activities in programming: understanding and reading code. Let's look at how we do that.

Support Code Reading

Most analyses focus on the design aspect of software. But reading code is a harder problem to solve. Let's take inspiration from other areas.

Think about how online sites tend to work. You check out a product and immediately get presented with similar products. What if we could do the same for code? You open a file and get presented with a "programmers who read this code also looked at the `UserStatistics` class and eventually ended up modifying the `ApplicationManager` module. Twice." Such reading recommendations are a natural next step to take.

Integrate Dynamic Information

Another promising area of research is to integrate dynamic analysis results into our development environments. We could use that information to present warnings for particular pieces of our codebase. Let's look at an example.

You have learned that a high degree of parallel development leads to lower quality. What if we hooked the results of such an analysis into our code editors? When we start to modify a piece of code, we would be presented with a

warning like “watch out—this code has been modified by three different programmers over the past day.”

To really work, you’d also need a time aspect. If you had a problem with parallel development in the past, you reacted, and then you fixed the problem, the warning should disappear automatically over time.

We already have all the building blocks we need. The next step is to integrate them with the rest of our workflow. Perhaps some of these new tools will be written by you?

Write to Evolve

I enjoyed writing this book, and I hope you enjoyed our journey through the fascinating field of evolving code. Ultimately, it’s all about writing better software that’s able to evolve as a reaction to the pressure of new features, novel usages, and changed circumstances.

Writing code of that quality will never be easy; software development is one of the hardest things we humans can put our brains to. We need all the support we can get, and I hope this modest collection of forensic techniques has inspired you to dive deeper into the subject. The scene is now yours. May the code be with you.

APPENDIX 1

Refactoring Hotspots

Refactor Guided by Names

Throughout this book, we've focused on detecting problems as early as possible. You learned that a hotspot analysis is an ideal first step toward understanding the overlap between complexity and programmer effort in large systems. In this appendix, you'll get some tips on how to tackle the hotspots you detect.

Back in Chapter 5, *Judge Hotspots with the Power of Names*, on page 47, you identified problematic hotspots like SessionImpl.java and SessionFactoryImpl.java in the Hibernate codebase. Since those modules are central to the system, you want to refactor them.

Hotspots are complicated by nature, so approach them with care. The safest way is to make your improvements in small increments so that you can experiment and roll back design choices that don't work.

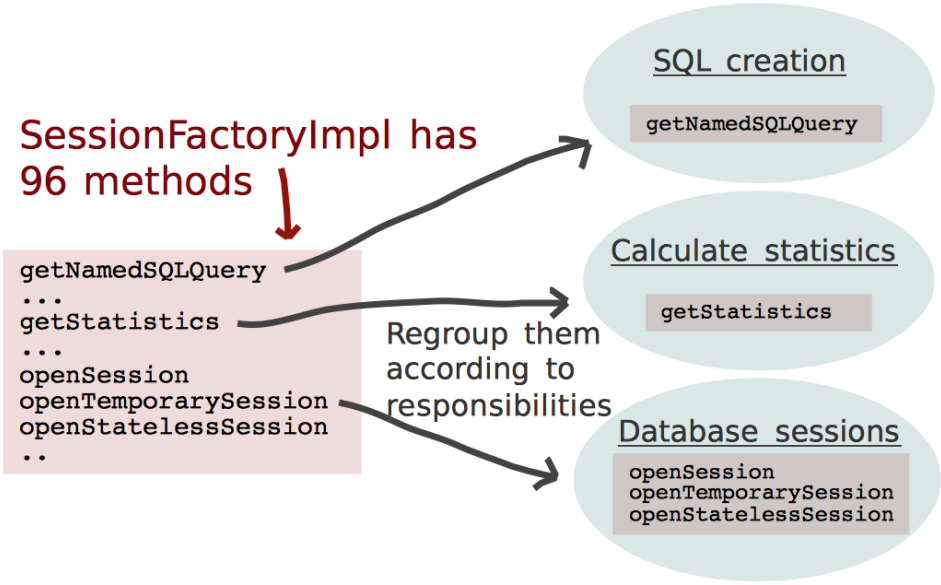
Even as you work iteratively, you want a general idea of where you're heading. Large-scale refactorings are challenging and require more discipline than local changes. It's way too easy to code yourself into a corner. Let's see what we can do to stay on course.

Group Functions by Tasks

As you identify a hotspot, look at the names of its methods and functions. They hold the key to the future.

If you use an IDE, you've probably noticed that it usually sort names alphabetically. It's an unfortunate convention—there's no order that's less relevant (even a random order would be preferable, since it at least doesn't pretend to matter) for our purposes.

What you want to do is group your functions and methods by task. When you do, as the figure shows, hidden responsibilities emerge.



The groups are ideal for identifying design elements. When you are refactoring, you make those responsibilities explicit and wind up with a design with higher cohesion and better modularity. You can make more radical improvements when needed. For readability, cohesion is king. (The other classic design aspect, coupling, isn't the main problem. Loosely coupled software may actually be harder to understand. It's always a tradeoff.)

Let Names Emerge from Wishful Thinking

Choosing good names is hard. As Martin Fowler points out, "There are two hard things in computer science: cache invalidation, naming things, and off-by-one errors."[1]

The best strategy is to let the correct names emerge. The tool you need is *wishful thinking*; defer the decision about how to represent your data and simply imagine you have all the functions to solve your problem in the simplest possible way.

With wishful thinking, you write your ideal code upfront. If you're test-driven, you start to play with a test. Don't worry about it if it doesn't compile or won't

1. http://martinfowler.com/bliki/TwoHardThings.html

run. Experiment with different variants until the code is as expressive as possible. Then you just have to make it compile by implementing your abstractions. This is often straightforward once you've come up with clear roles for your objects and functions.

The reason wishful thinking works is because it helps you get a new perspective on your problem. It's a perspective that fuels your creativity and makes it easier to come up with code that communicates intent.

I use the technique all the time as I get stuck with parts that don't read well. The concept is described along with examples in *Structure and Interpretation of Computer Programs [AS96]*—it's a brilliant read.

Kill the Distractions

A short note on development environments. If you're using an IDE, I recommend you turn off all syntax highlighting, background compilation, and other helpful features during your wishful-thinking session.

Because you are pretending to have code that isn't there yet, the IDE will get in your way. Few things are as disturbing as having your wishful code marked up with thick red syntax errors. A view like the following screenshot is a real productivity killer due to the distracting error markers that draw your attention away from what you're trying to achieve.

```
public MemoryUsage CollectMemoryStatistics(Machine machine, Process process)
{
    return RemoteCallTo(machine).BindTo(process).QueryFor(
}
```

Get Your Names Right

The main takeaway in this appendix is that naming is the most important thing in software design, which includes refactoring. Spending some extra time to get your names right pays off. Wishful thinking helps get them right.

The power of naming concepts goes deep. You saw that information-poor abstract names are magnets for extra responsibilities. When you come across such modules, group their methods and functions by responsibilities so that you know where to refactor.

Bibliography

[AS96] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, 2nd, 1996.

[Bac96] J. Bach. Test Automation Snake Oil. *Windows Tech Journal*. 1996.

[BH06] D. Beyer and A. E. Hassan. Animated Visualization of Software History using Evolution Storyboards. *Reverse Engineering, 2006. WCRE '06. 13th Working Conference on*. 199–210, 2006.

[BHS07] F. Buschmann, K. Henney, and D.C. Schmidt. *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*. John Wiley & Sons, New York, NY, 2007.

[BK03] R.S. Baron and N.L. Kerr. *Group Process, Group Decision, Group Action*. Open University Press, Berkshire, United Kingdom, 2003.

[BNMG11] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. Don't Touch My Code! Examining the Effects of Ownership on Software Quality. *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on foundations of software engineering*. 4-14, 2011.

[BOW04] R.M. Bell, T.J. Ostrand, and E.J. Weyuker. *Where the bugs are. Proceedings of the 2004 ACM SIGSOFT international symposium on software testing and analysis*. ACM Press, New York, NY, USA, 2004.

[BOW11] R.M. Bell, T.J. Ostrand, and E.J. Weyuker. *Does Measuring Code Change Improve Fault Prediction?*. ACM Press, New York, NY, USA, 2011.

[Bro95] Frederick P. Brooks Jr. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, Reading, MA, Anniversary, 1995.

[Con68] M.E. Conway. How do committees invent?. *Datamation*. 4:28–31, 1968.

[CY08] D. Canter and D. Youngs. *Applications of Geographical Offender Profiling*. Ashgate, Farnham, Surrey, UK, 2008.

[CY08a] D. Canter and D. Youngs. *Principles of Geographical Offender Profiling*. Ashgate, Farnham, Surrey, UK, 2008.

[DB13] F. Detienne and F. Bott. *Software Design: Cognitive Aspects*. Springer, New York, NY, USA, 2013.

[DHAQ07] M. Di Penta, M. Harman, G. Antoniol, and F. Qureshi. The Effect of Communication Overhead on Software Maintenance Project Staffing. *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*. 315–324, 2007.

[DL68] J.M. Darley and B. Latané. Bystander intervention in emergencies: diffusion of responsibility. *Journal of Personality and Social Psychology*. 8:377–383, 1968.

[DL99] Tom Demarco and Timothy Lister. *Peopleware: Productive Projects and Teams*. Dorset House, New York, NY, USA, Second edition, 1999.

[DLG05] M. D'Ambros, M. Lanza, and H Gall. Fractal Figures: Visualizing Development Effort for CVS Entities. *Visualizing Software for Understanding and Analysis, 2005. VISSOFT 2005. 3rd IEEE International Workshop on*. 1–6, 2005.

[DLR09] M. D'Ambros, M. Lanza, and R Robbes. On the Relationship Between Change Coupling and Software Defects. *Reverse Engineering, 2009. WCRE '09. 16th Working Conference on*. 135–144, 2009.

[FW08] S. M. Fulero and L. S. Wrightsman. *Forensic Psychology*. Cengage Learning, Boston, MA, 2008.

[GAL14] E. Guzman, D. Azócar, and L. Li. *Sentiment analysis of commit comments in GitHub. MSR 2014 Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM Press, New York, NY, USA, 2014.

[GK03] H. Gall and M. Krajewski. CVS release history data for detecting logical couplings. *Proc. International Workshop on Principles of Software Evolution*. 13–23, 2003.

[GKMS00] T. L. Graves, A. F. Karr, J. S. Marron, and H Siy. Predicting fault incidence using software change history. *Software Engineering, IEEE Transactions on*. 26[7], 2000.

[Gla92] Robert L. Glass. *Facts and Fallacies of Software Engineering*. Addison-Wesley Professional, Boston, MA, 1992.

[Har10] S. Harrison. *The Diary of Jack the Ripper: The Chilling Confessions of James Maybrick*. John Blake, London, UK, 2010.

[HGH08] A. Hindle, M.W. Godfrey, and R.C. Holt. *Reading Beside the Lines: Indentation as a Proxy for Complexity Metric. Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*. IEEE Computer Society Press, Washington, DC, 2008.

[HSSH12] K. Hotta, Y. Sasaki, Y. Sano, Y. Higo, and S. Kusumoto. An Empirical Study on the Impact of Duplicate Code. *Advances in Software Engineering*. Special issue on Software Quality Assurance Methodologies and Techniques, 2012.

[KG85] W. Kintsch and J. G. Greeno. Understanding and solving word arithmetic problems. *Psychological Review*. 92(1):109–129, 1985.

[Leh80] M. M. Lehman. On Understanding Laws, Evolution, and Conservation in the Large-Program Life Cycle. *Journal of Systems and Software*. 1:213–221, 1980.

[LR90] J. H. Langlois and L. A. Roggman. Attractive faces are only average. *Psychological Science*. 1:115–121, 1990.

[MW09] A. Meneely and L. Williams. Secure open source collaboration: an empirical study of Linus' law. *Proceedings of the 16th ACM conference on computer and communications security*. 453–462, 2009.

[NB05] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. *Proceedings of the 27th international conference on software engineering*. 284–292, 2005.

[NMB08] N. Nagappan, B. Murphy, and V. Basili. The Influence of Organizational Structure on Software Quality. *International Conference on Software Engineering, Proceedings*. 521–530, 2008.

[OW10] A. Oram and G. Wilson. *Making Software: What Really Works, and Why We Believe It*. O'Reilly & Associates, Inc., Sebastopol, CA, 2010.

[PM00] J.F. Pane and B.A. Myers. The Influence of the Psychology of Programming on a Language Design. *Proceedings of the 12th Annual Meeting of the Psychology of Programmers Interest Group*. 193–205, 2000.

[SEKH09] C.M. Schweik, R.C. English, M. Kitsing, and S. Haire. Brooks' versus Linus' law: an empirical test of open source projects. *Proceedings of the 2008 international conference on digital government research*. 423–424, 2009.

[SF08] V. Swami and A. Furnham. *The Psychology of Physical Attraction*. Routledge, New York, NY, USA, 2008.

[SFL88] P. Slovic, B. Fischhoff, and S. Lichtenstein. *Decision Making: Descriptive, Normative, and Prescriptive Interactions*. Cambridge University Press, Cambridge, United Kingdom, 1988.

[TT89] B. Tversky and M. Tuchin. A reconciliation of the evidence on eyewitness testimony: Comments on McCloskey and Zaragoza. *Journal of Experimental Psychology: General*. [118]:86–91, 1989.

[VDC94] J.S. Valacich, A.R. Dennis, and T. Connolly. Idea Generation in Computer-Based Groups: A New Ending to an Old Story. *Organizational Behavior and Human Decision Processes*. 57[3]:448–467, 1994.

[WMGS07] K. Weaver, D.T. Miller, S.M. Garcia, and N. Schwarz. Inferring the popularity of an opinion from its familiarity: A repetitive voice can sound like a chorus. *Journal of Personality and Social Psychology*. [92]:821–833, 2007.

[YMNC04] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. Software Engineering*. 9[30], 2004.

Index

D

G

H

M

N

O

S

T

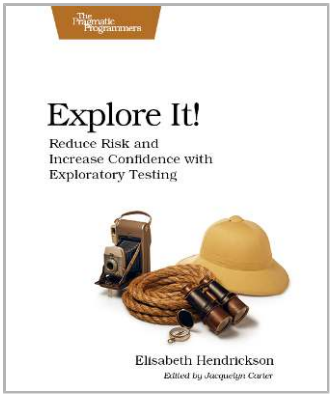
W

Y

Explore Testing and Cucumber

Explore the uncharted waters of exploratory testing and beef up your automated testing with more Cucumber—now for Java, too.

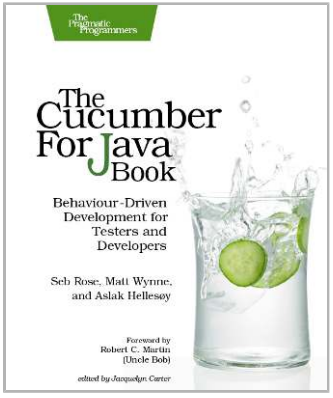
Explore It!



Uncover surprises, risks, and potentially serious bugs with exploratory testing. Rather than designing all tests in advance, explorers design and execute small, rapid experiments, using what they learned from the last little experiment to inform the next. Learn essential skills of a master explorer, including how to analyze software to discover key points of vulnerability, how to design experiments on the fly, how to hone your observation skills, and how to focus your efforts.

Elisabeth Hendrickson
(186 pages) ISBN: 9781937785024. $29
https://pragprog.com/book/ehxta

The Cucumber for Java Book



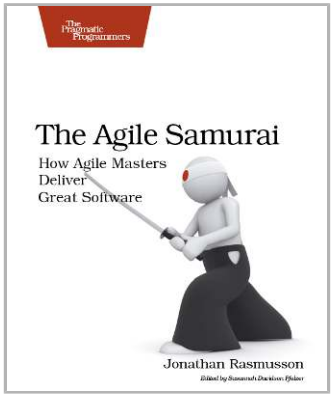
Teams working on the JVM can now say goodbye forever to misunderstood requirements, tedious manual acceptance tests, and out-of-date documentation. Cucumber—the popular, open-source tool that helps teams communicate more effectively with their customers—now has a Java version, and our bestselling *Cucumber Book* has been updated to match. *The Cucumber for Java Book* has the same great advice about how to deliver rock-solid applications collaboratively, but with all code completely rewritten in Java. New chapters cover features unique to the Java version of Cucumber, and reflect insights from the Cucumber team since the original book was published.

Seb Rose, Matt Wynne & Aslak Hellesoy
(338 pages) ISBN: 9781941222294. $36
https://pragprog.com/book/srjcuc

Be Agile

Don't just "do" agile; you want to *be* agile. We'll show you how, for new code and old.

The Agile Samurai



Here are three simple truths about software development:

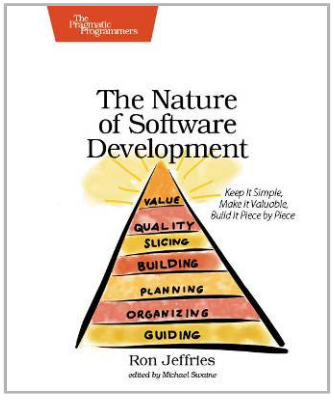
1. You can't gather all the requirements up front.
2. The requirements you do gather will change.
3. There is always more to do than time and money will allow.

Those are the facts of life. But you can deal with those facts (and more) by becoming a fierce software-delivery professional, capable of dispatching the most dire of software projects and the toughest delivery schedules with ease and grace.

This title is also available as an audio book.

Jonathan Rasmusson
(280 pages) ISBN: 9781934356586. $34.95
https://pragprog.com/book/jtrap

The Nature of Software Development



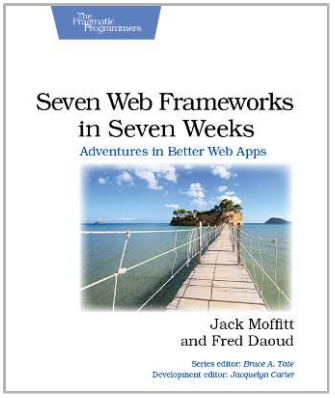
You need to get value from your software project. You need it "free, now, and perfect." We can't get you there, but we can help you get to "cheaper, sooner, and better." This book leads you from the desire for value down to the specific activities that help good Agile projects deliver better software sooner, and at a lower cost. Using simple sketches and a few words, the author invites you to follow his path of learning and understanding from a half century of software development and from his engagement with Agile methods from their very beginning.

Ron Jeffries
(178 pages) ISBN: 9781941222379. $24
https://pragprog.com/book/rjnsd

Seven in Seven

From Web Frameworks to Concurrency Models, see what the rest of the world is doing with this introduction to seven different approaches.

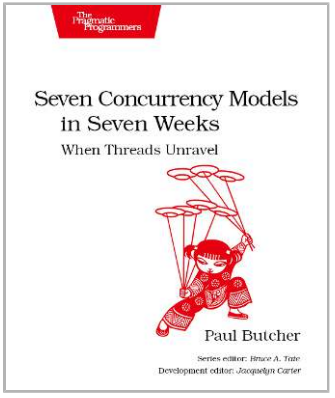
Seven Web Frameworks in Seven Weeks



Whether you need a new tool or just inspiration, *Seven Web Frameworks in Seven Weeks* explores modern options, giving you a taste of each with ideas that will help you create better apps. You'll see frameworks that leverage modern programming languages, employ unique architectures, live client-side instead of server-side, or embrace type systems. You'll see everything from familiar Ruby and JavaScript to the more exotic Erlang, Haskell, and Clojure.

Jack Moffitt, Fred Daoud
(302 pages) ISBN: 9781937785635. $38
https://pragprog.com/book/7web

Seven Concurrency Models in Seven Weeks



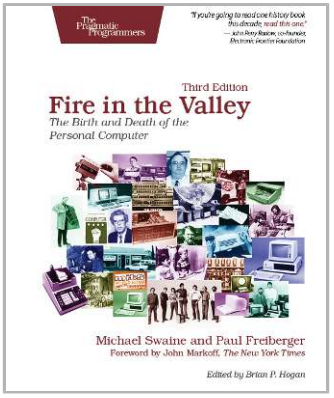
Your software needs to leverage multiple cores, handle thousands of users and terabytes of data, and continue working in the face of both hardware and software failure. Concurrency and parallelism are the keys, and *Seven Concurrency Models in Seven Weeks* equips you for this new world. See how emerging technologies such as actors and functional programming address issues with traditional threads and locks development. Learn how to exploit the parallelism in your computer's GPU and leverage clusters of machines with MapReduce and Stream Processing. And do it all with the confidence that comes from using tools that help you write crystal clear, high-quality code.

Paul Butcher
(296 pages) ISBN: 9781937785659. $38
https://pragprog.com/book/pb7con

Past and Present

To see where we're going, remember how we got here, and learn how to take a healthier approach to programming.

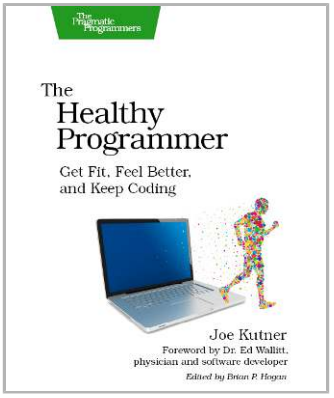
Fire in the Valley



In the 1970s, while their contemporaries were protesting the computer as a tool of dehumanization and oppression, a motley collection of college dropouts, hippies, and electronics fanatics were engaged in something much more subversive. Obsessed with the idea of getting computer power into their own hands, they launched from their garages a hobbyist movement that grew into an industry, and ultimately a social and technological revolution. What they did was invent the personal computer: not just a new device, but a watershed in the relationship between man and machine. This is their story.

Michael Swaine and Paul Freiberger
(424 pages) ISBN: 9781937785765. $34
https://pragprog.com/book/fsfire

The Healthy Programmer



To keep doing what you love, you need to maintain your own systems, not just the ones you write code for. Regular exercise and proper nutrition help you learn, remember, concentrate, and be creative—skills critical to doing your job well. Learn how to change your work habits, master exercises that make working at a computer more comfortable, and develop a plan to keep fit, healthy, and sharp for years to come.

This book is intended only as an informative guide for those wishing to know more about health issues. In no way is this book intended to replace, countermand, or conflict with the advice given to you by your own healthcare provider including Physician, Nurse Practitioner, Physician Assistant, Registered Dietician, and other licensed professionals.

Joe Kutner
(254 pages) ISBN: 9781937785314. $36
https://pragprog.com/book/jkthp

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

This Book's Home Page
https://pragprog.com/book/atcrime
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
https://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
https://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
https://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: *https://pragprog.com/book/atcrime*

Contact Us

Online Orders:	*https://pragprog.com/catalog*
Customer Service:	*support@pragprog.com*
International Rights:	*translations@pragprog.com*
Academic Use:	*academic@pragprog.com*
Write for Us:	*http://write-for-us.pragprog.com*
Or Call:	+1 800-699-7764