Extracted from:

Your Code as a Crime Scene

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

The Pragmatic Bookshelf

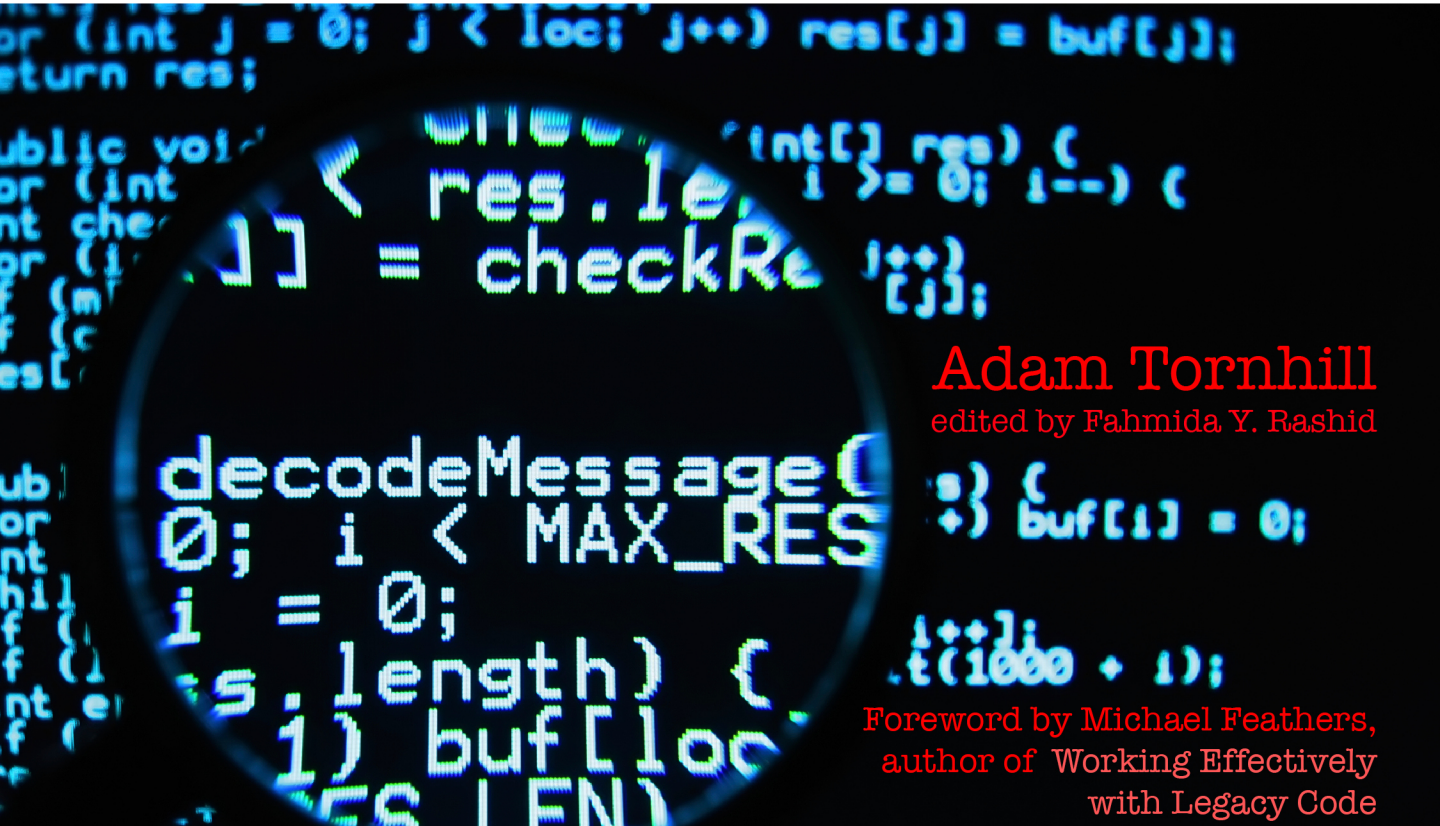
Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Investigator:
Date:
Case #:
Location:

Your Code As a Crime Scene

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



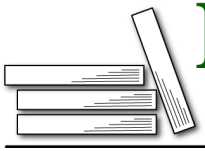
Your Code as a Crime Scene

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

Adam Tornhill

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

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

The team that produced this book includes:

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

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

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

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.

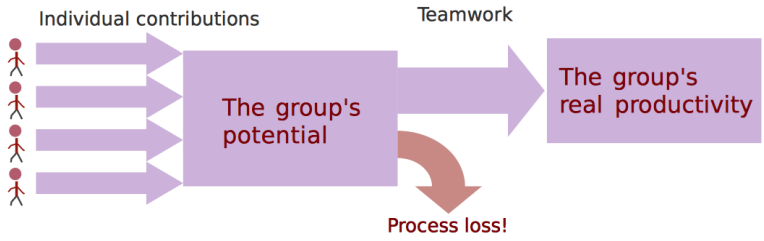
The story about Thomas Quick is a case study in the dangers of social biases in groups. The setting is much different from what we encounter in our daily lives, but the biases aren't. The social forces that led to the Thomas Quick disaster are present in any software project.

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 ?.) 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 ?? 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*.

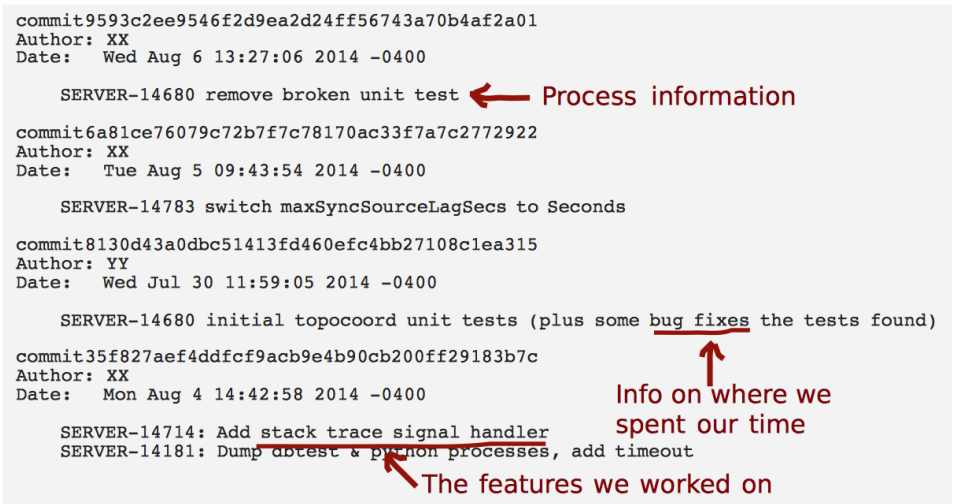
A modus operandi is like a criminal signature. For example, the gentleman bandit you read about in *Meet the Innocent Robber*, on page ?, was characterized by his polite manners and concern for his victims.

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 ?. 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.