Extracted from:

Software Design X-Rays

Fix Technical Debt with Behavioral Code Analysis

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Software Design X-Rays

Fix Technical Debt with Behavioral Code Analysis

Adam Tornhill

edited by Adaobi Obi Tulton

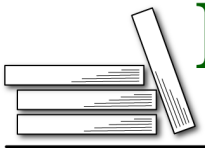
Software Design X-Rays

Fix Technical Debt with Behavioral Code Analysis

Adam Tornhill

The Pragmatic Bookshelf

Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

For sales, volume licensing, and support, please contact *support@pragprog.com*.

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

Printed in the United States of America.
ISBN-13: 978-1-68050-272-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: B1.0—October 25, 2017

The World of Behavioral Code Analysis

Welcome dear reader – I'm happy to have you here! Together we'll dive into the fascinating field of evolving software systems to learn how behavioral code analysis helps us make better decisions. This is important because our average software project is much less efficient than it could be.

The history of large scale software systems is a tale of cost overruns, death marches, and heroic fights with legacy code monsters. One prominent reason is technical debt, which represents code that's more expensive to maintain than it should be. Repaying technical debt is a hard problem due to the scale of modern software projects; With hundreds of developers and a multitude of technologies, no one has a holistic overview. We're about to change that.

Inside, you learn a set of techniques that give you an easily accessible overview of your codebase, together with methods to prioritize improvements based on the expected return on investment. That means you will be comfortable with picking up any large-scale codebase, analyze it and suggest specific refactorings based on how the developers have worked with the code so far.

Good code is as much social design as it is a technical concern. We reflect that by learning to uncover organizational inefficiencies, resolve coordination bottlenecks among teams, and assess the consequences of knowledge loss in your organization.

Why You Should Read This Book

We can never reason efficiently about a complex system based on its code alone. In doing so we miss out on long term trends and social data that are often more important than any property of the code itself. This means that we need to understand how we—as an organization—interact with the code we build.

This book shows you how as you learn to:

- Use data to prioritize technical debt and ensure your suggested improvements pay off.
- Identify communication and team coordination bottlenecks in code.
- Use behavioral code analysis to ensure your architecture supports your organization.
- Supervise the technical sprawl and detect hidden dependencies in a microservice architecture.
- Detect code quality problems before they become maintenance issues.
- Drive refactorings guided by data from how your system evolves.
- Bridge the gap between developers and business oriented people by highlighting the cost of technical debt and visualizing the effects of refactorings.

If all this sounds magical, we can assure you it's not; Rather than magic—which is usually a dead end for software—this book relies on data science and human psychology. Since we're part of an opinionated industry, it's hard to know up front what works and what doesn't. So we make sure to include references to published research so that we know that the techniques are effective before attempting them on our own systems.

We also make sure to discuss the limitations of the techniques, and suggest alternative approaches when applicable. As noted computer scientist Fred Brooks pointed out, there's no silver bullet (*No Silver Bullet—Essence and Accident in Software Engineering [Bro86]*). Instead, you want to view this book as a way of building a set of skills to complement your existing expertise and make decisions guided by data. The reward is a new perspective on software development that will change how you work with legacy systems.

Who Is This Book For?

To get the most out of this book you should be an experienced programmer, technical lead, or software architect. The most important thing is that you have worked on larger software projects and experienced the various pains and problems we try to solve in the book.

You don't have to be a programming expert, but you should be comfortable looking at smaller code samples. Most of our discussions are on a conceptual level and since the analyses are technology-neutral, the book will apply no matter what programming language you work with. This is an important aspect of the techniques you're about to learn, as most of today's systems are polyglot codebases.

You should also have experience with a version-control system. The practical examples assume you use Git, but the techniques themselves can be used with other version-control tools like *Subversion*, *TFS*, and *Mercurial* by performing a temporary migration to Git.[1]

How Should You Read This Book?

The book progresses from smaller systems to large-scale codebases with millions lines of code and thousands of developers. The earlier chapters lay the foundation for the more complex analyses by introducing fundamental concepts like hotspots and dependency analyses based on time and evolution of code. This means you want to read the first three chapters to build a solid toolset for tackling the more advanced material in Part II.

The last two chapters of Part I, Chapter 4, *Pay off Your Technical Debt*, on page ? and Chapter 5, *The Principles of Code Age*, on page ?, travel deeper into real code and are the most technical ones in the book. Feel free to skip them if you are more interested in maintaining a high-level strategic view of your codebase.

We will touch upon the social aspects of code early, but the full treatment is given in the first chapters of Part II. Modern software development is an increasingly collaborative and complex effort, so make sure you read Chapter 6, *Spot Your System's Tipping Point*, on page ? and Chapter 7, *Beyond Conway's Law*, on page ?.

No analysis is better than the data it operates on, so whatever path you chose through the book, make sure to read the section in chapter 10, "An Extra Team Member: Predictive and Proactive Analyses," which explains some special cases that you may come across in your work.

Most chapters also contain exercises that let you practice what you've learned and go deeper into different aspects of the analyses. If you get stuck, you just turn to Appendix 3, *Hints and Solutions to the Exercises*, on page ?.

To Readers of *Your Code as a Crime Scene*

If you've read my previous book, *Your Code as a Crime Scene [Tor15]*, you should be aware that there is an overlap between the two books and *Software Design X-Rays* expands upon the previous work. As a reader of my previous book you will get a head start as some techniques in Part I like hotspots and temporal coupling are familiar to you. However, you still want to skim through

1. https://git-scm.com/book/it/v2/Git-and-Other-Systems-Migrating-to-Git

those early chapters as we extend the techniques to work on the more detailed level of functions and methods. This is particularly important if you work in a codebase with large source code files that are hard to maintain.

Joe asks:

Who Am I?

Joe is a reading companion that shows up every now and then to question the arguments made in the main text. As such, Joe wants to make sure that we leave no stone unturned as we travel the world of behavioral code analysis.

How Do I Get Behavioral Data for My Code?

The techniques in this book build on the behavioral patterns of all the programmers that contribute to your codebase. However, instead of starting to collect such data we want to apply our analyses retrospectively to existing codebases. Fortunately, we already have all the data we need in our version-control system.

Historically, we've used version-control as a complicated backup system that—with good fortune and somewhat empathic peers—allows several programmers to collaborate on code. Now we are going to turn it inside out as we learn to read the story of our systems based on their historical records. The resulting information will give you insights that you cannot get from the code alone.

As you read through the book, you get to explore version-control data from real-world codebases; You learn to find duplicated code in the Linux kernel,[2] detect surprising hidden dependencies in Microsoft's ASP.NET MVC framework,[3] do some mental gymnastics as we suggest a refactoring of Google's TensorFlow codebase,[4] and much more.

These codebases represent some of the best work we—as a software community—are able to produce. The idea is that if we're able to come up with productivity improvements in code like this, chances are you'll be able to do the same in your own work.

All case studies use open source projects hosted on GitHub, which means you don't have to install anything to follow along with the book. The case

2. https://en.wikipedia.org/wiki/Linux_kernel
3. https://www.asp.net/mvc
4. https://www.tensorflow.org/

studies are chosen to reflect common issues that are found in many closed source systems too.

The Case Studies Have Stopped the World

The online analysis results represent the state of the codebases at this time of writing, and the repositories are available on a dedicated GitHub account.[5] This is important since popular open source projects evolve at a rapid pace, which means the case studies would become outdated faster than this week's JavaScript framework.

Most case studies use the analysis tool *CodeScene* to illustrate the examples.[6] CodeScene is developed by Empear, the startup where I work. CodeScene is free to use for open source projects.

We won't spend any time learning CodeScene, but rather use the tool as a portfolio – an interactive gallery. This saves you time as you don't have to focus on the mechanics of the analyses (unless you want to) and guarantees that you see the same results as we discuss in the book. The results are publicly accessible so you don't have to sign-up to CodeScene to follow along.

We also make sure to point out alternative tooling paths when such exist. Often, we go a long way with simple command line tools and we use them when feasible. We also point out other third-party tools that complement the analyses in the cases where such tools provide deeper information. Finally, there's another path to behavioral code analysis through the open source tool *Code Maat* that I developed to illustrate the implementation of the different algorithms. We cover Code Maat in appendix 2, "CodeMaat: An Open-Source Analysis Engine."

Finally, think of tooling as the manifestation of ideas and a way to put them into practice. Consequently, our goal in this book is to understand how the analyses work behind the scenes and how they help solve specific problems.

Online Resources

As mentioned earlier, the repositories for the case studies are available on a dedicated GitHub account. This book has its own web page[7] where you can find the community forum. There you can ask questions, post comments, and submit errata.

5. https://github.com/SoftwareDesignXRays
6. https://codescene.io/
7. https://pragprog.com/book/atevol

With the tooling covered, we're ready to explore the fascinating field of evolving systems. Let's dig in and get a new perspective on our code!