Mutation Testing

Better Code by Making Bugs

Filip van Laenen

Mutation Testing

Better Code by Making Bugs

Filip van Laenen

This book is available at https://leanpub.com/mutationtesting

This version was published on 2025-11-11

Tweet This Book!

Please help Filip van Laenen by spreading the word about this book on Twitter!

The suggested hashtag for this book is #mutestbook.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#mutestbook

Contents

Foreword

Preface

Many years ago, I started experimenting and doing hobby projects in Ruby. But with my Java background, I was used to measure code quality through static code analysis and test coverage reports, so I started looking for Ruby tools that could do the same thing. As I did some research on the Internet, I quickly found out that there weren't many useful tools in Ruby for static code analysis. The reason for this is very simple: Ruby isn't a typed language, and that makes it hard to do the same kind of static code analysis that's possible in languages like Java. But since I wanted to improve my Ruby programming skills, I was willing to try any tool that I could get running on my machine and that would help me get better in Ruby.

I did find a test coverage tool, but it didn't work very well. Together with a simple static code analysis tool that was able to give me some useful feedback, I had something running, together with a mysterious tool called "Heckle". I couldn't figure out what it really was, but somebody was recommending it on his blog. I was able to install it and run it against my Ruby source code, so I decided to give it a try. Somehow, it managed to point me to missing test cases from time to time, much like an automatic code reviewer. That didn't help me get better in Ruby per se, but it was still a great benefit while working on hobby projects at home. I got used to running it as part of my code quality check routine.

But then there was this one evening, when I was working on a piece of code that I thought I had tested completely. I had 100% test coverage, both for the lines and the branches, so what more could I aim for? Nevertheless, Heckle didn't seem to be happy with my unit tests, and kept telling me that something was wrong. In fact, it even gave me an alternative source code that it said would pass all my unit tests, even though I could see that that piece of source code simply wouldn't do the job. Clearly, this had to be a bug in Heckle, and I started to prepare a bug report so someone could fix the problem. I just had to prove first that Heckle was wrong. And what better way could there be to prove that Heckle was wrong, than pasting the source code it was proposing into my source code, and running all my unit tests against it?

Of course, Heckle wasn't wrong. The source code it proposed did pass all the unit tests. That annoyed me even more, so I had to find out what was really going on. As it turned out, there was indeed a test case missing, a test case that I hadn't thought of before. And yes, there was an alternative solution to the source code I had, one that passed all the unit tests I had written so far. Even more, after some refactoring it was also considerably simpler than the original soure code.

That evening, I learned that even if you try to follow all the rules of Test-Driven Development (TDD) carefully and to the detail, you will still make mistakes and write more source code than strictly needed to make all your unit tests pass. And it's hard to find out about this without an automatic tool that will systematically vet your source code and ask for every statement it can find whether it's really necessary to have it there or not. No human code reviewer can do that. And no human code reviewer can be so systematic and merciless as an automatic mutation testing tool.

There have been more evenings where mutation testing has pointed me to test cases I would never have discovered on my own. There have also been many evenings where mutation testing has pointed me to parts in my source code that turned out to be unnecessary, even though the test coverage reports where indicating these parts were covered. Over the years, I have probably deleted thousands lines of source code, and saved myself from a lot of maintenance and even bug fixing, thanks to mutation testing. Not only was I able to discover the unneeded lines of code, I also dared to delete them. If I really couldn't come up with a single unit test that would stop mutation testing from complaining about some unneeded code, then why keep it?

Mutation testing has thaught me that traditional test coverage reports, with line and branch coverage, simply aren't good enough. These reports only tell you which lines and branches your unit tests run through, not whether they really matter. Mutation testing has also thaught me to write unit tests in a different way, and care more about being able to test what the source code really does –or is supposed to do–, rather than trying to write unit tests that touch every line and branch in the source code. That makes a big difference. It makes for better unit tests, better source code, and also less source code. All three are positive effects in their own right, and it's amazing that mutation testing can do all three at once.

These days, I don't use Heckle any more. It stopped working after upgrading to Ruby 1.9, so I switched to Mutant, which works even better. When I'm programming in Java, I use PIT, which is a great tool too. Unfortunately, I'm not able to use mutation testing in every project I work on. But even if I can't use mutation testing, I still program as if mutation testing was lurking somewhere behind the door. I write plenty of unit tests, especially to test boundary cases, and I've become much more critical of putting in extra lines of source code if I'm not sure I really need them.

If you wonder what the little problem was that I was working on that evening, many years ago, have a look at the practical example later in this book. It's partly simplified and partly modified for pedagogical reasons, but the essence is still there. I've used it in my talks about mutation testing, and it seems to me that it's able to tell someting fundamental about mutation testing. I guess that anybody who starts using mutation testing takes software development and software quality seriously, and therefore sooner or later will run into a similar case: a mutation testing epiphany.

How To Read This Book

If you're new to mutation testing, you should start by reading the introduction to this book, the chapter about the basics of mutation testing, and work your way through the practical example. After that, you should be able to pick the right tool, and get started running mutation testing on your projects. As you gain more experience on mutation testing, you will want to browse through the other chapters to get a deeper understanding of mutation testing, how it works, the mutators that are available to you, and some of the history. But I tried to order the chapters such that they make the most sense if you want to read the book from the beginning to the end, as a sort of text book, so you may chose to do that too.

If you already have some experience with mutation testing you can probably skip the introduction and the practical example, and go straight to the more advanced chapters. In particular the chapters on how mutation testing works and how mutation testing tools are built should be of interest to you, as they will explain why the tool you're using behaves the way it behaves, and give you some ideas on how you can get more out of mutation testing as a technique, and the mutation testing tool you're using.

If you're thinking about creating a mutation testing tool on your own, perhaps because there isn't one for your favourite programming language, I would advice you to start experimenting with one of the tools for the other languages. Just pick the best one for the language that you're the most familiar with, and try it out. After that, the chapter explaining how mutation testing tools work should give you an overview over the things you'll have to think about. And if you finally decide to really build a new mutation tool, don't forget to notify me, so I can include it in this book!

The examples in this book are written in pseudocode, just to keep things as much as possible language neutral. That's also why I've chosen to keep it close to mathematical notations, which also reduces the number of keywords needed and helped keep it simple and concise. There's no formal definition for the language, and there isn't even a guarantee that it's completely consistent across all examples. It may even be inconsistent from one example to another just because of what I try to point out in the examples.

Who Should Read This Book

This book has been written for developers, senior developers and technologists, in order to give them an understanding of what mutation testing is about, how it can be used, and how they can get started working with it. However, if you're not so familiar with automatic unit tests, and your project still has low test coverage, you should probably go and fix that first.

If you're a tester or responsible for QA, you can also benefit from this book to understand the impact of mutation testing on your work, and give you some ideas on how the results of a mutation testing report can be helpful to you.

If you're a business analyst or a project leader, the introductory chapters in this book should be able to give you a basic understanding of what mutation testing is. But be warned that the rest of the material in this book is very technical, and therefore probably not so much of interest to you.

Acknowledgements

I want to thank Henry Coles, the author of PIT, for letting me use “Better Code by Making Bugs” as the subtitle for this book.

I also want to thank the folks at Leanpub for their great service. They were even so kind to add Deja Vu Mono to their set of fonts, so I could use all the glyphs I needed in the pseudocode listings.

1. Introduction

Quis custodiet ipsos custodes? Who is guarding the guards?

– Juvenal (Satire VI, lines 347–8)

Unit tests guard the source code, so we can be sure that the system behaves correctly. Indeed, each of them should test a little unit in the system, and make sure that given a certain input to the system, the resulting behavior is as expected. But how do we know that every part of our source code has been tested sufficiently by our unit tests? There are tools that can verify that the unit tests run through every line of the source code, every branch in the source code, and even all paths through the source code. But how do we know that the unit tests also verify every statement in the source code?

Traditional test coverage tools can easily be fooled –or "gamed"– by unit tests that run through all the lines, branches and paths of the source code, but never test anything useful. In fact, many of these test coverage tools will report full coverage of the source code even for unit tests that don't contain a single assertion. But even if these tools are smart enough to include only the unit tests that contain an assertion, there's no guarantee that the assertion verifies anything usefull. It could just as well be verifying that 1 still equals 1.

Even if all the unit tests contain an assertion, and all of them are written in good faith, i.e. in order to really test the system and not just reach a certain threshold, we still need to know whether our unit tests are doing a good job at verifying all the important aspects of our source code. There's a subtle difference between measuring that the unit tests *run through* every statement of the source code, measuring that the unit tests *verify the correctness* of every statement, and measuring that the unit tests verify *the necessity* of every statement.

Every statement in the source code should contribute to make at least one unit test pass. Not being the cause of the failure of a unit test is not a good enough reason for a statement to be present in the source code. If that's all a certain statement does, it could just as well be removed from the source code – and should be too. Mutation testing can, to a much larger extent than traditional code coverage tools, asses whether all statements in the source code are correct according to the unit tests, and whether they all are needed for the unit tests to pass.

But if mutation testing is so great, why hasn't it been in use for years? The truth is, until a few years ago, mutation testing tools didn't work very well. Many of them were proof of concepts, or academic tools to do research, and never built with performance or efficiency in mind. Today it's possible to use mutation testing in real-life projects, and in fact, is used by many projects. You still have to think through the set-up of the mutation testing tool though, and how you're going to integrate it with the rest of your build environment. If you don't, mutation testing can easily degrade into a job that requires a lot of computing power only to produce useless reports nobody ever reads.

Does that make mutation testing very different from the many other tools that are available to you as a software developer these days? Maybe not on the surface, but the reader should be warned that it really still is early days for the practical use of mutation testing in real-life projects. There are many aspects of mutation testing where there's still a lot of room for improvement, and best practices to use mutation testing are still evolving. Integration with common development environments and other tools in the software development eco-system is still lacking for many of the mutation testing tools. The community is still trying to find out what's the best way to report which mutants stayed alive during a mutation testing round. Unit test selection strategies are becoming better and better, and that means that the performance of the tools is greatly improving. But the biggest drawback of mutation testing is still that full rounds require a lot of computing power. Then again, that just means that you have to be a little bit smarter when you want to apply mutation testing to your project compared to using any of the other tools and techniques that are considered to be common goods in a software developement environment. But isn't that part of what it makes exciting to try out mutation testing in your project? I know for sure that the extra effort certainly is worth the while.