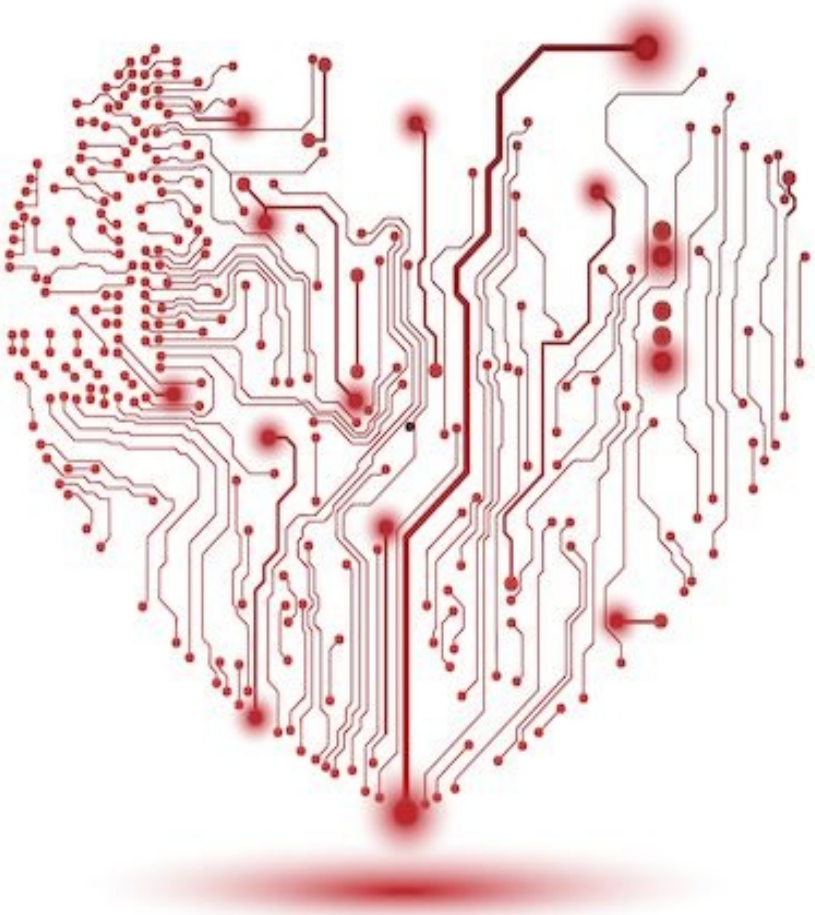
The Pragmatic Programmers

Rails 4 Test Prescriptions

Build a Healthy Codebase



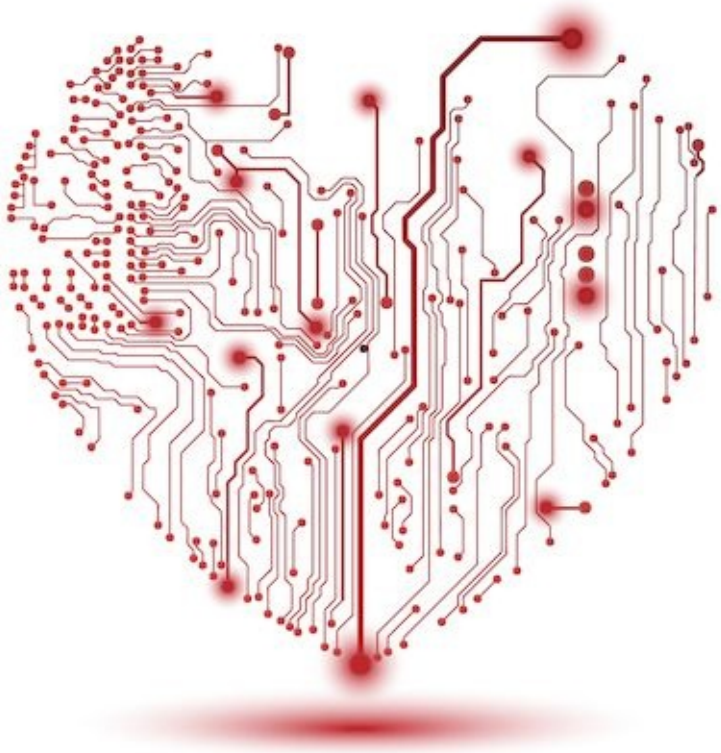
Noel Rappin

Edited by Lynn Beighley

The Pragmatic Programmers

Rails 4 Test Prescriptions

Build a Healthy Codebase



Noel Rappin

Edited by Lynn Beighley

Rails 4 Test Prescription

Build a Healthy Codeba

by Noel Rapp

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 http://pragprog.com.

The team that produced this book includes: Susannah Davidson Pfalzer (editor), Potomac Indexing, LLC (indexer), Candace Cunningham (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: 978-1-941222-19-5

Book version: P1.0—December 2014

Table of Contents

Early praise for *Rails 4 Test Prescriptions*

Rails 4 Test Prescriptions is quite simply the best book on the market on the topic of testing Rails applications. It's full of distilled wisdom from Noel's many years of experience. I especially love the emphasis on thinking through the tradeoffs involved in picking a test strategy; in my experience, thinking intentionally about those tradeoffs is one of the most important steps you can take toward building an effective test suite.

→ Myron Marston

Lead maintainer of RSpec and creator of the VCR gem

Rails 4 Test Prescriptions will benefit both developers new to test-driven development and those who are more experienced with it. Noel Rappin presents concepts like mocking and stubbing in a very detailed but also approachable and entertaining way. I loved the first edition of this book, and the second is even better. I highly recommend it!

→ Nell Shamrell-Harrington

Senior developer, PhishMe

If anyone asks me how to master testing in Rails applications, I will tell them to read this book first.

→ Avdi Grimm

Head chef, RubyTapas

Sometimes testing sucks. This book magically makes testing not suck; it makes it easy and rewarding with well-written explanations. It is the essential resource for any developer testing Rails application It's more than just a testing primer; developers will learn how to create optimal and efficient test suites for Rails. A must-read for beginners and seasoned programmers alike.

→ Liz Abinante

Software engineer, New Relic

Acknowledgment

It’s been six years since I first started working on a book called *Rails Test Prescriptions*. In that time many people have helped me make this book better than I could have made it on my own. This includes but is by no means limited to the following people.

Without the encouragement of Brian Hogan and Gregg Pollack, this might still be a self-published book. Brian also provided a valuable review of this edition.

Lynn Beighley has been my editor on this version of the project and has shaped the material this time around. I’ve worked with Susannah Pfalzer on all my Pragmatic projects, and she’s always been helpful and great to work with.

Many technical people reviewed this book and had their comments incorporated. They include Liz Abinante, Ashish Dixit, Mike Gehard, Derek Graham, Avdi Grimm, Sean Hussey, John Ivanoff, Evan Light, Myron Marston, Kerri Miller, Tim Morton, Matt Rohrer, Nell Shamrell, Brian Van Loo, and Andy Waite.

I’ve been very fortunate to be working at Table XI while writing this book. Not only have they been very supportive of the project; I’ve also had the chance to get insights and corrections from my very skilled coworkers.

This book is a commercial product built on the time and generosity of developers who create amazing tools and release them to the world for free. Thanks to all of you.

My family has always been encouraging. Thanks to my children, Emma and Elliot, who are more amazing and awesome than I ever could have hoped. And thanks to my wife Erin, the best part of my life and my favorite person. I love you very much.

Introductio

Test-driven development, or TDD, is the counterintuitive idea that developers will improve both the design and the accuracy of their code by writing the code test-first. When adding new logic to a program, the TDD process starts by writing an automated test describing the behavior of code that does not yet exist. In a strict TDD process, new logic is added to the program only after a failing test is written to prompt the creation of the logic.

Writing tests before code, rather than after, allows your tests to help guide the design of your code in small, incremental steps. Over time this creates a well-factored codebase that is easy to change.

We'll apply the TDD process to the creation of applications using Ruby and Rails. We'll talk about how to apply TDD to your daily coding and about the tools and libraries that make testing in Rails easier.

But first let me tell you a story.

A Test-Driven Fable

Imagine two programmers working on the same task. Both are equally skilled, charming, and delightful people, motivated to do a high-quality job as quickly as possible. The task is not trivial bu not wildly complex either; for the sake of discussion, let's say it's a new user logging in to a website and entering some detailed pertinent information.

The first developer, who we'll call Sam, says, "This is pretty easy, and I've done it before. I don't need to write tests." And in five minutes Sam has a working method ready to verify.

Our second developer is named Jamie. Jamie says, "I need to write some tests." Jamie starts writing test describing the desired behavior. The test is executable and passes if the corresponding code matches the test. Writing the test takes about five minutes. Five additional minutes later, Jamie also has a working method, which passes the test and is ready to verify. Because this is a fable, we are going to assume that Sam is allergic to automated testing, while Jamie is similarly averse to manuall verifying against the app in the browser.

At this point you might expect me to say that even though it has taken Jamie more time to write the method, Jamie has written code that is more likely to be correct, robust, and easy to maintain. That's true. I am going to say that. But I'm also going to say that there's a good chance Jamie will be done before Sam even though Jamie is taking on the additional overhead of writing tests.

Let's watch our programmers as they keep working. Sam has a five-minute lead, but both of them need to verify their work. Sam needs to test in a browser; we said the task requires a user to log in. Let's say it takes Sam one minute to log in and perform the task to verify the code in a development environment. Jamie verifies by running the test—that takes about ten seconds. (At this point Jamie h to run only one test, not the entire suite.)

Perhaps it takes each developer three tries to get it right. Since running the test is faster than verifyin in the browser, Jamie gains a little bit each try. After verifying the code three times, Jamie is only tw and a half minutes behind Sam. (In a slight nod to reality, let's assume that both of them need to verify one last time in the browser once they think they are done. Since they both need to do this, it's not an advantage for either one.)

At this point, with the task complete, both break for lunch (a burrito for Jamie, an egg salad sandwich for Sam). After lunch they start on the next task, which is a special case of the first task. Jamie has most of the test setup in place, so writing the test takes only two minutes. Still, it's not looking good for Jamie, even after another three rounds trying to get the code right. Jamie remains a solid two minutes behind Sam.

Let's get to the punch line. Sam and Jamie are both conscientious programmers, and they want to clean up their code with a little *refactoring*, meaning that they are improving the code's structure without changing its behavior. Now Sam is in trouble. Each time Sam tries the refactoring, it takes two minutes to verify both tasks, but Jamie's test suite still takes only about ten seconds. After three more tries to get the refactoring right, Jamie finishes the whole thing and checks it in three and a half minutes ahead of Sam. (Jamie then catches a train home and has a pleasant evening. Sam just misses the train and gets caught in a sudden rainstorm. If only Sam had run tests.)

My story is simplified, but look at all the things I didn't assume. I didn't assume that Jamie spent les actual time on task, and I didn't assume that the tests would help Jamie find errors more easily—although I think Jamie would, in fact, find errors more easily. (Of course, I also didn't assume that Jamie would have to track down a broken test in some other part of the application.)

It is frequently faster to run multiple verifications of your code as an automated test than to always check manually. And that advantage only increases as the code gets more complex. And the automate check will do a better job of ensuring steps aren't forgotten.

There are many beneficial side effects of having accurate tests. You'll have better-designed code in which you'll have more confidence. But the most important benefit is that if you do testing well, you work will go faster. You may not see it at first, but at some point in a well-run test-driven project, you'll notice that you have fewer bugs and that the bugs that do exist are easier to find. It will be easier to add new features and modify existing ones. You'll be doing better on the only code-quality metric that has any validity: how easy it is to find incorrect behavior and add new behavior. One reason why it is sometimes hard to pin down the benefit of testing is that good testing often just feels like you are doing a really good job programming.

Of course, it doesn't always work out that way. The tests might have bugs. They might be slow. Environmental issues may mean things that work in a test environment won't work in a development environment. Code changes will break tests. Adding tests to existing code is a pain. As with any othe programming tool, there are a lot of ways to cause yourself pain with testing.

Who You Are

This book's goal is to show you how to apply a test-driven process and automated testing as you buil your Rails application. Being test-driven allows you to use testing to explore your code's design. We will see what tools are available and discuss when those tools are best used. Tools come and tools go so I'm really hoping you come away from this book committed to the idea of writing better code through the small steps of a test-driven or behavior-driven development process.

I'm assuming some things about you.

I'm assuming you are already comfortable with Ruby and Rails and that you don't need this book to explain how to get started creating a Rails application in and of itself. I am *not* assuming you have ar particular familiarity with testing frameworks or testing tools used within Rails.

Over the course of this book, we'll go through the tools that are available for writing tests, and we'll talk about them with an eye toward making them useful in building your application. This is Rails, so naturally I have my own opinions, but the goal with all the tools and all the advice is the same: to hel you to write great applications that do cool things and still catch the train home.

Testing First Drives Design

Success with test-driven development starts with trusting the process. The classic process goes like this:

1. Create a test. The test should be short and test for one thing in your code. The test should run automatically.
2. Make sure the test fails. Verifying the test failure before you write code helps ensure that the tes really does what you expect.
3. Write the simplest code that could possibly make the test pass. Don't worry about good code yet Don't look ahead. Sometimes, write just enough code to clear the current error.
4. After the test passes, refactor to improve the code. Clean up duplication. Optimize. Create new abstractions. Refactoring is a key part of design, so don't skip this. Remember to run the tests again to make sure you haven't changed any behavior.

Repeat until done. This will, in theory, ensure that your code is always as simple as possible and is always completely covered by tests. We'll spend most of this book talking about how to best manage this process using the tools of the Rails ecosystem and solving the kinds of problems that you get in a modern web application. And we'll talk about the difference between "in theory" and "in practice."

If you use this process, you will find that it changes the design of your code.

Software design is a tricky thing to pin down. We use the term all the time without really defining it. For our purposes, *design* is anything in the way the code is structured that goes beyond the logical correctness of the code. You can design software for many different reasons—optimization for speed clarity of naming, robustness against errors, resistance to change, ease of maintenance….

Test-driven development enables you to design your software continuously and in small steps, allowing the design to respond to the changes in the code.

Specifically, design happens at three different points in the test-driven process:

- When you decide which test to write next, you are making a claim about what functionality your code should have. This frequently involves thinking about how to add that functionality to the existing code, which is a design question.
- As you write a test, you are designing the interaction between the test and the code, which is also the interaction between the part of the code under test and the rest of the application. This part of the process is used to create the API you want the code to have.
- After the test passes, you refactor, identifying duplication, missing abstractions, and other places where the code's design can be improved.

Prescription 1: Use the TDD process to create and adjust your code's design in small, incremental steps.

Continually aligning your code to the tests tends to result in code that is made up of small methods, each of which does one thing. These methods tend to be loosely coupled and have minimal side effects. As it happens, the hallmark of easy-to-change code is small methods that do one thing, are loosely coupled, and have minimal side effects.

I used to think it was a coincidence that tested code and easy-to-change code have similar structures, but I've realized the commonality is a direct side effect of building the code in tandem with the tests. In essence, the tests act as a universal client for the entire codebase, guiding all the code to have clea interactions between parts because the tests, acting as a third-party interloper, have to get in between all the parts of the code to work. Metaphorically, compared to code written without tests, your code has more surface area and less work happening behind the scenes where it is hard to observe.

This theory explains why testing works so much better when the tests come first. Even waiting a littl bit to write tests is significantly more painful. When the tests are written first, in very close intertwined proximity to the code, they encourage a good structure with low coupling (meaning different parts of the code have minimal dependencies on each other) and high cohesion (meaning code that is in the same unit is all related).

When the tests come later, they have to conform to the existing code, and it's amazing how quickly code written without tests will move toward low-cohesion and high-coupling forms that are much harder to cover with tests. If your only experience is with writing automated tests long after the initia code was written, the experience was likely quite painful. Don't let that turn you away from a TDD approach; the tests and code you will write with TDD are much different.

When you are writing truly test-driven code, the tests are the final source of truth in your application. This means that when there is a discrepancy between the code and the tests, your first assumption is that the test is correct and the code is wrong. If you're writing tests after the code, then your assumption must be that the code is the source of truth. As you write your code using test-driven development, keep in mind the idea that the tests are the source of truth and are guiding the code's structure.

Prescription 2: In a test-driven process, if it is difficult to write tests for a feature, strongly consider the possibility that the underlying code needs to be changed.

What Is TDD Good For?

The primary purpose of test-driven development is to go beyond mere verification and use the tests t improve the code's structure. That is, TDD is a software-development technique masquerading as a code-verification tool.

Automated tests are a wonderful way of showing that the program does what the developer thinks it does, but they are a lousy way of showing that what the developer thinks is what the program actually should do. "But the tests pass!" is not likely to be comforting to a customer when the developer's assumptions are just flat-out wrong. I speak from painful experience.

The kinds of tests written in a TDD process are not a substitute for acceptance testing, where users or customers verify that the code does what the user or customer expects. TDD also does not replace some kind of quality-assurance phase where users or testers pound away at the actual program trying to break something.

Further, TDD does not replace the role of a traditional software tester. It is a development process tha produces better and more accurate code. A separate verification phase run by people who are not the original developers is still a good idea. For a thorough overview of more traditional exploratory testing, read *Explore It!* [Hen13].

Verification is valuable, but the idea of verification can be taken too far. You sometimes see an argument against test-driven development that says, "The purpose of testing is to verify that my program is correct. I can never prove correctness with 100 percent certainty. Therefore, testing has no value." (Behavior-driven development and RSpec were created, in part, to combat this attitude.) Ultimately, though, testing has a lot of positive benefits to offer for coding, even beyond verification.

Preventing regression is often presented as one of the paramount benefits of a test-driven developme process. And if you are expecting me to disagree out of spite, you're out of luck. Being able to squash regressions before anybody else sees them is one of the key ways in which strict testing will speed up your development over time.

You may have heard that automated tests provide an alternate method of documenting your program —that the tests, in essence, provide a detailed functional specification of the program's behavior. That's the theory. My experience with tests acting as documentation is mixed, to say the least. Still, it's useful to keep this in mind as a goal, and most of the things that make tests work better as documentation will also make the tests work better, period.

To make your tests effective as documentation, focus on giving them names that describe the reason for their existence, keeping tests short, and refactoring out common features such as test setup. The documentation advantage of refactoring includes removing clutter from the test itself—when a test has a lot of raggedy setup and assertions, it can be hard for a reader to focus on the important feature. As you'll see, a test that requires a bunch of tricky setup often indicates a problem in the underlying code. Also, with common features factored out it's easier to focus on what's different in each individual test.

In a testing environment, blank-page problems are almost completely nonexistent. I can always think of something that the program needs to do, so I write a test for that. When you're working test-first,

the order in which pieces are written is not so important. Once a test is written, the path to the next one is usually more clear: find some way to specify something the code doesn't do yet.

When TDD Needs Some Help

Test-driven development is very helpful, but it won't solve all of your development problems by itself. There are areas where developer testing doesn't apply or doesn't work very well.

I mentioned one case already: developer tests are not very good at determining whether the application is behaving correctly according to requirements. Strict TDD is not great at acceptance testing. There are, however, automated tools that do try to tackle acceptance testing. Within the Rails community, the most prominent of these is Cucumber; see Chapter 10, *Integration Testing with Capybara and Cucumber*. Cucumber can be integrated with TDD—you'll sometimes see this called outside-in testing. That's a perfectly valid and useful test paradigm, but it's an extension of th classic TDD process.

Testing your application assumes that you know the right answer to specify. And although you sometimes have clear requirements or a definitive source of correct output, other times you don't know what exactly the program needs to do. In this exploratory mode, TDD is less beneficial, becaus it's hard to write tests if you don't know what assertions to make about the program. Often this lack direction happens during initial development or during a proof of concept. I also find myself in this position a lot when view-testing—I don't know what to test for until I get some of the view up and visible.

The TDD process has a name for the kind of exploratory code you write while trying to figure out the needed functionality: *spike*, as in, "I don't know if we can do what we need with the Twitter API; let' spend a day working on a spike for it." When working in spike mode, TDD is generally not used, but code written during the spike is not expected to be used in production; it's just a proof of concept to l thrown away and replaced with a version written using TDD.

When view-testing, or in other nonspike situations where I'm not quite sure what output to test for, I often go into a "test-next" mode, where I write the code first but in a TDD-sized small chunk, and the immediately write the test. This works as long as I make the switch between test and code frequently enough to get the benefit of having the code and test inform each other's design.

TDD is not a complete solution for verifying your application. We've already talked about acceptanc tests; it's also true that TDD tends to be thin in terms of the quantity of unit tests written. For one thing, in a strict TDD process you would never write a test that you expect to pass before writing mor code. In practice, though, you will do this all the time. Sometimes I see and create an abstraction in the code but there are still valid test cases to write. In particular, I'll often write code for potential error conditions even if I think they are already covered in the code. It's a balance because you lose some of the benefit of TDD by creating too many test cases that don't drive code changes. One way t keep the balance is to make a list of the test cases before you start writing the tests—that way you'll remember to cover all the interesting cases.

And some things are just hard. In particular, some parts of your application will be very dependent on an external piece of code in a way that makes it difficult to isolate them for unit testing. Test doubles which are special kinds of objects that can stand in for objects that are part of your application, are one way to work around this issue; see Chapter 7, *Using Test Doubles as Mocks and Stubs*. But there are definitely cases (though they're not common) in which the cost of testing a complex feature is higher than the value of the tests.

Recent discussions in the Rails community have debated whether TDD's design benefits are even valuable. You may have heard the phrase "test-driven design damage." I strongly believe that TDD, and the relatively smaller and more numerous classes that a TDD process often brings, do result in more clear and more valuable code. But the TDD process is not a replacement for good design instincts; it's still possible to create bad code when testing, or even to create bad code in the name of testing.

Words to Live By

* Any change to the program logic should be driven by a failed test.

- If it’s not tested, it’s broken.
- Testing is supposed to help for the long term. The long term starts tomorrow, or maybe after lunch.
- It’s not done until it works.
- Tests are code; refactor them, too.
- Start a bug fix by writing a test.
- Tests monitor the quality of your codebase. If it becomes difficult to write tests, it often means your codebase is too interdependent.

A Word About Tools, Best Practices, and Teaching TDD

There are two test libraries in general use in the Rails community—Minitest and RSpec—meaning I had a choice of which tool to use as the primary library in this book. The book is about how to test Rails in general, and therefore the details of the testing library in use are secondary to most of it, but still, the examples have to be presented using one tool or the other.

Minitest is part of the Ruby Standard Library and is therefore available everywhere you might use Ruby (1.9 and up). It has a straightforward syntax that is the Ruby translation of the original SUnit an JUnit libraries (for Smalltalk and Java, respectively), and it is the default alternative for a new Rails project.

RSpec is a separate testing tool designed to support an emphasis on specifying behavior rather than implementation, sometimes called behavior-driven development (BDD). Rather than using terms like "test" and "assert," RSpec uses "expect" and "describe." BDD emphasizes setting expectations about the code not yet written rather than assertions about code in the past.

RSpec has a quirky, metaprogrammed syntax with a certain "love it or hate it" vibe. It's more flexibl which means more expressive and more complicated, and it has a larger ecosystem of related tools.

The primary testing tool used in this book is RSpec because after going back and forth quite a bit, I've decided that its expressiveness makes it easier to work with over the course of the entire book, even i it has a slightly steeper learning curve. That said, there's a whole chapter on Minitest, and we'll discuss most of the extra tools in a way that references both RSpec and Minitest. Every RSpec example in the downloadable code has a corresponding Minitest version.

That leads to a more general point: sometimes the best practice for learning isn't the best practice fo experts. In some cases in this book we'll use relatively verbose or explicit versions of tools or tests t make it clear what the testing is trying to do and how. In particular, for clarity I've tried not to use multiple extra tools at once. For example, we discuss factory_girl as a way to create data for your tests, but we don't use factory_girl in examples that are intended to highlight, say, Capybara. I intend to focus each example on one tool or technique.