Extracted from:

Rails Test Prescriptions

Keeping Your Application Healthy

The Pragmatic Programmers

Rails Test Prescriptions

Keeping Your Application Healthy

Noel Rappin

Edited by Colleen Toporek

Chapter 1

The Goals of Automated Developer Testing

1.1 A Testing 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 but not wildly complex either; for the sake of discussion, we'll say it's behavior based on a new user registering for a website and entering pertinent information.

The first developer, who we'll call Ernie,[1] says, "This is pretty easy, and I've done it before. I don't need to write tests." And in five minutes Ernie has a working method ready to verify.

Our second developer is, of course, named Bert. Bert says, "I need to write some tests."[2] Bert starts writing a test, and in five minutes, he has a solid test of the new feature. Five minutes more, Bert also has a working method ready to verify. Because this is a fable, we are going to assume that Ernie is allergic to automated testing, while Bert is similarly averse to manually running against the app in the browser.

At this point, you no doubt expect me to say that even though it has taken Bert more time to write the method, Bert has written code that is more likely to be correct, robust, and easy to maintain. That's true.

1. Because that's his name.
2. Actually, if Bert is really into Agile, he probably asks, "Who am I going to pair with?" but that's an issue for another day.

But I'm also going to say that there's a good chance Bert will be done before Ernie.

Observe our programmers a bit further. Ernie has a five-minute lead, but both people need to verify their work. Ernie needs to test in a browser; we said the task requires a user to log in. Let's say it takes Ernie one minute to set up the task and run the action in his development environment. Bert verifies by running the test—that takes about ten seconds. (Remember, Bert has to run only one test, not the entire suite.)

Let's say it takes each developer three tries to get it right. Since running the test is faster than verifying in the browser, Bert gains a little bit each try. After verifying the code three times, Bert is only two and half minutes behind Ernie.[3]

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

Bear with me one more step, and we'll get to the punch line. Ernie and Bert are both conscientious programmers, and they want to clean their code up with a little refactoring. Now Ernie is in trouble. Each time he tries the refactoring, he has to spend two minutes verifying both tasks, but Bert's test suite still takes only about ten seconds. After three more tries to get the refactoring right, Bert finishes the whole thing and checks it in three and a half minutes ahead of Ernie.[4]

My story is obviously simplified, but let's talk a moment about what I didn't assume. I didn't assume that the actual time Bert spent on task was smaller, and I didn't assume that the tests would help Bert find errors more easily—although I think that would be true.[5] The main

3. 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.
4. Bert then catches his train home and has a pleasant evening. Ernie just misses his train, gets caught in a sudden rainstorm, and generally has a miserable evening. If only he had run his tests....
5. Of course, I didn't assume that Bert would have to track down a broken test in some other part of the application, either.

point here is that it's frequently faster to run multiple verifications of your code as an automated test than to always check manually. And that advantage is only going to increase as the code gets more complex.

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'll notice that your work goes faster. You may not see it at first, but at some point in a well-run test-driven project, you'll notice fewer bugs and that the bugs that do exist are easier to find. You'll notice that it's easier to add new features and easier to modify existing ones. As far as I'm concerned, the only code-quality metric that has any validity is how easy it is over time to find bugs and add new behavior.

Of course, it doesn't always work out that way. The tests might have bugs. 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 already existing code is a pain. Like any other programming tool, there are a lot of ways to cause yourself pain with testing.

1.2 Who Are You?

The goal of this book is to show you how to apply a test-driven process as you build your Rails application. I'll show you what's available and try to give you some idea of what kind of tools are best used in what circumstances. Still, tools come and tools go, so what I'm really hoping is that you come away from this book committed to the idea of writing better code through the small steps of a TDD or BDD process.

There are some things I'm assuming about you.

I'm assuming that 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 any particular familiarity with testing frameworks or testing tools used within Rails. If you do have familiarity, you may find some of the early chapters redundant. However, if you have tried to use test frameworks but got frustrated and didn't think they were effective, I recommend Chapter 3, *Writing Your First Tests*, on page 44 and Chapter 4, *TDD, Rails Style*, on page 63, since they walk through the TDD process for a small piece of Rails functionality.

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 all the tools have the same goal: to help you to write great applications that do great things and still catch the train home.

1.3 The Power of Testing First

The way to succeed with Test-Driven Development (TDD) is to trust 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 result of the test should be deterministic.
2. Make sure the test fails. Verifying the test failure before you write code helps ensure that the test 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, just write enough code to clear the current error.
4. Refactor. After the test passes. Clean up duplication. Optimize. This is where design happens, so don't skip this. Remember to run the tests at the end to make sure you haven't changed any behavior.

Repeat until done. This will, on paper at least, ensure that your code is always as simple as possible and always is completely covered by tests. We'll spend most of the rest of this book talking about the details of step 1 and how to use Rails tools to write useful tests.

If you use this process, you will find that it changes the structure of the code you write. The simple fact that you are continually aligning your code to the tests results 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 well-designed code is small methods that do one thing, are loosely coupled, and have minimal side effects. I used to think that was kind of a lucky coincidence, but now I think it's a direct side effect of building the code in tandem with the tests. In effect, the tests act as a universal client for the entire code base, guiding all the code to have clean interactions between parts because

A Historical Parallel

What's a Rails book without a good Franklin Roosevelt anecdote, right?

There's a widely told and probably apocryphal story about FDR meeting with a group of activists pushing a reform agenda—exactly what the group wanted seems to have been lost to history.

Anyway, when they were done with the meeting, FDR is supposed to have said to them, "I agree with you. I want to do it; now go make me do it."

Ignore for the moment the question of whether this statement makes sense as politics; it makes perfect sense as a test-driven development motto. Your requirements determine what your applications *want* to do. Your tests *make* the application do it.

the tests, acting as a third-party interloper, have to get in between all the parts of the code in order to work.

This theory explains why writing the code first causes so much pain when writing tests even if you just wait a little bit to get to the tests. When the tests are written first, or in very close intertwined proximity to the code, then the tests drive the code's structure and enable the code to have the good high-cohesion/low-coupling structure. When the tests come later, they have to conform to the existing code, and it's amazing how easily and 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 with writing unit tests comes only long after the initial 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.

1.4 What Is TDD Good For?

The primary purpose of this style of testing where the developer is writing tests for her own benefit is to improve the structure of the code. That is, TDD is a software development technique rather than a complete testing program. (Don't believe me, ask Kent Beck, who is most responsible for TDD as a concept and who said, "Correctness is a side effect" on a recent podcast.)[6]

Automated developer 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.[7]

Automated developer testing is not a substitute for *acceptance testing* with users or customers (which can itself be partially automated via something like Cucumber) or some kind of QA phase where users or testers pound away at the actual program trying to break something.

This goal can be taken too far, however. You sometimes see an argument against Test-Driven Development that runs something like this: "The purpose of testing is to verify that my program is correct. I can never prove this with 100 percent certainty. Therefore, testing has no value." (RSpec and Behavior-Driven Development were created, in part, to combat this attitude.) Ultimately, though, testing has a lot of positive benefits for coding, even beyond verification.

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

Another common benefit you may have heard in connection with automated tests is that they provide an alternate method of documenting your program. The tests, in essence, provide a detailed, functional specification of the behavior of the program.

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 your tests descriptive names, keeping tests short, and refactoring out com-

6. http://twit.tv/floss87. Good interview, recommended.
7. He says, speaking from painful experience....

mon setup and assertion parts. The documentation advantage of refactoring is 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 functional part. 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 actual order in which pieces are written is not so important. Once a test is written, the path to the next one is usually clear, and so on, and so on.

1.5 When TDD Needs Some Help

Test-Driven Development is very helpful, but it's not going to 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 very good 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 15, *Acceptance Testing with Cucumber*, on page 237. Cucumber can be integrated with TDD—you'll see this called *outside-in testing* or see the acronym ATDD for Acceptance Test–Driven Design. That's a perfectly valid and useful test paradigm, but it's an extension of the classic TDD process.

Testing your application assumes that you know the right answer. And although you will have clear requirements or a definitive source of correct output some of the time, other times you don't know what exactly the program needs to do. In this exploratory mode, TDD is less beneficial, because it's hard to write tests if you don't know what assertions to make about the program. Often this happens during initial development or during a proof of concept. I 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.

In classic Extreme Programming parlance, this kind of programming is called a *spike*, as in, "I don't know if we can do what we need with the Twitter API; let's spend a day working on a spike for it." When working in spike mode, TDD is generally not used, but it's also the expectation that the code written during the spike is not used in production; it's just a proof of concept.

When view testing, or in other nonspike situations where I'm not quite sure what output to test for, I tend to go into a "test-next" mode, where I write the code first, but in a TDD-sized small chunk, and then 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 acceptance tests, but it's also true that TDD tends to be thin in terms of the amount of unit tests written. For one thing, a strict TDD process would never write a test that you expect to pass. In practice, though, I 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 to 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 hey, some things are just hard. In particular, some parts of your application are going to be very dependent on an external piece of code in a way that makes it hard to isolate them for unit testing. Mock objects, described in Chapter 7, *Using Mock Objects*, on page 103, can be one way to work around this issue. But there are definitely cases where the cost of testing a feature like this is higher than the value of the tests. To be clear, I don't think that is a common occurrence, but it would be wrong to pretend that there's never a case where the cost of the test is too high.

1.6 Coming Up Next...

This book is divided into six parts.

Part I, which you are currently in the middle of, is an introduction to Rails testing. The next chapter, Chapter 2, *The Basics of Rails Testing*, on page 26, covers what you need to know to get started with unit testing in Ruby and Rails, covering Test::Unit, Test-Driven Design, and the basic workflow of a Ruby test. The following two chapters, Chapter 3, *Writing Your First Tests*, on page 44 and Chapter 4, *TDD, Rails Style*, on page 63, present a tutorial or walk-through of a basic Rails feature realized using TDD.

Words to Live By

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

A test should be as close as possible to the associated code.

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.

Part II of the book is about application data. Most of your Rails tests will cover model code, discussed in Chapter 5, *Testing Models with Rails Unit Tests*, on page 74. You'll often need sample data to run tests, and Chapter 6, *Creating Model Test Data with Fixtures and Factories*, on page 83 talks about the two most common ways to manage test data. Sometimes, though, you just need to bypass normal behavior entirely, and Chapter 7, *Using Mock Objects*, on page 103 talks about the standard way of replacing normal program behavior as needed in testing.

The models are the back room of your code, and Part III talks about testing the user-facing parts of your application. In Chapter 8, *Testing Controllers with Functional Tests*, on page 130, we'll talk about the standard Rails way of testing controllers, while Chapter 9, *Testing Views*, on page 141 discusses view testing. Increasingly, front-end code includes Ajax and JavaScript, discussed in Chapter 10, *Testing JavaScript and Ajax*, on page 157, which introduces the Jasmine framework for JavaScript testing.

The second half of the book is largely about extensions to core Rails testing. Part IV covers two of the biggest. Shoulda is covered in Chapter 11, *Write Cleaner Tests with Shoulda and Contexts*, on page 171, while RSpec gets its due in Chapter 12, *RSpec*, on page 188.

Part V of the book covers integration and acceptance testing that exercises your entire application stack. First, Rails core integration testing is covered in Chapter 13, *Testing Workflow with Integration Tests*, on page 217. Webrat and Capybara are tools that give integration tests more clarity and power, and they get their own chapter in Chapter 14, *Write Better Integration Tests with Webrat and Capybara*, on page 226. Cucumber has become a very popular tool for acceptance testing, and Chapter 15, *Acceptance Testing with Cucumber*, on page 237 tells you all about it.

The last part of the book is about evaluating your tests. The most common objective measure of tests is code coverage, which you will read about in Chapter 16, *Using Rcov to Measure Test Coverage*, on page 260. Coverage isn't everything in testing style, though, and Chapter 17, *Beyond Coverage: What Makes Good Tests?*, on page 272 talks about five other habits of highly successful tests. Adding tests to an existing application has its own challenges, discussed in Chapter 18, *Testing a Legacy Application*, on page 284. Finally, making your tests run faster is always a good thing, and Chapter 19, *Performance Testing and Performance Improvement*, on page 300 covers many different strategies.

Ready? Me too.

1.7 Acknowledgments

Over the course of the two years that I have been working on this project, I have had the guidance and support of many people. I hope I haven't forgotten anyone.

Back when this was just a DIY project, several people acted as early readers and offered useful comments including Paul Barry, Anthony Caliendo, Brian Dillard, Sean Hussey, John McCaffrey, Matt Polito, and Christopher Redinger. Alan Choyna and David DiGioia helped support the original Rails Prescriptions website. Alice Toth provided the original website design. Dana Jones made many, many valuable editorial corrections early in the life of the book.

Brian Hogan was the first person to suggest that this book might work for Pragmatic. Gregg Pollack was the second, and Gregg's kind words about this project on the official Rails blog were the push I needed to actually submit it.

Everybody I've worked with at Pragmatic has been outstanding. Dave Thomas and Andy Hunt said nice things about early chapters of the book, which was very encouraging. I doubt very much that Dave Thomas remembers when I introduced myself to him at Rails Edge in