Extracted from:

Rails 4 Test Prescriptions

Build a Healthy Codebase

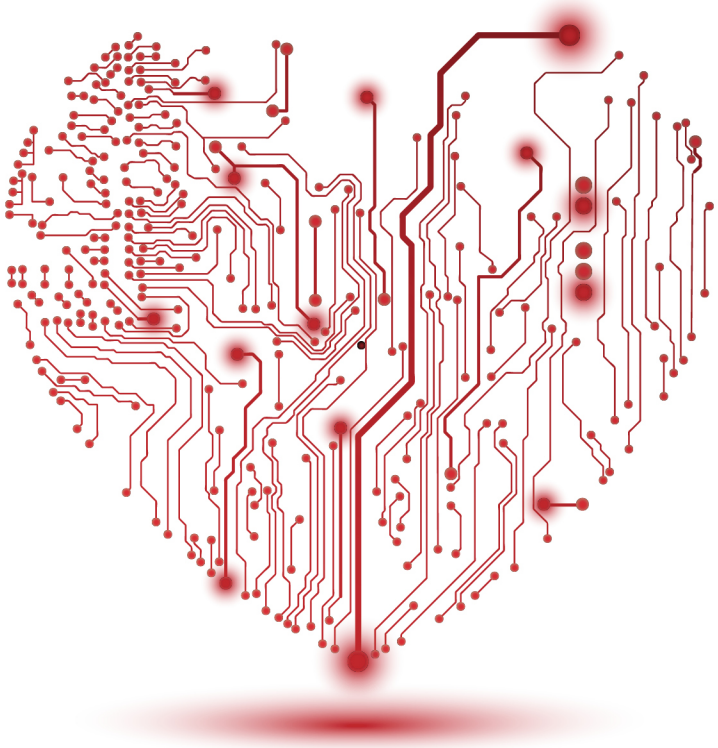
The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Rails 4 Test Prescriptions

Build a Healthy Codebase



Noel Rappin

Edited by Lynn Beighley

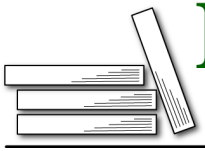
Rails 4 Test Prescriptions

Build a Healthy Codebase

Noel Rappin

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:

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
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—December 2014

As the Rails community has matured, Rails developers have become much more likely to work with codebases and test suites that contain many years' worth of work. As a result, there has been a lot of discussion about design strategies to manage complexity over time.

There hasn't been nearly as much discussion about what practices make tests and test suites continue to be valuable over time. As applications grow, as suite runs get longer, as complexity increases, how can you write tests that will be useful in the future and not act as an impediment to future development?

The Big One

The best, most general piece of advice I can give about the style and structure of automated tests is this:

Prescription 8	Your tests are also code. Specifically, your tests are code that does not have tests.

Your code is verified by your tests, but your tests are verified by nothing.

Having your tests be as clear and manageable as possible is the only way to keep them honest.

The Big Two

If a programming practice or tool is successful, following or using it will make it easier to:

- add the code I need in the short term.
- continue to add code to the project over time.

All kinds of gems in the Ruby and Rails ecosystem help with the first goal (including Rails itself). Testing is normally thought of as working toward the second goal. That's true, but often people assume the only contribution testing makes toward long-term application health is verification of application logic and prevention of regressions. In fact, over the long term test-driven development tends to pay off as good tests lead toward modular designs.

This means a valuable test saves time and effort over the long term, while a poor test costs time and effort. I've focused on five qualities that tend to make a test save time and effort. The absence of these qualities, on the other hand, is often a sign that the test could be a problem in the future.

The More Detailed Five: SWIFT Tests

I like to use five criteria to evaluate test quality. I've even managed to turn them into an acronym that is only slightly contrived: SWIFT.

- Straightforward
- Well defined
- Independent
- Fast
- Truthful

Let's explore those in more detail.

[S]traightforward

A test is *straightforward* if its purpose is immediately understandable.

Straightforwardness in testing goes beyond just having clear code. A straightforward test is also clear about how it fits into the larger test suite. Every test should have a point: it should test something different from the other tests, and that purpose should be easy to discern from reading the test.

Here is a test that is not straightforward:

```
## Don't do this
it "should add to 37" do
  expect(User.all_total_points).to eq(37)
end
```

Where does the 37 come from? It's part of the global setup. If you were to peek into this fake example's user fixture file, you'd see that somehow the totals of the points of all the users in that file add up to 37. The test passes. Yay?

There are two relevant problems with this test:

- The 37 is a magic literal that apparently comes from nowhere.
- The test's name is utterly opaque about whether this is a test for the main-line case, a test for a common error condition, or a test that exists only because the coder was bored and thought it would be fun.

Combine these problems, and it quickly becomes next to impossible to fix the test a few months later when a change to the `User` class or the fixture file breaks it.

Naming tests is critical to being straightforward. Creating data locally and explicitly also helps. With most factory tools (see *Factories*, on page ?), default

values are preset, so the description of an object created in the test can be limited to defining only the attributes that are actually important to the behavior being tested. Showing those attributes in the test is an important clue to the programmer's intent. Rewriting the preceding test with a little more information might result in this:

```
it "rounds total points to the nearest integer" do
  User.create(:points => 32.1)
  User.create(:points => 5.3)
  expect(User.all_total_points).to eq(37)
end
```

It's not poetry, but at the very least an interested reader now knows where that pesky 37 comes from and where the test fits in the grand scheme of things. The reader might then have a better chance of fixing the test if something breaks. The test is also more independent since it no longer relies on global fixtures—making it less likely to break.

Long tests or long setups tend to muddy the water and make it hard to identify the critical parts of the test. The same principles that guide refactoring and cleaning up code apply to tests. This is especially true of the rule that states "A method should only do one thing," which here means splitting up test setups into semantically meaningful parts, as well as keeping each test focused on one particular goal.

On the other hand, if you can't write short tests, consider the possibility that it is the code's fault and you need to do some redesign. If it's hard to set up a short test, that often indicates the code has too many internal dependencies.

There's an old programming adage that goes like this: "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it." (I got the quote from http://quotes.cat-v.org/programming/, but the original source is Brian W. Kernighan and P.J. Plauger's *The Elements of Programming Style*.) Because tests don't have their own tests, this quote suggests that you should keep your tests simple to give yourself cognitive room to understand them.

In particular, this guideline argues against using clever tricks to reduce duplication among multiple tests that share a similar structure. If you find yourself starting to metaprogram to generate multiple tests in your suite, you'll probably find that complexity working against you at some point. You never want to have to decide whether a bug is in your test or in the code. And when—not if—you do find a bug in your test suite, it's easier to fix if the test code is simple.

We'll talk more about clarity issues throughout the book. In particular, the issue will come up when we discuss factories versus fixtures as ways of adding test data in Chapter 6, *Adding Data to Tests*, on page ?.

[W]ell Defined

A test is *well defined* if running the same test repeatedly gives the same result. If your tests are not well defined, the symptom will be intermittent, seemingly random test failures (sometimes called Heisenbugs, Heisenspecs, or Rando Calrissians).

Three classic causes of repeatability problems are time and date testing, random numbers, and third-party or Ajax calls. In all cases the issue is that your test data changes from test to test. Dates and times have a nasty habit of monotonically increasing, while random data stubbornly insists on being random. Similarly, tests that depend on a third-party service or even test code that makes Ajax calls back to your own application can vary from test run to test run, causing intermittent failures.

Dates and times tend to lead to intermittent failures when certain magic time boundaries are crossed. You can also get tests that fail at particular times of day or when run in certain time zones. Random numbers, in contrast, make it somewhat difficult to test both the randomness of the number and that the randomly generated number is used properly in whatever calculation requires it.

The test plan is similar for dates, randomness, and external services—really, it applies to any constantly changing dataset. We test changing data with a combination of encapsulation and mocking. We encapsulate the data by creating a service object that wraps around the changing functionality. By mediating access to the changing functionality, we make it easier to stub or mock the output values. Stubbing the values provides the consistency we need for testing.

We might, for example, create a `RandomStream` class that wraps Ruby's `rand()` method:

```
class RandomStream
  def next
    rand()
  end
end
```

This example is a little oversimplified—normally we'd be encapsulating `RandomStream`. With your own wrapper class, you can provide more specific

methods tuned to your use case, something like def random_phone_number. First you unit-test the stream class to verify that the class works as expected. Then any class that uses RandomStream can be provided mock random values to allow for easier and more stable testing.

The exact mix of encapsulation and mocking varies. Timecop is a Ruby gem that stubs the time and date classes with no encapsulation—in Rails 4.1, ActiveSupport has a similar feature. This allows us to specify an exact value for the current time for testing purposes. That said, nearly every time I talk about Timecop in a public forum, someone points out that creating a time service is a superior solution.

We'll discuss this pattern for wrapping a potentially variable external service in more detail in Chapter 12, *Testing External Services*, on page ?. We'll cover mock objects in Chapter 7, *Using Test Doubles as Mocks and Stubs*, on page ?, and we'll talk more about debugging intermittent test failures in Chapter 14, *Troubleshooting and Debugging*, on page ?.

[I]ndependent

A test is *independent* if it does not depend on any other tests or external data to run. An independent test suite gives the same results no matter the order in which the tests are run, and tends to limit the scope of test failures to only tests that cover a buggy method.

In contrast, a very dependent test suite could trigger failures throughout your tests from a single change in one part of an application. A clear sign that your tests are not independent is if you have test failures that happen only when the test suite is run in a particular order—in fully independent tests, the order in which they are run should not matter. Another sign is a single line of code breaking multiple tests.

The biggest impediment to independence in the test suite itself is the use of global data. Rails fixtures are not the only possible cause of global data in a Rails test suite, but they are a common cause. Somewhat less common in a Rails context is using a tool or third-party library in a setup and not tearing it down.

Outside the test suite, if the application code is not well encapsulated it may be difficult or impossible to make the tests fully independent of one another.

[F]ast

It's easy to overlook the importance of pure speed in the long-term maintenance of a test suite or a TDD practice. In the beginning it doesn't make much

difference. When you have only a few methods under test, the difference between a second per test and a tenth of a second per test is almost imperceptible. The difference between a one-minute suite and a six-second suite is easier to discern.

From there, the sky's the limit. I worked in one Rails shop where nobody really knew how long the tests ran in development because they farmed the test suite out to a server farm that was more powerful than most production web servers I've seen. This is bad.

Slow test suites hurt you in a variety of ways.

There are startup costs. In the sample TDD session we went through in Chapter 2, *Test-Driven Development Basics*, on page ?, and Chapter 3, *Test-Driven Rails*, on page ?, we went back and forth to run the tests a lot. In practice I went back and forth even more frequently. Over the course of writing that tutorial, I ran the tests dozens of times. Imagine what happens if it takes even 10 seconds to start a test run. Or a minute, which is not out of the question for a larger Rails app. I've worked on JRuby-based applications that took well over a minute to start.

TDD is about flow in the moment, and the ability to go back and forth between running tests and writing code without breaking focus is crucial to being able to use TDD as a design tool. If you can check Twitter while your tests are running, you just aren't going to get the full value of the TDD process.

Tests get slow for a number of reasons, but the most important in a Rails context are as follows:

- Startup time
- Dependencies within the code that require a lot of objects to be created to invoke the method under test
- Extensive use of the database or other external services during a test

Not only do large object trees slow down the test at runtime, but setting up large amounts of data makes writing the tests more labor-intensive. And if writing the tests becomes burdensome, you aren't going to do it.

Speeding tests up often means isolating application logic from the Rails stack so that logic can be tested without loading the entire Rails stack or without retrieving test data from the database. As with a lot of good testing practices, this isolation results in more robust code that is easier to change moving forward.

Since test speed is so important for successful TDD, throughout the book we'll discuss ways to write fast tests. In particular, the discussion of creating data in Chapter 6, *Adding Data to Tests*, on page ?, and the discussion of testing environments in Chapter 15, *Running Tests Faster and Running Faster Tests*, on page ?, will be concerned with creating fast tests.

[T]ruthful

A *truthful* test accurately reflects the underlying code—it passes when the underlying code works, and fails when it does not. This is easier said than done.

A frequent cause of brittle tests is targeting assertions at surface features that might change even if the underlying logic stays the same. The classic example along these lines is view testing, in which we base the assertion on the creative text on the page (which will frequently change even though the basic logic stays the same):

```
it "shows the project section" do
  get :dashboard
  expect(response).to have_selector("h2", :text => "My Projects")
end
```

It seems like a perfectly valid test right up until somebody determines that "My Projects" is a lame header and decides to go with "My Happy Fun-Time Projects," breaking our test. You are often better served by testing something that's slightly insulated from surface changes, such as a DOM ID.

```
it "shows the project section" do
  get :dashboard
  expect(response).to have_selector("h2#projects")
end
```

The basic issue here is not limited to view testing. There are areas of model testing in which testing to a surface feature might be brittle in the face of trivial changes to the model (as opposed to tests that are brittle in the face of changes to the test data itself, which we've already discussed).

The other side of robustness is not just a test that fails when the logic is good, but a test that stubbornly continues to pass even if the underlying code is bad—a tautology, in other words.

Speaking of tautologies, mock objects have their own special robustness issues. It's easy to create a tautology by using a mock object. It's also easy to create a brittle test because a mock object often creates a hard expectation of what methods will be called on it. If you add an unexpected method call to

the code being tested, you can get mock-object failures simply because an unexpected method has been called. I've had changes to a login filter cause hundreds of test failures because mock users going through the login filter bounced off the new call.