Extracted from:

Rails 5 Test Prescriptions

Build a Healthy Codebase

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Rails 5 Test Prescriptions

Build a Healthy Codebase

Noel Rappin

Edited by Katharine Dvorak

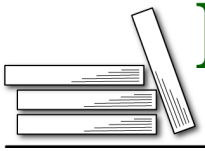
Rails 5 Test Prescriptions

Build a Healthy Codebase

Noel Rappin

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*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Brian MacDonald
Supervising Editor: Jacquelyn Carter
Development Editor: Katharine Dvorak
Indexing: Potomac Indexing, LLC
Copy Editor: Candace Cunningham
Layout: Gilson Graphics

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: 978-1-68050-250-3
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—February 2018

A standard Rails application uses a design pattern called *MVC*, which stands for model-view-controller. Each of the three sections in the MVC pattern is a separate layer of code, which has its own responsibilities and communicates with the other layers as infrequently as possible. In Rails, the model layer contains both business logic and persistence logic, with the persistence logic being handled by ActiveRecord. Typically, all of your ActiveRecord objects will be part of the model layer, but not everything in the model layer is necessarily an ActiveRecord object. The model layer can include various services, value objects, or other classes that encapsulate logic and use ActiveRecord objects for storage.

I'll start the tour of testing the Rails stack with the model layer because model tests have the fewest dependencies on Rails-specific features and are often the easiest place to start testing your application. Standard Rails model tests are very nearly vanilla RSpec. Features specific to Rails include a few new matchers and the ability to set up initial data.

I'll also talk about testing ActiveRecord features such as associations and models. And I'll talk about separating logic from persistence and why that can be a valuable practice for both testing and application development.

What Can You Do in a Model Test?

An RSpec file in the `spec/models` directory is automatically of `type: :model`, which gives you … not a whole lot of new behavior, actually. An add-on gem called rspec-activemodel-mocks,[1] which is maintained by the RSpec core team, includes some mock-object tools specific to use with ActiveModel.

What Should You Test in a Model Test?

Models. Next question?

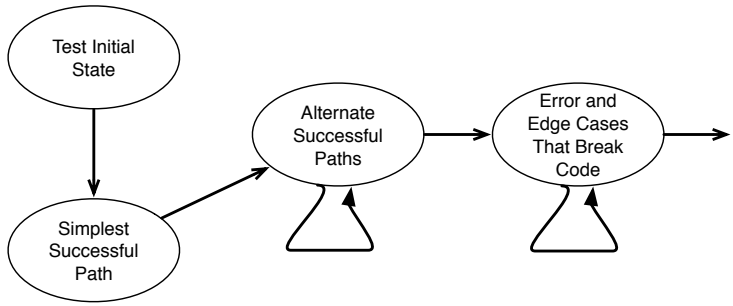
Okay, Funny Man, What Makes a Good Set of Model Tests?

There are a lot of different, sometimes conflicting, goals for tests. It's hard to know where to start your TDD process, how many tests to write, and when you're done. The classic description of the process—"write a simple test, make it pass, then refactor"—doesn't provide a lot of affirmative guidance or direction for the larger process of adding a feature.

1. https://github.com/rspec/rspec-activemodel-mocks

A TDD Metaprocess

The following diagram illustrates a metaprocess that reflects how I write a new business-logic feature. (I handle view logic a little differently.) This process is more of a guideline than a hard-and-fast checklist. As the logic gets more complex, and the less I know about the implementation when I start, the closer I stick to this process and the smaller the steps I take.



Often the best place to start is with a test that describes an initial state of the system, without invoking logic. This is especially useful if you're test-driving a new class, so the first test just sets up the class and verifies the initial state of instance variables. If you're working in an existing class, this step may not be necessary.

Next, determine the main cases that drive the new logic. Sometimes there will be only one primary case, like "calculate the total value of a user's purchase." Sometimes there will be many; "calculate tax on a purchase" might have lots of cases based on the user's location or the items being purchased.

Take that list and write tests for the main cases one at a time. I do not recommend writing multiple failing tests simultaneously—that turns out to be confusing. It can be helpful to use comments or pending tests to at least note what the future tests will be. Ideally these tests are small—if they need a lot of setup, you probably should be testing a smaller unit of code.

As you saw in the first few chapters, sometimes you'll pass the first test by putting in a solution that is deliberately specific to the test, like a one-line method that only returns the exact value that makes the test pass. Gary Bernhardt of Destroy All Software calls this technique *sliming*.[2] This can be helpful in keeping your code close to the tests—again, especially when the algorithm is complex.

2. http://www.destroyallsoftware.com

> **Prescription 12** If you find yourself writing tests that already pass given the current state of the code, that often means you're writing too much code in each pass.

The goal in these tests is to first make the test pass quickly without worrying much about niceties of implementation. Once each test passes, look for opportunities to refactor. (I'll talk more about that in the next section.) When the main cases are done, you try to think of ways to break the existing code. Sometimes you'll notice something as you're writing code to pass a previous test, like, "Hey, I wonder what would happen if this argument were nil?" Write a test that describes what the output should be and make it pass. Refactoring gets increasingly important here because special cases and error conditions tend to make code complex, and managing that complexity becomes really important to future versions of the code. The advantage of waiting to do special cases at the end is that you already have tests to cover the normal cases, so you can use those to check your new code each step of the way.

When you can no longer think of a way to break your code, you're likely done with this feature and ready to move on to the next. If you haven't been doing so, run the entire test suite to make sure you didn't inadvertently break something else. Then take one further look at the code for larger-scale refactoring.

Refactoring Models

In a TDD process, much of the design takes place during the refactoring step. A lot of this design happens under the guise of cleanup—looking at parts of the code that seem overly complicated or poorly structured and figuring out how best to rearrange them.

Just because the refactoring step includes cleanup doesn't mean you can skip this step when you're in a hurry. Don't do that. Refactoring is not a luxury. Refactoring is where you think about your code and how best to structure it. Skipping refactoring will slowly start to hurt, and by the time you notice the problem, it'll be much harder to clean up than if you had addressed it early.

> **Prescription 13** Refactoring is where a lot of design happens in TDD, and it's easiest to do in small steps. Skip it at your peril.

At the most abstract level, you're looking for three things: complexity to break up, duplication to combine, and abstractions waiting to be born.

Break Up Complexity

Complexity will often initially manifest itself as long methods or long lines of code. If you're doing a "quick to green" first pass of the code to make the tests pass, there will often be a line of code with lots of method chaining or the like. It's almost always a good idea to break up long methods or long lines by extracting part of the code into its own method. In addition to simplifying the original method, you have the opportunity to give the method you're creating a name that is meaningful in the domain and that can make your code self-documenting.

Booleans, local variables, and inline comments are almost always candidates for extraction:

- Any compound Boolean logic expression goes in its own method. It's much harder than you might think for people to understand what a compound expression does. Give people reading your code a fighting chance by hiding the expression behind a name that expresses the code's intent, such as `valid_name?` or `has_purchased_before?`.
- Local variables are relatively easy to break out into methods with the same name as the variable—in Ruby, code that uses the variable doesn't need to change if the variable becomes a method with no arguments. Having a lot of local variables is a huge drag on complex refactorings. You'll be surprised at how much more flexible your code feels if you minimize the number of local variables in methods. (I first encountered this idea, along with a lot of other great refactoring ideas, in *Refactoring: Improving the Design of Existing Code [FBBO99]*.)
- In long methods, sometimes a single-line comment breaks up the method by describing what the next part does. This nearly always is better extracted to a separate method with a name based on the comment's contents. Instead of one twenty-five-line method, you wind up with a five-line method that calls five other five-line methods, each of which does one thing and each of which has a name that is meaningful in the context of the application domain.

Prescription 14 Try to extract methods when you see compound Booleans, local variables, or inline comments.

Combine Duplication

You need to look out for three kinds of duplication: duplication of fact, duplication of logic, and duplication of structure.

Duplication of fact is usually easy to see and fix. A common case would be a "magic number" used by multiple parts of the code, such as a conversion factor or a maximum value. You saw this in an earlier example with the 21 days that you used to calculate velocity. Another common example is a status variable that has only a few valid values, and the list of those values is duplicated:

```
validates :size, numericality: {less_than_or_equal_to: 5}

def possible_sizes
  (1 .. 5)
end
```

The remedy for duplication of fact is also usually simple—make the value a constant or a method with a constant return value, like this:

```
MAX_POINT_COUNT = 5

validates :size, numericality: {less_than_or_equal_to: MAX_POINT_COUNT}

def possible_sizes
  (1 .. MAX_POINT_COUNT)
 end
```

Or, alternatively, like this:

```
VALID_POINT_RANGE = 1 .. 5
validates :size, inclusion: {in: VALID_POINT_RANGE}
```

The key metric is how many places in the code would need to change if the underlying facts change, with the ideal number being 1. That said, at some point the extra character count for a constant is ridiculous and Java-like in the worst way. For string and symbol constants, if the constant value is effectively identical to the symbol (as in `ACTIVE_STATUS = :active`), I'll often leave the duplication. I'm not saying I recommend that; I'm just saying I do it.

I also often make the constant value an instance method with a static return value rather than a Ruby constant, like this:

```
def max_point_count
  5
end
```

I do this because then `max_point_count` has the same lookup semantics as any other instance value, and it often reads better to make a value owned by the

instance rather than the class. It's also easier to change if the constant turns out to be less than constant in the future.

Duplication of logic is similar to duplication of fact, but instead of looking for simple values, you're looking for longer structures. This often includes compound Boolean statements used as guards in multiple methods (in the task-manager example, this might be something about whether a task has been completed) or simple calculations that are used in multiple places (such as converting task size to time based on the project's rate of completion).

In this example, the same Boolean test is applied twice and it's easy to imagine it being used many more times:

```
class User
  def maximum_posts
    if status == :trusted then 10 else 5 end
  end

  def urls_in_replies
    if status == :trusted then 3 else 0 end
  end
end
```

One way to mitigate this is to move the duplicated logic into its own method and call the method from each location (in this case, def trusted?). I recommend being aggressive about this—you'll sometimes see advice that you should just notice duplication on the second instance and refactor on the third instance. In my experience, that just means you wind up with twice as much duplication as you should have. (See the next section for other ideas about reused Booleans.)

Keep in mind that not every piece of logic that is spelled the same in Ruby is actually duplication. It's possible for early forms of two pieces of logic to look similar but eventually evolve in separate directions. A great example of this is Rails controller scaffolding. Every RESTful controller starts with the same boilerplate code for the seven RESTful actions. And there have been innumerable attempts to remove that duplication by creating a common abstraction. Most of those attempts eventually wind up in a tangle of special case logic because each controller eventually needs to have different features.