Extracted from:

Effective Testing with RSpec 3

Build Ruby Apps with Confidence

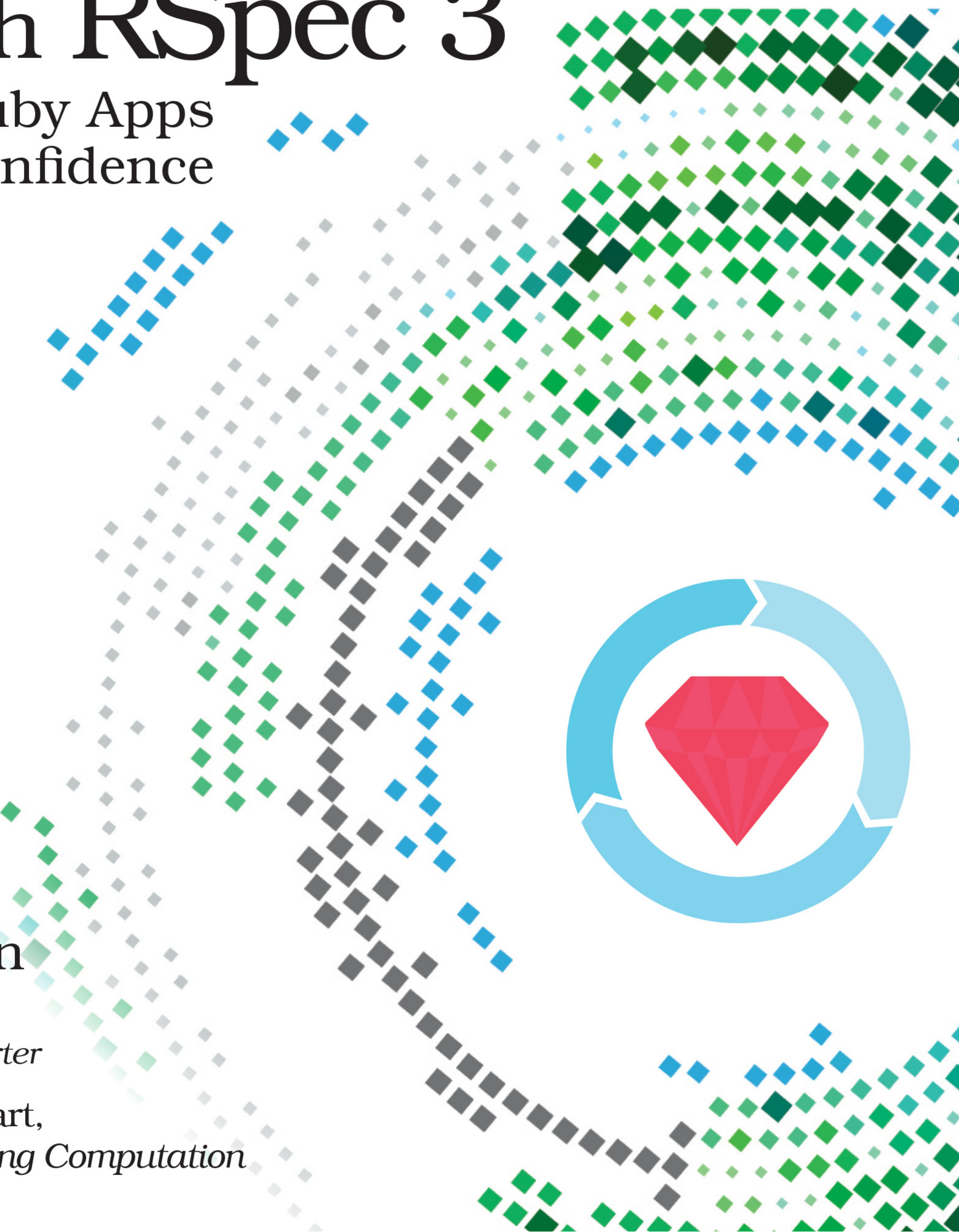
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Effective Testing with RSpec 3

Build Ruby Apps with Confidence



Myron Marston
and Ian Dees
edited by Jacquelyn Carter

Foreword by Tom Stuart,
author of *Understanding Computation*

Effective Testing with RSpec 3

Build Ruby Apps with Confidence

Myron Marston
Ian Dees

The Pragmatic Bookshelf
Raleigh, North Carolina

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Executive Editor: Susannah Davidson Pfalzer
Development Editor: Jacquelyn Carter
Indexing: Potomac Indexing, LLC
Copy Editor: Liz Welch
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: 978-1-68050-198-8
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—August 2017

You've installed RSpec and taken it for a test drive. You've written a few specs and gotten a feel for how they're different from test cases in traditional frameworks. You've also seen a few ways to trim repetition from your examples.

In the process, you've applied the following practices:

- Structuring your examples logically into groups
- Writing clear expectations that test at the right level of detail
- Sharing common setup code across specs

RSpec is designed around these habits, but you could learn to apply them to other test frameworks as well. You may be wondering if all that separates RSpec from the crowd is syntax.

In this chapter, we're going to show you that RSpec's usefulness isn't confined to how your specs *look*. It also applies to how they *run*. You're going to learn the following practices that will help you find problems in code more quickly:

- See your specs' output printed as documentation, to help your future self understand the intent of the code when something goes wrong

- Run a specific set of examples, to focus on one slice of your program at a time

- Fix a bug and rerun just the specs that failed last time

- Mark work in progress to remind you to finish something later

The tool that makes these activities possible—and even easy—is RSpec's *spec runner*. It decides which of your specs to run and when to run them. Let's take a look at how to make it sing.

Customizing Your Specs' Output

When you use RSpec on a real-world project, you'll build up a suite of dozens, hundreds, or even thousands of examples. Most test frameworks, including RSpec, are optimized for this kind of use. The default output format hides a lot of detail so that it can show your specs' progress.

The Progress Formatter

In this section, we're going to look at a different ways to view your specs' output. Create a new file called spec/coffee_spec.rb with the following contents:

02-running-specs/01/spec/coffee_spec.rb

```ruby
RSpec.describe 'A cup of coffee' do
  let(:coffee) { Coffee.new }

  it 'costs $1' do
```

```
    expect(coffee.price).to eq(1.00)
  end

➤ context 'with milk' do
    before { coffee.add :milk }

    it 'costs $1.25' do
      expect(coffee.price).to eq(1.25)
    end
  end
end
```

This spec file uses the same techniques we saw in the previous chapter, with one new twist: the context block starting on the highlighted line. This method groups a set of examples and their setup code together with a common description—in this case "with milk." You can nest these example groups as deeply as you want.

There's nothing mysterious going on behind the scenes here: context is just an alias for describe. You could use them interchangeably, but we tend to use context for phrases that modify the object we're testing, the way "with milk" modifies "A cup of coffee."

This spec will need a Coffee class to test. In a full project, you'd put its definition in a separate file and use require in your specs. But for this simple example, it's fine just to put the class at the top of your spec file. Here's the start of an implementation that's not quite enough to pass the specs yet:

02-running-specs/01/spec/coffee_spec.rb

```ruby
class Coffee
  def ingredients
    @ingredients ||= []
  end

  def add(ingredient)
    ingredients << ingredient
  end

  def price
    1.00
  end
end
```

When you run your specs, you'll see one dot for each completed example, with failures and exceptions called out with letters:

```
$ rspec
.F

Failures:

  1) A cup of coffee with milk costs $1.25
     Failure/Error: expect(coffee.price).to eq(1.25)
```

```
       expected: 1.25
            got: 1.0

       (compared using ==)
     # ./spec/coffee_spec.rb:26:in `block (3 levels) in <top (required)>'

Finished in 0.01222 seconds (files took 0.08094 seconds to load)
2 examples, 1 failure

Failed examples:

rspec ./spec/coffee_spec.rb:25 # A cup of coffee with milk costs $1.25
```

Here, we see one dot for the passing example, and one F for the failure. This format is good for showing the progress of your specs as they execute. When you've got hundreds of examples, you'll see a row of dots marching across the screen.

On the other hand, this output doesn't give any indication of which example is currently running, or what the expected behavior is.

When you need more detail in your test report, or need a specific format such as HTML, RSpec's got you covered. By choosing a different *formatter*, you can tailor the output to your needs.

A formatter receives events from RSpec—such as when a test fails—and then reports the results. Under the hood, it's just a plain Ruby object. You can easily create your own, and in *How Formatters Work*, on page ? you'll see how to do that. Formatters can write data in any format, and send the output anywhere (such as to the console, a file, or over a network). Let's take a look at another one of the formatters that ships with RSpec.

The Documentation Formatter

RSpec's built-in *documentation formatter* lists the specs' output in an outline format, using indentation to show grouping. If you've written example descriptions with legible output in mind, the result will read almost like project documentation. Let's give it a try.

To see the output in documentation format, pass --format documentation (or just -f d) to rspec:

```
$ rspec --format documentation

A cup of coffee
  costs $1
  with milk
    costs $1.25 (FAILED - 1)

Failures:
  1) A cup of coffee with milk costs $1.25
     Failure/Error: expect(coffee.price).to eq(1.25)

       expected: 1.25
            got: 1.0

       (compared using ==)
     # ./spec/coffee_spec.rb:26:in `block (3 levels) in <top (required)>'

Finished in 0.01073 seconds (files took 0.08736 seconds to load)
2 examples, 1 failure

Failed examples:

rspec ./spec/coffee_spec.rb:25 # A cup of coffee with milk costs $1.25
```

The test report is a list of the specifications of various cups of coffee that RSpec verified. There's a lot of information here, and RSpec uses spacing and capitalization to show you what's going on:

- An example group lists all of its examples indented underneath it.
- Contexts create additional nesting, the way the with milk example is indented further.
- Any failing examples show the text FAILED with a footnote number for looking up the details later on.

After the documentation at the top of the report, the Failures section shows the following details for each failure:

- The expectation that failed
- What result you expected versus what actually happened
- The file and line number of the failing expectation

This output is designed to help you find at a glance what went wrong and how. As we'll see next, RSpec can provide further cues through syntax highlighting.

Syntax Highlighting

We've seen how RSpec's color highlighting makes it *much* easier to scan the output for passing and failing specs. We can take it a step further by installing a code highlighter called CodeRay:[1]

```
$ gem install coderay -v 1.1.1
Successfully installed coderay-1.1.1
1 gem installed
```

1. https://github.com/rubychan/coderay

When this gem is installed, the Ruby snippets in your specs' output will be color-coded just like they'd be in your text editor. For example:

```
$ rspec -fd

A cup of coffee
  costs $1
  with milk
    costs $1.25 (FAILED - 1)

Failures:

  1) A cup of coffee with milk costs $1.25
     Failure/Error: expect(coffee.price).to eq(1.25)

       expected: 1.25
            got: 1.0

       (compared using ==)
     # ./spec/coffee_spec.rb:26:in `block (3 levels) in <top (required)>'

Finished in 0.0102 seconds (files took 0.09104 seconds to load)
2 examples, 1 failure

Failed examples:

rspec ./spec/coffee_spec.rb:25 # A cup of coffee with milk costs $1.25
```

Now, the line expect(coffee.price).to eq(1.25) has Ruby syntax highlighting. Normal method calls like coffee and price aren't shaded, but other elements are. In particular, both the key RSpec expect method and the number 1.25 are highlighted in color. This syntax highlighting is even more helpful for complex Ruby expressions.

RSpec will automatically use CodeRay if it's available. For Bundler-based projects, drop it into your Gemfile and rerun bundle install. For non-Bundler projects, install it via gem install as we've done here.

Identifying Slow Examples

Throughout this book, we're going to give you advice on how to keep your specs running quickly. To understand where the biggest bottlenecks are in your suite, you need to be able to identify the slowest examples.

RSpec's spec runner can help you do so. Consider the following group of examples that take too long to run:

02-running-specs/03/spec/slow_spec.rb

```
RSpec.describe 'The sleep() method' do
  it('can sleep for 0.1 second') { sleep 0.1 }
  it('can sleep for 0.2 second') { sleep 0.2 }
  it('can sleep for 0.3 second') { sleep 0.3 }
  it('can sleep for 0.4 second') { sleep 0.4 }
  it('can sleep for 0.5 second') { sleep 0.5 }
end
```

We can ask RSpec to list the top time-wasters by passing the --profile option along with the number of offenders we'd like to see:

```
$ rspec --profile 2
.....

Top 2 slowest examples (0.90618 seconds, 59.9% of total time):
  The sleep() method can sleep for 0.5 second
    0.50118 seconds ./spec/slow_spec.rb:6
  The sleep() method can sleep for 0.4 second
    0.40501 seconds ./spec/slow_spec.rb:5

Finished in 1.51 seconds (files took 0.08911 seconds to load)
5 examples, 0 failures
```

Just two examples are taking over half our test time. Better get optimizing!

Running Just What You Need

In the examples in this chapter, we've always run all the specs together. On a real project, you don't necessarily want to load your entire test suite every time you invoke RSpec.

If you're diagnosing a specific failure, for instance, you'll want to run just that one example. If you're trying to get rapid feedback on your design, you can bypass slow or unrelated specs.

The easiest way to narrow down your test run is to pass a list of file or directory names to rspec:

```
$ rspec spec/unit                  # Load *_spec.rb in this dir and subdirs
$ rspec spec/unit/specific_spec.rb # Load just one spec file
$ rspec spec/unit spec/smoke       # Load more than one directory
$ rspec spec/unit spec/foo_spec.rb # Or mix and match files and directories
```

Not only can you load specific files or directories, you can also filter which of the loaded examples RSpec will actually run. Here, we'll explore a few different ways to run specific examples.

Running Examples by Name

Rather than running all the loaded specs, you can choose a specific example by name, using the --example or -e option plus a search term:

```
$ rspec -e milk -fd
Run options: include {:full_description=>/milk/}

A cup of coffee
```

```
  with milk
    costs $1.25 (FAILED - 1)

Failures:

  1) A cup of coffee with milk costs $1.25
     Failure/Error: expect(coffee.price).to eq(1.25)

       expected: 1.25
            got: 1.0

       (compared using ==)
     # ./spec/coffee_spec.rb:26:in `block (3 levels) in <top (required)>'

Finished in 0.01014 seconds (files took 0.08249 seconds to load)
1 example, 1 failure

Failed examples:

rspec ./spec/coffee_spec.rb:25 # A cup of coffee with milk costs $1.25
```

RSpec ran just the examples containing the word *milk* (in this case, just one example). When you use this option, RSpec searches the full description of each example; for instance, A cup of coffee with milk costs $1.25. These searches are case-sensitive.

Running Specific Failures

Often, what you really want to do is run just the most recent failing spec. RSpec gives us a handy shortcut here. If you pass a filename and line number separated by a colon, RSpec will run the example that starts on that line.

You don't even have to manually type in which file and line to rerun. Take a look at the end of the spec output:

```
$ rspec
.F

« truncated »

2 examples, 1 failure

Failed examples:

rspec ./spec/coffee_spec.rb:25 # A cup of coffee with milk costs $1.25
```

You can copy and paste the first part of that final line (before the hash) into your terminal to run just the failing spec. Let's do so now:

```
$ rspec ./spec/coffee_spec.rb:25
Run options: include {:locations=>{"./spec/coffee_spec.rb"=>[25]}}
F

« truncated »

1 example, 1 failure

Failed examples:

rspec ./spec/coffee_spec.rb:25 # A cup of coffee with milk costs $1.25
```

RSpec ran only the single example you specified. This focusing ability becomes even more powerful when you add a key binding for it to your text editor. Several IDEs and editor plugins provide this behavior for you, including the following:

- ThoughtBot's rspec.vim plugin[2]
- Peter Williams's RSpec Mode for Emacs[3]
- The RSpec package for Sublime Text[4]
- Felipe Coury's Atom RSpec Runner[5]
- The RubyMine IDE from JetBrains[6]

With good editor support, you can quickly run the example under your cursor with a single keystroke.

Use Editor Integration for a More Productive Experience

Having to switch back and forth between your editor and a terminal window in order to run rspec really interrupts your workflow. We recommend taking the time to install an editor plugin so that running rspec is only a keystroke away.

2. https://github.com/thoughtbot/vim-rspec
3. https://www.emacswiki.org/emacs/RspecMode
4. https://github.com/SublimeText/RSpec
5. https://github.com/fcoury/atom-rspec
6. https://www.jetbrains.com/ruby/

Rerunning Everything That Failed

Using a line number works well when only one spec is failing. If you have more than one failure, you can run all of them with the --only-failures flag. This flag requires a little bit of configuration, but RSpec will coach you through the setup process:

```
$ rspec --only-failures

To use `--only-failures`, you must first set                              ↩
`config.example_status_persistence_file_path`.
```

RSpec needs a place to store information about which examples are failing so that it knows what to rerun. You supply a filename through the RSpec.configure method, which is a catch-all for lots of different runtime options.

Add the following lines to your coffee_spec.rb file between the Coffee class definition and the specs:

02-running-specs/06/spec/coffee_spec.rb

```
RSpec.configure do |config|
  config.example_status_persistence_file_path = 'spec/examples.txt'
end
```

You'll need to rerun RSpec once without any flags (to record passing/failing status):

```
$ rspec
.F

« truncated »

2 examples, 1 failure

Failed examples:

rspec ./spec/coffee_spec.rb:29 # A cup of coffee with milk costs $1.25
```

Now, you can use the --only-failures option:

```
$ rspec --only-failures
Run options: include {:last_run_status=>"failed"}
F

« truncated »

1 example, 1 failure

Failed examples:

rspec ./spec/coffee_spec.rb:29 # A cup of coffee with milk costs $1.25
```

Let's see what happens when the behavior gets fixed and the specs pass. Take a swing at modifying the Coffee class to pass both examples. Here's one possible implementation:

02-running-specs/06/spec/coffee_spec.rb

```
class Coffee
  def ingredients
    @ingredients ||= []
  end

  def add(ingredient)
    ingredients << ingredient
  end

  def price
    1.00 + ingredients.size * 0.25
  end
end
```

With your implementation in place, rerun RSpec with the --only-failures option:

```
$ rspec --only-failures
Run options: include {:last_run_status=>"failed"}
.

Finished in 0.00094 seconds (files took 0.09055 seconds to load)
1 example, 0 failures
```

RSpec reruns the formerly failing example and verifies that it passes. If we try this process once more, RSpec won't have any failing examples left to run:

```
$ rspec --only-failures
Run options: include {:last_run_status=>"failed"}

All examples were filtered out

Finished in 0.00031 seconds (files took 0.08117 seconds to load)
0 examples, 0 failures
```

Another command-line option, --next-failure, offers a twist on this idea. You'll get a chance to try it out in the exercise at the end of this chapter.

Passing options to the rspec command isn't the only way to run just a subset of your examples. Sometimes, it's more convenient to make temporary annotations to your specs instead.