Extracted from:

The RSpec Book

Behaviour-Driven Development with RSpec, Cucumber, and Friends

The Pragmatic Programmers

The RSpec Book

Behaviour-Driven Development with RSpec, Cucumber, and Friends

David Chelimsky
with Dave Astels, Zach Dennis, Aslak Hellesøy, Bryan Helmkamp, and Dan North

Foreword by Robert C. Martin (Uncle Bob)

Edited by Jacquelyn Carter

The Facets of Ruby Series

Chapter 5

Describing Code with RSpec

In the previous chapter, we introduced and used Cucumber to describe the behavior of our Codebreaker game from the outside, at the application level. We wrote step definitions for our first Cucumber feature that will handle the steps in the scenario, and we left off with a failing step: we're expecting Game to send a message to our fake Output, but its array of messages is empty.

In this chapter, we're going to use RSpec to *describe* behavior at a much more granular level: the expected behavior of instances of the Game class.

5.1 Getting Started with RSpec

To get going, create a spec directory, with a subdirectory named codebreaker. Now create a file named game_spec.rb in spec/codebreaker/. As we progress, we'll maintain a parallel structure like this in which each source file (for example, lib/codebreaker/game.rb) has a parallel spec file (for example, spec/codebreaker/game_spec.rb). See the *Joe Asks...* on the next page for more on this. Add the following to game_spec.rb:

Download cb/08/spec/codebreaker/game_spec.rb

```
Line 1  require 'spec_helper'
    2
    3  module Codebreaker
    4    describe Game do
    5      describe "#start" do
    6        it "sends a welcome message"
    7        it "prompts for the first guess"
    8      end
    9    end
   10  end
```

Joe Asks...

Shouldn't We Avoid a One-to-One Mapping?

Perhaps you've heard that a one-to-one mapping between objects and their specs is a BDD no-no. There is some truth to this, but the devil is in the details.

We want to avoid a strict adherence to a structure in which every object has a single example group and every method has a single code example. That sort of structure leads to long examples that take an object through many phases, setting expectations at several stopping points in each example. Examples like these are difficult to write to begin with and much more difficult to understand and debug later.

A one-to-one mapping of spec-file to application-code-file, however, is not only perfectly fine but actually beneficial. It makes it easier to understand where to find the specs for code you might be looking at. It also makes it easier for tools to automate shortcuts like the one in the RSpec TextMate bundle, which switches between spec-file and application-code-file with Ctrl+Shift+Down.

The first two statements are standard Ruby. We require a file named spec_helper.rb on line 1. We'll actually store that file in the spec directory, which RSpec adds to the global $LOAD_PATH. More on that in a minute.

The second statement declares a Ruby module named Codebreaker. This isn't necessary in order to run the specs, but it provides some conveniences. For example, we don't have to fully qualify Game on line 4.

The describe() method hooks into RSpec's API and returns a subclass of RSpec::Core::ExampleGroup. As its name suggests, this is a group of examples of the expected behavior of an object. If you're accustomed to xUnit tools like Test::Unit, you can think of an ExampleGroup as being akin to a TestCase.

The it() method creates an *example*. Technically, it's an instance of the ExampleGroup returned by describe(), but you really don't need to worry about that at this point. We'll get into the details of the underlying framework in Chapter 12, *Code Examples*, on page 152.

Connect the Specs to the Code

Before we can run this, we need to add the spec_helper.rb required on line 1. Create that now, and add the following:

Download cb/08/spec/spec_helper.rb

```
require 'codebreaker'
```

Similar to what we did with Cucumber's env.rb in the previous chapter, spec/codebreaker/game_spec.rb requires spec/spec_helper.rb, which requires lib/codebreaker.rb, which, in turn, requires lib/codebreaker/game.rb.

Open a shell and cd to the codebreaker project root directory, and run the game_spec.rb file with the rspec command,[1] like this:

```
rspec spec/codebreaker/game_spec.rb --format doc
```

You should see output similar to this:

```
Codebreaker::Game
  #start
    sends a welcome message (PENDING: Not Yet Implemented)
    prompts for the first guess (PENDING: Not Yet Implemented)

Pending:
  Codebreaker::Game#start sends a welcome message
    # Not Yet Implemented
    # ./spec/codebreaker/game_spec.rb:6
  Codebreaker::Game#start prompts for the first guess
    # Not Yet Implemented
    # ./spec/codebreaker/game_spec.rb:7
```

The --format doc option tells RSpec to format the output using the same nesting we see in the nested describe blocks in the file. We see Codebreaker::Game on the first line because we wrapped describe Game do inside the Codebreaker module.

The second line shows the string we passed to describe(), and the third and fourth lines show the strings we passed to it().

"PENDING: Not Yet Implemented" tells us that we have to implement those examples, which we do by passing a block to the it() method. Without the block, the example is considered pending.

1. The rspec command is installed when you install the rspec gem.

After RSpec outputs all the strings we passed to describe() and it(), it lists all the pending examples and their locations. This is followed by a summary that tells us how many examples were run, how many failed, and how many are pending.

5.2 Red: Start with a Failing Code Example

In game_spec.rb, we want to do what we've done in the feature: specify that when we start the game, it sends the right messages to the output. Start by modifying game_spec.rb as follows:

Download cb/09/spec/codebreaker/game_spec.rb

```
require 'spec_helper'

module Codebreaker
  describe Game do
    describe "#start" do
      it "sends a welcome message" do
        output = double('output')
        game = Game.new(output)

        output.should_receive(:puts).with('Welcome to Codebreaker!')

        game.start
      end

      it "prompts for the first guess"
    end
  end
end
```

Just as we did in the scenario, we want a test double to stand in for the real STDOUT. Instead of rolling our own as we did in the scenario, however, we're using RSpec's dynamic test double framework, RSpec::Mocks,[2] to create a dynamic test double on the first line of the example.

Next, we create a Game object, passing it the test double output we created on the previous line. These first two lines are the givens in this example.

The next line sets up a *message expectation*: an expectation that the output object should receive the puts message with the string "Welcome

2. See Chapter 14, *RSpec::Mocks*, on page 193 for more about RSpec::Mocks.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

The RSpec Book's Home Page
http://pragprog.com/titles/achbd
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/achbd.

Contact Us

Online Orders:	www.pragprog.com/catalog
Customer Service:	support@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com
Contact us:	1-800-699-PROG (+1 919 847 3884)