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 12

Code Examples

In this part of the book, we'll explore the details of RSpec's built-in expectations, mock objects framework, command-line tools, IDE integration, and extension points.

Our goal is to make Test-Driven Development a more joyful and productive experience with tools that elevate the design and documentation aspects of TDD to first-class citizenship. Here are some words you'll need to know as we reach for that goal:

subject code The code whose behavior we are specifying with RSpec.

expectation An expression of how the subject code is expected to behave. You'll read about state-based expectations in Chapter 13, *RSpec::Expectations*, on page 172, and you'll learn about interaction expectations in Chapter 14, *RSpec::Mocks*, on page 193.

code example An executable example of how the subject code can be used and its expected behavior (expressed with expectations) in a given context. In BDD, we write the code examples before the subject code they document.

The *example* terminology comes from Brian Marick, whose website is even named http://exampler.com. Using *example* instead of *test* reminds us that writing them is a design and documentation practice, even though once they are written and the code is developed against them, they become regression tests.

example group A group of code examples.

spec, aka **spec file** A file that contains one or more example groups.

Familiar Structure, New Nomenclature

If you already have some experience with Test::Unit or similar tools in other languages and/or TDD, the words we're using here map directly to words you're already familiar with:

- *Assertion* becomes *expectation*.
- *Test method* becomes *code example*
- *Test case* becomes *example group*

In addition to finding these new names used throughout this book, you'll find them in RSpec's code base as well.

In this chapter, you'll learn how to organize executable *code examples* in *example groups* in a number of different ways, run arbitrary bits of code before and after each example, and even share examples across groups.

12.1 Describe It!

RSpec provides a domain-specific language for specifying the behavior of objects. It embraces the metaphor of describing behavior the way we might express it if we were talking to a customer or another developer. A snippet of such a conversation might look like this:

You: *Describe a new account.*

Somebody else: *It should have a balance of zero.*

Here's that same conversation expressed in RSpec:

```
describe "A new Account" do
  it "should have a balance of 0" do
    account = Account.new
    account.balance.should == Money.new(0, :USD)
  end
end
```

We use the describe() method to define an example group. The string we pass to it represents the facet of the system that we want to describe (a new account). The block holds the code examples that make up that group.

The it() method defines a code example. The string passed to it describes the specific behavior we're interested in specifying about that facet

(should have a balance of zero). The block holds the example code that exercises the subject code and sets expectations about its behavior.

Using strings like this instead of legal Ruby class names and method names provides a lot of flexibility. Here's an example from RSpec's own code examples:

```
it "matches when actual < (expected + delta)" do
  be_close(5.0, 0.5).matches?(5.49).should be_true
end
```

This is an example of the behavior of code, so the intended audience is someone who can read code. With Test::Unit, we might name the method test_matches_when_value_is_less_than_target_plus_delta, which is pretty readable, but the ability to use nonalphanumeric characters makes the name of this example more expressive.

To get a better sense of how you can unleash this expressiveness, let's take a closer look at the describe() and it() methods.

The describe Method

The describe() method takes an arbitrary number of arguments and an optional block and returns a subclass of RSpec::Core::ExampleGroup. We typically use only one or two arguments, which represent the facet of behavior that we want to describe. They might describe an object, perhaps in a predefined state or perhaps a subset of the behavior we can expect from that object. Let's look at a few examples, with the output they produce so we can get an idea of how the arguments relate to each other:

```
describe "A User" { ... }
=> A User

describe User { ... }
=> User

describe User, "with no roles assigned" { ... }
=> User with no roles assigned

describe User, "should require password length between 5 and 40" { ... }
=> User should require password length between 5 and 40
```

The first argument can be either a reference to a class or module or a string. The second argument is optional and should be a string. Using the class/module for the first argument provides an interesting benefit: when we wrap ExampleGroup in a module, we'll see that module's name

in the output. For example, if User is in the Authentication module, we could do something like this:

```
module Authentication
  describe User, "with no roles assigned" do
```

The resulting report would look like this:

```
Authentication::User with no roles assigned
```

So, by wrapping the ExampleGroup in a module, we see the fully qualified name Authentication::User, followed by the contents of the second argument. Together, they form a descriptive string, and we get the fully qualified name for free. This is a nice way to help RSpec help *us* understand where things live as we're looking at the output.

You can also nest example groups, which can be a very nice way of expressing things in both input and output. For example, we can nest the input like this:

```
describe User do
  describe "with no roles assigned" do
    it "is not allowed to view protected content" do
```

This produces output like this:

```
User
  with no roles assigned
    is not allowed to view protected content
```

The context Method

The context() method is an alias for describe(), so they can be used interchangeably. We tend to use describe() for things and context() for context.

The User example, shown earlier, for example, could be written like this:

```
describe User do
  context "with no roles assigned" do
    it "is not allowed to view protected content" do
```

The output would be the same as when we used describe() on the second line, but context() can make it easier to scan a spec file and understand what relates to what.

What's It All About?

Similar to describe(), the it() method takes a single string, an optional hash, and an optional block. The string should be a sentence that,

when prefixed with "it," represents the detail that will be expressed in code within the block. Here's an example specifying a stack:

```
describe Stack do
  before(:each) do
    @stack = Stack.new
    @stack.push :item
  end

  describe "#peek" do
    it "should return the top element" do
      @stack.peek.should == :item
    end

    it "should not remove the top element" do
      @stack.peek
      @stack.size.should == 1
    end
  end

  describe "#pop" do
    it "should return the top element" do
      @stack.pop.should == :item
    end

    it "should remove the top element" do
      @stack.pop
      @stack.size.should == 0
    end
  end
end
```

This is also exploiting RSpec's nested example groups feature to group the examples of pop() separately from the examples of peek().

When run with the --format documentation command-line option, this would produce the following output:

```
Stack
  #peek
    should return the top element
    should not remove the top element
  #pop
    should return the top element
    should remove the top element


Finished in 0.00154 seconds
4 examples, 0 failures
```

Looks a bit like a specification, doesn't it? In fact, if we reword the example names without the word *should* in them, we can get output that looks even more like documentation:

```
Stack
  #peek
    returns the top element
    does not remove the top element
  #pop
    returns the top element
    removes the top element


Finished in 0.00157 seconds
4 examples, 0 failures
```

What? No *should*? Remember, the goal here is readable sentences. *Should* was the tool that Dan North used to get people writing sentences but is not itself essential to the goal.

The ability to pass free text to the it() method allows us to name and organize examples in meaningful ways. As with describe(), the string can even include punctuation. This is especially useful when we're dealing with code-level concepts in which symbols have important meaning that can help us understand the intent of the example.

12.2 Pending Examples

In *Test Driven Development: By Example* [Bec02], Kent Beck suggests keeping a list of tests that you have yet to write for the object you're working on, crossing items off the list as you get tests passing, and adding new tests to the list as you think of them.

With RSpec, you can do this right in the code by calling the it() method with no block. Let's say that we're in the middle of describing the behavior of a newspaper:

```
describe Newspaper do
  it "should be black" do
    Newspaper.new.colors.should include('black')
  end

  it "should be white" do
    Newspaper.new.colors.should include('white')
  end

  it "should be read all over"
end
```

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)