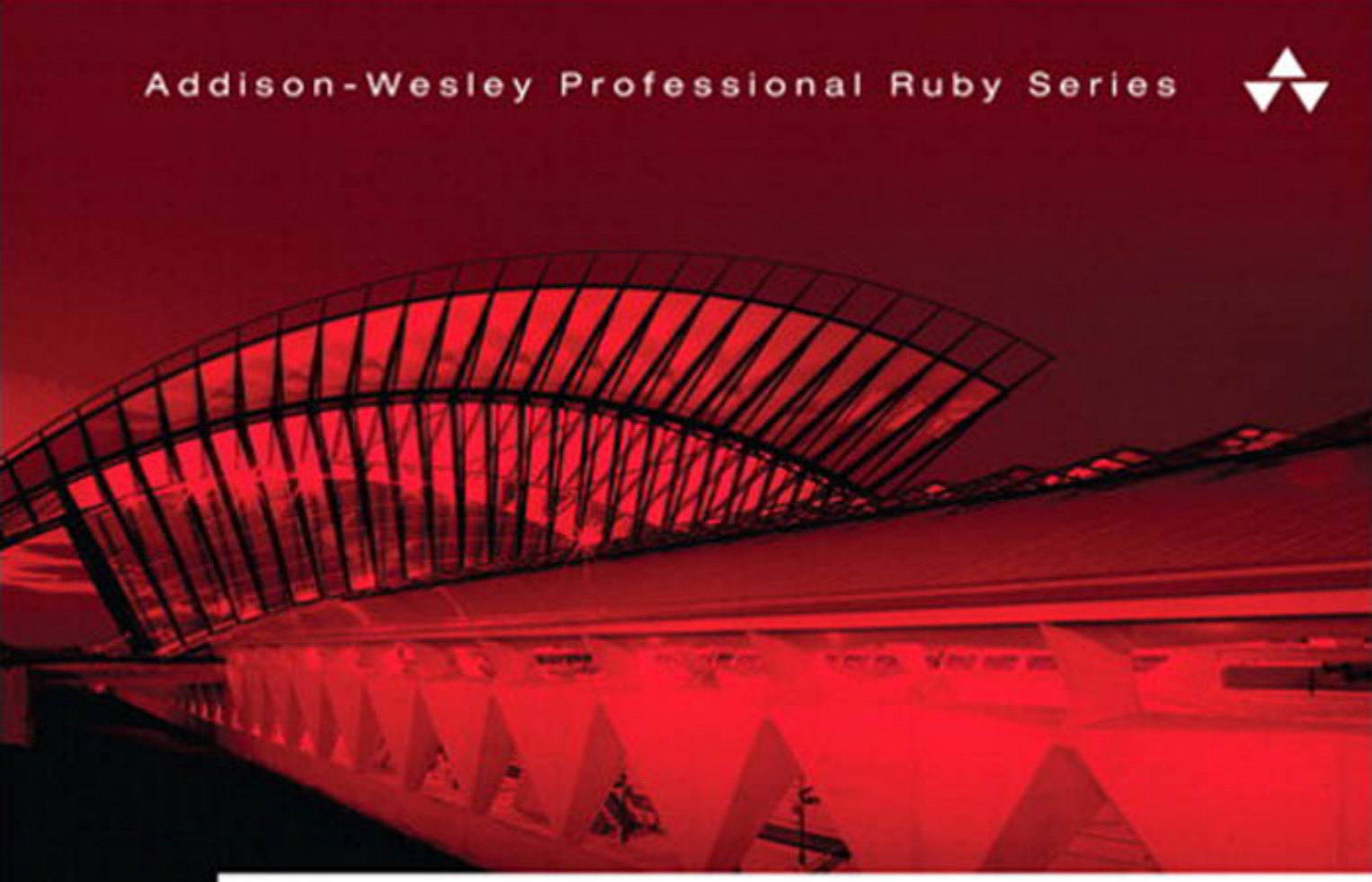
Addison-Wesley Professional Ruby Series



ELOQUENT RUBY

Foreword by **Obie Fernandez,** *Series Editor*

RUSS OLSEN

Praise for *Eloquent Ruby*

"Reading *Eloquent Ruby* is like programming in Ruby itself: fun, surprisingly deep, and you'll find yourself wishing it was always done this way. Wherever you are in your Ruby experience from novice to Rails developer, this book is a must read."

—Ethan Roberts
Owner, Monkey Mind LLC

"*Eloquent Ruby* lives up to its name. It's a smooth introduction to Ruby that's both well organized and enjoyable to read, as it covers all the essential topics in the right order. This is the book I wish I'd learned Ruby from."

—James Kebinger
Senior Software Engineer, PatientsLikeMe
www.monkeyatlarge.com

"Ruby's syntactic and logical aesthetics represent the pinnacle for elegance and beauty in the ALGOL family of programming languages. *Eloquent Ruby* is the perfect book to highlight this masterful language and Russ's blend of wit and wisdom is certain to entertain and inform."

—Michael Fogus
Contributor to the Clojure programming language and author of *The Joy of Clojure*

```
  it 'should know how many words it contains' do
    @doc.word_count.should == 4
  end

end
```

There is also an `after`, which is the RSpec cousin of `teardown` and allows you to get code executed after each example. The `:each` parameter means to run the code supplied before (or after) each example. Alternatively, you can use `before(:all)` and `after(:all)` to have some code run before or after any of the examples are run.

Easy Stubs

One of the banes of unit testing flows from the fact that an ideal test exercises exactly one class at a time. Doing this means that when the test fails we know there is something wrong with the class we are testing and not some other class that just happened to get dragged along. The trouble is that most classes will not function without other classes to help them: Programs tend to be complicated ecosystems, with the majority of classes relying on the kindness of other classes. It's this supporting software that is a problem for tests: How do you test just the one class when that class needs an entourage of other classes to work?

What you need are stubs and mocks. A **stub** is an object that implements the same interface as one of the supporting cast members, but returns canned answers when its methods are called. For a concrete example, imagine that we've created a subclass of our `Document` class, a subclass that supports printing. Further, suppose that the real work of printing is done by a printer class, which supports two methods. Method one is called `available?`, and it returns true if the printer is actually up and running. Method two is `render`, which actually causes paper to come spewing out of a real printer. With the printer class in hand, getting our document onto paper is pretty easy:

```
class PrintableDocument < Document
  def print( printer )
    return 'Printer unavailable' unless printer.available?
    printer.render( "#{title}\n" )
    printer.render( "By #{author}\n" )
    printer.render( content )
```

```
      'Done'
    end
  end
```

The idea of the `print` method is that we pass it a printer object and it will print the document—but only if the printer is actually running.

The question here is, how do we test the `print` method without having to get involved with a real printer? Conceptually this is easy: You just create a stub printer class, a sort of stand-in class that has the same `available?` and `render` methods of the real printer class but doesn't actually do any printing. In practice, coding stubs by hand can be tedious and, if you are testing a complex class with a lot of dependencies, tedious and error prone.

The RSpec `stub` method is there to reduce the pain of creating stubs. To use the `stub` method, you pass in a hash of method names (as symbols) and the corresponding values that you want those methods to return. The `stub` method will give you back an object equipped exactly with those methods, methods that will return the appropriate values. Thus, if you called `stub` like this:

```
stub_printer = stub :available? => true, :render => nil
```

You would end up with `stub_printer` pointing at an object with two methods, `available?` and `render`, methods that return `true` and `nil` respectively. With `stub` there are no classes to create, no methods to code; `stub` does it all for you. Here's an RSpec example that makes use of `stub_printer`:

```
describe PrintableDocument do
  before :each do
    @text = 'A bunch of words'
    @doc = PrintableDocument.new( 'test', 'nobody', @text )
  end

  it 'should know how to print itself' do
    stub_printer = stub :available? => true, :render => nil
    @doc.print( stub_printer ).should == 'Done'
  end

  it 'should return the proper string if printer is offline' do
    stub_printer = stub :available? => false, :render => nil
```

```
    @doc.print( stub_printer ).should == 'Printer unavailable'
  end
end
```

Along with stub, RSpec also provides the `stub!` method, which will let you stub out individual methods on any regular object you might have lying around. Thus, if we need a string that claimed to be a million characters long, we could say:

```
apparently_long_string = 'actually short'
apparently_long_string.stub!( :length ).and_return( 1000000 )
```

What all of this means is that with RSpec you will never have to write another class full of stubbed-out methods again.

. . . And Easy Mocks

Stubs, with their ability to quietly return canned answers, are great for producing the boring infrastructure that you need to make a test work. Sometimes, however, you need a stublike object that takes more of an active role in the test. Look back at our last printing test (the one with the stub) and you will see that the test doesn't verify that the `print` method ever called `render`. It's a sad printing test that doesn't check to see that something got printed.

What we need here is a **mock**. A mock is a stub with an attitude. Along with knowing what canned responses to return, a mock also knows which methods should be called and with what arguments. Critically, a disappointed mock will fail the test. Thus, while a stub is there purely to get the test to work, a mock is an active participant in the test, watching how it is treated and failing the test if it doesn't like what it sees.

In a boring bit of consistency, RSpec provides a `mock` method to go with `stub`. Here again is our `PrintableDocument` spec, this time enhanced with a mock:

```
it 'should know how to print itself' do
  mock_printer = mock('Printer')
  mock_printer.should_receive(:available?).and_return(true)
  mock_printer.should_receive(:render).exactly(3).times
  @doc.print( mock_printer ).should == 'Done'
end
```

In the code shown here we create a mock printer object and then set it up to expect that, during the course of the test, `available?` will be called at least once and `render` will be called exactly three times. As you can see, RSpec defines a little expectation language that you can use to express exactly what should happen. In addition to declaring that some method will or will not be called, you can also specify what arguments the method should see, RSpec will check these expectations at the end of each example, and if they aren't met the spec will fail.

These examples really just scratch the surface of what you can do with RSpec. Take a look at http://rspec.info for the full documentation.

In the Wild

Given the intense interest of the Ruby community in testing, it's not surprising that there are a lot of Ruby testing frameworks and utilities around. For example, if you decide to use Test::Unit, you might want to look into shoulda.[4] The shoulda gem defines all sorts of useful utilities for your Test::Unit tests, including the ability to replace those traditional test methods with RSpec-like examples:

```
require 'test/unit'
require 'shoulda'
require 'document.rb'

class DocumentTest < Test::Unit::TestCase
  context 'A basic document class' do
    def setup
      @text = 'A bunch of words'
      @doc = Document.new('a test', 'russ', @text)
    end

    should 'hold on to the contents' do
      assert_equal @text, @doc.content, 'Contents still there'
    end

    # Rest of the test omitted...
  end
end
```

4. http://github.com/thoughtbot/shoulda

Test::Unit users should also look into mocha,[5] which provides mocking facilities along the same lines as RSpec.

If you have settled on RSpec, a great place to look for examples of specs is the RubySpec[6] project. The fine folks behind RubySpec are trying to build a complete Ruby language specification in RSpec format. The beauty of this approach is that when they are done we will not only have a full specification of the Ruby language that people can read, but we will also have an executable specification, one that you can run against any Ruby implementation.

For our purposes, RubySpec is a great place to find real world, but nevertheless easy to understand specs. For example, here is a spec which makes sure that `if` statements work as advertised:

```
describe "The if expression" do
  it "evaluates body if expression is true" do
    a = []
    if true
       a << 123
    end
    a.should == [123]
  end

  it "does not evaluate body if expression is false" do
    a = []
    if false
      a << 123
    end
    a.should == []
  end

  # Lots and lots of stuff omitted
end
```

And here is a spec for the `Array.each` method:

```
describe "Array#each" do
  it "yields each element to the block" do
```

5. http://mocha.rubyforge.org

6. http://rubyspec.org

```
    a = []
    x = [1, 2, 3]
    x.each { |item| a << item }.should equal(x)
    a.should == [1, 2, 3]
  end

  # Lots of stuff omitted
end
```

There's a bit of irony in looking at the RubySpec project for tips on how to use RSpec given that RubySpec does not actually use RSpec. Instead, the RubySpec project uses a mostly compatible RSpec offshoot called MSpec. Although MSpec is close enough to RSpec for our "find me an example" purposes, it differs from RSpec in ways that are important if you are trying to test the whole Ruby language.

Finally, if you would like to do RSpec style examples but don't want to go to the trouble of installing the RSpec gem, you might consider Minitest, which is included in Ruby 1.9. MiniTest is a complete rewrite of Test::Unit, and it also sports MiniSpec, an RSpec like framework. Did I mention that Ruby people like testing?

Staying Out of Trouble

Simply having a set of automated tests is as close to a magical elixir for software quality as you are likely to find in this life. There are, however, a number of things you can do to ensure that you are getting the maximum testing magic for your effort. For example, unit tests, the ones that developers run as they develop, should be quick. Ideally the tests for your whole system should run in at most a few minutes. Think about it: Unit tests that take an hour to run will be run at most once per hour. Who am I kidding? Given the level of patience displayed by the average developer, unit tests that take an hour to run will get run once or twice a week if you are lucky. If you want developers to run your unit tests as often as they should, they gotta go quick.

Don't get me wrong—longer running tests are fine; in fact, they are great. Longer-running tests that pound away at the database, that require all the servers to be running, that stress the heck out of the system are great. They are just not unit tests. Unit tests should run quick with the setup that every developer has. They are your first line of defense, and in order to be any good they must be run often.