CHAPTER 9
Write Specs!

The past few decades have seen the software world argue endlessly about the best way to develop reliable programs: Should our languages be procedural or object oriented or functional? Should we build things top down or bottom up or inside out? When we code, should we be rational, agile, or simply pragmatic? Out of all this discussion, one simple idea has emerged: If you want to know that your code works, you need to test it. You need to test it early, you need to test it often, and you certainly need to test it whenever you change it. It turns out that with programs, seeing is believing.

Unless you want to spend all your waking hours running tests manually, you need a test framework, a framework that will let the computer exercise the code for you. Although the whole programming world has awakened to the need for automated tests, few language communities have embraced the concept with the passion of the Ruby community. A key part of the Ruby style of programming is that no class, and certainly no program, is ever done if it lacks automated tests.

In this chapter we are going to take a look at two different Ruby testing frameworks, starting with the very traditional Test::Unit and moving on to the very popular—and very nontraditional—RSpec. We'll also look at some of the do's and don'ts for writing tests, no matter what framework you use. Finally, we will also look at how the RubySpec project is using an RSpec-like approach to specifying the Ruby programming language itself.

Test::Unit: When Your Documents Just Have to Work

We are going to open our look at Ruby testing frameworks by starting with the familiar: Test::Unit. Test::Unit comes packaged with Ruby itself and is a member of the so-called XUnit family of testing frameworks, so called because there is a similar one for virtually every programming language out there.

The very simple idea behind Test::Unit is to exercise your code in a series of individual tests, where each test tries out one aspect of the program. In Test::Unit, each test is packaged in a method whose name needs to begin with `test_`. Here, for instance, is a little test that checks to make sure that the `Document` class can do the very basic thing that documents need to do, namely, to hold onto text:

```
def test_document_holds_onto_contents
  text = 'A bunch of words'
  doc = Document.new('test', 'nobody', text)
  assert_equal text, doc.content
end
```

There is not really a lot going on in this test: We make a document with some text, and then we check, using `assert_equal`, that the document does indeed still have the text. If the two arguments to `assert_equal` are not, in fact, equal, the test fails. One stylistic thing that his test does well is to have a nice descriptive name. If we do this testing thing right we are going to have a lot of test methods, and the last thing we want is to have to dig through the test code to see just what `test_that_it_works` or `test_number_four` is all about. In the same spirit, we could improve our use of `assert_equal` by adding the third, optional description parameter:

```
  assert_equal text, doc.content, 'Contents are still there'
```

Along with `assert_equal`, Test::Unit also allows you to assert that some arbitrary condition is true with the `assert` method. Thus we might check that our `words` method is returning what it should with:

```
  assert doc.words.include?( 'bunch' )
```

To really use Test::Unit you need to roll your tests up in a class, a subclass of `Test::Unit::TestCase`. Inside the class you can have any number of test methods:

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

class DocumentTest < Test::Unit::TestCase
  def test_document_holds_onto_contents
    text = 'A bunch of words'
    doc = Document.new('test', 'nobody', text)
    assert_equal text, doc.content, 'Contents are still there'
  end

  def test_that_doc_can_return_words_in_array
    text = 'A bunch of words'
    doc = Document.new('test', 'nobody', text)
    assert doc.words.include?( 'A' )
    assert doc.words.include?( 'bunch' )
    assert doc.words.include?( 'of' )
    assert doc.words.include?( 'words' )
  end

  def test_that_word_count_is_correct
    text = 'A bunch of words'
    doc = Document.new('test', 'nobody', text)
    assert_equal 4, doc.word_count, 'Word count is correct'
  end
end
```

Kicking off a Test::Unit test is about as easy as it comes: Just run the file containing the test class with Ruby:[1]

```
ruby document_test.rb
Loaded suite document_test
Started
...
Finished in 0.000261 seconds.

3 tests, 3 assertions, 0 failures, 0 errors
```

1. I'd like to invite you to stop for a minute and think about how this could possibly work. How do those tests get run when you just feed your test class declaration into the Ruby interpreter with no main program? I'm afraid you will have to wait until Chapter 20 for the answer.

Note that if one of the test methods does happen to fail, Test::Unit will keep soldiering along, running all the other tests. This is generally a good thing, since if there are multiple broken tests it allows you to survey all the wreckage after a single test run.

One problem with the document test is that it contains a fair bit of redundant code: We keep creating that `Document` instance over and over. To deal with this kind of thing, Test::Unit provides the `setup` method along with its friend, the `teardown` method. The `setup` method gets called before each test method is run; it's your opportunity to do what needs to be done to get ready for your tests. In our example we would probably just create the document:

```
class DocumentTest < Test::Unit::TestCase
  def setup
    @text = 'A bunch of words'
    @doc = Document.new('test', 'nobody', @text)
  end

  def test_that_document_holds_onto_contents
    assert_equal @text, @doc.content, 'Contents are still there'
  end

  def test_that_doc_can_return_words_in_array
    assert @doc.words.include?( 'A' )
    assert @doc.words.include?( 'bunch' )
    assert @doc.words.include?( 'of' )
    assert @doc.words.include?( 'words' )
  end

  def test_that_word_count_is_correct
    assert_equal 4, @doc.word_count, 'Word count is correct'
  end
end
```

In the same way, the `teardown` method gets called after each test method gets run. The `teardown` method is great for closing database connections, deleting temporary files, or any other general post-test tidying up. Note that `setup` and `teardown` get called around *each* test method, not before, and after all of the tests in the class get run.

A Plethora of Assertions

So far our tests have only needed to check that something is true with `assert` or that one thing is equal to something else with `assert_equal`. These two methods are certainly not the only assertions in the Test::Unit toolkit. To go with `assert` and `assert_equal` we have `assert_not_equal` as well as `assert_nil` and `assert_not_nil`, each of which does about what you would expect.

If you are dealing with a lot of strings, Test::Unit has a handy `assert_match`, which will fail if a string does not match a given regular expression:

```
assert_match /times.*/, 'times new roman'
```

You can also check that an object is an instance of some class:

```
assert_instance_of String, 'hello'
```

And you can assert that some code raises an exception:

```
assert_raise ZeroDivisionError do
  x = 1/0
end
```

Or doesn't raise (or throw, as the name of the assertion would have it) an exception:

```
assert_nothing_thrown do
  x = 1/2
end
```

In all, Test::Unit tests can call on about twenty different assertions.

Don't Test It, Spec It!

Clearly Test::Unit is a workmanlike chunk of testing support, and if you are only going to write an occasional bit of test code it would probably do. The trouble is that if you are doing the Ruby thing right, you are not just writing the occasional test. A good Ruby application comes with lots of tests, tests that try out everything that can be tried out.

Unfortunately, much of the code of a Test::Unit test is about the *test* and not about the code that you are testing. Take another look at `DocumentTest`: You will see that we have methods that do this and assert that result, but if you are reading the `DocumentTest` code, you need to *infer* what `Document` behavior is being tested from the test code.

In a more perfect world, the test would focus on the behavior itself, so that the test would read something like this:

> About the `Document` class: When you have a document instance, it should hang onto the text that you give it. It should also return an array containing each word in the document when you call the `words` method. And it should return the number of words in the document when you call the `word_count` method.

Rspec, possibly the Ruby world's favorite testing framework, tries to get us to that more perfect world. Here is the same set of `Document` tests expressed in RSpec:

```
describe Document do
  it 'should hold on to the contents' do
    text = 'A bunch of words'
    doc = Document.new( 'test', 'nobody', text )
    doc.content.should == text
  end

  it 'should return all of the words in the document' do
    text = 'A bunch of words'
    doc = Document.new( 'test', 'nobody', text )
    doc.words.include?( 'A' ).should == true
    doc.words.include?( 'bunch' ).should == true
    doc.words.include?( 'of' ).should == true
    doc.words.include?( 'words' ).should == true
  end

  it 'should know how many words it contains' do
    text = 'A bunch of words'
    doc = Document.new( 'test', 'nobody', text )
    doc.word_count.should == 4
  end
end
```

RSpec tries to weave a sort of pseudo-English out of Ruby: The code above isn't a test, it's a description. The description says that the `Document` class should hold on to the contents. We don't assert things; we say that they should happen. Thus, we don't assert that `word_count` returns 4; instead, we say that `word_count` should equal 4. Like Test::Unit assertions, RSpec `should`s come in a wide variety of forms. We could, for example, simplify the code above by using `should include`, turning this:

```
        doc.words.include?( 'bunch' ).should == true
```

Into:

```
        doc.words.should include( 'bunch' )
```

Similarly, there is `should match` for those times when a plain == will not do:

```
        doc.content.should match( /A bunch.*/ )
```

If you are feeling negative you can resort to `should_not.`[2]

```
        doc.words.should_not include( 'Excelsior' )
```

By convention, your RSpec code—generally just called a **spec**—goes in a file called `<<class name>>_spec.rb`, so the previous example would be best stored away in `document_spec.rb`. You can run the spec by using the `spec` command:[3]

```
    spec document_spec.rb
```

Do that and you should see something like this:

```
    ...

    Finished in 0.016846047 seconds

    3 examples, 0 failures
```

2. You can find out all about RSpec—including all the ways that you can say what should happen—at www.rspec.info.

3. Note that spec is an operating system command, along the lines of `ls` or `dir`, not something you would say in Ruby.

A very handy feature of the `spec` command is its ability to hunt down all of the spec files in a whole directory tree, assuming that you follow the `<<class name>>_spec.rb` convention. All you need to do is supply the path to a directory instead of a file to the `spec` command. So if you run:

```
spec .
```

RSpec will run all of the specs that live in the current directory and any of its subdirectories.

A Tidy Spec Is a Readable Spec

As it stands right now, our new document spec has the same problem as the original Test::Unit based `DocumentTest`: It has a lot of redundant code in it. Like the original test, each little chunk of the specification—called an **example** in RSpec parlance—creates the same document with the same text. RSpec deals with this problem the same way that Test::Unit does, by allowing you to supply code that is executed before each example. Here is a somewhat shorter-winded version of our spec:

```
require 'document'

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

  it 'should hold on to the contents' do
    @doc.content.should == @text
  end

  it 'should know which words it has' do
    @doc.words.should include( 'A' )
    @doc.words.should include( 'bunch' )
    @doc.words.should include( 'of' )
    @doc.words.should include( 'words' )
  end
  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.

Another thing that your tests should be is *silent*. When you run a test you want a simple answer to a simple question: Does it work or does it not work? You don't want to know how many elements are in the list, that you are now entering this method or leaving that method, and you certainly don't need help with today's date. If the test fails, and it tells you clearly that it failed, then you can pull out your tools and start looking at the problem. Until then, quiet please.

Tests also need to be independent of one another: You want to carefully avoid having one test rely on the output of a previous test. Don't create a file in one test (or RSpec example) and then expect it to be there in another. In particular, use RSpec's `before(:any)` in preference to `before(:all)`.

Do make sure that your tests will actually fail. It is all too easy to create a test that looks right but doesn't actually test anything worthwhile. For example, imagine that we implemented the `clone` method on `Document`. The `clone` method is another method that every object gets from the `Object` class: Call `clone` and you get a copy of your object. The default `clone` implementation makes a shallow copy; you get a second object from `clone`, but the instance variables of both the original and the clone will point at identically the same objects. We don't, however, have to live with the default: Here's a `Document` `clone` method that does a deep copy, duplicating `@title`, `@author`, and `@content` along with the `Document` instance:

```
class Document

  # ...

  def clone
    Document.new( title.clone, author.clone, content.clone )
  end
end
```

Having taken the advice of this chapter to heart, we make sure that we have a spec for our new `clone`. Here it is:

```
describe Document do
  it 'should have a functional clone method' do
    doc1 = Document.new( 'title', 'author', 'some stuff' )
    doc2 = doc1.clone
```

```
      doc1.title.should == 'title'
      doc1.author.should == 'author'
      doc1.content.should == 'some stuff'
    end
  end
```

The trouble with this last spec is that it is an impostor: It does indeed make a cloned copy of the document, but then it goes on to test the original (`doc1`) instead of the copy (`doc2`). The bottom line is that an important part of writing a test is making sure that it actually tests what you think it tests.

Let me also add a personal bit of testing heresy. Although the ideal set of unit tests will cover all of the significant features of your code, sometimes you just can't get to the ideal. Sometimes you can't get anywhere near it: The bug fix needs to go out, or the release is late, or your coworkers are just not into the whole testing thing. What to do? Well, write whatever unit tests you can. As an absolute minimum, just write a unit test that exercises the code a little, with no asserts at all. For example, look at this seemingly worthless spec:

```
require 'document'

describe Document do
  it 'should not catch fire when you create an instance' do
    Document.new( 'title', 'author', 'stuff' ).should_not == nil
  end
end
```

While far, far from ideal, this spec actually does more than you might think. If it runs successfully you will know:

- `Document` is a class.[7]
- The `Document` class is indeed found in the `document.rb` file.
- The `document.rb` file doesn't contain any really egregious Ruby syntax errors.
- `Document.new` will take three arguments.

7. Or at least it acts like a class.

- `Document.new` actually returns something.
- `Document.new` doesn't throw an exception.

Not bad for a few lines of code. At the risk of repeating myself, let me say it again: Write really, really good tests if you can. Write OK ones if that is all you can do. If you can't do anything else, at least write some tests that exercise your code, even just a little bit.

Given that you are going to have tests, or specs, there is also the question of *when* to write your tests. A large segment, perhaps a majority, of Ruby programmers adhere to the test-first philosophy of programming: Never, ever, add a feature to your code without first adding a test. Start out by writing a test that fails, the test-firsters say, and then write the code that makes the test pass. Repeat until you have the Great American Program, complete with a full suite of tests. The test-firsters make the very valid point that if you never write any code without first having a test for whatever that code is supposed to do, you will automatically write testable programs.

I have to say that I like the idea of test-first development, and I frequently practice it. But not—and here is the rub—always. There are times when I can be more productive by simply writing the code and then going back and adding the tests. When I hit those times I leave the tests for the end. Test-first development is a great idea, but we in the software industry have a habit of messing up really good ideas by deciding they are not just good but absolutely universal. And required. The take-away here is that you are not finished until you have both the software and the tests or specs to go with it. Write the tests first, or second, or third. But write the darned tests.

Wrapping Up

Let me say it again: Write the tests. Although I've tried to make the case in this chapter that Test::Unit is a fine, traditional testing framework and that RSpec, with its readable examples and easy mocking, is a great testing framework, the real message is that you need to write the tests. You will never know whether your code works unless you write the tests. Write thoroughly comprehensible tests if you can, write sketchy tests if you must, but write the tests.