Extracted from:

Effective Testing with RSpec 3

Build Ruby Apps with Confidence

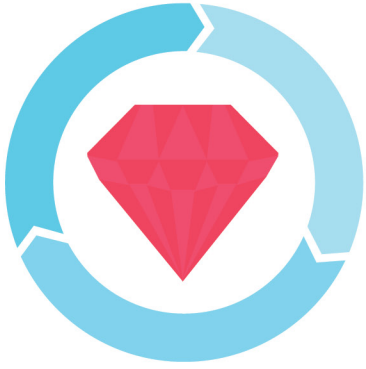
The Pragmatic Bookshelf

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

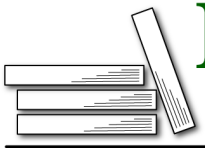
Foreword by Tom Stuart,
author of *Understanding Computation*

Effective Testing with RSpec 3

Build Ruby Apps with Confidence

Myron Marston
Ian Dees

The Pragmatic Bookshelf
Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

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

Origins: Pure, Partial, and Verifying Doubles

Now that we've seen the different usage modes of test doubles, let's look at where they come from.

Pure Doubles

All of the test doubles you've written so far in this chapter are *pure doubles*: they're purpose-built by rspec-mocks and consist entirely of behavior you add to them. You can pass them into your project code just as if they were the real thing.

Pure doubles are flexible and easy to get started with. They're best for testing code where you can pass in dependencies. Unfortunately, real-world projects are not always so tester-friendly, and you'll need to turn to more powerful techniques.

Partial Doubles

Sometimes, the code you're testing doesn't give you an easy way to inject dependencies. A hard-coded class name may be lurking three layers deep in that API method you're calling. For instance, a lot of Ruby projects call Time.now without providing a way to override this behavior during testing.

To test these kinds of codebases, you can use a *partial double*. These add mocking and stubbing behavior to existing Ruby objects. That means any object in your system can be a partial double. All you have to do is expect or allow a specific message, just like you'd do for a pure double:

```
>> random = Random.new
=> #<Random:0x007ff2389554e8>
>> allow(random).to receive(:rand).and_return(0.1234)
=> #<RSpec::Mocks::MessageExpectation #<Random:0x007ff2389554e8>.rand(any ↩
arguments)>
>> random.rand
=> 0.1234
```

In this snippet, you've created an instance of Ruby's random number generator, and then replaced its rand method with one that returns a canned value. All its other methods will behave normally.

You can also use a partial double as a spy, using the expect(...).to have_received form you saw earlier:

```
>> allow(Dir).to receive(:mktmpdir).and_yield('/path/to/tmp')
=> #<RSpec::Mocks::MessageExpectation #<Dir (class)>.mktmpdir(any arguments)>
>> Dir.mktmpdir { |dir| puts "Dir is: #{dir}" }
Dir is: /path/to/tmp
=> nil
```

```
>> expect(Dir).to have_received(:mktmpdir)
=> nil
```

When you used a pure double as a spy, you had a choice of how to specify up front which messages the spy should allow. You could permit *any* message (using spy or as_null_object), or explicitly allow just the messages you want. With partial doubles, you can only do the latter. RSpec doesn't support the notion of a "partial spy," because it can't spy on all of a real object's methods in a performant way.

When you use partial doubles inside your specs, RSpec will revert all your changes at the end of each example. The Ruby object will go back to its original behavior. That way, you won't have to worry about the test double behavior leaking into other specs.

Since you are experimenting in stand-alone mode, you will need to call RSpec::Mocks.teardown explicitly to get this same cleanup to happen:

```
>> RSpec::Mocks.teardown
=> #<RSpec::Mocks::RootSpace:0x007ff2389bccb0>
>> random.rand
=> 0.9385928886462153
```

This call also exits from the stand-alone mode you've been experimenting in. If you want to keep exploring in the same IRB session, you'll need to call RSpec::Mocks.setup to go back into stand-alone mode.

Test Doubles Have Short Lifetimes

RSpec tears down all your test doubles at the end of each example. That means they won't play well with RSpec features that live outside the typical per-example scope, such as before(:context) hooks. You can work around some of these limitations with a method named with_temporary_scope.[1]

Partial doubles are useful, but we consider them a *code smell*, a superficial sign that might lead you to a deeper design issue.[2] In *Using Partial Doubles Effectively, on page ?*, we'll explain some of these underlying issues and how to address them.

Verifying Doubles

The upside of test doubles is that they can stand in for a dependency you don't want to drag into your test. The downside is that the double and the

1. https://relishapp.com/rspec/rspec-mocks/v/3-6/docs/basics/scope
2. https://martinfowler.com/bliki/CodeSmell.html

dependency can drift out of sync with each other.[3] *Verifying doubles* can protect you from this kind of drift.

In *Test Doubles: Mocks, Stubs, and Others, on page ?*, you created a test double to help you test a high-level API when your lower-level Ledger class didn't exist yet. We later explained that you were using a verifying double for that spec; let's take a closer look at why it was important to do so.

Here's a simplified version of a similar double, *without* verification:

13-understanding-test-doubles/02/expense_tracker/spec/unit/ledger_double_spec.rb
```
ledger = double('ExpenseTracker::Ledger')
allow(ledger).to receive(:record)
```

When you tested your system's public API, your routing code called Ledger#record:

13-understanding-test-doubles/02/expense_tracker/app/api.rb
```
post '/expenses' do
  expense = JSON.parse(request.body.read)
➤ result = @ledger.record(expense)
  JSON.generate('expense_id' => result.expense_id)
end
```

The Ledger class didn't exist yet; the test double provided enough of an implementation for your routing specs to pass. Later, you built the real thing.

Consider what would happen if at some point you renamed the Ledger#record method to Ledger#record_expense but forgot to update the routing code. Your specs would still pass, since they're still providing a fake record method. But your code would fail in real-world use, because it's trying to call a method that no longer exists. These kinds of false positives can kill confidence in your unit specs.

You avoided this trap in the expense tracker project by using a verifying double. To do so, you called instance_double in place of double, passing the name of the Ledger class. Here's a stripped-down version of the code:

13-understanding-test-doubles/02/expense_tracker/spec/unit/ledger_double_spec.rb
```
ledger = instance_double('ExpenseTracker::Ledger')
allow(ledger).to receive(:record)
```

With this double in place, RSpec checks that the real Ledger class (if it's loaded) actually responds to the record message with the same signature. If you rename this method to record_expense, or add or remove arguments, your specs will correctly fail until you update your use of the method *and* your test double setup.

3. https://www.thoughtworks.com/insights/blog/mockists-are-dead-long-live-classicists

Use Verifying Doubles to Catch Problems Earlier

Although your unit specs would have had a false positive here, your acceptance specs would still have caught this regression. That's because they use the real versions of the objects, rather than counting on test doubles.

By using verifying doubles in your unit specs, you get the best of both worlds. You'll catch errors earlier and at less cost, while writing specs that behave correctly when APIs change.

RSpec gives you a few different ways to create verifying doubles, based on what it will use as an interface template for the double:

instance_double('SomeClass')
: Constrains the double's interface using the *instance methods* of SomeClass

class_double('SomeClass')
: Constrains the double's interface using the *class methods* of SomeClass

object_double(some_object)
: Constrains the double's interface using the methods of some_object, rather than a class; handy for dynamic objects that use method_missing

In addition, each of these methods has a _spy variant (such as instance_spy) as a convenience for using a verifying double as a spy.

Stubbed Constants

Test doubles are all about controlling the environment your specs run in: what classes are available, how certain methods behave, and so on. A key piece of that environment is the set of Ruby constants available to your code. With *stubbed constants*, you can replace a constant with a different one for the duration of one example.

For instance, password hashing algorithms are slow by design for security reasons—but you may want to speed them up during testing. Algorithms like bcrypt take a tunable *cost factor* to specify how expensive the hash computation will be. If your code defines this number as a constant:

13-understanding-test-doubles/03/stubbed_constants.rb

```
class PasswordHash
  COST_FACTOR = 12

  # ...
end
```

...your specs can redefine it to 1:

13-understanding-test-doubles/03/stubbed_constants.rb
```
stub_const('PasswordHash::COST_FACTOR', 1)
```

You can use stub_const to do a number of things:

- Define a new constant
- Replace an existing constant
- Replace an entire module or class (because these are also constants)
- Avoid loading an expensive class, using a lightweight fake in its place

Sometimes, controlling your test environment means *removing* an existing constant instead of stubbing one. For example, if you're writing a library that works either with or without ActiveRecord, you can hide the ActiveRecord constant for a specific example:

13-understanding-test-doubles/03/stubbed_constants.rb
```
hide_const('ActiveRecord')
```

Hiding the ActiveRecord constant like this will cut off access to the entire module, including any nested constants like ActiveRecord::Base. Your code won't be able to accidentally use ActiveRecord. Just as with partial doubles, any constants you've changed or hidden will be restored at the end of each example.

Your Turn

In this chapter, we discussed the differences between stubs, mocks, spies, and null objects. In particular, you saw how they deal with the following situations:

- Receiving expected messages
- Receiving unexpected messages
- Not receiving expected messages

We also looked at the different ways to create test doubles. Pure doubles are entirely fake, whereas partial doubles are real Ruby objects that have fake behavior added. Verifying doubles fall in between, and have the advantages of both with few of the downsides of either. They're the ones we use most often.

Now that you understand test doubles, you'll be ready to tackle the next chapter, where you'll configure *how* and *when* your doubles respond to messages. But first, we have a simple exercise that demonstrates a few nuances of verifying doubles.

Exercise

In this guided exercise, you're going to test a Skier class that collaborates with a TrailMap class. Starting in a fresh directory, put the following code in lib/skier.rb:

13-understanding-test-doubles/exercises/mountain/lib/skier.rb

```
module Mountain
  class Skier
    def initialize(trail_map)
      @trail_map = trail_map
    end

    def ski_on(trail_name)
      difficulty = @trail_map.difficulty(trail_name)
      @tired = true if difficulty == :expert
    end

    def tired?
      @tired
    end
  end
end
```

Now, create a file called lib/trail_map.rb with the following contents:

13-understanding-test-doubles/exercises/mountain/lib/trail_map.rb

```
puts 'Loading our database query library...'
sleep(1)

module Mountain
  class TrailMap
    def difficulty_of(trail_name)
      # Look up the trail in the database
    end
  end
end
```

The TrailMap class has a difficulty_of method, but the Skier class is incorrectly trying to call difficulty instead. If we use a verifying double to stand in for a TrailMap, it should be able to catch this kind of error; let's try that out.

Trying the Verifying Double

Create a file called spec/skier_spec.rb, and put the following spec in it:

13-understanding-test-doubles/exercises/mountain/spec/skier_spec.rb

```
require 'skier'

module Mountain
  RSpec.describe Skier do
    it 'gets tired after skiing a difficult slope' do
      trail_map = instance_double('TrailMap', difficulty: :expert)

      skier = Skier.new(trail_map)
      skier.ski_on('Last Hoot')
      expect(skier).to be_tired
    end
  end
end
```

This spec makes the same mistake the Skier class did with method names. It stubs the difficulty method instead of difficulty_of. However, you're using instance_double, so RSpec should catch the problem—right?

Try running your spec:

```
$ rspec
```

Surprisingly, the specs pass. RSpec can only verify against a real class if that class is actually loaded. With nothing to verify against, the verifying double acts just like a normal, non-verifying double. So, try running it again *with* the TrailMap class loaded; just pass -rtrail_map on the command line:

```
$ rspec -rtrail_map
```

The specs *still* pass. Moreover, they're running much more slowly (nearly 10x slower on our computers!) because of the time spent loading a heavyweight dependency. Before moving on, see if you can guess why RSpec isn't checking your trail_map double against the real Mountain::TrailMap class.

The Problem

The problem is that the constant name passed into instance_double doesn't match the real class. The TrailMap class's full name, including the module it's nested in, is 'Mountain::TrailMap'.

Change the instance_double call to use the correct name, and then rerun your specs (again, with -rtrail_map). This time, they should fail the way you'd expect them to: with an error message about the use of a nonexistent difficulty method.

There are two ways to catch these kinds of naming issues before they happen:

- Use Ruby classes instead of strings
- Configure RSpec to check that the class name exists

You're going to get the chance to try out both of these options. Undo the fix you just made before you start the next step of the exercise.

Using Ruby Constants

First, let's try using a Ruby constant to indicate which class you're faking. In the call to instance_double, change the string 'TrailMap' to the class TrailMap (without quotes).

Now, run your specs the same way you did at the beginning of this exercise: plain rspec with no command-line arguments. The first time you tried this, RSpec gave an incorrectly passing result. Now, you'll get an uninitialized constant Mountain::TrailMap error, because the TrailMap class isn't loaded.

To use the Ruby class directly like this, you'll have to make sure the dependency is loaded before your spec runs. If your specs use the class directly (as this one now does), you'll typically just add require 'trail_map' at the top of your spec file.

There are times, however, when you might *not* want to load your dependencies explicitly in this way:

- Your dependencies take a long time to load, like trail_map does
- You need to use a test double before the dependency even exists, as you did with the Ledger double in the expense tracker project

Now, back out the change you just made, and we'll look at the other way to catch class naming issues.

Configuring RSpec to Check Names

In *Library Configuration*, on page ?, you used an RSpec.configure block to set up rspec-mocks. Using the same kind of block, you can configure RSpec to make sure that all of your verifying doubles are based on real, loaded classes.

The setting you need is called verify_doubled_constant_names. You probably don't want to turn it on unconditionally in spec_helper.rb. If you did, you'd never be able to use a verifying double before its class existed! Instead, put the setting into a file you can load on demand; let's call it spec/support/verify_doubled_constants.rb:

13-understanding-test-doubles/exercises/mountain/spec/support/verify_doubled_constants.rb

```
RSpec.configure do |c|
  c.mock_with :rspec do |mocks|
    mocks.verify_doubled_constant_names = true
  end
end
```

When you want RSpec to be strict about your verifying doubles, just pass -rsupport/verify_doubled_constants on the command line:

```
$ rspec -rtrail_map -rsupport/verify_doubled_constants
```

Your specs will correctly fail, and RSpec will warn you that the class name doesn't exist. If you use this approach, we recommend that you develop with this setting *off*, but configure your continuous integration (CI) server to run with the setting *on*.

Make It Easy to Replicate Your CI Setup

Repeatability is important when you're setting up a CI system. Few things are more frustrating than a spec passing on your local machine but failing on the CI server.

If you're going to use certain options only with CI, such as the verify_doubled_constant_names setting, we recommend putting all of these options into a script or Rake task you can run locally. That way, when a spec fails on CI, you can just run something like ./script/ci_build and diagnose the issue on your machine.

We'll talk more about integration with Rake in Appendix 1, *RSpec and the Wider Ruby Ecosystem*, on page ?.

Wrapping Up

As we wrap up, let's look at the trade-offs we've seen. Verifying doubles do the following things:

- They raise errors when your code calls a dependency incorrectly.
- They can only do so when the dependency actually exists.
- They revert silently to regular doubles if the dependency doesn't exist.

To deal with that last item, you can create your doubles from Ruby class names instead of strings. It's only practical to do so if you've already written code for the dependency, and if it's not too expensive to load it. If you can't use a Ruby class, you can still double-check your constant names by setting verify_doubled_constant_names when you run your whole suite.

Using verifying doubles correctly takes a little extra up-front care. But the benefits to your project are well worth it.