Extracted from:

# Rails Test Prescriptions

## Keeping Your Application Healthy

The Pragmatic Programmers

# Rails Test Prescriptions

## Keeping Your Application Healthy

Noel Rappin

*Edited by Colleen Toporek*

in your test need to be in the mock_active_records() call, since an attempt to call the stubbed instantiate() method with a nonmatching hash would trigger an expectation error. In a factory universe, with only a couple of object defined, that may not be a difficult constraint to live with. Also, the internals of ActiveRecord may change in the future, causing this mechanism to stop working.

There is a simpler option if you have only one or two objects to mock and a simple method under test.

```
test "My projects might be properly saved" do
  @bluebook = Project.make(:name => "Project Bluebook")
  Project.stub(:find).return(@bluebook)
  @bluebook.stubs(:save => true)
  post :update, :id => @bluebook.id
  «»
end
```

All this does is stub the Project class to always return @bluebook when find() is called. That ensures that the controller method that looks up the object using find() returns the same object that you've set up in the test. There are sharp limitations here—basically, we're assuming that only one Project object needs to be created for the test. But there are a lot of cases, like a simple update or create method, where that assumption holds, and this is a reasonably clean way to share a stubbed object between the test and the method being tested.

## 7.4 Mock, Mock, Mock

A true mock object retains the basic idea of the stub—returning a specified value without actually calling a live method—and adds the requirement that the specified method must actually be called during the test. In other words, a mock is like a stub with attitude, expecting—nay, demanding—that its parameters be matched in the test or else we get a test failure.

As with stubs, Mocha provides a way to create a mock object from whole cloth, as well as a way to add mock expectations to an existing object. The method for bare mock creation is mock():

```
test "a sample mock" do
  mocky = mock(:name => "Paul", :weight => 100)
  assert_equal("Paul", mocky.name)
end
```

As it happens, this test fails:

```
  1) Failure:
test_a_sample_mock(ProjectTest) [/test/unit/project_test.rb:46]:
not all expectations were satisfied
unsatisfied expectations:
- expected exactly once, not yet invoked:
    #&lt;Mock:0x25550bc&gt;.weight(any_parameters)
satisfied expectations:
- expected exactly once, already invoked once:
    #&lt;Mock:0x25550bc&gt;.name(any_parameters)
```

It fails because the first line sets up two mock expectations, one for mocky.name() and one for mocky.weight(), but only one of those two mocked methods are called in the test. Hence, it's an unsatisfied expectation. To pass the test, add a call to mocky.weight():

```
test "a sample mock" do
  mocky = mock(:name => "Paul", :weight => 100)
  assert_equal("Paul", mocky.name)
  assert_equal(100, mocky.weight)
end
```

The method for adding a mock expectation to an existing object is expects():[5]

Download huddle_mocha/test/unit/project_test.rb

```
test "lets mock an object" do
  mock_project = Project.new(:name => "Project Greenlight")
  mock_project.expects(:name).returns("Fred")
  assert_equal("Fred", mock_project.name)
end
```

All the modifiers we've seen so far were applied to stubs, like returns(), raises(), any_instance(), and with(), or all the pattern matchers can be added to a mock statement. For example, the controller test for create and update failure can be changed to use true mocks:

Download huddle_mocha/test/functional/projects_controller_test.rb

```
test "mock fail create gracefully" do
  assert_no_difference('Project.count') do
    Project.any_instance.expects(:save).returns(false)
    post :create, :project => {:name => 'Project Runway'}
    assert_template('new')
  end
end

test "mock fail update gracefully" do
```

5. I have no idea why they didn't use *mocks*, which would seem more consistent.

```
  Project.any_instance.expects(:update_attributes).returns(false)
  put :update, :id => projects(:huddle).id, :project => {:name => 'fred'}
  assert_template('edit')
  actual = Project.find(projects(:huddle).id)
  assert_not_equal('fred', actual.name)
end
```

Again, the behavior of these tests is identical to the stub version, except for the additional, implicit test that the save() and update_attributes() methods are, in fact, called during the test.

By default, mock() and expects() set a validation that the associated method is called exactly once during the test. If that does not meet your testing needs, Mocha has methods that let you specify the number of calls to the method. These methods are largely self-explanatory:

```
proj = Project.new
proj.expects(:name).once
proj.expects(:name).twice
proj.expects(:name).at_least_once
proj.expects(:name).at_most_once
proj.expects(:name).at_least(3)
proj.expects(:name).at_most(3)
proj.expects(:name).times(5)
proj.expects(:name).times(4..6)
proj.expects(:name).never
```

In practice, the default behavior is good for most usages.

## 7.5 Mock Objects and Behavior-Driven Development

The interesting thing about using true mocks is that their usage enables a completely different style of testing. In the tests we've seen throughout most of this book, the test validates the result of a computation: it's testing the end state of a process. When using mocks, however, we have the opportunity to test the behavior of the process during the test, rather than the outcome.

An example will help clarify the difference. Back in Section 4.2, *Testing the View*, on page 67, the Huddle application had a controller test that was largely based on the results of a call to the model.

Without mock objects, the test looked like this (from test/functional/project_controller_test.rb):

Download huddle_mocha/test/functional/projects_controller_test.rb

```
test "project timeline index should be sorted correctly" do
  set_current_project(:huddle)
  get :show, :id => projects(:huddle).id
  expected_keys = assigns(:reports).keys.sort.map{ |d| d.to_s(:db) }
  assert_equal(["2009-01-06", "2009-01-07"], expected_keys)
  assert_equal(
      [status_reports(:ben_tue).id, status_reports(:jerry_tue).id],
      assigns(:reports)[Date.parse("2009-01-06")].map(&:id))
end
```

As the process played out in that section, the assertions in this test wound up being copied more or less identically to the model test that actually exercised the model call that is made by the controller show() action being tested here. At the time, we mentioned that a mock object package would be a different way of writing the test. The mocked version of the test could look something like this passing test:

Download huddle_mocha/test/functional/projects_controller_test.rb

```
Line 1  test "mock show test" do
    2    set_current_project(:huddle)
    3    Project.any_instance.expects(:reports_grouped_by_day).returns(
    4        {Date.today => [status_reports(:aaron_tue)]})
    5    get :show, :id => projects(:huddle).id
    6    assert_not_nil assigns(:reports)
    7  end
```

At first glance, that looks ridiculously minimalist. It doesn't seem to actually be asserting much of anything. The trick is the combination of the mock expectation set in lines 3–4, along with the rest of the tests that presumably exist in this system. This test validates that the controller calls the model method reports_grouped_by_day() exactly once, and it validates that the reports variable is set to some value. It also validates that the controller and view run without error, but that's secondary. The test is validating a behavior of the controller method—namely, that it calls a particular model method, not the state that results from making that call.

What this test doesn't do is attempt to validate features that are actually the purview of other tests. It doesn't validate the response from the model method; that's the job of the model test. What the view layer does

with this value is the job of a view test. This test validates that a particular instance variable is set to a value using a known model method, on the theory that the job of the controller method is to produce a set of known values for use by the view. But validating the exact value of the :reports variable would be pointless (at least in this case), since the value is completely generated by the mock expectation.

Using mock objects in this style of testing has advantages and disadvantages. Speed is a significant advantage: getting values from mocks is going to be a lot faster than getting values from either a fixture or a factory database. Another advantage is the encapsulation of tests. In the previous example, if a bug is introduced into the model object, the only tests that will fail will be the model tests—the controller tests, protected by the mock, will be fine. The nonmock version of the controller test, however, is susceptible to failure based on the results of the model method. Done right, this kind of encapsulation can make it easier to diagnose and fix test failures.

However, there are a couple of potential problems to watch out for. One is a mismatch between the mocked method and the real method. In the previous controller example, the mock call causes the method to return a hash where the key is a Date object and the values are lists of StatusReport objects. If, however, the model method really returns a hash with the keys as strings, then you can have a case where the controller method passes, the model method passes, but the site as a whole breaks. In practice, this problem can be covered by using integration or acceptance tests; see Chapter 13, *Testing Workflow with Integration Tests*, on page 217 and Chapter 15, *Acceptance Testing with Cucumber*, on page 237.

It's also not hard to inadvertently create a test that is tautological by setting a mock to some value and then validating that the mocked method returns that value (the earlier examples that show how stubbed methods work have this flaw).

Finally, an elaborate edifice of mocked methods runs the risk of causing the test to be dependent on very specific details of the method structure of the object being mocked. This can make the test brittle in the face of refactorings that might change the object's methods. Good API design and an awareness of this potential problem go a long way toward mitigating the issue.

I have to say, as much as I love using mocks and stubs to cover hard-to-reach objects and states, my own history with very strict behavior-based mock test structures hasn't been great. My experience was that writing all the mocks around a given object tended to be a drag on the test process. But I'm wide open to the possibility that this method works better for others or that I'm not doing it right. Or, to quote Stephen Bristol:[6] "RSpec, done properly, isn't testing. It is designing."

## 7.6 Mock Dos and Mock Don'ts

Here are some guidelines on the best usage of stubs and mocks:

- If you are using your fake objects to take the place of real objects that are hard or impossible to create in a test environment, it's probably a good idea to use stubs rather than mocks. If you are actually using the fake value as an input to a different process, then you should test that process directly using the fake value rather than a mock. Adding the mock expectation just gives you another thing that can break, which in this use case is probably not related to what you are actually testing.
- When you are using a true mock to encapsulate a test and isolate it from methods that are not under test, try to limit the number of methods you are mocking in one test. The more mocks, the more vulnerable the test will be to changes in the actual code. A lot of mocks may indicate that your test is trying to do too much or might indicate a poor object-oriented design where one class is asking for too many details of a different class.
- I've come to use mocks frequently in controller testing to isolate the controller test from the behavior of the model, essentially only testing that the controller makes a specific model call and using the model test to verify model behavior. Among the benefits of using mocks this way is you are encouraged to make the interface between your controllers and models as simple as possible. However, it does mean that the controller test knows more about your model than it otherwise might, which may make the model code harder to change.

6. http://twitter.com/stevenbristol/statuses/1221264618

- You also need to be careful of mocking methods that have side effects or that call other methods that might be interesting. The mock totally bypasses the original method, which means no side effect and no calling the internal method. Pro tip: saving to the database and outputting to the response stream are both side effects.
- Be very nervous if you are specifying a value as a result of a mock and then asserting the existence of the very same value. One of the biggest potential problems with any test suite is false positives, and testing results with mocked values is a really efficient way to generate false positives.
- A potentially larger problem is the type mismatch issue between the real method and values being used for mocks. Integration or acceptance testing can help with this problem, but that's not much help during development. I don't know that there's an automated way to ensure that mock values are actually valid possible results and still get the benefits of using mocks, so it's something to keep an eye on.

## 7.7 Comparing Mock Object Libraries

Now that we've spent some time exploring how mock objects work using Mocha, let's take a brief look at the various ways that the other popular Ruby mock libraries manage similar tasks. There are four packages that are currently popular:

FlexMock
: This is the original Ruby mock object package.

Mocha
: We've already discussed this at some length. It's quasi-official for Rails in that it is used in Rails core.

RSpec
: The RSpec library, described in more detail in Chapter 12, *RSpec*, on page 188, defines its own mock object package

RR
: Pronounced "Double Ruby," it's the newest entry, with a more concise syntax than the other packages and unique advanced features.

# 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

**Home page for Rails Test Prescriptions**
http://pragprog.com/titles/nrtest
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/nrtest.

# 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) |