The Pragmatic Programmers

Rails Test Prescriptions

Keeping Your Application Healthy

Noel Rappin

Edited by Colleen Toporek

Under Construction The book you're reading is still under development. As part of our Beta book program, we're releasing this copy well before a normal book would be released. That way you're able to get this content a couple of months before it's available in finished form, and we'll get feedback to make the book even better. The idea is that everyone wins!

Be warned. The book has not had a full technical edit, so it will contain errors. It has not been copyedited, so it will be full of typos and other weirdness. And there's been no effort spent doing layout, so you'll find bad page breaks, over-long lines with little black rectangles, incorrect hyphenations, and all the other ugly things that you wouldn't expect to see in a finished book. We can't be held liable if you use this book to try to create a spiffy application and you somehow end up with a strangely shaped farm implement instead. Despite all this, we think you'll enjoy it!

Download Updates Throughout this process you'll be able to download updated ebooks from your account on http://pragprog.com. When the book is finally ready, you'll get the final version (and subsequent updates) from the same address.

Send us your feedback In the meantime, we'd appreciate you sending us your feedback on this book at http://pragprog.com/titles/nrtest/errata, or by using the links at the bottom of each page.

Thank you for being part of the Pragmatic community!

► **Andy & Dave**

Rails Test Prescriptions

Keeping Your Application Healthy

Noel Rappin

The Pragmatic Bookshelf
Raleigh, North Carolina Dallas, Texas

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at http://www.pragprog.com.

Printed in the United States of America.

ISBN-10: 1-934356-64-6

ISBN-13: 978-1-934356-64-7

Printed on acid-free paper.

B4.0 printing, July 8, 2010

Version: 2010-7-7

Contents

Changes in the Beta Version

Beta 2: May 10

The primary addition for this beta release is two new chapters: Chapter 13, *Testing Workflow with Integration Tests*, on page 175, and Chapter 14, *Clean Up Integration Tests with Webrat*, on page 184. The setup appendix, Appendix A, on page 245, has also been reinstated. The Appendix starts the process of converting the book to Rails 3, however the Rails 3 instructions there are not yet complete. Several errata typos noticed on the forum have been cleared.

A small handful of you inadvertently received the entire book in Beta 1, rather than just the first ten chapters. The extra chapters have been temporarily removed, and will be reinstated when they are complete and fully edited.

Beta 3: May 24

The Cucumber chapter is new in this beta. Also several errata have been cleared.

Beta 4: July 8

There are two new chapters, Chapter 17, *Using Rcov to Measure Test Coverage*, on page 217 and Chapter 18, *Beyond Coverage: What Makes Good Tests?*, on page 229. In addition, the table of contents for Part Six of the book has been restructured a little. Nothing has gone away, but a couple of the smaller chapters have been consolidated together. In addition, several errata have been cleared.

Part I

Getting Started with Testing in Rails

Chapter 1

The Goals of Automated Developer Testing

1.1 A Testing Fable

Imagine two programmers working on the same task. Both are equally skilled, charming and delightful people, and motivated to do a high quality job as quickly as possible. The task is not trivial, but not wildly complex: for the sake of discussion, we'll say it's behavior based on a new user registering for a website and entering pertinent information.

The first developer, who we'll call Ernie,[1] says, "This is pretty easy, and I've done it before. I don't need to write tests." And in five minutes Ernie has a working method ready to verify.

Our second developer is, of course, named Bert. Bert says, "I need to write some tests."[2] Bert starts writing a test, and in five minutes, he has a solid test of the new feature. Five minutes more, Bert also has a working method ready to verify. Because this is a fable, we are going to assume that Ernie is allergic to automated testing, while Bert is similarly averse to manually running against the app in the browser.

At this point, you no doubt expect me to say that even though it has taken Bert more time to write the method, Bert has written code that is more likely to be correct, robust, and easy to maintain. That's true.

1. Because that's his name.
2. Actually, if Bert is really into Agile, he probably says, "Who am I going to pair with," but that's an issue for another day.

But I'm also going to say that there's a good chance Bert will be done before Ernie.

Observe our programmers a bit further. Ernie has a five-minute lead, but both people need to verify their work. Ernie needs to test in a browser; we said the task requires a user to log in. Let's say it takes Ernie one minute to set up the task and run the action in his development environment. Bert verifies by running the test—that takes about 10 seconds. (Remember, Bert only has to run one test, not the entire suite.)

Let's say it takes each developer three tries to get it right. Since running the test is faster than verifying in the browser, Bert gains a little bit each try. After verifying the code three times, Bert is only two and half minutes behind Ernie. [3]

At this point, with the task complete, both break for lunch (a burrito for Bert, an egg salad sandwich for Ernie, thanks for asking). After lunch, they start on the next task, which is a special case of the first task. Bert has most of his test setup in place, so writing the test only takes him two minutes. Still, it's not looking good for Bert, even after another three rounds trying to get the code right: he's still a solid two minutes behind Ernie.

Bear with me one more step and we'll get to the punch line. Ernie and Bert are both conscientious programmers, and they want to clean their code up with a little refactoring. Now Ernie is in trouble: each time he tries the refactoring, he has to spend two minutes verifying both tasks, but Bert's test suite still takes only about 10 seconds. After three more tries to get the refactoring right, Bert finishes the whole thing, and checks it in three and a half minutes ahead of Ernie.[4]

My story is obviously simplified, but let's talk a moment about what I didn't assume. I didn't assume that the actual time Bert spent on task was smaller, and I didn't assume that the tests would help Bert find errors more easily—although I think that would be true.[5] The main

3. In a slight nod to reality, let's assume that both of them need to verify one last time in the browser once they think they are done. Since they both need to do this, it's not an advantage for either one.
4. Bert then catches his train home, and has a pleasant evening. Ernie just misses his train, gets caught in a sudden rainstorm, and generally has a miserable evening. If only he had run his tests...
5. Of course, I didn't assume that Bert would have to track down a broken test in some other part of the application, either.

point here is that it's frequently faster to run multiple verifications of your code as an automated test than to always check manually. And that advantage is only going to increase as the code gets more complex.

There are other, beneficial side effects of having accurate tests—better-designed code in which you'll have more confidence—but the most important benefit is that if you do it well, you'll notice that your work goes faster. You may not see it at first, but at some point in a well-run test-driven project, you'll notice fewer bugs, and that those bugs that do exist are easier to find. You'll notice that it's easier to add new features and easier to modify existing ones. As far as I'm concerned, the only code quality metric that has any validity is how easy is it over time to find bugs and add new behavior.

Of course, it doesn't always work out that way. The tests might have bugs. Environmental issues may mean things that work in a test environment won't work in a development environment. Code changes will break tests. Adding tests to already existing code is a pain. Like any other programming tool, there are a lot of ways to cause yourself pain with testing.

That's where this book comes in. I'm assuming you are a Rails programmer, and you've often heard that Rails programs should be tested. You've probably even done some testing yourself. Maybe you got frustrated with it, maybe you kept up with it—but you are looking for some guidance on making testing work for you.

Over the course of this book, we'll go through the tools that are available for writing tests, and we'll talk about them with an eye toward making them useful in building your application. This is Rails, so naturally I've got my own opinions, but all the tools have the same goal: help you to write great applications that do great things, and still catch the train home.

1.2 The Power of Testing First

The way to succeed with Test-Driven Development (TDD) is to trust the process. The classic process goes like this:

1. Create a test. The test should be short, and test for one thing in your code. The result of the test should be deterministic.

2. Make sure the test fails. Verifying the test failure before you write code helps ensure that the test really does what you expect.

3. Write the simplest code that could possibly make the test pass. Don't worry about good code yet. Don't look ahead. Sometimes, just write enough code to clear the current error.

4. Refactor. After the test passes. Clean up duplication. Optimize. This is where design happens, so don't skip this. Remember to run the tests at the end to make sure you haven't changed any behavior.

Repeat until done. This will, on paper at least, ensure that your code is always as simple as possible, and always is completely covered by tests. We'll spend most of the rest of this book talking about the details of step one and how to use Rails tools to write useful tests.

If you use this process, you will find that it changes the structure of the code you write. The simple fact that you are continually aligning your code to the tests results in code that is made up of small methods, each of which does one thing. These methods tend to be loosely coupled and have minimal side-effects.

As it happens, the hallmark of well-designed code is small methods that do one thing, are loosely coupled, and have minimal side-effects. I used to think that was kind of a lucky coincidence, but now I think it's a direct side-effect of building the code in tandem with the tests. In effect, the tests act as a universal client for the entire code base, guiding all the code to have clean interactions between parts because the tests, acting as a third-party interloper, have to get in between all the parts of the code in order to work.

This theory explains why writing the code first causes so much pain when writing tests even if you just wait a little bit to get to the tests. When the tests are written first, or in very close intertwined proximity to the code, then the tests drive the code's structure and enable the code to have the good high-cohesion/low-coupling structure. When the tests come later, they have to conform to the existing code, and it's amazing how easily and quickly code written without tests will move toward low-cohesion and high-coupling forms that are much harder to write tests for. If your only experience with writing unit tests comes only long after the initial code was written, the experience was likely quite painful. Don't let that turn you away from a TDD approach, the tests and code you will write with TDD are much different.

A Historical Parallel

What's a Rails book without a good Franklin Roosevelt anecdote, right?

There's a widely told and probably apocryphal story about FDR meeting with a group of activists pushing a reform agenda—exactly what the group wanted seems to have been lost to history.

Anyway, when they were done with the meeting, FDR is supposed to have said to them, "I agree with you, I want to do it, now go make me do it."

Ignore for the moment the question of whether this statement makes sense as politics: it makes perfect sense as a test-driven development motto. Your requirements determine what your applications *want* to do. Your tests *make* the application do it.

1.3 What TDD Is and What TDD Is Not

What Is TDD Good For?

The primary purpose of this style of testing where the developer is writing tests for her own benefit is to improve the structure of the code. That is, TDD is a software development technique rather than a complete testing program. (Don't believe me, ask Kent Beck, who is most responsible for TDD as a concept and who said, "Correctness is a side effect" on a recent podcast.)[6]

Automated developer tests are a wonderful way of showing that the program does what the developer thinks it does, but they are a lousy way of showing that what the developer thinks is what the program actually should do. "But the tests pass!" is not likely to be comforting to a customer when the developer's assumptions are just flat-out wrong.[7]

Automated developer testing is not a substitute for *acceptance testing* with users or customers (which can itself be partially automated via something like Cucumber), or some kind of QA phase where users or testers pound away at the actual program trying to break something.

6. http://twit.tv/floss87. Good interview, recommended.
7. He says, speaking from painful experience...

This goal can be taken too far, however. You sometimes see an argument against Test-Driven Development that runs something like this: "The purpose of testing is to verify that my program is correct. I can never prove this with 100% certainty. Therefore, testing has no value." (RSpec and Behavior-Driven Development were created, in part, to combat this attitude.) Ultimately, though, testing has a lot of positive benefits for coding, even beyond verification.

Preventing regression is often presented as one of the paramount benefits of a test-driven development process. And if you are expecting me to disagree out of spite, you're out of luck. Being able to squash regressions before anybody outside of your laptop sees them is one of the key ways in which strict testing will speed up your development over time. In order to make this work best, of course, you need good tests.

Another common benefit you may have heard in connection with automated tests is that they provide an alternate method of documenting your program. The tests, in essence, provide a detailed, functional specification of the behavior of the program.

That's the theory. My experience with tests acting as documentation is mixed, to say the least. Still, it's useful to keep this in mind as a goal, and most of the things that make tests work better as documentation will also make the tests work better, period.

To make your tests effective as documentation, focus on giving your tests descriptive names, keeping tests short, and refactoring out common setup and assertion parts. The documentation advantage of refactoring is removing clutter from the test itself—when a test has a lot of raggedy setup and assertions, it can be hard for a reader to focus on the important functional part. Also, with common features factored out, it's easier to focus on what's different in each individual test.

In a testing environment, blank-page problems are almost completely nonexistent. I can always think of *something* that the program needs to do, so I write a test for that. When you're working test-first, the actual order in which pieces are written is not so important. Once a test is written, the path to the next one is usually clear, and so on, and so on....

When TDD Needs Some Help

Test-Driven Development is very helpful, but it's not going to solve all of your development problems by itself. There are areas where developer testing doesn't apply or doesn't work very well.

I mentioned one case already—developer tests are not very good at determining whether the application is behaving correctly according to requirements. Strict TDD is not very good at acceptance testing. There are, however, automated tools that do try to tackle acceptance testing. Within the Rails community, the most prominent of these is Cucumber, see Chapter 15, *Acceptance Testing with Cucumber*, on page 194. Cucumber can be integrated with TDD—you'll sometimes see the acronym "ATDD" for Acceptance Test-Driven Design. That's a perfectly valid and useful test paradigm, but it's an extension of the classic TDD process.

Testing your application assumes that you know the right answer. And while you will have clear requirements or a definitive source of correct output some of the time, other times you don't know what exactly the program needs to do. In this exploratory mode, TDD is less beneficial, because it's hard to write tests if you don't know what assertions to make about the program. Often this happens during initial development, or during a proof of concept. I find myself in this position a lot when view testing—I don't know what to test for until I get some of the view up and visible.

In classic Extreme Programming parlance, this kind of programming is called a *spike*, as in, "I don't know if we can do what we need with the Twitter API; let's spend a day working on a spike for it." When working in spike mode, TDD is generally not used, but it's also the expectation that the code written during the spike is not used in production, it's just a proof of concept.

When view testing, or in other non-spike situations where I'm not quite sure what output to test for, I tend to go into a "test-next" mode, where I write the code first, but in a TDD-sized small chunk, and then immediately write the test. This works as long as I make the switch between test and code frequently enough to get the benefit of having the code and test inform each other's design.

TDD is not a complete solution for verifying your application. We've already talked about acceptance tests, but it's also true that TDD tends to be thin in terms of the amount of unit tests written. For one thing, a strict TDD process would never write a test that you expect to pass.

Words to Live By

Any change to the logic of the program should be driven by a failed test.

A test should be as close as possible to the associated code.

If it's not tested, it's broken.

Testing is supposed to help for the long term. The long term starts tomorrow, or maybe after lunch

It's not done until it works.

Tests are code, refactor them too.

Start a bug fix by writing a test.

In practice, though, I do this all the time. Sometimes I see and create an abstraction in the code but there are still valid test cases to write. In particular, I'll often write code for potential error conditions even if I think they are already covered in the code. It's a balance, because you lose some of the benefit of TDD by creating too many test cases that don't drive code changes. One way to keep the balance is to make a list of the test cases before you start writing the tests—that way you'll remember to cover all the interesting cases.

And hey, some things are just hard. In particular, some parts of your application are going to be very dependent on an external piece of code in a way that makes it hard to isolate them for unit testing. Mock objects, described in Chapter 7, *Using Mock Objects*, on page 105, can be one way to work around this issue. But there are definitely cases where the cost of testing a feature like this is higher then the value of the tests. To be clear, I don't think that is a common occurrence, but it would be wrong to pretend that there's never a case where the cost of the test is too high.

1.4 Coming Up Next...

The next chapter is an introduction to the basic concepts of Ruby and Rails testing using Test::Unit. The following two chapters are going to be an extended walk through the basics of using TDD in a Rails program. We'll create a new Rails program and add some features to it using

TDD. After that, the next two sections of the book will explore Rails model testing and Rails user-facing testing in a lot more detail. Model testing chapters will discuss Rails models, of course, but also how to get sample data into your test, and using mock objects to simulate Rails objects. The user-facing section will cover controllers, view testing, testing your Rails helpers, and testing Ajax calls with Blue Ridge

With the basics covered, we'll then talk about RSpec and Shoulda, which are two large frameworks that add lots of power and flexibility to your tests. After that, we'll zoom out and talk about testing your entire program end-to-end using Rails integration tests, browser simulators like Webrat or Selenium, and Cucumber for acceptance tests. The last section of the book will talk about different ways to evaluate and improve your tests, including coverage testing, the autotest test runner, suggestions for improving test performance, and troubleshooting tests.

Ready? Me too.

Chapter 2

The Basics of Rails Testing

Let's start at the very beginning, which is, I hear, a very fine place to start. For Rails testing, the beginning is the set of tools, conventions, and practices that make up the test facilities provided by Core Rails. The basic Rails test functionality consists of the standard Ruby library Test:Unit, plus the Rails standard mechanisms for placing tests in the project, adding data to tests, and running tests.

All the basic features of building Rails tests are covered in this chapter. Once we have that foundation in place, we'll use these features in a Rails Test-Driven-Development process in Chapter 3, *Writing Your First Tests*, on page 39.

2.1 What's a Test

The individual test is the most basic unit of Rails testing. There are two ways to define an individual test in Rails, and they are functionally equivalent. In the older style, any method whose name starts with test_ is considered a test:

```
def test_that_a_new_user_has_a_valid_name
  # test logic here
end
```

In the newer style (Rails 2.2 and up), a more declarative syntax can be used with the method test():

```
test "that a new user has a valid name" do
  # test logic here
end
```

There's Always Some Version Confusion

Here's the short answer to what versions of different software we're talking about: by default, Rails 3, Test::Unit 1.3, and Ruby 1.8.7. Where Rails 2.x is substantially different, I'll note that. For the most part, the differences between Ruby 1.8.7 and 1.9.1 don't significantly impact the code in this book.

We're dealing with three separate entities that are in the process of transitioning between major versions. Rails is moving between the 2.x version stream and the 3.x versions. The Ruby language version 1.9.x is coming down the pike, and a recent fork of Test::Unit leaves 1.3 and a 2.0 version in the wild.

To sweeten the deal, Ruby 1.9.1 uses a different default test library called miniunit, which is mostly a smaller, faster replacement for Test::Unit 1.3. However, Ruby 1.9.1 keeps a module called Test::Unit as a wrapper around miniunit, for backwards compatibility purposes.

For its part, Rails smooths out the difference between miniunit and Test::Unit 1.3, and adds its own features on top of either. For our purposes, we don't need to worry about the difference, and to minimize confusion (too late?), I'll continue to refer to that library as Test::Unit. Since I'm boldly assuming you are writing a Rails application, I'm not going to sweat the difference between what's in Rails and what's in Test::Unit. Test::Unit 2.0 doesn't seem to have much of a constituency at the moment, so I'm going to ignore it.

Behind the scenes, those two definitions will result in the exact same test being executed.

You can't just slap a test method inside any old class. The tests you write need to be defined inside a subclass of Test::Unit::TestCase. Rails provides its own generic subclass called ActiveSupport::TestCase. You would use ActiveSupport::TestCase as the parent class of any of your Rails tests. Rails also defines its own subclasses of ActiveSupport::TestCase for testing controllers and integration testing.

2.2 What Goes in a Test

Anything you want, followed by some assertions.

When testing from within Rails, the entire Rails environment is automatically loaded as part of test startup, so any part of your Rails application along with any required plugin or gem is available. In general, a specific test class is tied to a specific application class, and tests in that test class exist to validate the behavior of the matching application class. That's good practice, and is enforced by the structure of Test::Unit or Rails. It's also good practice, and if you find yourself testing functionality outside the class your test is tied to, you should probably rethink your approach.

Inside each test, you are generally trying to do four things:

- Set up the data needed for the test. As a general rule, create as few data objects as possible for each test—it'll make the tests run faster. If you find yourself creating a lot of objects, it's often a sign you aren't testing small enough units.
- Perform the action that triggers the behavior being tested. In a Rails controller test, for example, it's generally the call to a controller action. In a model test, it's a call to the model method under test.
- Perform one or more assertions to verify that the behavior triggered in the previous step had the expected results. This usually involves making assertions about the state of the system after the action, although another style of testing, discussed in more detail in Chapter 7, *Using Mock Objects*, on page 105, makes assertions about the behavior of the application when the method under test is invoked. In either case, this is a step you want to take care to get right—a badly written assertion can leave you with a test that does not accurately reflect system behavior. A test that fails when the behavior is working is bad, but one that passes even though the underlying behavior is broken is even worse.
- Teardown any data structures that need to be removed before the next test runs. In Rails, this step is rarely needed, as most of the major bookkeeping—resetting the database state, for example—is handled by the framework. However, there are some testing tools that require a teardown statement in order to keep each individual test nicely independent.

Setup, teardown, and assertions all get some special structure from the testing framework. Let's talk about assertions first. The vanilla Test::Unit defines about 20 different methods that assert the presence

or absence of a particular state. From a programmer perspective, the simplest of these methods is the plain assert() method, which takes a boolean argument. If the argument is true, the assertion passes; if the argument is false, the assertion fails, and the current test stops execution at that point. Normally, you'd have an expression evaluating to a boolean as the argument to assert(), as in this example verifying how long a man's legs should be:

```
test "ask abe the length of a man's legs" do
  @user = User.new
  assert (@user.leg.length == "long enough to reach the ground")
end
```

I'll mention this once here, and you can apply it to all the assertions discussed in the next few pages: assert() and all the other Test::Unit assertion methods takes an optional last argument with a string message to be displayed on failure. In most cases, the default message and the resulting stack trace is plenty good enough to diagnose failures, so the messages are rarely used.

From a user perspective the simplest method is assert(), but inside the Test::Unit code, almost everything is built on top of assert_block(), which takes a no-argument block and passes if the block evaluates to true.

```
test "ask abe the length of a man's legs" do
  @user = User.new
  assert_block { @user.leg.length == "long enough to reach the ground" }
end
```

The most commonly used assertion is assert_equal(), which takes two arguments, an expected value and the actual computed value. The method passes if the two arguments are equal using Ruby's == operator.

```
test "ask abe the length of a man's legs" do
  @user = User.new
  assert_equal "long enough to reach the ground", @user.leg.length
end
```

While it is functionally irrelevant which order the arguments come in, the error message that you will receive on failure assumes that the first argument is the expected value and the second argument is the actual calculated value. Trust me, the only thing that comes from mixing those two is confusion when you are trying to track down a failed test. You can negate this assertion with the converse method assert_not_equal():

Those are the most commonly used assertions, but Test::Unit defines a handful of others. All of these take an optional message as a last

argument, which I'm leaving off because I want you not to use it. Let's take these in groups:

- assert_in_delta(expected, actual, delta)()

  Like assert_equal(), but for floating point numbers. Passes if the two floating points are within the delta value of each other.

- assert_instance_of(klass, object)()

  assert_kind_of(klass, object)()

  Passes if the object and the class have the relationship implied by the name of the method.

- assert_match(pattern, string)()

  assert_no_match(pattern, string)()

  Like assert_equal(), but for regular expressions.

- assert_operator(left, operator, right)()

  Never actually seen this in the wild. Passes if the left and right objects have the relationship stated by the operator, as in assert_operator 6, :<, 10. Uses send() to send the operator to the left-hand operand.

- assert_nil(object)()

  assert_not_nil(object)()

  At the risk of sounding snobby, these two really should be self-evident.

- assert_raise(*args, &block)()

  assert_nothing_raised(*args, &block)()

  The argument to the positive method is an exception class, and then the assertion passes if the associated block of code raises that exception. The negative method passes if the block does not raise an exception. The negative method does not care what kind of exception is raised.

- assert_same(expected, actual)()

  assert_not_same(expected, actual)()

  Like assert_equal(), but for actual object equality.

- assert_respond_to(object, method)()

Passes if object.respond_to?(method) is true.

- assert_send(array)()

  Really odd method that takes an array of the form [receiver, method, argument_list], and passes if the snippet receiver.method(argument_list) is true.

- assert_throws(symbol, &block)()

  assert_nothing_thrown(&block)()

  Like assert_raise, but for Ruby's rarely used catch/throw mechanism.

2.3 Setup and Teardown

Let's look at a pair of tests. The exact functionality isn't important right now; we're interested in the structure of the test:

```
test "a user should be able to see an update within the project" do
  fred = User.new(:name => "Fred")
  barney = User.new(:name => "Barney")
  project = Project.new(:name => "Project Runway")
  project.users << fred
  project.users << barney
  barney.create_status_report("I'm writing a test")
  assert_equal("I'm writing a test", fred.project_statuses[0].text)
end

test "a user should not be able to see from a different project" do
  fred = User.new(:name => "Fred")
  barney = User.new(:name => "Barney")
  project = Project.new(:name => "Project Runway")
  other = Project.new(:name => "Project Other")
  project.users << fred
  other.users << barney
  barney.create_status_report("I'm writing a test")
  assert_equal(0, fred.project_statuses.count)
end
```

These tests share some common code. The common setup is only a few lines, but a real set of tests could wind up with far more duplication. Your first signal that something is wrong is that the setup has probably been copied and pasted from one test to the next. Copying and pasting multiple lines of code is almost always a heads-up to at least consider what you are doing, and see if there's a commonality that you can refactor.

The classic way of managing duplicate setup using Test::Unit is to move the common code to the setup() method, which is automatically called by the test framework before each test:

```
def setup
  fred = User.new(:name => "Fred")
  barney = User.new(:name => "Barney")
  project = Project.new(:name => "Project Runway")
  project.users << fred
end

test "a user should be able to see an update from a friend" do
  project.users << barney
  barney.update_status("I'm writing a test")
  assert_equal("I'm writing a test", fred.project_statuses[0].text)
end

test "a user should not be able to see an update from a non-friend" do
  other = Project.new(:name => "Project Other")
  other.users << barney
  barney.update_status("I'm writing a test")
  assert_equal(0, fred.project_statuses.count)
end
```

Moving the common setup code to the setup() method solves a couple of problems. The setup() code is automatically executed before each test, guaranteeing that each test is executed in the same environment. Also, moving the setup out of the test method makes it easier to write each individual test, and also easier to follow the unique purpose of each test. There's some debate over whether the setup methods really are clearer—there's also a school of thought that says that moving anything into a setup method makes the test harder to follow. In general, if the setup gets too complex, you start to have problems—but in most model and controller tests, you can keep the common setup simple enough to be clear.

Over time, the setup() method, like any initializer, can become cluttered with multiple independent small setups jammed together in the same method. And don't forget that right below the sign that says "Don't Repeat Yourself" is another one that says "A Method Should Do Exactly One Thing" (the acronym AMSDEOT is nowhere near as catchy as DRY, though).[1] A confused setup method violates the AMSDEOT principle. Relief for this arrived in Rails 2.2, which converted setup code to a block declaration similar to the way that before_filter() is handled in

1. I'm also a big fan of CAPITROAE: "Cut And Paste Is The Root Of All Evil."

controllers. In Rails 2.2, you can declare methods to be run during setup by using the setup() call and placing something like this in your test class or in test/test_helper.rb:

```
setup :setup_users

def setup_users
  @fred = User.new(:name => "Fred")
  @barney = User.new(:name => "Barney")
end
```

What's particularly nice about this is that—as with before filters—you can have multiple setup blocks or methods, and they will all be executed before each test:

```
setup :create_fred
setup :create_barney

def create_fred
  @fred = User.new(:name => "Fred")
end

def create_barney
  @barney = User.new(:name => "Barney")
end
```

Setup methods are executed in the order in which they are declared. Calls to setup() in the test/test_helper.rb file will always be declared and thus executed before any method in the actual test file.

You can also define the setup as a block:

```
setup do
  User.create(:name => "Fred")
  User.create(:name => "Barney")
end
```

This is not recommended, though, because the inside of the block is evaluated in class context—you can initialize global or class settings, but you can't create instance variables that are accessible from your tests.

There is a similar mechanism to control what happens at the end of tests. The teardown() method, has the same declaration rules as setup():

```
teardown :reset_globals

def reset_globals
  #whatever
end
```

Because Rails handles the reset of database data in testing, it's not all that common to see teardowns used in Rails testing. Normally you'd use it to reset third-party tools outside of Rails. For example, certain mock object packages, such as Flexmock, require a method to be called at teardown to reset object status.

Eventually, you are likely to have one or more basic setup methods that are shared among multiple controller or unit tests. If the setup methods start to crowd out the test_helper.rb file, you can create a test/setup_methods.rb file with a module containing all the setup methods:

```
module SetupMethods
  def setup_users
    fred = User.new(:name => "Fred")
    barney = User.new(:name => "Barney")
  end

  # put more setup methods here
end
```

Then the test class can include the module, and declare any relevant methods as setups:

```
class UserTest < ActiveSupport::TestCase

  include SetupMethods
  setup :setup_users

  test "my test" do
  end
end
```

The include and the setup call can also go in the test/test_helper.rb file.

2.4 What Can You Test in Rails?

When testing a Rails application, Rails specifies the default location of tests based on the class being tested. All Rails tests go in the test directory of the application. Rails assumes a consistent relationship between controllers, models, and their test files. This makes it easy to know where to put new tests, and enables external tools like autotest (see Chapter 21, *Using Autotest as a Test Runner*, on page 243) to know what tests to run when an application file changes. Figure 2.1, on the following page, shows a screenshot of the app directory of a Rails application and its associate test directory.

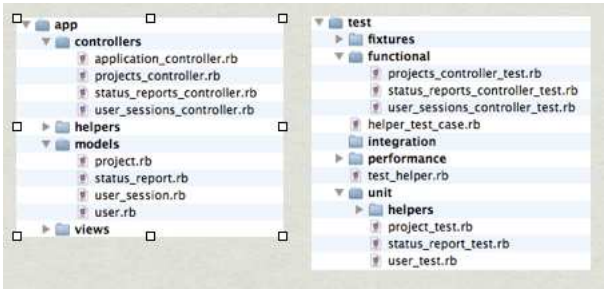


Figure 2.1: Directory Comparison

Tests for Rails models are in the test/unit directory. Each test is named for its model, so by convention the test file test/unit/user_test.rb contains the class UserTest and is expected to correspond to the Rails model file in app/models/user.rb. In Rails 2.0 and up, unit tests are subclasses of the Rails ActiveSupport class ActiveSupport::TestCase, which is a subclass of the Ruby standard library class Test::Unit::TestCase. When you create a model using a Rails generator, the associated unit test class is also created for you.

Functional tests and Rails controllers have a similar relationship. Tests for Rails controllers are in the test/functional directory. The file app/controllers/users_controller.rb contains the class UsersController. The tests for that class are in test/functional/users_controller_test.rb, and the test class is named UsersControllerTest. In current versions of Rails, all controller tests are subclasses of the Rails class ActionController::TestCase, which is a subclass of the same ActiveSupport::TestCase used for models. Anytime you create a controller using one of the three or four standard Rails generators that include controllers, an associated functional test file is created for you. (In Rails 2.3 and up, a test file is also created for the helper module, and stored in test/unit/helpers.)

Rails also creates a file called test/test_helper.rb, which contains features and settings common to all of your tests. This file is required by any Rails test file.[2] The provided file re-opens the class ActiveSupport::TestCase to allow you to add your own methods, which are then available to all your tests. Typically, this involves initialization and tear-

2. Not automatically, however: Rails places the 'require' statement at the top of its generated files. If you create your own files, you need to add the statement yourself.

down, complex data setups, and complex assertions. Other sections in this book will cover those possibilities in more detail.

There are two other kinds of tests in the standard Rails toolbox. The first, *integration tests*, are perhaps the most ignored feature of Rails testing. By design, integration tests are used to test sequences of events that span multiple actions or controllers. However, they don't easily map to the various test-first methodologies, and Rails developers tend to overlook them. That's unfortunate, because integration tests are a good way to validate complex interactions in an application, as well as ensure that there are no holes in the controller tests. Integration tests are created using a Rails generator script/generate integration_test, and are not automatically created by any other Rails generator. Integration tests will be discussed in more detail in Chapter 13, *Testing Workflow with Integration Tests*, on page 175.

Performance tests are different than the other automated test types. They are not intended to verify the correctness of your program; instead, they give access to profiling information about the actions called during the test. Essentially, performance tests are wrappers around the ruby-prof profiler, but if you've ever tried to get ruby-prof working, you'll appreciate the help. Performance tests are created using the Rails generator, script/generate performance_test, and are not automatically created by other Rails generators. Unlike the other Rails tests—functional, unit, and integration—performance tests are not automatically run by the Rails test runners.

2.5 What Happens When Tests Run?

Each time you run a test task in Rails, the following steps take place. Figure 2.2, on the following page shows the entire work flow.

The test database, as determined by the entry with the symbol test in the config/database.yml file, is cleared of all data.[3]

Based on which test task is being run, a list of test files that matches the criteria for the task is generated. For example, running just the rake test:functionals task generates a list of all the test files in the test/functionals directory, and no others.

3. If you run your test via the command-line rake task, then Rails will automatically apply any pending migrations. If you run via a different method—an IDE, for example—you may need to apply pending migrations to the test environment yourself.

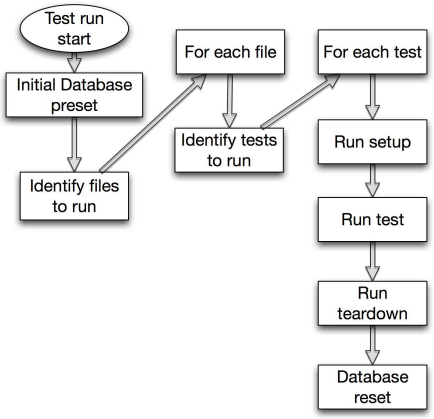


Figure 2.2: Test Flow

Once the list of test files is created, each test file is loaded one by one. Like any other Ruby file, loading the file causes any module or class level code to be interpreted. Although it's pretty rare to have any class-level initialization in a test file, you'll sometimes have additional classes in the file besides the test class itself (for example, a specialized mock object class).

After a file is loaded, all the test methods in the file are identified. In versions of Rails before 2.2, a test method is any method in the test class that starts with test_. In Rails 2.2 and up, an additional test method is added that allows you to also create tests with a more natural block syntax (test "should pass this" do end). Test add-ons like Shoulda also offer different ways to define tests.

For each test method that has been identified by the test framework, the test method is executed. However, execution does not mean just running the test method itself—the test framework also executes setup code before the test method and teardown code after the test method. Here are the steps for running an individual test method:

1. Reset fixture data. By default, fixtures are loaded once per test run inside a database transaction. At the end of each test method, the transaction is rolled back, allowing the next test to continue with a pristine state. More details on fixture loading are available in Section 6.2, *Loading Fixture Data*, on page 81.

2. Run any defined setup blocks. In versions of Rails before 2.2, there is only one setup method per test case. In Rails 2.2 and up, multiple setup methods can be declared. Note that setup blocks can be in the actual test class or in any parent classes or included modules. There are more details in Section 2.3, *Setup and Teardown*, on page 26.

3. Run the actual test method. The method execution ends when a runtime error or a failed assertion is encountered. If neither of those happens, then the test method passes. Yay!

4. Run all teardown blocks. Teardown blocks are declared similarly to setup blocks.

5. Rollback or delete the fixtures, as described in step 1. The result of each test is passed back to the test runner for display in the console or IDE window running the test. Typically, failures and errors return stacktraces from the offending point in the code.

2.6 Running the Rails Tests

Rails provides several commands to run all or part of your test suite.

The most common test to run is the rake default testing task, invoked with either rake test or simply rake. The default task combines three subtasks, which can be individually invoked as rake test:functionals, rake test:units, and rake test:integration. Each of these tasks runs any file matching the pattern *_test.rb in the appropriate directory. The command-line output looks something like this:

```
$ rake
(in /Users/noel/Projects/huddle)
/System/Library/Frameworks/Ruby.framework/Versions/1.8/usr/bin/ruby
-Ilib:test
"/Library/Ruby/Gems/1.8/gems/rake-0.8.3/lib/rake/rake_test_loader.rb"
<ALL TESTS>
Loaded suite
/Library/Ruby/Gems/1.8/gems/rake-0.8.3/lib/rake/rake_test_loader
Started
.....EE...E.................E.E.................
Finished in 2.138619 seconds.
```

Each successful test method is represented by a dot. If a test triggers an actual exception or error, it's represented by an E; if the test merely caused an assertion inside the test to fail, it's represented by an F. After

the run-through, each error and failure will have a message and a stack trace.

Rails provides two helpful but often-overlooked convenience tasks for testing the files you are currently working on. The task rake test:recent looks for any controller or model file that has changed in the last 10 minutes, and runs the associated functional or unit test. The task rake test:uncommitted works similarly on any controller or model file that has been changed since you last committed back to your source control repository. You must be using Subversion or Git to take advantage of the uncommitted task.

To run performance tests, use rake test:benchmark or rake test:profile. The two test types differ primarily in the output they present. A *benchmark test* outputs about five simple values for each performance run, including elapsed overall time and memory used. On first run, each test also generates a CSV file for output values. Further benchmark test runs append values to the CSV file, allowing for easy visualization of performance changes over time.

A *profile test* splits the timing data for each test based on the methods in which the test execution takes place, and returns the amount of time spent in each method for the purpose of trying to determine where your application bottlenecks are. The output of a profile test can either be a list of methods and their data, or a call graph associating each method with the methods that call it and the methods it calls.

Tests for all your plugins can be run using rake test:plugins. Individual test files can also be run just by invoking them from the command line, as in ruby test/unit/user_test.rb. (In recent versions of Rails, you may need to adjust your Ruby load path such that the require test_helper line at the beginning of the test files finds the test helper file.)

If you are using an IDE such as Eclipse or NetBeans, the IDE should provide a command to run tests within the IDE itself—typically, this will either run the rake command line in a console window (NetBeans), or invoke a custom test runner that essentially does the same thing, but prettier (Eclipse, RubyMine). The IDE should also have commands for running an individual test file or an individual test method.

2.7 More Info: Getting Data into the Test

Often, testing a feature properly requires data to be created in order to build a meaningful test setup. A reporting feature might best be tested with a number of different data items that fill different columns in a report. Or a social networking feature might require the creation of several users in various relationship permutations for proper testing.

There are a number of ways to conveniently create sample data for tests. Rails core offers an easy-to-use, if somewhat limited, feature called *fixtures*. Generically, a fixture is any predefined set of data used by multiple tests as a baseline. Fixtures in Rails are a core mechanism for defining and using known data in tests. Specifically, every ActiveRecord model gets an associated set of fixture data in test/fixtures, where data objects can be specified in YAML format.

Rails fixtures give you a consistent, potentially complex data set that is automatically created before each test. Although fixtures have been a part of Rails since the beginning, a number of Rails developers have decided to use other tools for generating sample data, especially in complex applications. We're going to largely sidestep that discussion here, and focus on fixtures because they are a) easy to use and b) part of Rails core. You will find much more discussion on the positives and negatives of using fixture data in Chapter 6, *Creating Model Test Data with Fixtures and Factories*, on page 78.

The directory test/fixtures is expected to contain a YAML file for each ActiveRecord model. So, app/models/user.rb is attached to test/fixtures/user.yml. The fixture files are created automatically by the Rails generators when a model is created. If you create an ActiveRecord model manually, you'll also need to create the associated YAML file. The reverse is also true: if you remove an ActiveRecord file, you need to remove the YAML file. Otherwise, you will be unable to run tests, since Rails will try to load the fixture data into the test database for the missing ActiveRecord class. The data placed in the fixture files is automatically loaded into the test database before each test.[4]

Rails fixtures are described in YAML, which has a strict, nested format: at the top level, each individual model gets a name, followed by a colon.

4. Fixture data can also be loaded into the development database using the rake db:load:fixtures task.

After that, the data for that model is indented, Python-style. Each line starts with the key, followed by a colon, followed by the value, like this.[5]

Download huddle/test/fixtures/projects.yml

```
huddle:
  name: Huddle Project
```

Add this snippet to the test/fixtures/projects.yml file to create a single project (don't delete the existing fixtures in that file; there are Rails-generated controller tests that depend on them).

To start a new model, outdent back to the left edge and start again with a name for the model. Those top-level names, such as huddle in the above file, have no particular meaning in Rails beyond being an identifier to that particular fixture within the test environment. The keys in the YAML file must be columns in the database table for that model (meaning that they can't be arbitrary methods in the Ruby code the way they can be when calling new() or create()).

Once this code is in the YAML file, then an object is generated from it and is loaded into the database before every test,[6] and is by default accessible anywhere, in any test, using the method call projects(:huddle).

Here's some sample fixture data for a status report class we'll use in a later example:

Download huddle/test/fixtures/status_reports.yml

```
ben_tue:
  project: huddle
  user: ben
  yesterday: Worked on Huddle UI
  today: Doing some testing
  status_date: 2009-01-06

ben_wed:
  project: huddle
  user: ben
  yesterday: Did Some Testing
  today: More Testing
  status_date: 2009-01-07
```

There are a couple of things to note about the fixtures. We do not need to specify an id for each fixture; Rails automatically generates one for

5. There's more to YAML syntax, of course, but there's no need to go beyond the basics here.
6. Well... it's a little more complicated than that. See Section 6.2, *Loading Fixture Data*, on page 81 for more.

us. Also, when I said that all the keys had to be database columns, that wasn't strictly true: they can also be associations. If the key is an association, the value represents the name of a fixture in the related table (or a comma-separated list of fixture names for a one-to-many relationship). If we do specify an id in the YAML file, the nifty auto-association feature will not work.[7] Also, as you can see here, dates are automatically converted from string representations.

Fixture files are interpreted by Rails as ERb, so you can loop or dynamically generate data with the full power of Ruby. Also, when copying and pasting YAML data into a text file, remember that YAML files require a specific whitespace layout: the outdented fixture names need to be in the left-most column of the line. Give fixtures meaningful names; it'll help later on. (The Rails default names are a pain in the neck to keep straight.)

Although fixtures have many wonderful qualities—they are always available, relatively easy to set up, and consistent across all tests—they can also be kind of brittle. For example, if you are testing a reporting function, the results you are expecting are sensitively dependent on the makeup of the data in the fixtures. This dependency can cause at least two serious problems. First, if there's a lot of data in the fixtures, the results you are testing against can easily become opaque and hard to verify. Second, if anybody ever adds a fixture to the data, it can easily break all the reporting tests—a bit of a momentum-killer. As such, a number of alternatives for fixtures have been developed that make it easier to define test data specific to individual tests. We'll cover those in Section 6.4, *Using Factories to Fix Fixtures*, on page 83.

2.8 Beyond The Basics

Test::Unit and Rails combine to provide a common set of testing tools that are always available from any Rails application. If you've used other frameworks, particularly from a few years back, you may remember that setting up a group of tests into test cases and test suites was always something of a pain. Not so in Rails, where the design goal is to reduce unnecessary duplication of effort, especially when creating tests. Rails provides standard ways to define tests, a standard location

7. All this fancy id mapping is a Rails 2.0-and-up feature. Before, we had to track the id values manually—which was a total pain for many-to-many join tables.

for each test, a way to add data into the test, and easy ways to run the tests in different combinations.

With this introduction to the basic core of Rails testing in hand, it's time to see a more detailed example. Over the next two chapters, we'll take an almost new Rails application and add new features to it using a test-driven approach.

Chapter 3

Writing Your First Tests

You have a problem. You are the team leader for a development team that is distributed across multiple locations. As an agile development team, your project has a daily stand-up meeting, sometimes called a *scrum*, where everybody briefly describes what they did yesterday, what they plan on doing today, and if anything is blocking them from getting their work done.

However, since your team is geographically distributed, you need to do these scrums via email. That's not the worst thing ever, but it does lead to annoying email threads, and I think we can all do better with a little web application magic. Let's create an application called Huddle, which will support entering and viewing these daily status messages.

Since you are an agile developer, and use test-driven methods, the first thing you should ask is, "What do I test?" Test-driven developers start an application by writing tests. In that spirit, we're going to initiate our tour of Rails testing by writing lots of tests. Specifically, we're going to walk through the first few test-driven feature cycles of the Huddle application, to give you the feel of Test-Driven Development (TDD) using Rails.

We'll use a hands-on approach and walk through the specifics of how to write your first tests. We'll talk about how the practice of working test first improves development, but more importantly, we'll show what working in a test-driven style looks like. This chapter uses the testing tools that are available in core Rails, and is largely limited to common Rails tasks like creating and submitting a web form. At the end of this chapter, you should have a good sense of how TDD development works

A Word About Best Practices

There's a tension in this section between making the introduction to Rails testing as simple and clear as possible, and presenting the tests in what I would consider to be best practices. In particular, many of my regular testing practices depend on third-party tools that we're not going to cover in this walkthrough.

In this chapter, I decided to focus on making testing as easy as possible to explain with decent coding practice, with some discussion of where improvements might come. We'll go over coding style and practice considerations again at the end of the book.

in Rails, and you'll be ready to explore the third-party tools and more detailed topics in the rest of the book.

Appendix A, on page 245, contains the steps for creating the skeleton application we're starting with—including the initial setup, creation of Rails scaffolds, addition of Authlogic, and other things that are necessary to the application, but beside the point for our tutorial. If you'd like to start at the same place, the code samples for this chapter are available for download at http://www.pragprog.com/titles/nrtest/source_code. The code for this application was written and tested against Rails 2.3.4.[1]

We're going to do this in a reasonably strict test-driven style, meaning no new logic will be added to the application except in response to a failing test. We'll be a little more lenient with view code. We're assuming a basic understanding of standard Rails concepts, i.e., you don't need to be told what a controller is. For the moment, we're also going to limit ourselves to test tools provided by core Rails. Later in the book, we'll spend a lot of time covering third party tools, especially in Chapter 12, *RSpec*, on page 173 and Chapter 11, *Write Cleaner Tests with Shoulda and Contexts*, on page 172. But in the name of keeping it simple, we'll start with vanilla core Rails.

1. A future beta of this book will support Rails 3 (as I write this, Authlogic still has Rails 3 issues).

3.1 First Test First

The first question to ask is, "What do I test?" The answer to that question comes from your requirements. Without some sense of what your program should be doing, it's hard to write tests that describe that behavior in code.

The form and formality of your requirements will depend on the needs of your project. In this case, you are your own client, and it's kind of a small project, and we don't have space in this book for military-level precision. So, the informal list of the first three stories in the application looks something like this:

- A user is part of a project. A user can enter his scrum status for that project.
- For the purpose of adding a testable constraint, let's say the user's status report has yesterday's status and today's expected work, and the user must include text in at least one of these items.
- Members of the project can see a timeline of status reports. This one will get covered in Chapter 4, *TDD, Rails Style*, on page 58.

Over the rest of this tutorial, we'll go after these stories one by one. Anytime we add or change the logic of the application, we'll write a test. The exact starting point of the first test is not important (although it's helpful to have at least some sense of where you are going); you can start with any requirement or feature in the program that can be objectively specified.

Our starting point for Huddle is the need to have a status report that is created as part of a project. The report should have all its values, including the date, set correctly. Because I think the code for this feature might be in the StatusReportsController, I'm going to put the test for this feature in test/functional/status_reports_controller_test.rb.

```
Line 1  test "creation of status report with data" do
        -   assert_difference('StatusReport.count', 1) do
        -     post :create, :status_report => {
        -       :project_id => projects(:one).to_param,
        5       :user_id => users(:ben).to_param,
        -       :yesterday => "I did stuff",
        -       :today => "I'll do stuff"}
        -   end
        -   actual = assigns(:status_report)
       10   assert_equal(projects(:one).id, actual.project.id)
        -   assert_equal(users(:ben).id, actual.user.id)
```

```
-   assert_equal(Date.today.to_s(:db), actual.status_date.to_s(:db))
-   assert_redirected_to status_report_path(actual)
- end
```

Let's walk through this test in detail.

Line 3 simulates a post call to the create action of the StatusReportsController. The second argument to this call simulates the URL parameters of the call—effectively, you are setting up the params hash that will be used in the action. As part of that hash, the call references users(:ben), which is a *fixture*, or set of known sample data that can be used in testing. This particular fixture data comes from the Authlogic sample project at http://github.com/binarylogic/authlogic_example, and looks like this:

Download huddle/test/fixtures/users.yml

```
ben:
  login: bjohnson
  password_salt: <%= salt = Authlogic::Random.hex_token %>
  crypted_password: <%= Authlogic::CryptoProviders::Sha512.encrypt(
      "benrocks" + salt) %>
  persistence_token: 6cde0674657a8a313ce952df979de2830309aa4c11ca65805
```

Section 2.7, *More Info: Getting Data into the Test*, on page 35 has more detail on fixtures.

The block that starts in line 2 and ends in line 8 uses the assert_difference() method to assert that there is one more StatusReport object in the database at the end of the block than at the beginning. More plainly, the method is asserting that a new StatusReport instance has been created.

Line 9 uses the Rails test framework assigns() method, which allows access to instance variables set in the controller being tested—in this case, the controller variable @status_report, which should be the newly created instance. You don't need the @ symbol in the argument to the assigns() method.

Starting with line 10, there are three lines asserting that a project, user, and status date are added to the newly created object.[2] Line 13 asserts that the result of the controller call is a redirect to the show page of the newly created StatusReport.

2. If this test is run at just the right moment before midnight, 12 will fail because the date has changed during the running of the code. Section 6.12, *Managing Date and Time Data*, on page 98 discusses working around this problem in more detail.

Although people will certainly quibble with the style and structure of this test, it is a basic, straightforward test of the desired functionality. This is the maximum amount of complexity that I'm comfortable having in a single test. In some cases, the amount of data or validation needed in a test suggests the need to refactor some of the complexity into *setup methods* or *custom assertion methods*.

Rather than start with a controller test, I could start by testing the model behavior. The model test is probably closer to the eventual code that will be written, since good Rails style places complexity in the model. However, I sometimes find that it is easier to specify the desired result when I start testing in via the controller. Another option would be to start with an integration or Cucumber-based acceptance test (described in more detail in Chapter 15, *Acceptance Testing with Cucumber*, on page 194).

We're testing status_date because we know new code will be needed to add that attribute to the object, and we're testing the existence of the project and user objects because the requirements need relationships to be set up between the models. We're not testing the today and yesterday texts because that's part of core ActiveRecord—we could test it, but it would be redundant. Redundancy is not always bad in testing, but right now it's unnecessary.

I often use a testing style that limits each individual test to a single assertion, and might therefore separate this test into four different tests sharing a common setup. The advantage of this one-assertion-per-test style is that each assertion is able to pass or fail separately. As written, the first failure prevents the rest of the tests from running. While it's a good point that assertions should be independent, in this case it's easier to follow the intent of the test when similar assertions are grouped. Also, single assertion tests are easier to write with a little help. In Section 11.3, *Single Line Assertion Tools*, on page 172, we'll see some tools that make it easier to write single-assertion tests.

When we run the tests, we get an error. The stack trace for the error looks like this:

```
  1) Error:
test_creation_of_status_report_with_data(StatusReportsControllerTest):
ArgumentError: wrong number of arguments (1 for 0)
    /test/functional/status_reports_controller_test.rb:58:in `to_s'
    /test/functional/status_reports_controller_test.rb:58:in
     `test_creation_of_status_report_with_data'
```

The line with the error is assert_equal(Date.today.to_s(:db), actual.status_date.to_s(:db)), and strictly speaking, the error message says that to_s, which converts the object to a string, is being called with the wrong number of arguments: (1 for 0), which means the method was called with one argument but expected zero.

This error message is technically true, but misleading. The real error is that actual.status_date is nil and not Date.today. That error manifests itself as a "wrong number of arguments" because the test converts both dates to strings. The method to_s() takes no arguments for most classes, but Rails ActiveSupport overrides the method for Date with an optional format argument. Since our test results in a nil value instead of a Date, the extra argument causes an error.[3]

Also, notice that the user and project parts of the test already pass. This is a Rails feature. With the use of user:references in the script/generate command line (the exact setup commands are listed in Appendix A, on page 245), Rails automatically adds the belongs_to association to the StatusReport class. As we'll see later on, it doesn't add the relationship in the other direction.

Now let's make the test pass. The classic process says to do the simplest thing that could possibly work. It's a good idea to just make the immediate error or failure go away, even if we suspect there are further errors waiting in the test. Doing so keeps the test/code cycle short, and prevents the code from getting unnecessarily complex.

To get past the test failure, change the beginning of the create() method in app/controllers/status_reports_controller.rb to this:

```
def create
  @status_report = StatusReport.new(params[:status_report])
  @status_report.status_date = Date.today         # ==> the new line
  ## the rest of the method as before
end
```

3.2 The First Refactor

We fixed the immediate problem, and the test passes. We now enter the refactoring step. There isn't much here to refactor, but we have one detail we can tweak: it's better not to set the status_date in the controller.

3. Why convert to strings, you ask? Because you get much more readable error messages when the values are both dates.

Good Rails practice moves complexity from controllers to models where possible. For one thing, placing code in the models tends to decrease duplication where functionality is used by multiple controller actions. For another, code in the model is easier to test.

Ordinarily, we would not be writing tests during refactoring, just using existing tests to verify that behavior hasn't changed. However, when moving code from one layer, the controller, to another, the model, it helps to create tests in the new class. Especially here, because our new behavior will be slightly different—we want the status_date to be automatically set whenever the report is saved.

The unit test goes in test/unit/status_report_test.rb:

```
Line 1  test "saving a status report saves the status date" do
2     actual = StatusReport.new
3     actual.save
4     assert_equal(Date.today.to_s, actual.status_date.to_s)
5   end
```

The test, of course, fails. Once again, in line 4, we're comparing literal string objects rather than the dates.

To pass the test, we add a before_save() callback to the StatusReport class:

```
class StatusReport < ActiveRecord::Base
  belongs_to :project
  belongs_to :user

  before_save :set_status_date

  def set_status_date
    self.status_date = Date.today
  end
end
```

Now the test passes. But there's one more thing to worry about—if the status_date has already been set before the report is saved, the original date should be used. As the code stands now, the status_date will change whenever the model is edited. In the TDD process, we force ourselves to make that code change by exposing the error with a test. Here's how, in test/unit/status_report_test.rb:

```
Line 1  test "saving a status report that has a date doesn't override" do
2     actual = StatusReport.new(:status_date => 10.days.ago.to_date)
3     actual.save
4     actual.reload
5     assert_equal(10.days.ago.to_date.to_s, actual.status_date.to_s)
```

```
6 end
```

The to_date() methods in lines 2 and 5 are there to convert between 10.days.ago, which is a Ruby DateTime object, and the status_date, which is a Ruby Date object. Without that conversion, we'll get an error because the string formats won't match in line 5.

The reload() call in line 4 forces ActiveRecord to re-retrieve the record from the database. ActiveRecord does not prevent a database record from having multiple live objects pointing to it. In this particular case, the controller creates a new instance from the database and saves that instance, without touching the actual variable created for the test. As a result, the database version has typecast the status_date to a Date when saving, but the live version in memory hasn't gotten that change.

In general, it's a good idea to reload any object being tested and saved. This is most commonly an issue in controller tests, where setup might create and hold onto an object that is later updated in the database during the controller action. In that case, the unreloaded object does not reflect changes made in the controller action, leading to hours of fun as you try to figure out why your test is failing.

One way to make the new test pass is this very slight change to the model:

Download huddle/app/models/status_report.rb

```
def set_status_date
  self.status_date = Date.today if status_date.nil?
end
```

And now the scary part: removing the status-changing line from the controller and making sure that the tests pass again. This involves removing the line of code that we just added to the controller a couple of seconds ago.

It just takes a second to take care of that... and it works, which we can verify by running rake.

3.3 More Validations

While we're looking at the status report model, there is another one of our original three requirements we can cover, namely the requirement that a user must enter text in at least one of the yesterday and today boxes. Back in test/unit/status_report_test.rb:

Download huddle/test/unit/status_report_test.rb

```
test "a test with both blank is not valid" do
  actual = StatusReport.new(:today => "", :yesterday => "")
  assert !actual.valid?
end
```

The simplest way to pass this test is by placing the following line of code in app/models/status_report.rb:

```
validates_presence_of :yesterday, :today
```

That's great! With that line of code in place, everything will be swell. Nothing can go wrong. (Cue ominous music). Let's run rake:

```
  1) Failure:
test_saving_a_status_report_saves_the_status_date(StatusReportTest)
[/test/unit/status_report_test.rb:9]:
<"2009-08-26"> expected but was
<"">.

  2) Error:
   test_saving_with_a_date_doesn't_override(StatusReportTest):
ActiveRecord::RecordNotFound: Couldn't find StatusReport without an ID
    /test/unit/status_report_test.rb:17:in
   `test_saving_with_a_date_doesn't_override'

  1) Failure:
test_should_create_status_report(StatusReportsControllerTest)
   [/test/functional/status_reports_controller_test.rb:17]:
"StatusReport.count" didn't change by 1.
<3> expected but was
<2>.
```

What? Well, you've probably figured it out, but adding the validation causes problems in other tests. Specifically, status reports that were created by other tests without either text field being set are now failing their saves because they are invalid. This is admittedly annoying, because it's not really a regression in the code: the actual code in the browser probably still works fine. It's more that the shifting definition of what makes a valid StatusReport is now tripping up older tests that used insufficiently robust data.

Fixing the failing tests is straightforward. First, to fix the two tests in test/unit/status_report_test.rb, add the arguments (:today => "t", :yesterday => "y") to each StatusReport.new() method call, giving:

Download huddle/test/unit/status_report_test.rb

```
Line 1  test "saving a status report saves the status date" do
    -     actual = StatusReport.new(:today => "t", :yesterday => "y")
```

```
  actual.save
  assert_equal(Date.today.to_s, actual.status_date.to_s)
end

test "saving with a date doesn't override" do
  actual = StatusReport.new(:status_date => 10.days.ago.to_date,
      :today => "t", :yesterday => "y")
  actual.save
  actual.reload
  assert_equal(10.days.ago.to_date.to_s, actual.status_date.to_s)
end
```

This puts enough data in the report to make the test pass—we don't need to care what the data actually is. A similar change needs to be made in test/functional/status_reports_controller_test.rb:

Download huddle/test/functional/status_reports_controller_test.rb

```
test "should create status_report" do
  assert_difference('StatusReport.count') do
    post :create, :status_report => {:today => "t", :yesterday => "y"}
  end
  assert_redirected_to status_report_path(assigns(:status_report))
end
```

And now we're back at all passing. This is, frankly, the kind of thing that causes people to develop an aversion to testing: sometimes it seems like a boatload of busywork to have to go back in and change all those older tests. And, well, it can be. There are a couple of ways you can minimize the annoyance and keep the benefits of working test-first.

One helpful technique is to keep a very tight loop between writing tests and writing code, and run the test suite frequently (ideally, we'd run it constantly using autotest or a similar continuous-test execution tool, Chapter 21, *Using Autotest as a Test Runner*, on page 243). The tighter the loop and the fewer lines of code we write in each back-and-forth, the easier it is to find and track down these structural test problems.

Second, and more specific to these kinds of validation problems, using some kind of factory tool or common setup method to generate well-structured default data makes it much easier to keep data in sync with changing definitions of validity. Much more on that topic in Section 6.4, *Using Factories to Fix Fixtures*, on page 83.

Anyway, fixing the older data is a distraction: we have a larger problem. Remember, we wanted the status to only be invalid if *both* today and yesterday were blank. We need to write a couple of follow-up tests to confirm that we haven't overshot the mark. The tests go in test/unit/status_report_test.rb.

Download huddle/test/unit/status_report_test.rb

```
test "a test with yesterday blank is valid" do
  actual = StatusReport.new(:today => "today", :yesterday => "")
  assert actual.valid?
end

test "a test with today blank is valid" do
  actual = StatusReport.new(:today => "", :yesterday => "yesterday")
  assert actual.valid?
end
```

Oops.

```
  1) Failure:
test_a_test_with_today_blank_is_valid(StatusReportTest)
[/test/unit/status_report_test.rb:36]:
<false> is not true.

  2) Failure:
test_a_test_with_yesterday_blank_is_valid(StatusReportTest)
[/test/unit/status_report_test.rb:31]:
<false> is not true.
```

At this point, we want to move to a custom validation, because the validation functions provided by Rails aren't quite going to get this right for us.

Replace the validation line in app/model/status_report.rb with the following:

Download huddle/app/models/status_report.rb

```
validate :validate_has_at_least_one_status

def validate_has_at_least_one_status
  if today.blank? and yesterday.blank?
    errors.add_to_base("Must have at least one status set")
  end
end
```

And we're back to passing.[4]

A couple of points on the question of what to test and when:

- The general situation here is very important. Always try to test a boundary from both sides. If you are testing that an administrator should see a certain link, you also need to test that a regular user can't see it. Your tests will only give you an accurate picture of

4. One early reviewer pointed out that this can, in fact, be done with the core Rails validations, namely a pair of validates_presence_of() calls with the if option.

your application if they cover the requirement boundaries from both sides.

- While we don't need to test the Rails validation methods as such, we do need to verify the operational behavior that a model object in a certain state is invalid. In a strict TDD process, it's the test for validity that causes us to add the Rails validation method in the first place.
- Whether to go back and add a controller test to validate behavior for invalid objects is an open question. As a matter of course, we insert a generic test into our controller scaffold using mock objects to cover the general failure case (shown in detail in Chapter 7, *Using Mock Objects*, on page 105), which means we don't need to go back and test the controller behavior for each and every different possible kind of model failure. Unless, of course, each specific failure actually dictates different controller behavior.

Now, this may seem like a lot of work because we've been going through every step in excruciating detail. In practice, though, each of these test cycles is very quick—in the 5 to 15-minute range for relatively simple tests like these.

3.4 Security Now!

Let's take a look at Huddle's login and security models that use the Authlogic plugin. Authlogic has its own set of tests, so we don't need to write tests for the basic behavior of login and logout. We do need to write tests to cover parts of the application-specific security model for who can see and edit what different things. Let's say that our authentication requirements are as follows:

1. Users must be logged in to view or create a status report.
2. Users must always have a current project chosen. Right now, any user can see and create a status report on any project. Assigning users to projects may or may not happen later. At the moment, we don't care.
3. Users can only edit their own reports. Again, there may or may not be admin functionality later; we'll cross that bridge when we get to it.

In order to enforce an Authlogic login globally throughout the app, we need to add the following inside the ApplicationController. In a slight break from normal procedure, we'll implement the forced login in the code first.

Download huddle/app/controllers/application_controller.rb

```
before_filter :require_user
```

The user session controller needs to have the following line to remove this global filter; otherwise, we're requiring a user to log in in order to be able to log in. That feature would make for a nifty Escher painting, but an annoying application.

Download huddle/app/controllers/user_sessions_controller.rb

```
skip_before_filter :require_user, :only => [:new, :create]
```

Why not do this test first? Because most of the functionality is already tested by Authlogic, and because the model is super-basic and application-wide. If and when the login model gets more complex (if, for example, there were public reports that did not require a login), we'd start adding some tests.

Despited not adding any new tests, we suddenly have no shortage of failing tests just from adding the login requirement. Running rake, the unit tests pass, but the controller tests, well:

```
18 tests, 17 assertions, 15 failures, 2 errors
```

The test failures are all due to the login requirement: every controller test is now being redirected to the login page. What we want is for all our tests to take place in the context of an active login.

There are two things we need to do to get Authlogic to play nicely with our tests. First, we need to add the following lines to the top of our test/test_helper.rb file:

Download huddle/test/test_helper.rb

```
require "authlogic/test_case"

class ActionController::TestCase
  setup :activate_authlogic
end
```

Then, in the body of the ActiveSupport::TestCase declaration in the same file, add the following methods:

Download huddle/test/test_helper.rb

```
def set_session_for(user)
```

```
  UserSession.create(user)
end

def login_as_ben
  set_session_for(users(:ben))
end
```

In the two controller tests where the controller requires a login (ProjectsControllerTest and StatusReportsControllerTest), place the following line inside the test class at the top of the class declaration:

```
setup :login_as_ben
```

This is the first time we've used the setup/teardown mechanism as implemented in Rails 2.2 and up; you can see the mechanism in more detail in Section 2.5, *What Happens When Tests Run?*, on page 31. The login_as_ben() method is called before each and every individual test in those controllers, using the mechanism Authlogic provides to fake a user login. Setups are invoked in the order declared, which is why we can't just put this setup line in the test helper and only declare it once. If we try, the login_as_ben() method is invoked before the setup in the actual controller test class. Since the controller test setup hasn't been called, the session object used for tests hasn't been created, and the login_as_ben() method will cause an error.

The authlogic method itself is simple. The call to login_as_ben() directly creates an Authlogic UserSession object. Essentially, we simulate the result of a login without requiring the actual login.

With this setup in place, the tests pass again.

3.5 Applying Security

Now that user login is required, the form for creating a status report should no longer have the user_id as an entry in the form, since the currently logged-in user is assumed to be the creator. Similarly, although we haven't specified a mechanism for setting it, the project_id will always be an implicit current project, and also doesn't need to be specified in a form. Which leads us to a new test... well, actually an edit of an existing test. In test_helper.rb, add the following helper method to set a current project in the test session:

Download huddle/test/test_helper.rb

```
def set_current_project(symbol)
  @request.session[:project_id] = projects(symbol).id
```

```
end
```

Then change the previously written status report test to remove the project and user from the test we wrote a few pages ago:

Download huddle/test/functional/status_reports_controller_test.rb

```
Line 1  test "creation of status report with data" do
        -   set_current_project(:one)
        -   assert_difference('StatusReport.count', 1) do
        -     post :create, :status_report => {
      5 -       :yesterday => "I did stuff",
        -       :today => "I'll do stuff"}
        -   end
        -   actual = assigns(:status_report)
        -   assert_equal(projects(:one).id, actual.project.id)
     10 -   assert_equal(users(:ben).id, actual.user.id)
        -   assert_equal(Date.today.to_s(:db), actual.status_date.to_s(:db))
        -   assert_redirected_to status_report_path(actual)
        - end
```

The test helper method created in the first snippet, then called in line 2 above, sets the current project in the test session; for the moment, we don't care that there's no UI way to set the current project.

Now, the first draft of this section of the book actually presented this as a separate test; as a general rule, it's not a good idea to edit existing tests unless they are actually broken by some change to the code. In this case, though, the behavior of the original test is really dangerous and must change—we don't want the user_id from the form to be acknowledged at all, in order to prevent a malicious user from posting nasty things under your good name. So we edit the existing test instead.

Preventing the malicious user is the next test, but let's pass this one first. The create() method in StatusReportsController needs to be changed so that it starts:

```
def create
  params[:status_report].merge!(:user_id => current_user.id,
      :project_id => current_project.id)
  @status_report = StatusReport.new(params[:status_report])
  ## rest of method as before
end
```

The current_user() method is defined by Authlogic and, since we are requiring a login to get this far, is guaranteed to be non-nil. We also need a current_project() that will always be non-nil. Put this in the ApplicationController:

Download huddle/app/controllers/application_controller.rb

```
helper_method :current_project
def current_project
  project = Project.find(session[:project_id]) rescue Project.last
end
```

If there is no project_id in the session, the method returns the most recently created project. That's almost certainly not the final logic, but we can get by with it for now.

3.6 Punishing Miscreants

The time has come to punish evildoers who try to get around the site by putting somebody else's user_id into their form submit. Right now, based on the above code, the submitted id is ignored, and the actually-logged-in user is used instead. But why not kick out any user trying to fake a user_id? The test, in status_reports_controller_test.rb, looks like this:

Download huddle/test/functional/status_reports_controller_test.rb

```
Line 1  test "redirect and logout if the user tries to snipe a user id" do
  -       noel = User.create(:login => "nrappin", :password => "banana",
  -           :password_confirmation => "banana")
  -       set_current_project(:one)
  5       assert_no_difference('StatusReport.count') do
  -         post :create, :status_report => {
  -           :user_id => noel.id,
  -           :yesterday => "I did stuff",
  -           :today => "I'll do stuff"}
 10       end
  -       assert_nil session[:user_id]
  -       assert_redirected_to(new_user_session_path)
  -     end
```

There are a few differences between this test and the previous one. In line 7, we've added the user_id for a newly created user to the params. In line 5, we're explicitly testing that StatusReport.count does not change—that no new report is created. Finally, lines 11 and 12 assert that the user is logged out and bounced back to the login page.

There are several things about that test that will fail at the moment (although it makes sense long term to keep it as one test). It's best to fix these things one at a time. The first failure is the assert_no_difference() call, so we need to prevent the creation of the new object.

The first attempt is a) ugly and b) doesn't work:

```
Line 1  def create
```

```
-   if params[:status_report][:user_id].nil? ||
-         (params[:status_report][:user_id] == current_user.id)
-     params[:status_report].merge!(:user_id => current_user.id,
5         :project_id => current_project.id)
-     @status_report = StatusReport.new(params[:status_report])
-   end
-   respond_to do |format|
-     if @status_report && @status_report.save
10      flash[:notice] = 'StatusReport was successfully created.'
-       format.html { redirect_to(@status_report) }
-       format.xml  { render :xml => @status_report, :status => :created,
-           :location => @status_report }
-     else
15      format.html { render :action => "new" }
-       format.xml  { render :xml => @status_report.errors,
-           :status => :unprocessable_entity }
-     end
-   end
20 end
```

In lines 2–6, we're attempting to only create the object if there is a user ID; and if the ID differs from the current user, the existence of the object is a prerequisite for the save test in line 9.

It doesn't work because in the failure case, the code tries to render the new action, which assumes the existence of @status_report. We could try to make that a redirect in line 16, but that's getting down a rathole that we'd only have to walk away from in just a second. Let's try a more general solution:

```
Line 1 def create
-   if (params[:status_report][:user_id] &&
-     params[:status_report][:user_id] != current_user.id)
-     current_user_session.destroy
5     redirect_to new_user_session_path
-     return
-   end
-   params[:status_report].merge!(:user_id => current_user.id,
-       :project_id => current_project.id)
10  @status_report = StatusReport.new(params[:status_report])
-   respond_to do |format|
-     if @status_report.save
-       flash[:notice] = 'StatusReport was successfully created.'
-       format.html { redirect_to(@status_report) }
15      format.xml  { render :xml => @status_report, :status => :created,
-           :location => @status_report }
-     else
-       format.html { render :action => "new" }
-       format.xml  { render :xml => @status_report.errors,
20          :status => :unprocessable_entity }
```

```
    end
  end
end
```

Now the tests all pass. The if statement on line 2 grabs invalid forms, and the logout and redirect happen there, with the **return** on line 6 preventing further processing.

It would be nice to make this a more general method—the problem is that if we just convert the initial if statement into a helper method, the return statement still needs to be in the actual controller action. If it's not, the action continues to process, and we get a double-render error.

Here's one solution: the general part of method goes in the ApplicationController. It performs the same test against the current user's ID, does the logout and redirect if they don't match, and returns true:

Download huddle/app/controllers/application_controller.rb

```
def redirect_if_not_current_user(user_id)
  if user_id && user_id != current_user.id
    current_user_session.destroy
    redirect_to new_user_session_path
    return true
  end
  false
end
```

The controller method now needs to take that result and use it to stop its own processing:

Download huddle/app/controllers/status_reports_controller.rb

```
Line 1  def create
        redirect_if_not_current_user(params[:status_report][:user_id]) and return
        params[:status_report].merge!(:user_id => current_user.id,
            :project_id => current_project.id)
    5   @status_report = StatusReport.new(params[:status_report])
        respond_to do |format|
          if @status_report.save
            flash[:notice] = 'StatusReport was successfully created.'
            format.html { redirect_to(@status_report) }
   10       format.xml  { render :xml => @status_report, :status => :created,
                :location => @status_report }
          else
            format.html { render :action => "new" }
            format.xml  { render :xml => @status_report.errors,
   15           :status => :unprocessable_entity }
          end
        end
      end
```

Line 2 is the key here: it calls the general method, then returns if the result is true. That version is the most readable, even if it's a bit Perl-ish. We could also try:

```
return if redirect_if_not_current_user(params[:status_report][:user_id])
```

Or:

```
if redirect_if_not_current_user(params[:status_report][:user_id])
  return
end
```

Or, using a before_filter at the top of the controller:

```
before_filter :redirect_if_not_current_user, :only => [:create]
```

In which case, we don't need anything in the create() method, but the helper method needs to change in a couple of ways. First, the return value needs to flip and return false if the redirect happens in order to stop further processing, and true if the user is OK. Second, filter methods can't take arguments, so the extraction from params[:status_report][:user_id] needs to take place in the filter method itself.

With that refactoring, the tests pass again.

Testing for security is generally time well-spent—security tests are relatively easy to write, since in many cases we're just testing for a redirection to a login page, or that key data didn't display. The trick is to be creative and try to think of all the oddball ways that somebody might be able to submit data to flummox our application.

3.7 Roadmap

In this chapter we've started a new Rails application and written the first couple of features of that application using Test-Driven Development. We've added features to our controller, then moved them to the model, and validated the model more tightly. We also tested and added authentication logic.

We're not done touring Rails core TDD procedures, though. In the next chapter, we'll look at view logic, and show how to use TDD within Rails to create a feature end-to-end.

Chapter 4

TDD, Rails Style

Now that we've got some successful test/code/refactor cycles down, we'll go through a more complicated TDD example, this time adding display logic to the application. In that example, we'll cascade our tests from layer to layer, between the controller, model, and view.

4.1 Now for a View Test

We've tested some model logic and some controller logic, but we still have two view tests that need to be written to close out the initial data entry feature. We'd like to validate that the form being generated actually contains the field names that the create() method expects—this little gap is one of the easiest ways for a problem to slip through the cracks of an otherwise airtight Rails TDD process. Then let's test that the project show page actually displays the timeline of status reports for the project.

Testing view logic is a little bit different than testing model or controller logic. It's more impressionistic and less precise. When testing views, it's important to keep the larger goal of specifying the application logic in mind, and not fritter away huge amounts of time trying to get the view tests ultra-complete and correct. It's easy to write view tests that are detailed but add little to the amount of coverage in the test suite, and are prone to breaking any time a web designer looks at the HTML code cross-eyed.

These issues are covered more completely in Chapter 9, *Testing Views*, on page 142, but here are three tips to keep in mind as we embark on two sets of view tests:

Make Those Pop-Culture References Work for You

This could well be the goofiest tip I give in this whole book, but when I create test data for users that have some kind of relationship, I find it valuable to give those users names that are meaningful in terms of some reference or other; for instance, Fred and Barney, Homer and Marge, or Lois and Clark. This simple trick helps to keep the expected relationship between the various pieces of test data straight in your mind. Obviously, you can take this too far, and there will always be somebody on the team who doesn't know the reference. It sounds silly, but at the very least, it'll make you smile every now and then looking at your code.

- Consider moving view logic to helpers where possible. In the core Rails setup, it's much easier to individually test a helper than it is to individually test an view partial (although if you use RSpec you can test view partials directly—more on this in Chapter 12, *RSpec*, on page 173). On the other hand, moving view code from HTML/ERb and into Ruby methods could cause a web-only design team to come after you with torches and pitchforks. So beware.
- Remember that the purpose of view testing is for the developer to validate the logical correctness of the application. Automated view testing isn't (and in Rails, really can't be) a substitute for walking though the site and checking that things line up and look nice.
- Test at the *semantic* level, not the display level. Liberally assign DOM IDs and classes to the elements of the view, and test for the existence of those elements, rather than the actual text. Obviously, there are limits; sometimes the actual text is critical. But the advantage of DOM ID testing is that it doesn't break every time the content team updates the text, or when the designer changes the CSS styles.

Our first view test will just validate that the form elements actually exist in as we want them to, which means the attributes that will be automatically added by the controller shouldn't be in the form. With core Rails test structures, this test goes with the controller in test/functional/status_reports_controller_test.rb:

Download huddle/test/functional/status_reports_controller_test.rb

```
Line 1  test "new form has expected elements" do
2     get :new
3     assert_select "form[id=new_status_report][action=/status_reports]" do
4       assert_select "#status_report_project", :count => 0
5       assert_select "#status_report_user", :count => 0
6       assert_select "textarea#status_report_today"
7       assert_select "textarea#status_report_yesterday"
8       assert_select "#status_report_status_date_1i", :count => 0
9     end
10  end
```

This is a reasonably straightforward view test, which simulates a GET request for the new status report form in line 2 and then uses assert_select() to verify several features of the output of the call. The assert_select() method has about seventy-eleven different options, all of which are lovingly detailed in Chapter 9, *Testing Views*, on page 142. For now, the basic point to remember is that the first argument to assert_select() is a CSS-like selector, and the remaining arguments are assertions about elements in the output that match the selector.[1] For our purposes, the most useful option is :text, which is either a string or a regular expression. If this option is specified, the internal contents of tags that match the selector are checked to see if any matches the argument. The second most useful option is :count, which is the number of elements in the view that match the selector. If :count is not specified, the expectation is that there will be at least one match. If :count and :text are specified, :count is the number of items that match both the selector and the text.

You can also nest assert_select() calls using blocks; this step adds the additional constraint that the assertions inside the block must all be true inside the body of a tag that matches the outer assert_select(). This constraint can also be described in a single assert_select() call using compound selectors like ul > li. In the above test, the outer assert_select() matches a form tag with the ID and action that we would expect by Rails convention for a new form action. All the other assert_select() calls validate the elements inside that form tag. Specifically, we are testing that there will not be a project, user, or date tag of any kind, and also that there will be text area entry tags for the yesterday and today status report fields.

1. One handy feature of assert_select() is that it parses the HTML markup, and will spit out a warning if the HTML is badly formed—for instance, if it is missing an end tag.

There could be some quibbling back and forth on the exact structure of these tests. We've chosen to test the form fields based on their DOM IDs, knowing that Rails has a consistent pattern for IDs and the name field of the tag. For me, DOM IDs work a little bit better with the assert_select() syntax.[2] Still, a clueless programmer who was breaking the Rails conventions could cause a bug without failing this view test. That said, chasing down every way the view could break would take all our efforts, forever: try to limit view testing to things that are likely sources of error or regression.

Passing the test is quite easy: just remove the form elements for the project, user, and date. However, unlike the model tests, even with the passing test, this view is in no way, shape, or form fit to be in a production application. You need styling; the Rails scaffold HTML is not optimal—that sort of thing. . .

At this point in a "real" development environment, the feature is essentially finished, but we *must* go into the browser and test it, no matter how complete we think the tests are. Tests are great and wonderful, but the user isn't going to care about our code coverage if it doesn't work in the browser. Writing tests minimizes the amount of time we spend cycling through the browser, but in no way does automated testing mean we can stop using the browser.

But here in tutorial world, let's move on to the display functionality for an entire project within Huddle. So far, we've covered the major features of Rails TDD, we've tested models, controllers, and views, but we've only tested them in isolation. To close out this walkthrough, let's take a slightly larger piece of functionality and show how you might move back and forth between code and tests, and between the different kinds of test, when building a complete slice of your application.

4.2 Testing the Project View: A Cascade of Tests

In Huddle, a project page should show a timeline of status reports for that project. This functionality requires at least a controller and a view test to start, and we may use some model tests later on. We are going to need some status report data for this test, and we haven't discussed fixtures yet. In order to get these tests to work, we first need to get

2. Another approach is to say that these tests should track the name directly, as in assert_select "textarea[name = ?]", "status_report[today]".

some sample data into our test, then we'll use that data to specify the behavior of the application.

Testing the View

In order to test the view, we need some fixture data. Add the following to test/fixtures/projects.yml to create some projects.

Download huddle/test/fixtures/projects.yml

```
huddle:
  name: Huddle Project
```

We'll need another user, so here's a snippet for test/fixtures/users.yml.

Download huddle/test/fixtures/users.yml

```
jerry:
  login: jjones
  password_salt: <%= salt = Authlogic::Random.hex_token %>
  crypted_password: <%= Authlogic::CryptoProviders::Sha512.encrypt("jj" + salt) %>
  persistence_token: x
```

Finally, some status reports, in test/fixtures/status_reports.yml.

Download huddle/test/fixtures/status_reports.yml

```
ben_tue:
  project: huddle
  user: ben
  yesterday: Worked on Huddle UI
  today: Doing some testing
  status_date: 2009-01-06

ben_wed:
  project: huddle
  user: ben
  yesterday: Did Some Testing
  today: More Testing
  status_date: 2009-01-07
```

Download huddle/test/fixtures/status_reports.yml

```
jerry_tue:
  project: huddle
  user: jerry
  yesterday: set up huddle schema
  today: pair programming with ben
  status_date: 2009-01-06

jerry_wed:
  project: huddle
  user: jerry
  yesterday: sick
```

```
  today: trying to pair again
  status_date: 2009-01-07
```

We can start testing using this fixture data. The project controller method for show() should gather the reports for that project and group them by date. This goes in test/functional/projects_controller_test.rb:

Download huddle/test/functional/projects_controller_test.rb

```
Line 1  test "project timeline index should be sorted correctly" do
  2       set_current_project(:huddle)
  3       get :show, :id => projects(:huddle).id
  4       expected_keys = assigns(:reports).keys.sort.map{ |d| d.to_s(:db) }
  5       assert_equal(["2009-01-06", "2009-01-07"], expected_keys)
  6       assert_equal(
  7           [status_reports(:ben_tue).id, status_reports(:jerry_tue).id],
  8           assigns(:reports)[Date.parse("2009-01-06")].map(&:id))
  9     end
```

In lines 4–5, this asserts that an object called @reports is created, and its keys are the dates of the reports that are found. Lines 6–8 assert that each key contains its reports, sorted by the name of the user (well, technically, the email of the user—the user model created by Authlogic doesn't have name fields, and we haven't added them yet).

That test will, of course, fail, because assigns(:reports) is nil.

Moving to the controller itself, let's just defer the assignment to the model. In app/controllers/projects_controller.rb:

Download huddle/app/controllers/projects_controller.rb

```
def show
  @project = Project.find(params[:id])
  @reports = @project.reports_grouped_by_day
  respond_to do |format|
    format.html # show.html.erb
    format.xml  { render :xml => @project }
  end
end
```

Which means we now need a model test in test/unit/project_test.rb:

Download huddle/test/unit/project_test.rb

```
test "should be able to retrieve projects based on day" do
  actual = projects(:huddle).reports_grouped_by_day
  expected_keys = actual.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],
      actual[Date.parse("2009-01-06")].map(&:id))
end
```

This is a direct swipe of the code we just put in the controller test, which opens up the question of whether we need both the controller and model test. The model test is important because it's closest to the actual implementation, and is the easiest place to write error-case tests. The controller test adds just the piece of information that the model method is, in fact, called by the controller—strictly speaking, the controller test does not need to revalidate the model logic. You could potentially remove the model-specific assertions from the controller test if you wanted. This is a classic place for a *mock object call* in the controller, which would prevent the controller test from depending on the specific code in the model. Mock objects are a huge topic in their own right, and are covered in more detail in Chapter 7, *Using Mock Objects*, on page 105. For our purposes right now, the duplicate test is not a problem.[3]

Anyway, passing this test requires some code. In app/models/project.rb:

Download huddle/app/models/project.rb

```
has_many :status_reports

def reports_grouped_by_day
  status_reports.by_user_name.group_by(&:status_date)
end
```

Previously, this model was blank: we added the has_many line here because it's needed to pass the test. Note that we don't need to separately test what has_many does—that's part of the Rails framework itself. The need for the association line came immediately as we started writing tests for functionality of that model.[4]

The actual method uses the Rails ActiveSupport group_by(), which returns the hash structure we want. The by_user_name will be a named scope inside StatusReports() that sorts the reports based on the user's login. As I worked on this method, it became clear that the named scope was going to be complex enough to require its own test. In test/unit/status_report_test.rb:

Download huddle/test/unit/status_report_test.rb

```
test "by user name should sort as expected" do
  reports = StatusReport.by_user_name
  expected = reports.map { |r| r.user.login }
```

3. Another option is to use an integration test or a Cucumber acceptance test in place of the controller test.
4. A framework like Shoulda gives easy, one-line tests for associations—in which case, it might be worth throwing in the single line.

```
  assert_equal ["bjohnson", "bjohnson", "jjones", "jjones"], expected
end
```

You only do this because the named scope crosses an association. For a simple scope, it might not be worth the trouble, since the functionality is mostly covered by Rails, and is also easily covered by the existing test. Essentially, you might write the test if you're nervous that you might not write the scope correctly.

There are two overlapping issues here: when to test functionality provided by the framework, and when to test private or subordinate methods that are only called by other tested methods. The guideline is that a new test only needs to be written if the logic of the application has changed. For instance, if we refactor a smaller method out of a larger, already tested method, we won't need to also write targeted tests against the smaller method—it is just a restructuring of the already-tested logic. Should the subordinate method later gain additional logic, we'll need to write tests for that method. If we find a bug in the smaller method, then it should have tests immediately.

The scope in app/model/status_report.rb looks like this:

Download huddle/app/models/status_report.rb

```
named_scope :by_user_name, :include => "user",
    :order => "users.login ASC",
    :conditions => "user_id IS NOT NULL"
```

And all the tests pass.

This is the order in which I actually wrote this code the first time I went through it: controller test, model test in Project, Project model implementation (after checking, the one liner was written directly), scope test, then scope code (with a little flailing in there about exactly how to manage the users table). It may not be strictly test-first, but in the entire process, I never wrote more than about five lines of code on one side of the test/code divide without jumping to the other, and I never went more than a minute or two without running the test suite.

And that's the key to success with TDD. Keep a tight feedback loop between the code and tests: don't ever let one or the other get too far out in front.

Now we need a genuine view test to validate that something reasonable is going into the view layer, back in test/functional/projects_controller_test.rb:

Download huddle/test/functional/projects_controller_test.rb

```
test "index should display project timeline" do
  set_current_project(:huddle)
  get :show, :id => projects(:huddle).id
  assert_select "div[id *= day]", :count => 2
  assert_select "div#2009-01-06_day" do
    assert_select "div[id *= report]", :count => 2
    assert_select "div#?", dom_id(status_reports(:ben_tue))
    assert_select "div#?", dom_id(status_reports(:jerry_tue))
  end
  assert_select "div#2009-01-07_day" do
    assert_select "div[id *= report]", :count => 2
    assert_select "div#?", dom_id(status_reports(:ben_wed))
    assert_select "div#?", dom_id(status_reports(:jerry_wed))
  end
end
```

Continuing with the idea of semantic-level tests for the view layer, this test checks to see that there is some kind of div tag for each day, and that inside that tag is a div tagged for each status report. Even putting in a div might be overly specific—it's possible these might be table rows, or something. The count tests are to prevent more content showing up than expected, something that is not caught by just testing for the existence of known tags. In this case, we're also testing that there are exactly two days worth of reports, and that each day has two reports. This is calibrated to match the fixture data we just set up, and gives a quick look at one weakness of fixtures—a change to that fixture data could break this test.

Also, for this to work, add this line inside the class definition in test/test_helper.rb:

Download huddle/test/test_helper.rb

```
include ActionView::Helpers::RecordIdentificationHelper
```

This allows us to use the dom_id() method in tests, which is handy.

Putting aside that there are a jillion ways the view code could pass the letter of this test and violate the spirit, here's the basic structure of a passing view, in app/views/projects/show.html.erb:

Download huddle/app/views/projects/show.html.erb

```
<h2>Status Reports for <%= @project.name %></h2>
<% @reports.keys.sort.each do |date| %>
  <div id="<%= date.to_s(:db) %>_day">
    <h3>Reports for <%= date.to_s(:long) %></h3>
    <% @reports[date].each do |report| %>
      <div id="<%= dom_id(report) %>">
        Yesterday I: <%= h report.yesterday %>
```

```
        Today I will: <%= h report.today %>
      </div>
    <% end %>
  </div>
<% end %>
```

This gives the view a div for each date, and a nested div for each individual report.

4.3 So Far, So Good

At this point, we've completed our initial walk-through of Rails test-driven development. You should be able to add tests to a Rails application, use fixture data, test your controllers, models, and views, and have a feel for how the TDD quick test/code feedback loop works in practice.

The rest of the book is divided up based on what part of the application is under test. Part II, Testing Application Data, deals with testing ActiveRecord models, getting data into your tests via fixtures and factories, and other data-related topics. Part III, Testing User-Facing Sections, discusses testing controllers, views, and helpers, each of which has some special mechanisms to make the process easier.

The big third-party test frameworks, Shoulda and RSpec, are discussed in Part IV, while Part V covers integration testing, both using Rails itself, and using external tools like Webrat, Cucumber, and Selenium. Finally, Part VI talks about how to evaluate and improve your tests, using coverage testing, managing performance, and troubleshooting.

Part II

Testing Application Data

Chapter 5

Testing Models with Rails Unit Tests

The overwhelming majority of data in a typical Rails application is accessed via ActiveRecord objects that connect to a SQL database. These objects are tested using *model tests*. Model tests, which core Rails calls *unit tests*, are the most basic level of the Rails testing functionality—by which I mean they are the closest to Ruby's Test::Unit and the foundation on which the test structures for controllers and views are built. Model tests in Rails are just Test::Unit plus a) the ability to set up data in fixtures, b) the block syntax for describing setup, teardown, and tests, and c) a couple of additional assertion methods. Model tests that you write are placed in the test/unit directory.[1] We're starting our tour of the Rails stack with model tests because model tests have the fewest dependencies on Rails-specific features and are usually the easiest place to start testing your application. Later on, we'll move forward to controller testing, view testing, integration testing, and other Rails tools.

5.1 What's Available in a Model Test

Rails model tests are subclasses of ActiveSupport::TestCase, which is a subclass of the core Ruby Test::Unit::TestCase. Model tests also include a couple of modules from the Rails core mixed in to provide additional functionality. All told, the following functionality is added to Ruby unit tests to make them Rails model tests (as we'll see in Chapter 8, *Testing*

1. While I'm here, does it bother anybody else that app/controllers and app/models are plural, while test/functional and test/unit are singular?

What's in a Name, Part One

Tests for Rails models are usually referred to as "unit tests," but as much as I hate being pedantic and technical, I prefer to call them "model tests." (Who am I kidding, I *love* being pedantic and technical.) My basic problem is that "unit test" has a specific meaning that, depending on how you look at it, either includes Rails functional tests, or doesn't include either kind of test. (Jay Fields, for example, argues that Rails model tests aren't really unit tests because they require an external database.) Personally, I find it less confusing to use "unit test" as a generic term for all developer testing, and "model test" for tests that actually validate Rails models.

Controllers with Functional Tests, on page 131, controller tests have even more additions):

- The ability to load data from fixtures before each test.
- The Shoulda and RSpec-inspired test syntax: test "do something" do.
- Multiple setup and teardown block syntax, discussed in more detail in Section 2.3, *Setup and Teardown*, on page 26.
- The assert_difference() and assert_no_difference() methods.
- The assert_valid() test, which verifies that an ActiveRecord model is—wait for it—valid according to the rules of that model.
- A couple of little-known database helper methods that you can gain access to by making your test case a subclass of ActiveRecord::TestCase: assert_date_from_db(), assert_sql(), assert_queries(), and assert_no_queries(). These are used internally by the Rails core test system, and don't seem have a lot of value outside core.
- The test/test_helper.rb, required by all Rails-generated model tests and part of your application, injects some additional methods into ActiveSupport::TestCase.[2] This is a good place to put common setup and assertion methods needed by all tests.

2. Before Rails 2.2, the injects were into Test::Unit::TestCase, even in Rails 2.1, where ActiveSupport::TestCase was already being used by the model tests.

5.2 What to Test in a Model Test

Models. Next question?

5.3 Okay, Funny Man, What Makes a Good Model Test Class?

The goal is to have each model's individual test file cover pretty much 100% of the code in that model. Other models or controller tests will probably incidentally touch code in the model, but that's not a replacement for effective tests for a particular model in its own test file. Tests should be as close as possible to the code being described.

You don't need to write a separate test for the existence of relationships like belongs_to or has_many—Rails tests those features thoroughly, and if you don't have a relationship you expect, you'll get failures all over the place.[3] The existence of the relationship should be driven by a test that needs the relationship in order to deliver functionality. You should write tests to cover validations, in order to ensure that the difference between a valid and invalid object is what you expect. Named scopes are somewhere between relationships and methods. Simple named scopes can probably be treated like relationships, but anything complicated—with a lambda block, for example—should probably be tested. More on that in Section 5.5, *Testing Named Scopes*, on page 74.

How many tests do you need? The strict test-driven answer is that all new logic should be driven by a failing test, and conversely, each new test should fail and trigger a new piece of code logic. A typical progression looks something like this (if you keep your methods smaller and simpler, you'll tend to need fewer tests for each method):

- One test of the normal, happy-path case.
- One test for each alternate branch through the code. Refactoring here often causes the method to be split into multiple smaller methods.
- At least one test for known error cases, such as being passed nil arguments, as needed.[4] My position, which is perhaps a lit-

3. Brian Hogan pointed out in review—I paraphrase here—that this may be an overly optimistic view of the stability of Rails core. True enough. The place to catch that is probably at the integration test level.

4. I'm tempted to go on a rant about programmers putting too much error-checking in their code, but this really isn't the place.

tle cranky, is that you should only include this if you either really need the error case to do something specific, or if you have reason to doubt that an expected error case is really being treated as an error.

Keep each individual test small. In many cases, you only need to create a single model object for a unit test.

If a single call to the method causes multiple changes in the model, it's fine to have all the assertions in a single test; you don't have to be a purist about keeping only a single assertion in a test method. In this example from a tracking system, marking a story complete triggers several different changes in the method. It can be awkward and hard to follow to have each of these assertions in a different method. For example:

```
test "mark a story complete" do
  story = stories(:incomplete)
  assert_difference "story.task_logs.count", 1 do
    story.complete!(users(:quentin))
  end
  assert story.completed?
  assert !story.blocked?
  assert_equal(Date.today.to_s(:db), story.end_date)
  assert_equal("completed", story.most_recent_log.end_state)
end
```

Versus the one assertion per test version:

```
test "mark a story complete and add a task log" do
  assert_difference "story.task_logs.count", 1 do
    stories(:incomplete).complete!(users(:quentin))
  end
end

test "mark a story complete and the story should be completed" do
  story = stories(:incomplete)
  story.complete!
  assert story.completed?
end
```

And so on. To be fair, the one-at-a-time tests are verbose because I'm only using core Rails methods. With *contexts* and a couple of other tricks, these tests can be written more compactly (see Section 11.1, *Contexts*, on page 172 and Section 11.3, *Single Line Assertion Tools*, on page 172). Written with those tools, the one-assertion-per-test version can be more readable than the all-in-one-test version.

There's a tradeoff: by putting all the assertions in the same test method, you gain clarity and cohesion benefits, but you prevent the tests from running independently. In the all-in-one test, if assert story.completed? fails, you won't even get to the check for assert !story.blocked. If all the assertions are in separate tests, everything runs independently, but it's harder to determine how tests are related. (The multiple test version also runs more slowly.)

That said, when you are writing separate tests that cover different branches of the method or the error tests, they should be different test methods (unless the method setup is *extremely* simple):

```
test "full names" do
  u1 = User.create(:first_name => "Fred", :last_name => "Flintstone")
  assert_equal("Fred Flintstone", user.full name)
end

test "full names with a middle initial" do
  u1 = User.create(:first_name => "Fred", :last_name => "Flintstone"
      :middle_initial => "D")
  assert_equal("Fred D. Flintstone", user.full name)
end

test "full name where there's no first name" do
  u1 = User.create(:last_name => "Flintstone")
  assert_equal("Flintstone", user.full name)
end
```

In this case, you do want each test to run independently; trying to stuff all the branches into a single test will be very hard to read going forward.

5.4 Asserting a Difference, or Not

The two assertions that Rails adds to the basic unit test are powerful replacements for a common test pattern. The following test asserts that after a create() call, there is one more user than there was previously:

```
test "creating creates a user" do
  pre = User.count
  User.create(:first_name => "Noel")
  post = User.count
  assert_equal(pre + 1, post)
end
```

With assert_difference(), the test can be written without the duplicate call to User.count():

```
test "creating creates a user" do
  assert_difference 'User.count' do
    User.create(:first_name => "Noel")
  end
end
```

The first argument to assert_difference() is a string of Ruby code. The value of the string is calculated using eval(). The code inside the block is executed, and the value of the string is re-calculated. By default, the expectation is that the new value will be one more than the old value, but you can adjust this by passing in a second argument, such as assert_difference("User.count", 0), in order to assert no change, or assert_difference("User.count", -1), to assert the removal of a user.

If you'd like to check multiple code snippets, you have two options. The calls to assert_difference() can be nested:

```
test "create a user and a log entry" do
  assert_difference 'User.count' do
    assert_difference 'LogEntry.count' do
      User.create(:first_name => "Noel")
    end
  end
end
```

Or, if the difference number is the same for all the snippets, the snippets can be passed in a list:

```
test "creating creates a user" do
  assert_difference ['User.count', 'LogEntry.count'] do
    User.create(:first_name => "Noel")
  end
end
```

Finally, assert_no_difference('User.count') is syntactic sugar for assert_difference('User.count', 0).

5.5 Testing Named Scopes

Named scopes are an exceptionally useful piece of ActiveRecord functionality that allows you to encapsulate pieces of database logic. For example, if you often find yourself needing, say, users sorted by email

address, or a list of all active users, or to only get five users out of your database find call, you can write all of those as named scopes: [5]

```
named_scope :by_email, :order => "email ASC"

named_scope :active_only, :conditions => {:active => true}

named_scope :limit_to, lambda {|x| {:limit => x}}
```

The scopes can then be used as class methods, just like any other ActiveRecord find() command.

```
User.by_email
User.active_only
User.limit_to(5)
```

But the best part is that scopes can be composed, which gives you a very readable way to express complex database queries:

```
User.active_only.by_email.limit_to(5)
```

Named scopes, therefore, are awesome. But they occupy an awkward place between methods you might write and Rails core features, leading to the question of how best to test functionality you've placed in a named scope declaration. Here are some guidelines.

Named scopes are often extracted during a refactoring step. In this case you may not need any new test to cover the scope—like any other method extracted in refactoring, it's not a change in logic, so it's already covered for TDD purposes by the original test. Even so, if the named scope winds up in a different class than the original method, it's often useful to transfer the test logic dealing with the scope to a test in the new class.

The pitfall you want to avoid when testing named scopes (or any database find behavior) is testing the nature of the SQL call to the database rather that testing the results of the call. In other words, it's not all that hard to extract the parameters the named scope object is going to use to contact the database. However, only testing that your named scope winds up with :conditions => {:active => true} as its find parameters actually doesn't help you any. Just testing the parameters says nothing about the actual database behavior, and it's the actual behavior that you are normally looking to validate.

5. In Rails 3, all ActiveRecord finders behave as named scopes do in Rails 2.x. This section will be rewritten to reflect that fact. In the meantime, you can mentally apply the discussion here to Rails 3 finder methods.

The following method can be helpful; it can actually be used to test any method that extracts a set of records from the database (which means that it doesn't help testing scopes that just affect, say the output sort order). This method is in Shoulda syntax, and takes the named scope under test as a symbol, any arguments that get passed to the scope, and then a block:

Download huddle/test/test_helper.rb

```
Line 1  def self.should_match_named_scope(named_scope, *args, &block)
  -       should "match named scope #{named_scope}" do
  -         ar_class = self.class.model_class
  -         scoped_objects = ar_class.send(named_scope, *args)
  5         assert !scoped_objects.blank?
  -         scoped_objects.each do |obj|
  -           assert block.call(obj)
  -         end
  -
 10         non_scoped_objects = ar_class.all - scoped_objects
  -         assert !non_scoped_objects.blank?
  -         non_scoped_objects.each do |obj|
  -           assert !block.call(obj)
  -         end
 15       end
  -     end
```

This code does three things. First, in line 3 and line 4, it extracts the model class being tested and calls the named scope on that model class, resulting in a set of instances of that model. Then in lines 5-8, each instance in the list of matching objects is tested against the block and must return true for the test to pass. Just as importantly, lines 11-14 run the block against all the instances that weren't returned by the named scope and assert that the block is false for each one.

A sample usage of this test might look like this—notice that the test is a class-level method that assumes the user population has already been created in setup:

```
setup :create_users

def create_users
  active_user = User.create(:active => true)
  inactive_user = User.create(:inactive => true)
end

should_have_named_scope :active_only { :active == true }
```

This test verifies that the named scope correctly sorts the universe of users into active and inactive groups. You don't need to create dozens

and dozens of user objects for this test to work—you just need at least one in each category. Creating extra objects just slows the test down. A nice feature of this test style is that the test will be relatively robust against new objects being created. For example, if you also have fixture data in your tests, any new users created in fixtures will simply be split into the correct group and validated. The test will continue to pass.

5.6 Coming Up Next

Rails unit tests are not the only important part of testing data. Over the remaining chapters in this section, we'll compare fixture data against factory data and see when you might use one or the other. We'll show how to use mock objects to test parts of your application that might otherwise be hard to reach. And we'll cover some tricky kinds of data, such as date and times, that isn't fully test-covered.

Chapter 6

Creating Model Test Data with Fixtures and Factories

One of the most valuable ways in which Ruby on Rails supports automated testing is through the use of easily created data that is accessible to all the tests in your system, no matter when or where you write those tests. It's sometimes hard for an experienced Rails programmer to remember just how exciting the YAML fixtures used to seem. You can just set up data once? In an easy format? And it's always there? Amazing.

Over time, the infatuation with fixtures dims a bit, but fixtures are still a quick and easy way to get data into your tests. In this chapter, we'll discuss how to use fixtures, then we'll discuss the problems with fixtures. Many of the shortcomings with fixtures have been addressed by a variety of tools that use the *factory* pattern to create data objects. We'll discuss those factory tools, and how using them differs from using fixtures.

6.1 Defining Fixture Data

A *fixture* is the baseline, or fixed state, known to exist at the beginning of a test. The existence of a fixed state makes it possible to write tests that make assumptions based on that particular set of data. In Rails, the fixtures that are available out of the box are a defined in a set of YAML files that are automatically converted to ActiveRecord objects and loaded for each test.

Under normal circumstances, each model in your application will have an associated fixture file. The fixture file is in YAML format, a data-description format often used as an easier-to-type alternative to XML.[1] The details of YAML syntax are both way outside the scope of this book and largely irrelevant to fixtures—YAML contains a number of advanced features that don't need to concern us here.

Each entry in a fixture file starts with an identifier for that entry, followed by the attributes for that entry. This sample contains two entries for a hypothetical User class, and would go in test/fixtures/users.yml :

```
Line 1  fred:
2         first_name: Fred
3         last_name: Flintstone
4         email: fflint@slaterockandgravel.com
5
6       barney:
7         first_name: Barney
8         last_name: Rubble
9         email: brubble@slaterockandgravel.com/
```

YAML syntax is somewhat reminiscent of Python, both in the colon used to separate key/value pairs (also a feature of Ruby 1.9), and in the use of indentation to mark the bounds of each entry. The fact that line 6, barney:, is outdented two spaces indicates to the YAML parser that a new entry has begun. Strings do not need to be enclosed in quotation marks, although it doesn't hurt if you find it more readable.[2] A multiline string can be specified by putting a pipe character (|) on the line with the attribute name. The multiline string can then be written over the next set of lines; each line must be indented relative to the line with the attribute name. Once again, outdenting indicates the end of the string.

```
fred:
  first_name: Fred
  last_name: Flintstone
  description: |
    Fred is very tall.
    He is not very small.
```

The Rails fixture creation process uses information in your database to coerce the values to the proper type. I write dates in SQL format

1. YAML stands for Yet Another Markup Language. Which you probably figured out already.
2. Quotation marks are necessary around a string if the YAML parser would find the string ambiguous, for instance, if the string itself contains a colon followed by a space.

(yyyy-mm-dd), though any format readable by Ruby's Date.parse() will work.

The identifier that introduces each record is then used to access the individual fixture entry within your tests. Assuming that this is the User class, you'd be able to retrieve these entries throughout your test suite as users(:fred) and users(:barney), respectively. Unless you like trying to figure out what's special about users(:user_10), I recommend meaningful entry names, especially for entries that expose special cases: users(:user_with_no_first_name).

An older mechanism for allowing access to fixtures as instance variables (in this case, @fred and @barney) can be turned on by setting self.use_instantiated_fixtures = true in the test/test_helper.rb file. This style is largely deprecated; it's rather slow, and when all fixtures are loaded, it requires all your fixture entries to have different names across all classes. The feature is still there, however, and you might occasionally see it used in (really) old legacy code.

Unlike the normal way of creating ActiveRecord models, the YAML data is converted to a database record directly, without going through the normal ActiveRecord creation methods. (To be clear, when you use the data in your tests, those are ActiveRecord models—only the original creation of the data to the database bypasses ActiveRecord). This means you can't use arbitrary methods of the model as attributes in the fixture the way you can in a create() call. Fixture attributes have to be either actual database columns or ActiveRecord associations explicitly defined in the model. Removing a database column from your model and forgetting to take it out of the fixtures is a good way to have every single one of your tests error out. The fixture loading mechanism also bypasses any validations you have created on your ActiveRecord, meaning that there is no way to guarantee the validity of fixture data on load, short of explicitly testing each fixture yourself.

You do not need to specify the id for a fixture (although you can if you want to). If you do not specify an id explicitly, the id is generated for you based on the YAML identifier name of the entry. If you allow Rails to generate these ids, then you get a side benefit: an easier way of specifying relationships between fixture objects. If your models have a relationship with models in another fixture file, the other object can be referenced using the name of the relationship and the identifier of the YAML entry in the other file. In other words, if we have a company.yml with:

```
slate:
  name: SlateCo
```

And we also have a user model that belongs_to: company, then we can do the following in our user.yml file.

```
fred:
  first_name: Fred
  last_name: Flintstone
  company: slate
```

If the relationship is has_many, the multiple values in the relationship can be specified as a comma-delimited list. This is true even if the two objects are in a has_and_belongs_to_many relationship via a join table, although a has_many :through relationship does need to have the join model entry explicitly specified.

```
fred:
  first_name: Fred
  last_name: Flintstone
  company: slate
  roles: miner, digger, dino_wrangler
```

This is very handy, and a vast improvement over the older functionality, where the all the id columns had to be explicitly filled with the id number of the other model.

Fixture files are also interpreted as ERb files, which means you can have dynamic attributes like this:

```
fred:
  last_login_time: <%= 5.days.ago %>
```

Or specify multiple entries dynamically, like this:

```
<% 10.times do |i| %>
task_<%=i%>:
  name: "Task <%= i %>"
<% end %>
```

In the second case, notice that the identifier still needs to be at the left-most column; you can't indent the inside of the block the way that normal Ruby style would suggest.

6.2 Loading Fixture Data

Fixture loading is covered by a few parameters that have default values, which are set in the test/test_helper.rb file. The most important is the fixtures :all method call, which ensures that all your fixture files are

loaded in all your tests. Back in the day,[3] the prevailing style was to declare which fixtures needed to be loaded in each individual test file. That got to be annoying once your models were intertwined enough to need to load bunches of them in each test file, so in the fullness of time, loading all fixtures, all the time became the default. (Speculating wildly, this was about the same time that transactional fixtures were added, minimizing the performance cost of loading all that data.)

By default, fixtures are loaded just once, and every test method takes place inside a database transaction. At the end of the test method, the transaction is rolled back and the initial fixture state is thereby restored. This dramatically reduces test time[4] unless your database doesn't support transactions (most likely because you are using MySQL with MyISAM tables).

Fixture transactions are also a problem if you are actually trying to test transactional behavior in your application, in which case the fixture transaction will overwhelm the transaction you are trying to test. If you need less aggressive transaction behavior, you can go into the test/test_helper.rb file and change the value in the assignment self.use_transactional_fixtures = true to false. That will change the value for all tests, but you can also override the value on a class-by-class basis by including the assignment (set to false) in your individual class. There's no way to change this behavior to be fine-grained enough to only use the nontransactional behavior for a single method.

6.3 Why Fixtures Are a Pain

As great as fixtures are when you are starting out, using them long-term on complex projects exposes problems. Here are some things to keep an eye on.

Fixtures Are Global

There is only one set of fixtures in a default Rails application. So the temptation to keep adding new data points to the fixture set every time you need a corner case is pretty much overwhelming. The problem is that every time you add a user because you need to test what happens when a left-handed user sends a message to another user with a friend

3. In other words, Rails 1.x. You know, 2006.
4. I almost wrote, "This is super-nifty," but decided that wasn't professional enough.

relationship who happens to live in Fiji, or whatever oddball scenario you need, every other test has to deal with that data point being part of the test data.

Fixtures Are Spread Out

Fixtures live in their own directory, and each model has its own fixture file. That's fine, until you start needing to manage connections, and a simple setup of a user commenting on a post related to a given article quickly spans across four different fixture files, with no easy way to trace the relationships. I'm a big fan of "small and plentiful" over "large and few" when it comes to code structure, but even I find fixtures too spread out.

Fixtures Are Distant

If you are doing a complex test based on the specific fixture lineup, you'll often wind up with the end data being based on the fixture setup in such a way that, when reading the test, it's not clear exactly how the final value is derived. You need to go back to the fixture files to understand the calculation.

Fixtures Are Brittle

Of course, once you add that left-handed user to your fixture set, you're guaranteed to break any test that depends on the exact makeup of the entire user population—tests for searching and reporting are notorious culprits here. There aren't many more effective ways to kill your team's enthusiasm for testing like having to fix 25 tests on the other side of the world every time you add new sample data.

Sounds grim, right? It's not. Not only are fixtures perfectly suitable for simple projects, the Rails community has responded to the weaknesses of fixtures by creating factory tools that can replace fixtures in creating test data.

6.4 Using Factories to Fix Fixtures

The goals of a fixture replacement system are to take the three largest problems with fixtures—they are global, spread out, and brittle—and turn them into strengths. We want the system to be:

- *Local*: Each individual test should have its setup data tuned to the needs of that test. The setup data should be defined as closely as possible to the actual test.
- *Compact*: The setup data should be easy and quick to generate, otherwise lazy programmers (like me) just won't do it. It should be possible to generate a complex network of objects in just a few lines.
- *Robust*: Tests should not be dependent on changes made to setup data in other tests. We should even be able to specify more data in the current test without breaking other tests.

Generically, the answer to the fixture problem is a *data factory*.

6.5 Data Factories

The idea behind a data factory is that rather than specifying all the test data exactly, you provide a blueprint for creating a sample instance of your model. When you need data for a specific test, you call a factory method, which gives you an element based on your blueprint. You can override the blueprint to specify any data attributes required to make your test work out. Calling the factory method is simple enough to make it feasible to set up a useful amount of data in each test.

As I write this, there are at least four similar factory tools available for Rails, none of which seems to have a really dominant mindshare. The oldest of the three is Scott Taylor's FixtureReplacment (http://replacefixtures.rubyforge.org/). More recently, the ThoughtBot team behind Shoulda and other great tools provided factory_girl (http://github.com/thoughtbot/factory_girl/tree/master). There is also Pete Yandell's Machinist (http://github.com/notahat/machinist/tree/master) and OG Consulting's Object Daddy (http://github.com/flogic/object_daddy/tree/master). Of these, factory_girl probably has the largest user base.

The basic structure of all four tools is similar. Each gives you a syntax to create the factory blueprints and an API for creating the new objects. Both factory_girl and Machinist also provide a mechanism for creating unique streams of values according to a pattern, while FixtureReplacement has the most flexible creation syntax.

In describing how factory tools work, I'm going to start by focusing on Machinist, which is something of an editorial recommendation; it has my favorite combination of syntax and features. After we've covered the

basics via Machinist, we'll take a quick tour of all four packages and compare and contrast their syntax.

6.6 Installing Machinist

Machinist is distributed as a gem or as a Rails plugin, and is typically installed into your project using one or the other of those options:

```
$ sudo gem install machinist --source http://gemcutter.org
$ script/plugin install git://github.com/notahat/machinist.git
```

Data models in Machinist are called "blueprints," and you will typically place them in a test/blueprints.rb file. The blueprints file should start with the following two lines:

```
require 'machinist/active_record'
require 'sham'
```

Require the blueprint file from within your test/test_helper.rb file, like so:

```
require File.expand_path(File.dirname(__FILE__) + "/blueprints")
```

Finally, add a setup block to the test/test_helper.rb file:

```
setup { Sham.reset }
```

Why, you ask, is it critically important to reset your Sham before each and every test? All in good time, my friend, all in good time.

6.7 Creating and Using Simple Blueprints

Machinist adds a class method to ActiveRecord called blueprint(), which defines the default data for each class. The blueprint takes a block in which you can define default values on an attribute-by-attribute basis.

A very simple example for our Huddle network might look like this; each attribute has a simple default value. We'll get to associations in a moment:

```
Project.blueprint do
  name "Dog Meet Dog Dot Com"
  start_date Date.parse("2009-01-23")
end
```

Note the absence of equals signs—these are not assignments. Technically, they are function calls, so if it makes it more readable to write the lines like name("Dog Meet Dog Dot Com"), go for it.

A simple use of this blueprint calls the make() method, also created by Machinist, and is easily used in the setup of a test case or in a test itself:

```
setup do
  @project = Project.make
end
```

This creates a Project instance using the default values and saves it to the database. Validations on your ActiveRecord model will be called. (To be specific, Machinist will call save!() and reload() on the object before it is returned.) If you don't want to save the instance to the database, use make_unsaved() instead. The plan() method does not create an ActiveRecord object; it just returns a hash of the attributes defined by the blueprint. This is especially useful for use simulating an HTTP POST call in a controller test.

If you don't want to use the default values, you can pass a set of key/value pairs to the Machinist creation methods; these values will override the blueprint values:

```
setup do
  @project = Project.make(:name => "Soups Online")
  @other_project = Project.make(:name => "Google Thumbnail")
end
```

There are three important points to notice. First, the values in the blueprint can be blocks, rather than static values—in most cases, if you are using a random value, it's best to use a Sham object, but you can do the following:

```
Project.blueprint do
  name "Dog Meet Dog Dot Com"
  start_date { Date.today - rand(50) }
end
```

You can also refer to a previously assigned value later in the blueprint, which is where these factories start to get powerful:

```
Project.blueprint do
  name "Dog Meet Dog Dot Com"
  url { "#{name.downcase}.gsub!(" ", "_")" }
end
```

What's nice about this is that the blueprint will still use the value in the name attribute to calculate the URL, even if you pass the name in yourself:

```
test "machinist url" do
  soup = Project.make(:name => "Soups Online")
```

```
  assert_equal("soups_online", soup.url)
end
```

Inside the blueprint, you can call any attribute in the model that has a setter method; in other words, any virtual attribute in the model (like the password attribute of a RESTful Authentication User model) is fair game.

6.8 This Attribute Is a Sham!

Being able to specify default data can be especially useful (or at least, especially fun), when the default data is random. One problem with randomized data is that it's possible that when you create several objets from a limited set of possibilities, you might get duplicate data in a field that should be unique, leading to intermittent test failures. Random data also makes it difficult to reproduce a specific test case that might be causing a failure. Machinist avoids both problems with Sham objects.

Declaring a sham is simple. In your blueprint file, just call the Sham class with the name of the attribute you want shamified and a block that calculates the sham values. For instance:

```
Sham.label_color { %w(brown black white).rand }
```

Then the sham is referenced in the blueprint by calling the same name without the block:

```
Project.blueprint do
  name "Dog Meet Dog Dot Com"
  label_color { Sham.label_color }
end
```

If the name of the sham is the same as the name of the attribute, you can leave the block out and Machinist will infer the user of the sham. This makes blueprint declaration pretty succinct:

```
Project.blueprint do
  name "Dog Meet Dog Dot Com"
  label_color
end
```

In exchange for the added complexity of defining and calling the Sham object, you get a) a guarantee that the values coming out of the Sham are unique for each individual test and b) a guarantee that the values coming out will always be in the same order. If the colors are “black”, “white”, and “brown” the first time around, then for each test, the first

color created is “black”, the second is “white”, and you can probably guess the third.[5]

Hence, the Sham.reset() call in each setup block, which resets the random number generator, and then resets all the lists of values back to their initial index.

If for some reason you get in a state where there are no more unique values—in the above color example, if you tried to generate four different projects, for instance—Machinist throws an exception. You can avoid this for values where uniqueness is not important by passing an options argument to the Sham definition, like so:

```
Sham.color(:unique => false) { %w(brown black white).rand }
```

Unfortunately, there’s no way (yet) to turn off the guaranteed same sequence. That’s a shame—every now and then random values do find a bug in my code that might be hidden if I was getting the same values every time.

You can pass an index argument to the sham block, so as to allow sequential values:

```
Sham.title { |i| "Article_#{i}" }
```

6.9 Freedom of Association

Machinist makes it easy to specify related objects in your blueprints. In this example, the blueprint for the Project class calls the User blueprint to create the associated object by simply calling the make() function on the related class. (In a one to many relationship, you probably want to do this from the belongs_to() side. In a many to many relationship, it doesn’t matter what side creates the items.) However, you do need to make sure that only one side of the relationship creates items, otherwise you’ll get a circular dependency and a stack-too-deep exception.

```
Sham.first_name { Faker::Name.first_name }
Sham.last_name { Faker::Name.last_name }
Sham.label_color { %w(brown black white).rand }

User.blueprint do
  first_name
```

5. Internally, Machinist simply reseeds the random number generator with the same constant value every time. If you are actually doing something random in your project, this might conflict.

```
  last_name
end

Project.blueprint do
  name "Dog Meet Dog Dot Com"
  label_color
  user { User.make }
end
```

This example also uses the Faker gem (http://faker.rubyforge.org/) to generate random structured data. If you are following Rails conventions, and the model name matches the attribute name in the expected way, the blueprint can be simplified:

```
Project.blueprint do
  name "Dog Meet Dog Dot Com"
  label_color
  user
end
```

In either case, creating a machined Project object via Project.make() implicitly creates, verifies, and saves a machined User object. As with regular attributes, the value of the associated object is available for later attribute blocks:

```
Project.blueprint do
  name "Dog Meet Dog Dot Com"
  label_color
  user
  label { "#{user.first_name}'s project"}
end
```

You can pass in your own associated object just as with any other attribute defined in the blueprint. This is how you can create multiple sibling objects:

```
setup do
  me = User.make(:first_name => "Noel", :last_name => "Rappin")
  my_project = Project.make(:user => me)
  my_other_project= Project.make(:name => "Soups Online", :user => me)
end
```

If you don't want the related object to be created, you need to explicitly set the object to nil when calling make():

```
Project.make(:user => nil)
```

The Machinist documentation, for some reason, recommends abstracting common setups and placing them in the ActiveRecord class reopening the class by putting something like the following in the blueprint file itself:

```
Class Project < ActiveRecord::Base
  def make_my_users
    me = User.make(:first_name => "Noel", :last_name => "Rappin")
    my_dog = Project.make(:user => me)
    my_other_project= Project.make(:name => "Soups Online", :user => me)
  end
end
```

That's one way to go, but it's not the best idea to mess around in the ActiveRecord namespace more than strictly necessary (granted, everybody's definition of "strictly necessary" is different). It's a better choice to place the setup methods in a setup module either inside the blueprint file or outside of it.

6.10 Factories of the World Unite: Preventing Factory Abuse

The temptation when converting a project from fixtures to factories is to replicate your entire fixture setup from factory objects. You will get some benefits: the factory object will probably be easier to read and maintain than the fixtures were, and all your existing tests will pass. However, the factory tests will probably be significantly slower than transactional fixtures, and you still have the problem of global, far away data definitions—though at at least with factories, new tests can avoid the global data.

The way to use factories is to create less data for each test. Create only the smallest amount of data needed to expose the issue in each test. This practice speeds up the test, makes the issue easy to see rather than burying it among dozens of fixtures, and makes the correctness of the test itself easier to verify.

6.11 Factory Tool Comparison

As promised, here's a comparison guide to the main features of the four most popular factory tools. Each of these tools has subtle or more advanced features that are outside of the scope of this quick guide—check out the tools that look appealing to you.

Defining Factories

The basic structure of each of these tools is similar. You create a file made up of factories, each of which defines the basic data for a model. First, let's discuss that file. Each tool places it in a different location.

Factories and RESTful Authentication

The RESTful Authentication plugin, which is commonly used for managing user logins, has a couple of mild testing gotchas when using factories.

First of all, the UserTest class generated by RESTful Authentication contains a private method called create_user(). If you are using FixtureReplacement, that's a name crash with the user-generation method automatically created from your User class blueprint. In this case, you can change the name of the method in the UserTest class, adjust the tests that call it, and move on from there.

More generally, RESTful Authentication provides test user accounts as fixtures. If you are truly going to avoid fixtures, and you are using an older version of RESTful Authentication, then you may run into a problem with the login_as() method provided by RESTful Authentication and used throughout tests to set up a logged-in user. The login_as() method expects to take a symbol and look up the user data using the fixture-based users(:symbol)() method. Even if you convert the RESTful Authentication fixture data to a factory-based setup method (useful if only because the RESTful Authentication accounts have known, encrypted passwords), you still need to change the login_as() method in lib/authenticated_test_helper.rb to take an actual User object rather than a symbol:

```
def login_as(user)
  @request.session[:user_id] = user ? user.id : nil
end
```

You can then use this with your factories with code like this:

```
setup do
  login_as(User.make(:role => "admin"))
end
```

factory_girl
: Autoloaded if in test/factories.rb or spec/factories.rb() or test/factories/*.rb or spec/factories/*.rb; otherwise, manually loaded.

FixtureReplacement
: The file db/example_data.rb, automatically loaded by module. Also, code in that file must be declared inside module FixtureReplacement. Some users report that factories can be defined inline in your tests as well, but that feature appears undocumented.

Machinist
: The file test/blueprints.rb, by convention, but it's manually loaded, so it could go anywhere.

Object Daddy
: Typically, in /spec/examplars/model_exemplar.rb. Object Daddy puts the factories inside the ActiveRecord model class definition, so this is a place that monkey patches the model and will be auto-loaded during testing; however, the data can also be placed in the actual class definition in app/models.

Just as a note, my personal preference here is to keep the tools in the test directory—the placement for FixtureReplacement in the db directory bugs me. Object Daddy is the only tool that expects the factories to be within the model class.

Creating an Individual Factory

Within the factory file, the general idea is that multiple blocks are defined, one to each model—more or less. The following descriptions contains the syntax for creating a block; we'll cover what goes inside that block in a moment. Let's assume here a model called Project.

factory_girl
: For a factory matching a model name:

```
Factory.define :project do |p|
  «factory definition»
end
```

To define a factory with an arbitrary name:

```
Factory.define :active_project, :class => Project do |p|
  «factory definition»
end
```

FixtureReplacement

```
attributes_for :project do |p|
  «attribute definition»
end
```

Machinist

```
Project.blueprint do
  «definition»
end
```

To give an optional label:

```
Project.blueprint(:active) do
 «definition»
end
```

Object Daddy

N/A. Since Object Daddy puts the generators in the actual ActiveRecord model class, it doesn't need an explicit block.

There's a slight difference in structure here. Since factory_girl and Fixture Replacement are essentially defined in their own namespaces, the factory blocks take a single argument, which is the model being created. Machinist adds a class method to ActiveRecord base, so it doesn't need that argument—the block will be evaluated inside the class that the blueprint method is called on.

Defining Attribute Defaults

Within each factory block, individual attributes can have default values specified. Typically, these are either static values or random values (the Faker gem is very helpful here). Each tool has a slightly different syntax for defining basic attribute defaults.

Unless otherwise specified, these lines defining the defaults would go inside the factory blocks listed above. Also, unless otherwise specified, you can do other arbitrary code inside the factory definition to calculate any intermediate values you want.

factory_girl

For static default values:

```
p.name 'Test Project'
```

For dynamic default values:

```
p.name { «dynamic value»  }
```

Attributes can depend on earlier defined ones, if the block takes an argument:

```
p.long_name { Faker::Company.name }
p.short_name { |a| a.long_name[0, 2] }
```

FixtureReplacement

```
p.name = "Test Data"
```

The right-hand side can be an arbitrary expression, and can, I think, reference previously defined attributes.

Machinist

Static data:

```
name "Test Data"
```

Dynamic data can take a block:

```
name { "Test Data" }
```

Previously defined attributes can be referenced. All dynamic random data should be described in a Sham.

Object Daddy

Inside the active record model, there are four forms:

```
generator_for :name, "Test Name"
generator_for :name do "Test Name" end
generator_for :name, :method => :fake_name
generator_for :name, :class => NameGenerator
```

See next section for how to use these with sequences.

There is some slight difference here between FixtureReplacement's use of an assignment statement, Machinist using a method that takes a block, and factory_girl using a method that takes a value or a block. Object Daddy does everything a little differently.

Defining Sequences of Data

One nice feature of these factory tools is the ability to create sequences of related, nonrepeating data, whether that's just calling projects "Test 1", "Test 2", and so on, or a more complex sequence. Here is how to define a data sequence in each tool. Note that even though these examples take an index and are sequential, they can also be used with just random data.

factory_girl
: The sequence is defined in two steps. Outside the block defining the factory, define the sequence:

```
Factory.sequence :name do |n| "Test Project #{n}" end
```

Then, inside the define block for a factory:

```
p.name { Factory.next(:name) }
```

FixtureReplacement
: No explicit support, but does define a 'random_string' method.

Machinist
: The sequence is defined in two steps. First, outside the blueprint block:

```
Sham.name { |index| "Test Project #{index}" }
```

Then, inside the blueprint index:

```
name { Sham.name }
```

But if the Sham and the attribute have the same name, this can be written as just:

```
name
```

And Machinist will make the right connection. Note that the Sham class needs to be reset in your setups.

Object Daddy
: The generator_for method takes a :start option to define the initial value of the sequence. The block and method versions take an optional argument, :prev; this is the previous value of the sequence, which can be used to generate the next value. The class version can maintain its own sequence, as long the class provides a next() method.

Defining Associations

A factory also allows you to define default associations with dependent objects. This isn't always what you want—frequently, you're better off specifying the associations in the actual test. For the purposes of the example, let's assume that the Project belongs to a Client, and again, all this code is in the block defining the example, unless otherwise specified. These also assume that there is a separate factory definition for Client.

factory_girl

```
p.client { |a| a.association(:client) }
```

Which can be abbreviated:

```
p.association :client
```

In each case, extra arguments are treated as overrides to the default data of the other object.

FixtureReplacement

```
p.client = default_client
```

Additional arguments to the default_client() method are treated as overrides.

Machinist

```
client { Client.make }
```

Again, additional arguments can be specified as overrides. If the attribute and the blueprint have the same name, this can be shortened to:

```
client
```

Object Daddy
: A belongs_to association is automatically created (by calling generate() on the associated class) when needed.

Also note that has_many style relationships cannot be created directly in any of these tools, because ActiveRecord needs the object to be saved first. These relationships need to be created in the actual test setup method.

Importing the Factory Data

Having spent all this time defining attributes, you are probably going to want to declare some in your tests.

First, you actually need to include or require some things to allow your tests to be aware of the factory replacement data file.

factory_girl
: No further steps once installed, unless you want to include a factory file from a nonstandard location.

FixtureReplacement
: In Test::Unit, Test::Unit::TestCase must include FixtureReplcement. In RSpec, the Spec::Runner.configure block must have the line config.include FixtureReplacement.

Machinist
: The blueprint file must be required in test_helper.rb or spec_helper.rb. You must add a setup or before block called before each test containing the line Sham.reset.

Object Daddy
: No further steps once installed, unless you include generator files in a nonstandard location.

Using Factory Data

Now let's write the lines in the test that will create the data.

In general, all these methods allow you to override the defined default values by passing them as key/value attributes to the method creating the object. In general, this works for attributes as well as associations. The examples below show the creation of the object with the name field overridden, just to give a sense of the syntax. Some of the tools have different options, depending on whether you want the object saved to the database.

factory_girl
: Create and save an object:

```
Factory.create(:project, :name => "New Project")
```

Create but don't save:

```
Factory.build(:project, :name => "New Project")
```

Create an object as series of stubs:

```
Factory.stub(:project, :name => "New Project")
```

Return the attributes as a hash:

```
Factory.attributes_for(:project, :name => "New Project")
```

FixtureReplacement
: Create and save:

```
create_project(:name => "New Project")
```

Create and don't save:

```
new_project(:name => "New Project")
```

Create as a dependent object:

```
default_project
```

Machinist

Create and save:

```
Project.make(:name => "New Project")
```

Create and don't save:

```
Project.make_unsaved(:name => "New Project")
```

Return the attributes as a hash:

```
Project.plan(:name => "New Project")
```

Named blueprints can be invoked by having the name be the first argument to any of the above methods:

```
Project.make(:active)
```

Object Daddy

Create and save:

```
Project.generate(:name => "New Project")
```

Create and save, but raise exception if invalid:

```
Project.generate!(:name => "New Project")
```

Create and don't save:

```
Project.spawn(:name => "New Project")
```

All four of these tools are exceptionally good at allowing you to quickly create sample data for use in a test. You should choose the one whose syntax and features match your own preferences. Let's move on to a slightly different topic in model data that causes problems of its own.

6.12 Managing Date and Time Data

Calendar logic has a well-deserved reputation as one of the most annoying parts of a program that doesn't actually involve Unicode. Testing calendar logic—time-based reports, automatic logouts, "1 day ago" text displays—can be a headache, but there are a couple of things you can do to simplify the time logic beast.

You're Doing It Wrong

Picture this. You've got a YAML file with some projects:

```
runway:
  name: Project Runway
  start_date: 2010-01-20

greenlight:
  name: Project Greenlight
  start_date: 2010-02-04

gutenberg:
  name: Project Gutenberg
  start_date: 2010-01-31
```

You'd like to test some time-based code, as might be used in a search or report result; this goes in test/unit/project_test.rb:

```
test "reports based on start date" do
  actual = Project.find_started_in_last(6.months)
  assert_equal(3, actual.size)
end
```

And here's the code that makes the test pass, from app/models/project.rb:

```
def self.find_started_in_last(time_span)
  old_time = Date.today - time_span
  all(:conditions => ["start_date > ?", old_time.to_s(:db)])
end
```

On January 20, 2010, the test passes. And on the 21st it will pass, and the day after...

Six months from now, though, on about June 20th, when you've probably long forgotten about this test, this data, and maybe even this project, the test will suddenly fail. And you'll spend way too much time trying to figure out what happened, until you remember the date issue, and realize that the January 20th project has moved out of the six-month time span specified in the test. Of course, changing all the dates just pushes the problem forward, and gives you time to forget all about it again.

This issue may sound silly to some, but like many of the more ridiculous examples in the book, this is a mistake that a) happened to me and b) can end up costing a lot of time.

Long ago, when I was young and foolish, I solved this problem by adding an optional argument to just about every method that used Date.today(), allowing an optional time to be passed to the method and allowing an explicit date to be used for testing. This was way more work than was actually needed, so here are a couple of better ideas.

Relative Dates in Fixtures

As mentioned in Section 2.7, *More Info: Getting Data into the Test*, on page 35, fixture files are evaluated as ERb files before loading. For our purposes, that's helpful because it allows us to specify dates dynamically, like so:

```
runway:
  name: Project Runway
  start_date: <%= Date.today - 1.month %>

greenlight:
  name: Project Greenlight
  start_date: <%= Date.today - 1.week %>

gutenberg:
  name: Project Gutenberg
  start_date: <%= Date.today - 1.day %>
```

With fixtures written like this, the above test will always work, since the start_date of the projects will never fall out of the six-month range. (If you are using a factory tool instead of fixtures, you can do something similar in your factory blueprint.)

While this technique works quite well for keeping test data a consistent relative distance from the test time, it's less helpful if you are actually trying to test the exact value of one of the dates—when testing, say, output display. With the first, static, set of fixture data, you could write:

```
test "that project dates are displayed in this goofy format" do
  assert_equal("2010 1 January", projects(:runway).goofy_start_date)
end
```

This test is a lot more difficult to write if you don't explicitly know the value of the project's start_date. But keep reading...

Timecop

Recently, I've been solving my time problems with the help of a nice little gem called Timecop written by John Trupiano, which you can find at http://github.com/jtrupiano/timecop. Timecop can be installed with gem install timecop, or placed in your environment.rb file.

Timecop is essentially a super-specific mock object package: it stubs out Date.today(), DateTime.now(), and Time.now(), allowing you to explicitly set the effective date for your tests. Using Timecop, the original test could be rewritten as follows:

```
test "reports based on start date" do
```

Explicit Timestamps

One trick worth mentioning when testing dates is explicitly setting the created_at attribute of your ActiveRecord model. Normally, created_at is a timestamp automatically generated by Rails, and it's often used for the kind of time-based reporting alluded to in the rest of this section. Since it's automatically created at the current time, you can get into some weird situations if other dates are specified in the past. Even without that complication, you may still need to explicitly set created_at to use the attribute to test time-based features.

You can set created_at in the fixture file, just like any other attribute, or it can be specified in ActiveRecord::create() or ActiveRecord::new(), in a factory blueprint, or just plain reset with an assignment or update method.

Setting updated_at is trickier. Under normal circumstances, if you try to explicitly set updated_at, Rails will just automatically reset it on save, which completely defeats the purpose. To change this behavior, set the class variable Model.record_timetamps = false sometime before you save the object with modified update time. Instead of Model use the model class that is actually being saved. After the save, reset things back to normal with Model.record_timestamps = true.*

*. See http://www.neeraj.name/blog/articles/800-override-automatic-timestamp-in-activerecrod-rails for some ways to make this call a little friendlier using Ruby Eigenclasses.

```
  Timecop.freeze(Date.parse("2010-02-10"))
  actual = Project.find_started_in_last(6.months)
  assert_equal(3, actual.size)
end
```

The Timecop.freeze() command stubs the current date and time methods back to the date passed as the argument—in this case, February 10, 2010. Time does not move for the duration of the test. A separate method, Timecop.travel(), resets the time, but lets the system time move forward from that point onward.

Why both options? Because keeping time constant for the life of a test makes the test environment more consistent. (For example, RESTFul Authentication has an intermittent test failure if the time rolls over in just the right way during one test.) But sometimes, it is necessary for

time to move forward, so Timecop offers both options. Along those lines, it's sometimes useful to put the following line in a setup method:

```
Timecop.freeze(Date.today)
```

With the following line in a teardown block:

```
Timecop.return
```

Why? Because it ensures that the current time doesn't change for the duration of each test. Again, with certain kinds of timing-related issues, that consistency eliminates a possible source of intermittent test failures or just plain confusion.

The argument to freeze() or travel() is an instance of Date, DateTime, Time, or a series of arguments of the form (year, month, day, hour=0, minute=0, second=0). Both methods also take blocks such that the fake time is only good for the duration of the block:

```
test "reports based on start date" do
  Timecop.freeze(Date.parse("2010-02-10")) do
    actual = Project.find_started_in_last(6.months)
    assert_equal(3, actual.size)
  end
end
```

The time travel methods can be in your setup, or in an individual test. You can also change the time in the middle of a test, to speed up an ongoing process:

```
test "is the project over" do
  p = Project.new(:start_date => Date.today,
      :end_date = Date.today + 8.weeks)
  assert !p.complete?
  Timecop.freeze(Date.today + 10.weeks)
  assert p.complete?
end
```

Timecop lets you keep explicit dates in your test data without causing problems later on. The only downside is that if you have many tests setting time to different days, it can get somewhat confusing in the aggregate. It's easier if you use the same start date consistently. (On a solo project, you might use your birthday, for instance, but that's probably overly cute for a team project.) A more minor problem is that the line at the end of your test runs saying how long the test suite took will be hopelessly messed up due to the continued messing with Time.now.

Comparing Dates and Times

Ruby, not content with a simple date and time system, has three separate classes that manage date and time data. The Time class is a thin wrapper around the same Unix C library that pretty much every language exposes. (Motto: Annoying programmers since 1983!) There are also the Ruby-specific classes Date and DateTime, which are more flexible and have a more coherent API, but are slower.

For testing purposes, the relevant points are a) ActiveRecord uses Date and DateTime, depending on the specifics of the underlying database column, b) comparing a Date to a DateTime instance will always fail (as will trying to add or subtract them), and c) most of the Rails ActiveSupport methods (think 5.days.ago) return DateTime.

In testing, this can lead to a lot of annoying failures, especially when you have a Date column with no time information—which is recommended if the time is not important.

In general, it's a good idea to compare dates and times by converting them using to_s(:db). It avoids the irritating question of object equality, and you tend to get more readable tests and error messages. When the exact time of the time object is in question, try to force the issue by using the Rails ActiveSupport methods to_date(), to_time(), and to_datetime(). Most commonly, this means something like 5.days.ago.to_date.to_s(:db), which may read a touch on the awkward side, but is a robust test with a decent error message on failure.

6.13 Model Data Summary

To sum up, Rails provides fixtures as an exceptionally simple way to create a set of test data that can be shared across multiple tests. However, fixtures are so simple that they tend to not be adaptable to more complex product needs. Factory tools, which take a little bit more initial setup, allow for more flexibility in use at some cost in test performance. The two structures don't have to be mutually exclusive. One pattern for combining them is to create exactly one complex scenario in fixtures for use in integration or complex controller tests, and to use factories for unit tests or simpler controller tests.

Fixtures and factory tools allow you to get test data into your database in order to create a known baseline for testing. However, in some cases you may not want to actually place data in the database. Using the

database from a test may be undesirable for performance reasons, for philosophical reasons (some people don't consider it reasonable to touch the database in a "unit" test), or where logistical reasons make objects hard to create. In the next chapter, we'll explore mock objects, which allow tests to proceed by faking not the data, but rather the actual method calls that produce the data.

Chapter 7

Using Mock Objects

A *mock object* is a "fake" object used in place of a "real" object for the purposes of automated testing. A mock might be used when the real object is unavailable or difficult to access from a test environment—a common example is an external credit-card payment system. A mock might also be used to easily re-create a specific application state that would be otherwise difficult to trigger in a test environment, like a database or network failure. Mocks can be used strategically to limit the scope of a test to the object and method specifically under test. Used in that manner, mocks drive a different style of testing, where the test is verifying the behavior of the system during the test, rather than the state of the system at the end of the test.

7.1 What's a Mock Object?

One complicating factor in dealing with mock objects is that pretty much everybody who creates a mock framework feels perfectly free to use slightly different naming conventions than everybody else. Here are the names that I use, which are—of course—also the correct ones.[1]

The generic term for any object used as a stand-in for another object is *test double*, by analogy to "stunt double," and with the same connotation of a cheaper or more focused replacement for a more expensive real object. Colloquially, *mock object* is also used as the generic term but—confusingly—is also the name of a specific type of test double.

1. Actually, I believe this naming structure is the creation of Gerard Meszaros in *xUnit Test Patterns* [Mes07].

A *stub* is a fake object that returns a predetermined value for a method call without calling the actual object. We can create a stub as follows (This uses the Ruby gem Mocha, but you don't need to worry about the exact syntax just yet):

```
thing.stubs(:name).returns("Fred")
```

That line of code says that if you call thing.name, you'll get Fred as a result. Crucially, the actual thing.name method is not touched, so whatever value the "real" method would return is not relevant; the Fred response comes from the stub, not the actual object. If thing.name is not called in the test, nothing happens.

A *mock* is similar to a stub, but in addition to returning the fake value, a mock object also sets a testable expectation that the method being replaced will actually be called in the test. If the method is not called, the mock object triggers a test failure. So, when you write the following snippet to create a mock object instead of a stub:

```
thing.expects(:name).returns("Fred")
```

Then if you call thing.name in your test, you still get Fred, and the actual thing.name method is still untouched. But if you don't call thing.name in the test, the test fails with what's generally called a MockExpectationError, or some such.

In other words, setting a stub on a method is passive, and just says, "Ignore the real implementation of this method and return this value," while setting a mock on a method is more aggressive, and says, "This method will return this value, and you better call the method, OR ELSE!"

The reason you might set such an expectation is that once you've stubbed the method, it makes no sense to write an assertion on it like this one:

```
thing.stubs(:name).returns("Fred")
assert_equal "Fred", thing.name
```

In this case, you're just testing that the stub works as advertised—this test can't fail. But if you use the mock:

```
thing.expects(:name).returns("Fred")
```

Then your code actually has to behave a certain way to pass the test.

One of the nice side effects of Ruby's openness and metaprogramming functionality is that mock object libraries are easier to write and have

more flexibility and power than similar libraries in other languages (Java, I'm looking at you...). There are at least four widely used Ruby mock packages as I write this. All have broadly similar features with slight differences in emphasis and syntax. FlexMock (http://flexmock.rubyforge.org/) is the oldest and the one I have the most actual experience with. Mocha (http://mocha.rubyforge.org/) is in use by the Rails core team in integration testing. Double Ruby (http://github.com/btakita/rr/tree/master) is a newer library with perhaps a cleaner syntax and a couple of unique features. Finally, RSpec has its own mock library (discussed in Chapter 12, *RSpec*, on page 173), although it allows you to use any of the other three if you want.

In the absence of any compelling constituency for any of these three, we'll use Mocha in these examples, on the grounds that it's pretty close to actually being part of core Rails. However, in Section 7.7, *Comparing Mock Object Libraries*, on page 121, we'll compare and contrast the syntax of the various mock frameworks.

Thus ends the blathering. Here's how you actually use the things.

Install Mocha as a gem (gem install mocha) or as a plugin (script/plugin install git://github.com/floehopper/mocha.git). In order to use Mocha, place:

```
require 'mocha'
```

In any test file that will need it or in test/test_helper.rb.

7.2 Stubs

Although basically similar, stubs and mocks fit into the pattern of your tests very differently. I find it easier to start by describing stubs. A stub is a replacement for all or part of an object that prevents a normal method call from happening, and instead returns a value that is preset when the stub is created.

In Mocha, you can create an object that only exists as a set of stubbed methods by using the stub() method, which is available throughout your test cases. Since Ruby uses duck-typing and therefore only cares whether objects respond to the messages sent to them, a stub object created in such a way can be injected into your application as a replacement for a real object.

```
test "here's a sample stub" do
  stubby = stub(:name => "Paul", :weight => 100)
  assert_equal("Paul", stubby.name)
```

```
end
```

The hash arguments to stub() list the methods that the stubbed object responds to and the values returned. So the assertion in the second line is true because the stub has been preset to respond to the name message with "Paul". If you call the stub with a method that is not in the hash argument, Mocha will return an error. However, Mocha provides the stub_everything() method, which instead returns nil for methods not in the hash argument. Using stub_everything() makes sense in the case where there are a large number of potential methods to be stubbed, but where the values make so little difference that specifying them reduces the readability of the test

In case it's not clear, this test is a very stupid way to use stubs; I've set up a nice little tautology and I haven't actually learned anything about any larger system around this test.

You would use a bare stub object to stand in for an object that is unavailable or prohibitively expensive to create or call in the test environment. In Ruby, though, you would more often take advantage of the way Ruby allows you to open up existing classes and objects for the purposes of adding or overriding methods. It's easy to take a "real" object, and only stub out the methods that you need. This is extraordinarily useful when it comes to actual uses of stub objects.

In Mocha, this is managed with the stubs() method, which is mixed in to any Ruby object:

Download huddle_mocha/test/unit/project_test.rb

```
Line 1  test "lets stub an object" do
2         stub_project = Project.new(:name => "Project Greenlight")
3         stub_project.stubs(:name)
4         assert_nil(stub_project.name)
5       end
```

This test passes: line 3 sets up the stub, and the stub_project.name call in line 4 is intercepted by the stub to return nil, and never even gets to the actual project name.

Having a stub that always returns nil is a little pointless, so Mocha allows you to specify a return value for the stubbed method using the following syntax:

Download huddle_mocha/test/unit/project_test.rb

```
Line 1  test "lets stub an object again" do
2         stub_project = Project.new(:name => "Project Greenlight")
```

```
3   stub_project.stubs(:name).returns("Fred")
4   assert_equal("Fred", stub_project.name)
5 end
```

Line 3 is doing the heavy lifting here, tying the return value Fred to the method :name. Technically, stubs() returns a Mocha Expectation object, which is effectively a proxy to the real object. The returns() method is a method of that Expectation object that associates the return value with the method.

Since classes in Ruby are really just objects themselves, you'd probably expect that you can stub classes just like stubbing instance objects. You'd be right:

Download huddle_mocha/test/unit/project_test.rb

```
Line 1 test "let's stub a class" do
2   Project.stubs(:find).returns(Project.new(:name => "Project Greenlight"))
3   project = Project.find(1)
4   assert_equal("Project Greenlight", project.name)
5 end
```

In this test, the class Project is being stubbed to return a specific project instance whenever find() is called. In line 3, the find() method returns that object via the stub when Project.find() is called.

Now we may be getting somewhere... you'll notice that this test uses the results of a find() method without actually touching the database. It's not hard to find Rails programmers who would consider the database to be prohibitively expensive to use in a test environment, and this is one—admittedly, over-simplified—strategy for avoiding it. Again, remember that this stub shouldn't be used to verify that the find() method works; it should be used by other tests that need the find() method along the way to the other logic that is actually under test.

There are a couple of advanced usages of returns() that might be interesting now and again. If you have multiple return values specified, the stubbed method returns them one at a time, as the following irb sessions shows:

```
>> stubby = Project.new
=> #<Project id: nil .... >
>> stubby.stubs(:user_count).returns(1, 2)
=> #<Mocha::Expectation:0x221e470... >, side_effects[]
>> stubby.user_count
=> 1
>> stubby.user_count
=> 2
```

```
>> stubby.user_count
=> 2
```

The return values of the stubbed method walk through the values passed to returns(). Note that the values don't cycle; the last value is repeated over and over again.[2]

You can get the same effect with a little more syntactic sugar by using the then() method—you can chain together as many of these as you want:

```
stubby.stubs(:user_count).returns(1).then.returns(2)
```

A very common use of stub objects is to simulate exception conditions. If you want your stubbed method to raise an exception, you can use the raises() method, which takes an exception class and an optional message:

```
stubby.stubs(:user_count).raises(Exception, "oops")
```

You can even chain returns() and raises():

```
stubby.stubs(:user_count).returns(1).then.raises(Exception)
```

Another common use case is if you want all instances of a class created during a test to respond to the same stub. This is managed with the class method any_instance(), followed by any returns() or raises() expectation you care to add. As in:

```
Project.any_instance.stubs(:save).returns(false)
```

With this little trick, you can rectify a nagging annoyance in the standard Rails scaffolds. As currently constituted (at least, as of this writing), the Rails generated tests for a standard script/generate scaffold controller do not cover 100% of the controller methods. Specifically, the failure conditions for create() and update() are not covered. I've always assumed, with no real justification, this oversight was because the easiest way to test these is with a mock package, and the Rails team didn't want to mandate one particular package.[3]

Since we've already mandated a mock package, here's a couple of sample tests that use the any_instance() call to validate the error behavior

2. For some reason, the Mocha RDoc says that returns([1, 2]) is the same as returns(1, 2)—not true, according to my testing. returns([1, 2]) returns the array [1, 2], and returns(1, 2) returns 1, then 2 on successive calls.
3. Alternate possibility: they just figured it was too minor to care about.

for create() and edit(). This is for the Huddle Project class—you'll need a slight tweak for your own classes:

Download huddle_mocha/test/functional/projects_controller_test.rb

```
Line 1  test "fail create gracefully" do
        -   assert_no_difference('Project.count') do
        -     Project.any_instance.stubs(:save).returns(false)
        -     post :create, :project => {:name => 'Project Runway'}
      5     assert_template('new')
        -   end
        - end
        -
        - test "fail update gracefully" do
     10   Project.any_instance.stubs(: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)
     15 end
```

These two tests have the same basic format. The first command in each one sets an any_instance() expectation (lines 3 and 10); then the actual controller command is run (lines 4 and 11). After that, validation: first that the error-appropriate template is used (lines 5 and 12); then that the actual creation or update did not take place. For create, that's the assert_no_difference() call validating that Product.count doesn't change, and for update, it's validating that the :name => 'fred' in the update form doesn't actually get sent to the database.

The truly sharp-eyed among you have probably realized that, while the create() version of this test needs to use any_instance() because the exact instance being created is not known at runtime, the update() version could, in fact, include a stub on project(:huddle).id, since that's the only instance under consideration, and its identity is known before the controller call. Fair point. In practice, though, there's no guarantee that the find() method in the controller will return the exact same object as is used in the test—it will most likely create a brand-new instance that is a copy of the fixture data, but loaded from the database. If so, a stub limited to the particular instance created in the test would not apply to the instance created in the controller. After we introduce with() in the next section, we'll see one potential hack/workaround for this issue.

A related gotcha to watch out when using any_instance() is that a stub or mock declared via any_instance() only applies to instances that are created after the declaration. Specifically, Rails fixture objects, accessed via one of the special fixture methods like users(:fred) that have already

been generated when the test starts will not reply to the stub or mock—the object needs to be re-created from the database for the double to apply.

In addition, the find() method call in line 13 is required in order to force a check all the way back to the database to see if the database record is changed—you could get the same effect by calling reload() on projects(:huddle). Otherwise, changes made to the database won't have been reflected on the instance already created and in memory. And, last and perhaps most obviously, when you adapt this to your own classes, the form part of the call in line 11 needs to have attributes that are actually part of the class under consideration.

7.3 Stubs with Parameters

The next level in tuning the stub is to have it return different values based on the input parameters. In Mocha, this is managed using the with() method:

Download huddle_mocha/test/unit/project_test.rb

```
test "let's stub a class again" do
  Project.stubs(:find).with(1).returns(
      Project.new(:name => "Project Greenlight"))
  Project.stubs(:find).with(2).returns(
      Project.new(:name => "Project Blue Book"))
  assert_equal("Project Greenlight", Project.find(1).name)
  assert_equal("Project Blue Book", Project.find(2).name)
end
```

In its simplest form, shown in the example above, the with() method takes one or more arguments. When the stub() method is called, Mocha searches for a match between the arguments passed and the declared stubs, and returns the value matching those arguments.

One thing to be careful of is that by setting expectations tied to specific input values, you are limiting the Mocha stub to only those input values. In other words, if we were to try Project.find(3) in this test, the test would fail—which is a counterintuitive result for a stub. The failure triggers the following rather cryptic error message:

```
test_let's_stub_a_class_again(ProjectTest)
      [/test/unit/project_test.rb:43]:
unexpected invocation: Project(id: integer, name: string,
created_at: datetime, updated_at: datetime, start_date: date,
end_date: date).find(3)
satisfied expectations:
```

```
- allowed any number of times, already invoked once:
Project(id: integer, name: string, created_at: datetime,
updated_at: datetime, start_date: date, end_date: date).find(2)
- allowed any number of times, already invoked once:
Project(id: integer, name: string, created_at: datetime,
updated_at: datetime, start_date: date, end_date: date).find(1)
```

The guts of this message will perhaps be a little clearer after we've discussed mocks a little bit more, but the gist is simple: we did something Mocha didn't expect, and Mocha doesn't like surprises.

A with() descriptor can be attached to either return values or raised exceptions:

```
Project.stubs(:find).with(1).returns(
    Project.new(:name => "Project Greenlight"))
Project.stubs(:find).with(nil).raises(Exception)
assert_equal("Project Greenlight", Project.find(1).name)
assert_raises(Exception) { Project.find(nil).name }
```

The with() declaration can be made more complicated in several ways—frankly, practical application of some of these eludes me, but we'll run through them quickly.

Most generally, we can pass a block to 'with' instead of an argument:

```
proj = Project.new()
proj.stubs(:status).with { |value| value % 2 == 0 }.returns("Active")
proj.stubs(:status).with { |value| value % 3 == 0 }.returns("Asleep")
```

When the stubbed method is called, if the with() block returns true, then the expectation is considered matched:

```
>> proj.status(2)
=> "Active"
```

If more than one block returns true, it seems as though the last one declared wins:

```
>> proj.status(3)
=> "Asleep"
>> proj.status(6)
=> "Asleep"
```

If none of the blocks returns true, we get a unexpected invocation error, as we did just a second ago.

Mocha also defines a bunch of parameter matchers that give more flexible with() behavior. This is an incomplete list of the ones that seem most useful. Note that all of these behaviors can be implemented using the block syntax.

The instance_of() matcher, and its related cousin is_a(), match any incoming parameter that is of the given class. Use it like this:

```
proj = Project.new()
proj.stubs(:tasks_before).with(instance_of(Date)).returns(3)
proj.stubs(:tasks_before).with(instance_of(String)).raises(Exception)
```

This or any other Mocha matcher can be negated with the Not() method. (Yes, it's capitalized. Maybe to avoid weird parse collisions with the keyword **not**...):

```
proj = Project.new()
proj.stubs(:tasks_before).with(Not(instance_of(Date))).returns(3)
```

We can apply a stub to more than one possible argument with the any_of`() matcher:

```
proj.stubs(:thing).with(any_of('a', 'b')).returns('abababa')
```

Which would match against either of:

```
proj.thing('a')
proj.thing('b')
```

We can also nest any_of() with other matchers, though we can quickly get tangled in a pile of syntax:

```
proj.stubs(:thing).with(any_of(instance_of(String)),
    instance_of(Integer)).returns("Argh")
```

Another useful matcher is regexp_matches(), which allows us to match against—guess what? a regular expresssion:

```
proj.stubs(:thing).with(regexp_matches(/*_user/)).returns("A User!")
```

A hash argument can be matched against the existence of a specific key/value pair with the has_entry() matcher.

```
proj.stubs(:options).with(has_entry(:verbose => true))
```

The stub in this snippet will match any hash argument that contains a :verbose => true entry, no matter what the other contents of the hash might be.

There's about a dozen more of these matchers, many of which seem to be, shall we say, somewhat lacking in real-world practical value. Rather than cluttering your head with a bunch of stuff you'll never use, I invite you to check out the Mocha docs at http://mocha.rubyforge.org for a full listing.

One possible use of with() is to help work around the issue with ActiveRecord objects mentioned in the last section. Anthony Caliendo came up

with the following clever solution for creating a mock or stub on an ActiveRecord object in your test and getting it to still be applied to the ActiveRecord object created by the controller.[4]

Remember, the problem is that the database call from the controller creates a completely different Ruby object than the one you have attached a stub to. But you can dig into the ActiveRecord internals and define this in your test helper:

```
Line 1  def mock_active_records(*records)
2     records.each do |record|
3       record.class.stubs(:instantiate).with(
4           has_entry('id' => record.id.to_s)).returns(record)
5       record.stubs(:reload).returns(record)
6     end
7   end
```

The key phrase here is stubs(:instantiate).with in line 3. That method is called with a set of key/value pairs used to create an ActiveRecord object. Then in line 4, Mocha's has_entry decorator is used to declare that if the set of key/value pairs contains an entry for the id that matches the known record's ID, then return that object directly. The instantiate() method is called from ActiveRecord::Base::find(), so any mechanism for trying to retrieve this object from the database will be caught here such that if the ID of the object you are requesting matches one of the known objects, that object is returned without a new trip to the database. The reload() method is similarly stubbed.

A sample usage might look like this:

```
test "My projects might be properly saved" do
  @bluebook = Project.make(:name => "Project Bluebook")
  @runway = Project.make(:name => "Project Runway")
  mock_active_records(@bluebook, @runway)
  @bluebook.stubs(:save => true)
  @runway.stubs(:save => false)
end
```

Note that you have to mark the records you are going to use with the mock_active_records() method as well as actually specify any other stub or mock on those objects. There are a couple of things to watch out here, the most glaring of which is that all ActiveRecords you might find in your test need to be in the mock_active_records() call, since an attempt to call the stubbed instantiate() method with a non-matching hash would trigger an expectation error. In a factory universe, with

4. This method is described in more detail at http://www.pathf.com/blogs/2009/08/using-mocha-for-activerecord-partial-mocks-w

only a couple of object defined, that may not be a difficult constraint to live with. Also, the internals of ActiveRecord may change some time in the future, causing this mechanism to stop working.

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, 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 that were applied to stubs, such as returns(), raises(), any_instance(), 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
  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)
```

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

```
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 62, 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 itself. 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 above 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 non-mock 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 above 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 175 and Chapter 15, *Acceptance Testing with Cucumber*, on page 194.

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 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. 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

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

might indicate a poor object-oriented design where one class is asking for too many details of a different class.

- 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
: 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 173, 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.

This is a quick tour of common features, and not a complete look at each of these packages. Check the documentation for all the details and quirks—RR, in particular, has features that don't map to the other tools.

Loading into Test Suite

The first part of using any of these packages is installing and integrating with Test::Unit. The RSpec mocks, of course, can't be integrated with Test::Unit; however, any of the other packages can be integrated with RSpec by adding the line config.mock_with :rr or :flexmock, or :mocha in the spec_helper.rb file.

Flexmock

```
% sudo gem install flexmock
```

Then, in test_helper.rb:

```
require 'flexmock/test_unit'
```

Mocha

```
sudo gem install mocha
```

Then, in test_helper.rb:

```
require 'mocha'
```

RSpec

N/A

RR

```
sudo gem install rr
```

Then, inside the test case declaration in test_helper.rb:

```
include RR::Adapters::TestUnit
```

Creating Blank Stubs

The most basic function of any of these packages is creating a simple stub object in which you can specify the return value of one or more methods. When those methods are called, the specified value is returned. If the methods are not called, nothing happens. Here's the syntax to create a stub object that is not connected to any pre-existing object in the application.

Flexmock

```
stub = flexmock("name", :method => result)
```

Mocha

```
object = stub(:method => result)
```

The method/result pairs can also be specified in a block argument as shown in the next section.

RSpec

```
stub = stub("name", :method => value)
```

RR

```
double = stub(Object.new).method { value }
```

Which can be abbreviated as:

```
double = stub!.method { value }
```

Creating Stubs from Existing Objects

Most of the time, though, you'll want to create stubs that replace methods on existing objects. In Flexmock and RR, this involves calling a special method with the object as an argument, as in Flexmock's flexmock(object), while in Mocha and RSPec, this involves calling a method of the object itself, as in object.stubs. In either case, further information about the method being stubbed and its return value is usually chained after the declaration of the stub.

Remember, classes are just another kind of object in Ruby, so class methods can be treated like any other method, as in stub(User).should_receive(:find).

Flexmock

```
stub = flexmock(project).should_receive(:method).and_return(value)
```

If the object being mocked is a string or symbol, use :base as the first argument to prevent a confusion with a simple test double containing just the name. In other words, a call flexmock("fred") is ambiguous. As written, it is just a bare flexmock object with the name fred. If you acutally want to stub methods on the actual string "fred", then use flexmock(:base, "fred").

The :safe argument is used in the case where your object being mocked might already define methods with the same name as the ones used by Flexmock. When called via :safe, Flexmock will not

add extra methods to the existing object. Since those methods don't exist, a Flexmock object declared with :safe can only be used by having any expectations defined in an attached block.

```
mock = flexmock(:base, object)
mock = flexmock(:safe, project) { |mock| mock.should_receive(:a) }
```

Multiple should_receive calls can be chained to a single stub, in which case they cycle in the same way as Mocha values. Values can also be set in a block argument to flexmock(), as in:

```
flexmock(obj) do |m|
  m.should_receive(:method).and_return(value)
end
```

To specify errors, and_raise(exception) is used. The syntax :method => value can be used as a short cut for and_return() in either of the following forms:

```
flexmock(obj).should_receive(:method => value)
flexmock(obj, :method => value)
```

Mocha

```
obj.stubs(:method).returns(value)
obj.stubs(:method, value)
```

The two methods above are identical. To specify errors, use raises(exception) instead of returns. Multiple values in the returns method cycle can also be written returns(1).then.returns(2).

RSpec

```
project.stub!(method).and_return(1)
```

The and_return() method can also take a block or a list of values, which is treated as Mocha or FlexMock. Use and_raise() to raise an error.

RR

```
stub(project).method { value }
```

Which can also be written as:

```
stub(project).method.returns(value)
```

Creating Mocks with Expectations

All these packages allow you to create a mock object with an expectation that the method will be called a specified amount of times. The biggest difference here is that FlexMock does not have a separate syntax for creating objects with expectations; any doubled object can have an expectation added by appending a method like once() to the description chain. In Mocha, RSpec, and RR, test doubles that will have expectations must be declared as such, using mock() or (in Mocha) expects(). In those libraries, specifying an method with mock() implicitly assumes that the method will be called exactly once.

Each library offers options to change the expected number of times a method will be called.

RR has a unique feature called a *proxy*, where the method is actually called (as opposed to the return value being set by RR), but you can still specify an expectation on how many times the method is called.

Flexmock

```
mock.should_receive(:method).and_return(value).once
```

Other options include: zero_or_more_times(), twice(), never(), and times(n). You can also use combination methods, as in at_least.once() or at_most.twice().

Mocha

Bare mock objects are just like bare stub objects, except all methods are expected to be called once. Existing objects use the expects() method to be converted to mocks.

```
mock = mock()
project.expects(:method)
```

The default is that the method is called exactly once, equal to project.expects(:method).once. Other options include: twice(), at_least_once(), at_most_once(), at_least(x), at_most(x), times(x), times(x..y), and never().

RSpec

```
obj.should_receive(:method).and_return(1)
```

The default expectation is that the method will be called once. If the method should never be called:

```
obj.should_not_receive(:method)
```

Other method count expectations can be set with once(), twice(), exactly(n).times(), at_least.once(), at_least(n).times(), at_most.once(), at_most(n).times(), and any_number_of_times().

RR

This sets an expectation for a single call. The expected value is the result of the block.

```
mock(obj).method { value }
```

To set an expectation that a method is called more than once:

```
mock(obj).method.times(n) { value }
```

To set an expectation that a method is not called:

```
do_not_call(obj).method
```

RR lets you create a *proxy* that creates an expectation that a method will be called, but unlike a mock, it actually calls the method. A block argument to the proxy call allows you to post-process the output of the actual method:

```
mock.proxy(project).method { |actual| "#{actual}_mocked" }
```

Filtering Methods by Argument

All four libraries offer similar syntax for specifying arguments that must match for the doubled method to be invoked. This allows you to specify different return values based on the arguments. For example, you could specify multiple stubs of the find method, each returning a different model object. In addition to matching based on the exact value of the arguments, each library offers some more generic matchers based on class, or matching a regular expression, or whatnot.

Flexmock

```
mock.should_receive(:method).with("a")
```

Also with_any_args() and with_no_args(). If the argument to with() is a class, any instance of the class matches. If it's a regular expression, any string matching the regular expression matches. There is also a mechanism for more complex logic.

Mocha

```
project.expects(:method).with(1)
```

If with() is passed a block, the method matches if the block returns true. Several other matchers can be combined with the argument, including instance_of(), Not(), any_of(), and regexp_matchers().

RSpec

```
project.should_receive(:method).with(1)
```

Other filters include anything(), any_instance_of(), hash_including(), boolean(), duck_type(:message), or a regular expression.

RR

```
mock(project).method(1) { value }
```

Just set the arguments to the method when defined. There are special matchers that can be placed as an argument, including anything, is_a(), numeric, boolean, and duck_type. You can also put in a range or a regular expression.

Doubling Any Instance of a Class

Three of the libraries also have special syntax that allow you to specify stub or mock behavior for any instance of the class that is created subsequent to that declaration. (Be careful, instances of the class previously created will not have the double behavior.) Typically, after the method declares that this double applies to all new instances, any other filter or expectation can be applied. In each example, the Project class is being decorated to return false when save() is called.

Flexmock

```
flexmock(Project).new_instances.should_receive(:save => false)
```

After new_instances(), any Flexmock expectation or filter can be used.

Mocha

```
Project.any_instance.expects.save.returns(false)
```

RSpec

RSpec doesn't have an exact match for this feature; the closest workaround seems to be this:

```
Project.stub!(:find).and_return(
    mock_model(Project, :save => false))
```

RR

```
mock.instance_of(Project).save(false)
```

ActiveRecord Mock Features

A couple of the libraries offer special features for ActiveRecord.

Flexmock

```
flexmock(:model, Project)
```

Works like a normal stub, with the methods id(), to_params(), new_record?(), errors(), is_a?(), instance_of?(), kind_of?(), and class() already stubbed to consistent defaults.

Mocha

N/A

RSpec

```
mock_model(Project, :method => value)
```

This requires the rspec-rails plugin, and stubs id(), to_param(), new_record?(), and errors().

RR

N/A

Method Chains

Some of the libraries offer the ability to set a stub or mock on an entire chain of method calls in one line, without having to explicitly set the intermediate mock object. This can make code clearer in the odd case where you need to mock across several objects.

Flexmock

```
flexmock(project).should_receive("project.leader.address.city")
```

The resulting mock acts like any other Flexmock object.

Mocha

N/A

RSpec

```
stub_chain(project.leader.address.city).and_return("Chicago")
```

RR

```
stub(project).leader.stub!.address.stub!.city { "Chicago" }
```

7.8 Mock Object Summary

In this section we covered model testing. First, we talked about the services Rails provides for testing models, and we discussed fixtures and factories as mechanisms for creating consistent test data. With this chapter, we've started to transition from testing models to testing the user-facing parts of the application. Mock testing is useful for testing models, but it becomes especially useful when trying to shield the various layers of your application from each other.

In the next section, we'll be discussing testing the controller and view layers. Mock objects can be a very important part of controller testing; creating mock models allows the controller tests to proceed independently of the model test.

Part III

Testing User-Facing Layers

Chapter 8

Testing Controllers with Functional Tests

Even more than model testing, Rails controller testing depends on Rails-specific structures and methods. The goal of controller testing is to simulate a single request to a specific controller method and make assertions about the result. This may or may not include assertions about the view output, depending on your taste. Standard Rails functional testing includes both controller and view tests, but there are add-ons that allow for separate testing of views and partials.

8.1 What's Available in a Controller Test?

Rails-generated controller tests are subclasses of ActionController::TestCase, itself a subclass of the ActiveSupport::TestCase used for model testing.[1]

1. Before I let go of the naming topic entirely, why is it *Action*Controller and *Action*View, but *Active*Support and *Active*Record?

What's in a Name, Part Two

I don't want to belabor the point (too late), but as much as I try to hold to standard Rails naming conventions here, "functional test" still doesn't sit right with me. I'll stick to "controller test" and "view test" in order to be specific about what the actual goal of the test is.

All the features available for model testing are still present in controller and view testing. In addition, some additional toys are added to ActionController::TestCase via a few modules that are mixed into it. This is what is at your disposal:

- Three instance variables that are the mock object versions of the @controller, the @request, and the @response. Functionality that would require a real user request or client browser is stubbed out. My experience is that I don't use those directly a whole lot, except for @response.body, which is helpful for view testing and debugging by inspecting the body text from inside a test method.
- Four pseudo-hash variables representing control structures. You've got your session, cookies, and flash, each of which represents the Rails construct of the same name, such as session[:user_id]—although the cookie variable has string keys, not symbol keys. And you've got assigns, which allows access to any instance variable set in the controller method, so that, say, @user in the controller method can be verified in the test by using assigns(:user). You're supposed to access that as a method and not as a hash—assigns[:user] won't work (but assigns["user"] will...).
- A method to simulate each HTTP verb for the purpose of pretending to call a controller: get(), post(), put(), delete(), plus the bonus xhr() for Ajax calls.
- Several assertions aimed at the specifics of controller and view testing, the most valuable of which will be covered in more detail in Chapter 9, *Testing Views*, on page 142.

8.2 What to Test

The controller part of controller tests is generally straightforward. If you are following good development practice, the complicated functionality is in the model and is being tested in your model tests; one of the reasons this is a best practice is that models are easier to test. Views, which are genuinely a pain in the neck to test, we'll cover in Chapter 9, *Testing Views*, on page 142.

A controller test should have one of the following goals:

1. Verifying that an normal, basic user request triggers the expected ActiveRecord calls and passes the necessary data to the view.

2. Verifying that an ill-formed or otherwise invalid user request is handled properly, for whatever definition of "properly" fits your app.

3. Verifying that your security roles work as expected, such as requiring logins for pages as needed, and testing that users who enter a URL for a resource they shouldn't be able to see is blocked or diverted. These tests often have a view component: admins should see a "Delete" button, but nobody else should.

8.3 Simulating a Controller Call

The general structure of a controller test includes a setup method that puts the data and session context in place, and then an individual test method that simulates a call to the controller and validates the controller response.[2] The most simplified version of this structure looks something like this:

```
setup :generic_setup

def generic_setup
  @task = Task.create
  login_as :admin
end

test "should show a task" do
  get :show, :id => @task.id.to_s
  assert_equal(@task.id, assigns(:task).id)
  assert_response :success
end
```

Rails provides a test method for each HTTP verb: get(), post(), put(), and delete(). Each of these methods works the same way. The first argument to the simulated call is the controller method to invoke. The second argument contains the key/value pairs that become the params of the call. The Rails conventions for placing complex data types into parameter names holds here, so :task => {:project => {:id => "3"}} creates params[:task][:project][:id] = "3".

In most cases, the parameters argument is the only one passed to the controller, and so is written as a list of key/value pairs rolled into a hash by Ruby. However, there are optional third and fourth arguments

2. If you are heavily into contexts, then the simulated call can go in the context. See Section 11.1, *Contexts*, on page 172 for a look at this structure.

to these test methods that set hashes for the session and flash, respectively. In the following snippet, the session is getting a user ID and current project, and the flash is getting a notice:

```
get :show, {:id => @task.id.to_s}, {:user_id => "3",
    :current_project => @project.id.to_s}, {:notice => "flash test"}
```

The xhr() method simulates an Ajax call to the controller. The signature of the method is a little different; the first argument is the HTTP verb, the second is the controller method, and the remaining arguments match the order of the other HTTP mimic methods:

```
test "my ajax call" do
  xhr :post, :create, :task => {:id => "3"}
end
```

The controller will respond to a test call made with the xhr() exactly the way it would respond to an actual Ajax request. That is to say, if you use the more modern respond_to blocks, then the request will match the format.js block. Alternately, the controller method xhr? will return true for the action being tested.

There are two gotchas in simulating controller calls. The first is that the controller method is called directly without working through the routes file. This means that the test is not verifying whether the method in question is actually reachable using the given HTTP verb—that can be done via a separate routing test that we'll cover later Section 8.6, *Testing Routes*, on page 140. This can be particularly annoying for RESTful controllers, where the HTTP verb is salient, and the test can mask a potential issue by using an unexpected HTTP verb without catching the problem.

I get snared by this trap when testing the intersection of Ajax and REST If the method called via Ajax in the previous example actually doesn't expect an HTTP POST (for example, it's a standard RESTful edit method), the test passes, but the code fails in the browser.

Another distinction between the test environment and the real request is that real request parameters are always strings, whereas these test invocations do not convert the key/value values to strings before calling the controller method. Under many circumstances this isn't a big deal—if you are just doing, say, a find() on an ID parameter, ActiveRecord does the right thing regardless. However, if you are depending on an exact match of the variable (for example, directly comparing the parameter to

Testing File Uploads

When you want to fake a file upload in a test, Rails provides the helper method fixture_file_upload() to help out. It takes two arguments: a path to an actual file relative to the Rails root, and a MIME type. You use this as the value part of a key/value pair being passed to a Rails method in a controller test, as in post :upload_icon, :icon => fixture_file_upload('/public/images/test_icon.png', 'image/png'). The controller treats that value as though it was an uploaded file.

an ID), the type may matter, and you might have a test that passes in the test environment, while the code in the browser fails.

This may become an issue in security testing. If the controller method looks like this:

```
def create
  if current_user.id == params[:id]
    # allow
  else
    # deny
  end
end
```

And the test looks like this:

```
test "I can create"
  login_as(@user)
  put :create, @user.id
  #assert that allowed branch was taken
end
```

The test will pass because params[:id] in the test is an integer, but the create() method will fail in the browser because params[:id] will be a string. The workaround is to ensure that the test values are converted to strings in the test. One recommendation is to use @user.to_param as the argument to put(); this method is preferable to the @user.id.to_s because it gives you some flexibility if you override the to_param() method later on.

8.4 Testing Controller Response

The most basic thing you usually want to verify in a controller method is that it returns the HTTP status code you expect and that the appropriate Rails template is placed in charge of returning the response. Rails provides three assertion methods to help: assert_redirected_to(), assert_response(), and assert_template().

When you're expecting a normal HTTP response and not a redirection, you can indicate your expectation in the test with a combination of assert_response() and assert_template(), like so:

```
test "successful index request" do
  get :index
  assert_response :success
  assert_template "index"
end
```

Let's break that down. assert_response() actually verifies the response code sent by Rails to the browser. Typically, the expected value from the test is :success, meaning code 200, or :redirect, which matches any of the 300-399 range that indicates a redirect of some form or other. Other special values are :missing to match a 404 error, and :error, which matches any of the 500-599 error range. Instead of a special value, the expected argument can also be the exact integer of the expected response. All the individual response codes also have their own specific, if rarely used, symbolic equivalent.[3]

The assert_template() verifies which template Rails uses to generate the response. The template name is specified, exactly as it would be in the controller using render :action—the template name can be a string or a symbol. If the argument is just a single string or symbol, then it is checked against the name of the main template that rendered the action. If you call assert_template() with a key value argument of the form assert_template :partial => '_user_data_row', then you are testing whether the specified partial is called when the controller action is rendered. The partial name in the test method must include the leading underscore. Adding the :count option verifies that the specified partial was called a specific number of times, which is potentially useful for a partial that is rendered inside a loop.

3. There are about 50 different status codes; the entire list of can be found in the Rails source in actionpack/lib/action_controller/status_codes.rb.

For a redirect, Rails provides assert_redirected_to(), which takes as an argument any object that can be resolved into a URL. (assert_redirected_to() implicitly makes a call to assert_response :redirect, so you don't need to write a separate assertion for that.)

If the argument to assert_redirected_to() is a hash, as in:

```
assert_redirected_to :controller => :task, :action => :show, :id => 3
```

Then if the actual redirect URL as specified in the controller also uses a hash, the Rails test only checks the keys that you actually use in your assertion, allowing you to check for a partial match. However, if the actual redirect URL is specified via a named route or RESTful route mechanism, Rails will insist on an exact match.

You can check against a named or RESTful route method:

```
assert_redirected_to new_task_url
```

Or an object that corresponds to a RESTful route:

```
assert_redirected_to @task
```

A Rails controller test does not—repeat, does not—follow the redirect.[4] Any data validation tests you write apply only to the method before the redirect occurs. If you need your test to follow the redirection for some reason, you are cordially invited to try something in an integration test; see Chapter 13, *Testing Workflow with Integration Tests*, on page 175.

8.5 Testing Returned Data

Rails allows you to verify the data generated by the controller method under test through the four collections mentioned earlier: assigns, session, cookies, and flash. Of these, assigns, which is a hash of instance variables created in the controller, is the most commonly used. A typical use might look like this, with a common use of assigns, and a frankly contrived use of session:

```
test "should show task" do
  get :show, :id => @task_1.id
  assert_response :success
  assert_equal @task_1.id, assigns(:task).id
  assert_equal "task/show", session[:last_page]
end
```

4. Prior to Rails 2.2, there was a method that let you follow a redirect from a test, but only within the same controller. It was deprecated in Rails 2.2.

Gotcha: Requiring a Login

Here's something that happens to me with disturbing frequency. On sites that require user logins, I often code the site up a little bit before adding the filter that requires a user login, usually to get the site architecture correct before forcing the security. I add the line before_filter :require_user for Authlogic, and boom—all the controller tests fail because they are redirected to session/new for the lack of login. It seems it always takes me longer to remember why this has happened than it should (since it should, by now, only take about 10 seconds). To sum up: remember to put the login in your tests. With Authlogic, by default that means putting UserSession.create(user) in a setup block, along with some other setup that is specified in Section 3.4, *Security Now!*, on page 50.

The cookies and flash special variables are used similarly, though I don't write tests for the flash very often.[5]

When testing the model data in a controller, keep in mind that you usually don't want to duplicate the actual model tests, and you definitely don't want the controller test to replace the model test. It's easier and more effective to test the model in the model test—there's less setup, and the test is generally clearer and more focused. See Section 4.2, *Testing the Project View: A Cascade of Tests*, on page 61 for a description of how to interleave controller and model tests with a Test-Driven Development process.

To Mock or Not to Mock

While you want to make sure and cover any error conditions specific to the controller itself, you don't need to test all the error and condition paths of the model: again, that is best managed in the model test. However, you do want to verify that the controller gets the expected object. In many cases, it's enough to know that the controller object is calling the expected model method and sets the expected variable. Since

5. One thing to note about cookies: in Rails 2.3 and up, cookies in controller tests are just strings. In prior versions, you could also specify them as CGI::Cookie objects if you wanted to test cookie attributes. It doesn't seem like you can test cookie attributes in Rails 2.3 without manually accessing the request header.

the model test covers the result of the model method, knowing that the method is called with the expected arguments is enough to verify the state of the controller.

This is a potentially great use case for mock objects. Mock object tests are covered in more detail in Chapter 7, *Using Mock Objects*, on page 105, but here's a sample of how they can be used in controller testing, with the Ruby mock object tool Mocha:

```
test "should show task" do
  task = Task.new
  Task.expects(:find).with(1).returns(task)
  get :show, :id => 1
  assert_response :success
  assert_equal task, assigns(:task)
end
```

The Task.expects line has two separate functions:

1. The stub function. Bypassing the real find() method so that if the object receives a call for find(1), it will return task without calling the actual Task.find() method. Note that this bypasses the database entirely.

2. The mock function. Setting up an expectation that find(1) will be called exactly once during this test. If that method is not called, the test will fail.

The upshot of this is that you can test the controller behavior without having to worry about the finer points of the model implementation—in fact, you can test the controller before the model method is even written. There's a downside, common to any mock testing. Setting up the mocks can be time-consuming and brittle, and if the object that you return from the mock doesn't match the real method, you may be obscuring bugs in the controller or view.

One compromise is the concept of a *spy*, which the Double Ruby mock object framework allows. Essentially, you can identify the model method as being of interest, and then set an assertion that the method has actually been called. In Double Ruby syntax, the test might look like this:

```
Line 1  test "should show task" do
2     stub.proxy(Task).find('1')
3     get :show, :id => 1
4     assert_response :success
5     assert_received(Task) { |t| t.find('1') }
6     assert_equal task, assigns(:task)
```

```
7 end
```

There are just two special lines in this snippet. In line 2, a special Double Ruby construct called a *proxy* is created. Basically, this just calls the method normally, but marks the method as one for Double Ruby to keep track of.[6] Since Double Ruby is now tracking calls to the method, you can make assertions based on method counts, in this case asserting that the Task class actually received a method call find('1'). I find this to be a nice way to split the difference between real mocks and actually testing the model method in the controller.[7]

Much, much more on the costs and benefits of mock object testing in Chapter 7, *Using Mock Objects*, on page 105. Also, see Section 7.2, *Stubs*, on page 107 for a description about how mock objects can be used to shore up test coverage of the failure states in code generated by standard Rails scaffolds.

8.6 Testing Routes

Although the basics of Rails routing is simple, the desire to customize Rails response to URLs can lead to confusion about exactly what your application is going to do when converting between an URL and a Rails action, and vice versa. Rails provides two core methods that you can use to specify or validate routing behavior.

The two Rails methods, assert_generates() and assert_recognizes(), are mirror images of each other. Let's take them in alphabetical order.

The assert_generates() method takes an expected path string as its first element and a hash of key/value pairs as the second. The test passes if the Rails router would convert the key/value pairs into the expected string—it's doing a string comparison. An example from Huddle:

```
assert_generates "/status_reports/1",
    {:controller => "status_reports", :action => "show", :id => "1"}
```

In this case, Rails runs the router against the hash and compares it to the string.

6. You can also choose to post-process the returned result, though an obvious use case doesn't leap to mind.
7. There is a fork of Mocha, available at http://github.com/jferris/mocha, which adds a basic spy implementation.

The inverse method, assert_recognizes() flips the order of the arguments and also flips the execution.

```
assert_recognizes {:controller => "status_reports",
    :action => "show", :id => "1"}, "/status_reports/1"
```

In this case, the second argument is run through the Rails router, and the parameters that it converts to are compared against the hash in the first argument.

There is a little bit more flexibility to the arguments in assert_recognizes(). The second argument can also be a hash with two keys, a :path key and an :method key, which allows you to test RESTful routing like this:

```
assert_recognizes {:controller => "status_reports",
    :action => "update", :id => "1"},
    {:path => "/status_reports/1", :method => :put}
```

For the purposes of this test, the first argument can be written as a RESTful method or named route method, though it's not immediately clear to me why this is an improvement:

```
assert_recognizes new_status_report_url, 'status_reports/new'
```

If you want to test items that would be in the query string, you must pass them as a hash in an optional third argument—just appending them to the string path argument won't work:

```
assert_recognizes {:controller => "status_reports",
    :action => "show", :id => "1", :all => true},
    "/status_reports/1", {:all => true}
```

RSpec automatically generates route tests as part of its controller scaffold, but the default tests don't. I don't usually make a habit of including these tests, but they are handy in cases where the routing instructions get complicated.

8.7 Coming Up

In this chapter, we covered how to simulate controller actions for test purposes. Next, we're going to run through other user-facing parts of the Rails stack. The next chapter will talk about view testing, and when and why you need it. After that, we'll discuss testing helpers; nearly every Rails application I've ever seen has untested helpers, and we'll show some ways to tackle them. In the following chapter, we'll close with an example of how to integrate JavaScript Ajax testing with your Rails test suite.

Chapter 9

Testing Views

View testing is fundamentally different from testing controllers and models in that it's prohibitively time-intensive to fully test the view output, and even when that's accomplished, it's still hard to validate the actual in-browser look of the view. Not to mention that overly detailed view tests are notoriously brittle in the face of consistent redesign of the look of your application. Using Rails' tools, you are best served by trying to specify and verify the logical and semantic structure of your application's output.

9.1 The Goals of View Testing

Successful view testing is an art—it's hard to find the appropriate balance between testing so little that the tests have minimal value, and testing so much that they break every time your HTML designer looks at them cross-eyed.

Create a view test when you want to do one of the following things:

- Validate that the view layer runs without error. This is of course the bare minimum you'd expect from a view test. That said, you can save yourself some production deployment embarrassment if you know that every branch of your views actually runs.
- Validate that the data gathered by the controller is expressed as expected in the view. You'll especially want to test that any view logic dependent on the presence or absence of particular data works as expected.
- Validate security-based output; for example, administrators may have links on pages allowing them access to edit mode. Or users

may be able to edit their own posts, but not a different user's. And so on.

A view test, within Rails itself, should not attempt to do any of the following:

- Validate the exact text or HTML markup of the response. This is susceptible to random breakage.
- Validate the user-facing look of the site. Even if you could figure out a reasonable way to to this automatically in Rails, again, it's too easily broken. Other tools, such as Selenium, can handle some of this testing.

9.2 Keys to Successful View Testing

The lists in the previous section demonstrate that view testing is somewhat different from model or controller testing. View testing is more impressionistic than model or controller testing. In order to keep view testing manageable, you need to assume that the HTML keeps true to the spirit of the test. That is, if you are testing for the existence of an element with a DOM ID project_description, then for your own sanity, you need to assume that the content of that DOM ID will actually be something related to the project description and not, say, the phase of the moon.

With that in mind, here are some guidelines for keeping your eye on the ball when writing view tests:

Try not to test for the existence of specific inner text of the HTML tags. If you feel the need for this kind of double-check, odds are your test is too specific. Testing for more general regular expression matches is sometimes useful, though. You may be tempted to say that testing for the inner text of an HTML tag is okay if you are testing for the exact contents of a field that you have specifically created in your test data. Maybe, but don't blame me when somebody decides to add the phrase "date of birth" into the actual text field, and you have to fix a bunch of tests. You know your code better than I do, but I've seen this movie, and it rarely ends well.

Instead, test views at the semantic level. Put DOM IDs on HTML tags all over the place and test for the existence or nonexistence of tags with the expected DOM IDs. Alternately, consider tagging with CSS classes that describe the semantic role of some output, even if the CSS class

Validating HTML

A very strong side benefit of assert_select() is that the method parses the HTML and will emit warnings if the HTML is badly formed—for example, if an end tag is missing. (Unfortunately, the warning message doesn't come with a stack trace, so if a warning suddenly pops up in the middle of your test suite, it can be hard to track down.) For this reason, assert_select() is valuable to run on your output even if you aren't doing any other validations in the call. If you want to test for valid markup more formally, you can check out the html_test plugin on GitHub.

doesn't have specific styling associated (for instance, giving all the data rows of a table a data_row CSS class). This lets you verify the structure of your output without being caught up in the content. The extra DOM IDs also help in Ajax, CSS, and browser debugging later on.

The only exceptions to testing for DOM IDs are form elements; you can test those using their name attribute (which allows you to directly verify the keys that will be in the ensuing POST call). You can even test the value attribute, at least sometimes.

In addition to testing for what is in the response, testing for what is not in the response is very useful, especially when dealing with users who have different roles, and therefore different output.

9.3 Using assert_select

The assert_select() method is a powerful and flexible method of validating HTML content. The basic idea is to use the syntax of CSS selectors to specify HTML structures to be found in the output. The assert_select() method automatically looks at @response.body to find the output to test against. Once you've found matching structures, you can further test for the content of the structures or the number of matches found.

The simplest use of assert_select() is merely to test for the existence of a specific HTML tag:

```
assert_select("form")
```

This assertion is true if there is at least one HTML form element in the response body. (By the way, the tag selector must be a string, not a symbol.)

There are three ways to make an assert_select() test more specific. You can augment the selector using various CSS-style syntax decorations, you can specify the content of the tag, and you can specify the number of matching tags that should exist, based on the selector and content. Or you can do all three.

The least complicated of these three options is to specify the content of the tag. If the next argument to assert_select() is a string or regular expression, then the assertion is only true if there is at least one matching HTML tag with inner content that matches the argument. String arguments must match the content exactly. Regular expression arguments must =~ match the contents. The following two examples both match against Happy Birthday:

```
assert_select("span", "Happy Birthday")
assert_select("span", /Birthday/)
```

If the second argument after the selector is a number or a range, the test validates that the number of matching elements in the response is either the exact number or in the range. In practice, I use this to validate that, say, an table in an index view has one row for each element being displayed:

```
assert_select("tr", 5)
assert_select("tr", 2 .. 3)
```

You can also write the tests with true and false as the second argument. True indicates the default "one or more" behavior, and false indicates that there are no matching elements: assert_select(:form, false). A common scenario is a view in which different text elements display for different users. Testing for just the existence of options leaves a hole in the tests: a view that displays all options at all times will pass the tests. Testing for the nonexistence of text that shouldn't exist exposes that particular bug.

If you want to be bold and specify both text and a count, the final argument is a hash or some key/value pairs. The keys :text and :count match the features you've already seen:

```
assert_select("span", :text => "User Name", :count => 5)
```

Personally, I find the keys more readable, and I tend to use them even when specifying only one option. You can also specify range behavior with the keys :minimum and :maximum.

Which brings us to modifying the selector. There are many different selector modifiers, most of which don't get used very often. The modifiers boil down to five types, described in the following sections.

Specify the DOM ID of the Element

You can specify an element by DOM ID using an #, as in "div#user_name_3". You can leave off the HTML tag if you want; "#user_name_3" matches any DOM element with that ID, regardless of tag type. In any assert_select() selector, a ? behaves as it does in an ActiveRecord SQL condition string, as a placeholder for a value specified later. So, the common use case of specifying an element using the Rails dom_id() method can be written:[1]

```
assert_select("div#?", dom_id(@user, :name))
```

This is often easier to read than duplicating the hash symbol to interpolate the string. As mentioned previously, I recommend testing for IDs frequently.

Specify the CSS Class of the Element

The CSS class of the element being searched for can be specified using ., as in "div.headline". Again, you can leave off the HTML tag. Aside from looking at class rather than ID, the behavior is exactly as described above.

Specify the Value of an Arbitrary HTML Attribute

An assert_select() method can be made to look for any arbitrary HTML attribute. This is most useful when searching for the name attribute of input tags. Other, noncontrived examples include the action attribute of a form tag, the src attribute of an img tag, and the href attribute of an a tag. The basic syntax looks like this:

```
assert_select("input[name=user_id]")
```

Note than when testing forms that use Rails naming conventions, you probably want to use the ? syntax for the case where the name actually contains a bracket; otherwise, you can put the entire name in quotes:

1. Remember that you need to include ActionView::Helpers::RecordIdentificationHelper in order to use dom_id() in your test class.

```
assert_select("input[name=?]", "user[email]")
assert_select("input[name='user[email]']")
```

The = operator indicates an exact match with the value of the attribute. There are a couple of modifiers to the = for different match behavior. The one I use most often is *=, which is true if the attribute value contains the specified text, as in assert_select("input[name *= email]"). You can also use ^= and $= for starts with and ends with, respectively.

Use a Pseudo-Class

There are a number of pseudo-class modifiers that can be added to a selector, including:

```
assert_select("div:first-child")
assert_select("div:last-child")
assert_select("div:only-child")
assert_select("div:nth-child(3)")
assert_select("div:nth-last-child(3)")
```

To clear up any confusion: you read these as "a div element that is a first child of its parent," not as "the first child of a div element." I find these helpful when I am testing for output that is in a specific order, such as a sorted list. It's easier to test for "li:nth-child(2)" than to try to extract all the list element text via regular expression and test for the elements of that resulting list. Also, notice those are dashes and not underscores in those pseudo-class names. Seems like I'm always typing them as underscores and then being surprised when the pseudo-class is not recognized.

Each of these has a more specific modifier, replacing child with of-type, such as first-of-type, which means the matching tag must be the first of its siblings of the given HTML type. Read div:first-of-type as "a div element that is the first div element child of its parent." (Do I need to mention that you can use any number as the argument to the nth modifiers?)

Other modifiers that aren't in the first/last/only genre include:

```
assert_select("div:root")
assert_select("div:empty")
assert_select("div:not(.headline)")
```

The first modifier in the list checks for an element of the given type that also happens to be the root element. The second looks for an empty tag, while the not modifier looks for an element of that type that does not match the given selector—in this case, a div element that does not have a CSS class of headline. I tend to feel the first two of these are too

fiddly for useful tests, but I'm sure there's a valuable use case out there somewhere. The not construct can be helpful for negative testing.

Combine Multiple Selectors

You can use assert_select() to test for different combinations of tags. Again, the syntax is very similar to standard CSS.

```
assert_select("div.headline span")
```

That snippet finds all span elements that are inside a div with the class `headline`. Putting a > operator between the two elements means that the second element must be a direct child of the first, not an arbitrary descendant. The + operator means the second element must come immediately after the first in the document, while the ~ operator means that the second element comes somewhere after the first.

You can do further testing on the elements that match your selector. This is most valuable when you want to perform a second match that is limited to only a particular part of your HTML body; for example, if you want to test for the existence of a particular text input field, but only within a specific form on the page. The most common way to do this kind of search is by nesting assert_select() calls. Nested assert_select() calls have an explicit form that shows the general idea:

```
assert_select("div.headline") do |matching_elements|
  matching_elements.each do |element|
    assert_select(element, "span")
  end
end
```

To begin with, this example exposes a feature of assert_select() that hasn't been mentioned yet. If the first argument to the assert_select() call is a Rails HTML::Node class, then the search is limited to content within that HTML Node. If no HTML Node is specified, then as you have seen, the default is the response body for the response being tested.

If assert_select() is passed a block element, then the argument to the block is an array containing all the HTML nodes that match the selector—in this case, any div element with the DOM class headline. Inside the block, the matching elements can be iterated on. In this case, they are iterated on for the purpose of verifying that all headline elements contain a span subelement.

This pattern is so common that it has a shortcut, which is used by passing a block that takes no arguments:

```
assert_select("div.headline") do
  assert_select("span")
end
```

In this case, any assert_select() method called within the block is automatically applied to all the matching elements and must pass all those elements in order for the entire assertion to pass—meaning that this snippet and the previous one are exactly equivalent, and each verifies that every div.headline element contains at least one span element.

The form where the block takes an argument is more flexible, although to be honest, I've used assert_select() for years without needing that flexibility—frankly, without even knowing that flexibility existed. If you need even more flexibility, assert_select() also returns the same array of matches, so the same test could be written like this:

```
matching_elements = assert_select("div.headline")
matching_elements.each do |element|
  assert_select(element, "span")
end
```

That's arguably an improvement over the first form, in that it's shorter. This structure is most helpful in debugging an assert_select() test that is failing.

Using assert_select() well is the difference between view tests that actually help you and view tests that make you tear your hair out, so point it at those DOM IDs and take advantage of the shortcuts, and you're off to a great start.

9.4 Testing Outgoing Email

The process of testing outgoing email is closely related to view testing. Typically, you are interested in validating two separate pieces of logic: first, that your application sends an email when expected; and second, that the email content is what you want. The somewhat indirect nature of the Rails ActionMailer makes testing email somewhat less obvious than it might be, but it's not that hard. This section assumes that you are using ActionMailer—if you're using a non-core tool for managing email, you may need to find a different way to test.[2]

In order to test emails in Rails, configure your test setup so that ActionMailer saves emails in a query-able data structure rather than actu-

2. This section may be unusually affected by Rails 3 changes. Updates to come.

ally mailing them. This step should be done for you in the Rails config/environments/test.rb file:

```
config.action_mailer.delivery_method = :test
```

Remember to put this next line in your test setup; doing so ensures that the data structure holding the mailings is emptied. Otherwise, emails from other tests will linger and make your test results invalid:

```
ActionMailer::Base.deliveries.clear
```

The quickest test you can create when dealing with email is one to find out whether an email has been sent. The method ActionMailer::Base.deliveries.size() returns the number of emails that have been sent since the last time the ActionMailer::Base.deliveries object was cleared. So a simple way to determine if an email has been sent is just to query it:

```
assert_equal 1, ActionMailer::Base.deliveries.size
```

Rails provides a shortcut, along the lines of assert_difference. This method takes a block and determines how many emails are sent in the course of executing the block:

```
assert_emails 1 do
  get :forgot_password
end
```

The argument to the assert_emails() method is the number of emails that need to be sent in the block for the assertion to pass. The assert_emails() method does not depend on the clear() method having been called; it does its own tracking of count before and after the block. There's also an assert_no_emails(), which is equivalent to assert_emails(0).

If you want to specify the content of the email messages and not just the count, Rails provides the assert_select_email() method, which you might use like this:

```
assert_select_email do
  assert_select "div", :text => "Email Reset"
end
```

What assert_select_email() allows you to do is make assert_select() assertions that will be applied to the body of every email currently in the ActionMailer::Base.deliveries repository. All emails must pass the assertions in the block for the whole thing to pass. This method only works if your outgoing emails are of content type text/html, which is a significant limitation.

You can test the features of the sent emails directly using code like this:

```
email = ActionMailer::Base.deliveries.first
assert_equal "Forgot Password Notice", email.subject
assert_equal @user.email, email.to
assert_match /new password/, email.body
```

Rails uses the Ruby TMail library to manage mail objects. The basic accessors have expected names. You can find out more about how to inspect a mail object at http://tmail.rubyforge.org.

Outside of core Rails, Shoulda provides a couple of helper methods for testing email; these are described in Section 11.2, *Shoulda Assertions*, on page 172. The most robust way of testing emails, though, is linked to RSpec (see Chapter 12, *RSpec*, on page 173) and also works great with Cucumber (see Chapter 15, *Acceptance Testing with Cucumber*, on page 194). The email-spec library provides a number of very useful helpers. For the most part, they are nice, RSpec-ways of performing the tests we've already examined, but the library also provides the ability to follow a link in an email back to the site, which is very helpful for acceptance testing of user interactions that include email. The library's home is http://github.com/bmabey/email-spec.

9.5 Testing Helpers

Helper methods are the storage attic of most Rails applications. Typically, helper modules contain reusable bits of view logic, such as the logic to control what output is printed, or change the display based on model data. In practice, helper modules tend to get filled with all kinds of clutter that doesn't seem to belong anywhere else. Worse, since the Rails mechanism for testing helpers was underpowered and underpublicized, helper methods often aren't tested even when they contain significant amounts of logic.

Unfortunately, helpers occupy a somewhat ambiguous place in the Rails MVC pattern structure. Helpers are sort of in the view layer, but they are frequently used as a conduit between views and models, or even views and controllers. Helper tests have a lot in common with model tests—for instance, individual helper methods can be unit-tested one at a time. However, in order for helper methods to be properly run in a test context, at least *some* of the test controller framework used for Rails functional tests needs to be loaded, and getting the setup correct takes a couple of extra steps.

There is a core Rails mechanism for testing helpers, added in Rails 2.1 or so; it's one of the best kept secrets in the entire core stack. (Or was... as of Rails 2.3, helper tests are generated whenever you generate controllers.) You can find it in ActionView::TestCase. This class creates a fake controller environment so that helpers can be loaded, called, and tested.

In order for helper tests to work if you're using anything prior to Rails 2.3, you must explicitly require 'action_view/test_case' either in your test/test_helper.rb file or in the helper test itself. Your actual helper test class is a subclass of ActionView::TestCase. The Rails generated test shell looks like this:

```
require 'test_helper'

class UsersHelperTest < ActionView::TestCase
end
```

The naming convention is just like everything else in Rails: add Test to the end of the name of the helper module to get the class name, and _test to get the filename. So your UsersHelper module is tested in users_helper_test.rb and is declared as class UsersHelperTest < ActionView::TestCase. As of Rails 2.3, Rails places helper tests in the test/unit/helpers directory.

At this point, you are basically good to go, and can test helper methods exactly as though they were regular model methods (with an exception or two described below). For example, if you have the following helper module:

```
module UsersHelper
  def display_name(user)
    "#{user.first_name} #{user.last_name}"
  end
end
```

The test is straightforward:

```
class UsersHelperTest < ActionView::TestCase

  test "a users display name" do
    @user = User.new(:first_name => "Ron", :last_name => "Lithgow")
    assert_equal("Ron Lithgow", display_name(@user))
  end

end
```

Inside your test class, you should have access to all the helpers in your project as well as all the regular Rails helpers, which are automatically loaded by ActionView::TestCase.

9.6 Testing Block Helpers

In Rails 2.2, the private _erbout variable in templates that stored the output stream was changed to a public instance variable. This change caused very little stir outside of _erbout's immediate family, but it did have two very nice effects. It made it possible to inject text from a helper directly into the ERb output without fussing around with block variables, and it also made it easier to test block helpers.

A *block helper* is a helper function that, when invoked in the ERb file, takes a block made up of ERb text. Two common uses of block helpers are as access control, in which the logic in the helper determines whether the code in the block is invoked, and as wrapper code for HTML that might surround many different kinds of text—a rounded rectangle effect, for example.

Here's a small example of a block helper:

```
def if_logged_in
  yield if logged_in?
end
```

Which would be invoked like so:

```
<% if_logged_in do %>
  <%= link_to "logout", logout_path %>
<% end %>
```

And which you would test like this (in Rails 2.2 and higher):

```
test "logged_in" do
  assert !logged_in?
  assert_nil(if_logged_in {"logged in"})
  login_as users(:quentin)
  assert logged_in?
  assert_equal("logged in", if_logged_in {"logged in"})
end
```

What we're taking advantage of here is that the last value in the block becomes the return value of the block, which becomes the last value in the method, which becomes the return value of the method. So if nobody is logged in, the block doesn't fire, and the method returns nil. If a user is logged in, then whatever gets passed in the block—in this case, the literal string logged in—is the returned value of the helper.

This also works for helpers that concat values directly into the ERb stream. The somewhat contrived helper:

```
def make_headline
  concat("<h1 class='headline'>#{yield}</h1>")
```

```
end
```

And the test:

```
test "make headline" do
  assert_dom_equal("<h1 class='headline'>fred</h1>",
      make_headline { "fred" })
end
```

Again, we're taking advantage of concat() returning the final string as well as placing it into the ERb output. Be careful when using this kind of testing: you aren't actually testing the ERb output—just the return value of the helper method.

If that's not good enough, you can actually test against the ERb output by taking advantage of the fact that concat adds its text to an output buffer that you can also access in your test:

```
test "make headline with output buffer" do
  make_headline { "fred" }
  assert_dom_equal("<h1 class='headline'>fred</h1>", output_buffer)
end
```

In this test, the helper call places the text in the output_buffer, which is then validated in the final line of the test. The method assert_dom_equal() tests whether two strings of HTML are equivalent even if their attribute lists are differently formed.

9.7 Using assert_select in Helper Tests

The output buffer trick is kind of neat, but it would be even nicer if you could bring the full flexibility of assert_select() to bear on the output. Under normal circumstances, assert_select() looks to @response.body to get the text to parse. There are two ways to work around this. You can fake a @response.body object, or you can point assert_select() at an arbitrary string.

The arbitrary string mechanism is perhaps simpler—include the following method in your test/test_helper.rb.

```
def assert_select_string(string, *selectors, &block)
  doc_root = HTML::Document.new(string).root
  assert_select(doc_root, *selectors, &block)
end
```

This method creates an HTML selector object of the kind that Rails accepts as the selector first argument of assert_select(). Using this method would look like this:

```
test "make headline with response body" do
  assert_select_string(display_name(user) "div.first_name")
end
```

In other cases, you might want to have a more complete controller setup, because you are testing session logic. You might also want to call assert_select() directly. If so, you can add the controller setup easily. Place this module somewhere, then require it in place of action_view/test_case:

Download huddle/test/helper_test_case.rb

```
require 'action_view/test_case'

module ActionView
  class TestCase

    setup :setup_response
    def setup_response
      @output_buffer = ""
      @request    = ActionController::TestRequest.new
      @response   = ActionController::TestResponse.new
      @session = {}
      @request.session = @session
    end

    def session
      @request.session
    end

    def make_response(text)
      @response.body = text
    end
  end
end
```

What this does is add a TestResponse object; it also gives you a session object, should you need that for testing. The new class now has the make_response() method, which takes arbitrary text and slaps it right into @response.body for you. All you need to do is call make_response() with whatever text you want to assert:

```
test "make headline with response body" do
  make_headline { "fred" }
  make_response output_buffer
  assert_select("h1.headline")
end
```

I'm not prepared to defend this as the most elegant way to get assert_select() into your helper tests, but it does work. I think this mechanism probably results in more readable tests for block helpers or for nested assert_select()

Gotcha: url_for

Although all core helpers are automatically loaded into the ActionView test environment, there are one or two that have significant dependencies on the real controller object and therefore fail with opaque error messages during helper testing. The most notable of these is url_for(). The recommended workaround is to override url_for() by defining it in your own test case (the method signature is def url_for(options = {})). The return value is up to you; I find a simple stub response is often good enough.

calls. Note that the ActionView::TestCase() creates its own brand-new TestResponse object every time you call a helper method—so if for some odd reason you have a test that depends on the exact @response object in a helper, you could run into problems with the response object not holding values from helper call to helper call.

9.8 How Much Time Should You Spend on Helpers?

The amount of time you spend testing helpers really depends on the helper, or, more generally, on how you use helpers. If you have a lot of view logic in helpers, and they are shading towards being presenters of one kind or another, it's a good idea to validate any complex logic. Simple HTML methods, like the make_headline() example here, don't require a lot of testing (certainly not if the testing is bogging you down). However, if you do find a bug in a helper, pull the method into the garage and cover it with tests.

9.9 When to View Test

It's important to test view logic in a considered, careful way. In all TDD, any change to the logic of the program should be driven by a test. Although you shouldn't have the bulk of your logic in the view layer, you'll need to have some logic surrounding display, and that should be tested.

However, complex view logic should be moved to helpers,[3] which are easer to test; see Section 9.5, *Testing Helpers*, on page 151. A strict TDD process often breaks down a little in the view. A view program sometimes feels more exploratory to me, so I usually do a little bit of view code, then cycle back to write a test. As long as the feedback between test and code is relatively tight, that process won't hurt you. View testing can also be managed in an integration test layer using Cucumber (see Chapter 15, *Acceptance Testing with Cucumber*, on page 194) or Rails integration tests; see Chapter 13, *Testing Workflow with Integration Tests*, on page 175. Find the work flow that works best for you.

3. There are a number of third-party tools that enlarge the helper concept to a full-fledged object called a *Presenter*, which effectively mediates between a view and one or more models.

Chapter 10

Testing JavaScript and Ajax

I've had this conversation more than once:

ME: *What test tools do you use?*

SOMEBODY ELSE: *Oh, you know, Shoulda, Factory Girl.*

ME: *Do you test your views?*

SOMEBODY ELSE: *Sometimes. When necessary.*

ME: *How do you test your Ajax?*

SOMEBODY ELSE: *[With sad resignation] We don't.*

To be fair, sometimes the roles are reversed, and it's me observing that we don't test our JavaScript.

And really, can you blame me? Testing JavaScript and Ajax is a pain. For one thing, JavaScript is frequently, well, written in JavaScript. It's a nice little language and all, but it doesn't have anywhere near the level of test framework support and commitment as Ruby.[1] Not only that, but Ajax actions often only happen in the browser. The actions are difficult to extract for a user test if they only exist in the view layer. Then there's the whole browser dependence thing. It's hard to blame somebody for chucking the whole testing thing and just writing the code already.

1. For some reason, JavaScript tool support has always lagged a few years behind similar languages, probably due to the mistaken impression that JavaScript is just some kind of toy language. This has started to change over the last few years as frameworks like Prototype and jQuery have enabled really powerful JavaScript tools.

And yet: as more application logic moves to the browser to create cool and useful user experiences, leaving that code untested becomes a bigger and bigger risk. In this chapter, we'll walk through some strategies and tools for testing JavaScript and Ajax from within a Rails application. Later, the chapter Chapter 16, *Browser Testing with Selenium*, on page 215 will discuss automated testing of Ajax from within a browser.

This is not meant to be a complete guide to all possible JavaScript tools; we simply don't have the space for that. The tool structure that is outlined in this chapter is one method that I've had some success with and which I think you can be successful with as well. The advantage of the tool chain described in this chapter is that it allows you to test your Ajax both in the browser and as part of your regular command-line Ruby testing.

10.1 First Off, RJS

Even in the notoriously fickle-about-new-tools Rails community, the rise and fall of Ruby JavaScript (RJS) from "cool new tool for writing Ajax" to "tool whose usage is a slightly embarrassing admission that you don't like writing JavaScript" seemed to happen quickly. RJS came along at a time when the JavaScript toolkits were so clunky that it made sense to make what was potentially an extra server call, and write simple client-side commands in Ruby so as to avoid writing scary JavaScript. Since then, the JavaScript tools have made legitimate strides, the idea of unobtrusive JavaScript has spread, and the Ruby/Rails community has gotten more comfortable with the need to write more complex client-side interactions in JavaScript, all of which have combined to make RJS seem unneeded at best.

That said, RJS is still nice for quick Ajax features that are inside the limited RJS feature set and are simple enough not to need the weight of an entire JavaScript framework, such as a simple dynamic update of a specific point on the page based on current server data. Plus, since RJS executes essentially as a Rails controller action, it's much easier to test than other JavaScript that exists exclusively in the browser.

Your weapon of choice in testing RJS is the assert_select_rjs() method. This method is less like the normal assert_select() than you might imagine. The meaning of an assert_select_rjs() call depends slightly on the number of arguments the method is called with. In what I think of as the standard way of calling the method, it takes two arguments, the

first of which is a symbol matching an existing RJS Rails method, and the second of which is a DOM ID. The DOM ID can be static, or can be calculated, as in this example:

```
assert_select_rjs :replace, dom_id(@project)
```

This assertion passes if the controller call in question returns an RJS JavaScript snippet, and that snippet contains a call to the RJS replace() method with a DOM ID matching dom_id(@project). This does not make any claim—yet—about the text that is being injected into that DOM element. If the first argument is the RJS method :insert, then the second method can be the position of the insertion (:top, :bottom, :before, or :after), in which case the assertion only passes if the method and the position are called from the RJS. If there is no position argument, any insertion to the given DOM ID will match.

The assertion in assert_select_rjs() can be made weaker. If there's no method symbol, any RJS call to the DOM ID will pass the assertion. If there are no arguments to assert_select_rjs(), the assertion will pass if any RJS call is made in the output—not the most helpful assertion in terms of actually saying something useful.

Clearly, though, the RJS assertion is more valuable, or at least more detailed, if you can specify details about the HTML being sent back to the client page. In order to manage that, you need to pass a block to assert_select_rjs(). As a block container, assert_select_rjs() behaves just like plain old assert_select(), meaning that you can place assert_select() calls inside the block and they will be evaluated against the HTML extracted from the RJS call sent to the client.

```
assert_select_rjs :replace, dom_id(@project) do
  assert_select "div#project_name", :text => @project.name
end
```

In addition to testing for the existence of the RJS call, the above snippet also tests that the HTML sent to replace the existing text contains a div element with @project.name as its content. This works for any RJS call that is associated with sending text back to the browser—so, for example, remove() doesn't need a block call.

If you happen to be using RJS, then the RJS should be tested as though it was a view, and since RJS pretty much always indicates a change in application logic, then that means RJS calls nearly always should have tests. Exactly how detailed the tests need to be is up to you—normally I do test for the specifics of the text coming back to the browser, but then

most of the time I only use RJS if the call is on the simple side. When the call gets more complicated, then we move to actual JavaScript, which needs its own test framework.

10.2 Testing JavaScript from Rails with BlueRidge

Once you outgrow the RJS training wheels and start riding the big-kid bicycle that is actual JavaScript, the testing awkwardness becomes apparent. Many JavaScript test frameworks run in a browser, which is certainly nice for assuring fidelity in that browser but is slow and unwieldy for TDD-style tight-feedback testing. If only you could work in both modes... Well, you can, using a Rails plugin called Blue Ridge, from Relevance, which is a team name that's attached to a lot of great testing tools.[2]

Blue Ridge is not the only way to test JavaScript; it's certainly not the only JavaScript tool available; and it's probably not the only way to integrate Rails and JavaScript. I'm going to present one tool that I'm reasonably comfortable with, rather than survey the entire space.

Blue Ridge makes a few assumptions. First off, it assumes you are using jQuery as your JavaScript framework of choice.[3] You are also going to be much happier if you write using the *unobtrusive JavaScript* style, which I'll define as the separation of JavaScript from the HTML markup. When writing JavaScript unobtrusively, no JavaScript behavior is specified in the body of the HTML page. Instead, event handlers are bound to DOM IDs in functions written elsewhere and injected to the page on load.

There a number of advantages to an unobtrusive style—generally speaking, it opens browser client JavaScript up to the full range of modern software techniques. For our purposes here, the biggest advantage is that JavaScript separated from the HTML view layer is much easier to isolate and test.

Which brings us back to Blue Ridge, which isn't so much a tool in its own right as it is a clever bit of glue tying together a few JavaScript test-

2. The original RailsConf talk on Blue Ridge was entitled "JavaScript Testing in Rails: Fast, Headless, In-Browser. Pick Any Three." I mention this because I'm a sucker for that joke structure.
3. In theory, you can use Prototype—the Blue Ridge docs show a sample snippet—but I personally gave up before I could get it to work for me.

ing frameworks with the ability to run tests from a Rails command line. Blue Ridge starts with Screw.Unit, which is a JavaScript testing framework with RSpec-like syntax, and adds in Smoke, a mock framework that may be familiar to fans of Mocha. The env.js library is a JavaScript DOM implementation and the final major piece is Rhino, the JavaScript interpreter that runs on top of the Java Virtual Machine. The combination of tools gives you access to the same kinds of test tools you would expect in a Rails environment as well as the means to run the tests both in and outside of a browser.

10.3 Getting Started with Blue Ridge

Blue Ridge is distributed as a Rails plugin, and after you install you need to generate a few files.

```
% ./script/plugin install git://github.com/relevance/blue-ridge.git
% ./script/generate blue_ridge
  create  test/javascript
  create  test/javascript/application_spec.js
  create  test/javascript/spec_helper.js
  create  test/javascript/fixtures
  create  test/javascript/fixtures/application.html
  create  test/javascript/fixtures/screw.css
```

What that gives you is a test/javascript directory where your Screw.Unit tests go, plus a test/javascript/fixtures directory[4] where you can put HTML setups that act as test fixtures, in the sense that they create a DOM structure that you can use to build tests on. You also get one skeleton test, application_spec.js, which is automatically associated with the fixture file test/javascript/fixtures/application.html. The spec_helper.js file is exactly analogous to the standard test_helper.rb or RSpec's spec_helper.rb, and contains methods or other dependencies to be used across tests. The screw.css file can be used to place common styles that might be referenced in tests.

The assumption is that you'll have a test file and fixture pair matching every JavaScript file in your application; obviously, this one is targeted at application.js. The contents of the example files will give you a good feel for how Blue Ridge pulls all its contents together. Here's the application_spec.js file.

4. If you are using RSpec, the directories are spec/javascript and spec/javascript/fixtures. The rake task is spec:javascripts.

Download huddle_mocha/test/javascript/application_spec.js

```
Line 1  require("spec_helper.js");
-       require("../../public/javascripts/application.js");
-
-       Screw.Unit(function(){
5         describe("Your application javascript", function(){
-           it("does something", function(){
-             expect("hello").to(equal, "hello");
-           });
-
10          it("accesses the DOM from fixtures/application.html", function(){
-             expect($('.select_me').length).to(equal, 2);
-           });
-         });
-       });
```

Let's break that down. Lines 1 and 2 load in the spec helper file and the actual JavaScript file under test, respectively. The latter line is particularly important, as it's what ties the Screw.Unit file to the items it's testing.

If you are familiar with RSpec, you'll notice that the syntax is similar. (If you aren't yet familiar with RSpec, we'll cover it in Chapter 12, *RSpec*, on page 173). Where RSpec or Test::Unit use Ruby blocks, Screw.Unit uses JavaScript anonymous functions to package a chunk of executable code for later use. The basic functionality is the same, although the JavaScript is a bit more verbose.

The describe method in lines 5 to 13 sets up a context where multiple tests can share a common setup. For more on contexts, see Section 11.1, *Contexts*, on page 172. The describe() method takes a JavaScript anonymous function as an argument. Inside that anonymous function, you define the actual tests using the it() method, as shown in lines 6 and 10. The it() method takes an anonymous function in which you define the actual expectations. You can also define before() and after() methods that handle setup and teardown by—guess what—taking anonymous functions as arguments, with the function executed at the appropriate time in the test cycle.

Within each test, you set up expectations to be validated via the combination of expect() and to(). The expect() method sets up the actual value in a comparison, and the to() method sets up the other part. So, in line 11, the test is setting the expectation that the number of items of class .select_me will be 2. In addition to equal as the first argument of the to() function, Screw.Unit also provides be_empty, be_false,

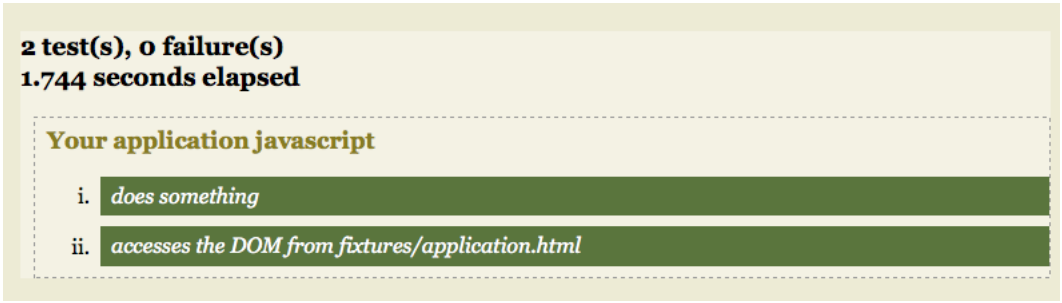


Figure 10.1: Blue Ridge HTML Results

be_null, be_true, be_undefined, contain_selector, have_length, match, and match_selector. Most of these are self-explanatory. The contain_selector, and match_selector test the value to see whether it completely matches or just contains a jQuery selector, while match compares against a regular expression. You can also use to_not() in place of to().

10.4 Running Blue Ridge Tests

Blue Ridge has not just one, but two helpful ways to run your JavaScript tests. The base Screw.Unit functionality is to run the tests in the browser. You do this by simply opening the HTML fixture page in your browser.[5] You can see the result in Figure 10.1.

The HTML fixture page that generated this test—again, we're looking at the Blue Ridge generated boilerplate—looks like this:

Download huddle_mocha/test/javascript/fixtures/application.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
  "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">

<head>
  <title>Application | JavaScript Testing Results</title>
  <link rel="stylesheet" href="screw.css" type="text/css" charset="utf-8" />
  <script type="text/javascript"
    src="../../../vendor/plugins/blue-ridge/lib/blue-ridge.js"></script>
```

5. Unless your browser happens to be Safari or Chrome. As I write this, some kind of JavaScript timing issue prevents the HTML fixture page from displaying results in WebKit.

```
</head>

<body>
  <!-- Put any HTML fixture elements here. -->
  <div class="select_me"/>
  <span class="select_me"/>
  <div class="dont_select_me"/>
</body>
</html>
```

The only really important line here is the script tag that loads blue-ridge.js. That file handles all the requirements, including locating and loading the application_spec.js file where the tests are defined, and running the actual Screw.Unit loop. The Screw.Unit run loop finds all the defined tests and executes them using the DOM structure defined in the HTML fixture as the backdrop. This particular HTML fixture file doesn't have any elements that actually display text. If there were display elements, they would show up above the Screw.Unit test results.

The test display has one nice trick up its sleeve. Clicking on any test causes that test to be re-executed (including re-loading the JavaScript test file and the JavaScript application file under test). Clicking on the label denoting a describe() block—in this case "Your application javascript"—reruns all the tests in that describe() block. The runner will also handle nested describe() calls by indenting them in display page.

That's all well and good, and pretty helpful in confirming cross-browser behavior, but it's a little awkward for TDD purposes. Which brings us to Blue Ridge's real value added: the ability to run the JavaScript tests from the command line.

```
% rake test:javascript
(in /Book/code/huddle_mocha)
Running application_spec.js with fixture 'fixtures/application.html'...
2 test(s), 0 failure(s)
0.26 seconds elapsed
```

The rake:javascript task does everything that the in-browser test does, but it does it using the Rhino interpreter and env.js DOM implementation to simulate the browser environment. The same HTML fixture file is used to create the DOM background for the tests, and the same Screw.Unit test runner is called. The results are then converted back to the familiar terminal format. You can use this rake task as is, add it to the default rake task, or add it to your continuous integration tool, all of which serve to make the JavaScript tests a fully integrated part of your test suite.

Figure 10.2: GitHub Search Box

10.5 A Blue Ridge Example

Let's take a short trip to the Blue Ridge Mountains—see what I did there? it's almost like a joke—and try out a simple example that goes a little beyond boilerplate. In this example, we'll be showing a search box that has some default text that explains what goes in the box. When the user puts focus on the box, the default text goes away. This is by no means unusual behavior; as I write this http://www.github.com has an example of this interaction (see Figure 10.2). Although relatively simple, this is technically a real-world example, in that I actually wrote it in the course of starting a genuine client project. True JavaScript masters are advised to keep their snickering to themselves.

We can start by telling Blue Ridge to generate the test files for our new set of tests:

```
% script/generate javascript_spec search_box
      exists  test/javascript
      exists  test/javascript/fixtures
      create  test/javascript/search_box_spec.js
      create  test/javascript/fixtures/search_box.html
```

Somewhat unsurprisingly, we got a new Blue Ridge JavaScript file and the associated HTML fixture file. In order to get our tests to work, we'll need an HTML fixture file that contains a search box for us to work with. In this file, the input is pre-set to the default text for the search box.

Download huddle_mocha/test/javascript/fixtures/search_box.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
  "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">

<head>
  <title>SearchBox | JavaScript Testing Results</title>
  <link rel="stylesheet" href="screw.css" type="text/css" charset="utf-8" />
  <script type="text/javascript"
  src="../../../vendor/plugins/blue-ridge/lib/blue-ridge.js"></script>
</head>
```

```
<body>
  <input class="search_default"
    id="unified_search"
    name="unified_search"
    type="text"
    value="Type Name or Company" />
</body>
</html>
```

The key here is to keep the fixtures as simple as possible so as to make it abundantly clear what the test is working on. Notice that if you want to use Blue Ridge for true end-to-end testing of your JavaScript, you are out of luck, at least at the moment. Ideally, for a true end-to-end test, the input to the Blue Ridge test would be actual content from your Rails application. Since Blue Ridge takes the DOM for the test from the fixture file, it's incumbent on you to ensure that the relevant parts of that DOM are replicated in your application.

I'll spare you all my floundering trying to get these tests written, although we'll discuss at least one specific flounder point in just a moment. I wrote three Screw.Unit tests to cover this feature.

Download huddle_mocha/test/javascript/search_box_spec.js

```
require("spec_helper.js");
require("../../public/javascripts/search_box.js");

Screw.Unit(function(){
  describe("With my search box and default", function() {
    it("should switch the default", function() {
      search_focus($('#unified_search'));
      expect($('#unified_search').attr('value')).to(equal, '');
    });

    it("should switch a non-default", function() {
      $('#unified_search').addClass('search_entry');
      search_blur($('#unified_search'));
      expect($('#unified_search').attr('value')).to(equal, default_text);
    });

    it("should not switch if there is a value", function() {
      $('#unified_search').attr('value', 'Fred');
      search_blur($('#unified_search'));
      expect($('#unified_search').attr('value')).to(equal, 'Fred');
    });

  });
});
```

These tests assume two JavaScript functions, search_focus() and search_blur(). The first test is for the case where the user takes focus of the search box while the default text is displayed, and validates that the text switches to blank. The second test is the inverse: the user moves focus away from the search box without a value, and the test validates that the default text is restored. The third test validates that if the user has entered text when the search box loses focus, the user-entered text stays in the box.

As I look at these tests, it occurs to me that there are things missing. We never wrote a test for the focus condition if there is existing text, and we never added a test for the fact that the CSS styling of the text box changes from grey text for default to black text for user-entered text.[6]

Anyway, here's the JavaScript code that passes the test. It'll pass the missing tests, too—we verified everything in the browser.

Download huddle_mocha/public/javascripts/search_box.js

```
var default_text = "Type Name or Company";

function search_focus(element) {
  if (element.attr('value') == default_text) {
    element.attr('value', '');
    element.addClass('search_entry');
    element.removeClass('search_default');
  };
}

function search_blur(element) {
  if (element.attr('value') == '') {
    element.attr('value', default_text);
    element.addClass('search_default');
    element.removeClass('search_entry');
  };
}
```

We're not done, though. The code still needs to be integrated into the app. Somewhere in our header, the following search field was placed.

```
<%= text_field_tag "unified_search", "Type Name or Company",
    :class => "search_default search" %>
```

And then the jQuery document loading function needs to tie the actual DOM events to the methods we created to pass the tests.

6. Wait, I've always wanted to say this. These are left as exercises for the reader. I *totally* meant to do that.

```
$(document).ready(function(){
 $(".search").focus(function(event) {
   search_focus($(this));
 });

 $(".search").blur(function(event) {
   search_blur($(this));
 });
});
```

Which brings me to the thing that frustrated me most about learning Blue Ridge. I was determined to try and end-to-end test, and if I couldn't have actual output from my Rails app, then I at least wanted to actually trigger the DOM events, and have the jQuery handlers get called inside the Blue Ridge tests. In other words, rather than having my test call search_focus(), I wanted to actually trigger a DOM focus event on that item and let jQuery call search_focus().

Doesn't work that way, apparently—at least, not at the moment. The Blue Ridge folks don't recommended testing via the actual event, as getting all the timing of the various loading and event bindings happening in the test environment appears to be prohibitively painful. Which is not to say that extracting the functions out of the jQuery event handlers is a bad practice—it's a perfectly fine practice—just that testing only the function leaves another potential gap between your tests and your app, and bugs just love to crawl into that kind of gap.

10.6 When and How to Use Blue Ridge

Blue Ridge has a lot of great qualities. With Blue Ridge, it's possible to integrate your frontend JavaScript and your backend Ruby into the same TDD development process. For many teams, that's going to be a significant improvement.

I think—speculating a little bit—that the key to being successful with Blue Ridge is to be aware of its limitations and try to keep the test setup as simple as possible. You are much better off, I think, having many small Blue Ridge test setups, each with a short HTML fixture file, than with a few large and entangled ones. Your biggest risk using Blue Ridge is that the DOM in your actual application is different from the test. The smaller and cleaner the test HTML is, the easier that problem is going to be to diagnose. Later, in Chapter 16, *Browser Testing with Selenium*, on page 215, we'll see how to perform acceptance testing of your actual application JavaScript in a browser.

This brings us to the end of our discussion of testing the user-facing parts of your application in Rails. We've covered controllers, views, email, helpers, and Ajax, and ideally given you a sense of the best way to use these features to help you design and implement solid code for this critical part of your application. Next, we're going to look at two larger add-ons that don't so much augment the way core Rails testing works as replace it. Shoulda is technically an enhancement to Test::Unit, but it introduces a completely different style of writing tests, and RSpec is a framework completely separate from Test::Unit, which has its own special syntax and testing style.

Part IV

Testing Framework Extensions

Chapter 11

Write Cleaner Tests with Shoulda and Contexts

Chapter 12

RSpec

Part V

Testing Everything All Together

Chapter 13

Testing Workflow with Integration Tests

Integration tests seem to be used more infrequently than other Rails built-in test types. Certainly there is much less online chatter about how and when to use the built-in integration tests, probably because integration tests don't quite map to the classic TDD/BDD process. On the one hand, this is a shame, because integration tests are a powerful and flexible way to do high-level testing on your application. Done right, they can even serve as acceptance tests. On the other hand, I'm as guilty as anybody of underusing them. For what it's worth, integration tests have been extremely handy when I've used them.

What's in a Name, Part Three

I like the name "integration tests." So I've got nothing here.

Well, I should point out the difference between an integration test and an acceptance test. An integration test is written by the developer for a developer audience, whereas an acceptance test is written by or in close consultation with the customer, for a customer audience. Acceptance tests often have some kind of Domain Specific Language magic that allows for something readable by the customer to be automatically executed as a test; see Chapter 15, *Acceptance Testing with Cucumber*, on page 194 for details.

13.1 What to Test in an Integration Test

Integration tests are the tool of choice in Rails to test any process that spans one or more controller actions. Two common use cases are a multi-step registration process and a purchase/checkout e-commerce function. In both cases, the entire user task flow has an integrity that can only be completely tested when all the steps can be executed together. As a rule of thumb, if one controller action is communicating with a future controller action via session state or other data sharing method, you probably should have an integration or acceptance test covering the entire procedure.

Integration tests tend to be written separately from the normal tight loop of the test-driven development process. Often, a whole bunch of them are written up front to serve as requirements, a guide to development, or proof of completion. Sometimes they are written after primary development to ensure that pieces that have only been tested separately are in fact making the expected communications—for example, to catch a potential issue created when one method saves to the session with one key, while the later method looks for a different key in the session. Oops.

13.2 What's Available in an Integration Test?

Rails integration tests are similar to controller tests. They import the same assertion modules, so any assertion available in a controller test can also be called from an integration test. The same set of method names are available to simulate HTTP requests (get, post, put, delete, head, and xhr), although we'll see in a second that there are some minor differences from the controller versions of these methods.

The two biggest differences between integration tests and controller tests are:

- Integration tests are not tied to a specific controller. This means the action argument to the HTTP request methods needs to be a an object that resolves to a known URL in the system. Most often, this is simply the URL string, such as get 'tasks/show'.
- Integration tests maintain one or more separate sessions that persist between the different simulated user requests—in a controller test, the session is a characteristic of the response and is not guaranteed to behave correctly if you simulate multiple calls.

With the HTTP simulation methods, you can use any object that Rails can convert into a URL by using the method url_for(). The argument could be a hash or a resource object, as in get url_for(tasks(:one)).

The second argument to all the HTTP request methods is a hash of parameters. Using the second argument is the preferred mechanism for adding parameters in an integration test, rather than building up a query string for the URL or using a complex url_for() call. The third argument (rarely used) contains any arbitrary HTTP headers that you wish to set for the request. As with controller methods, the xhr() method takes a first argument specifying the HTTP verb, then includes the same arguments as the other request methods in order.

By default, integration test requests do not follow redirects. However, each method except xhr() has a via_redirect variant, such as post_via_redirect(), that performs the request and follows all redirects until it gets to a result that isn't a redirect. Any and all session changes it hits along the way will go into the session object, and the output to be tested will be the output of the final destination. If you want to control the redirect behavior less automatically from inside your test, the integration test method follow_redirect!() will, as you might expect, follow exactly one redirect, raising an exception if the last request did not end in a redirect. You can use the integration test method redirect?() to determine if the last request was or was not a redirect.[1]

There are also two methods that allow you to change a setting for all future requests in a test. The https!() method will make all future requests behave as though they use HTTPS. Note that this won't actually encrypt anything, since there isn't a real server request being made, but any other controller-side logic that is dependent on the secure nature of the request will be appropriately tested. Switching back to non-HTTPS requests is done using https!(false). Similarly, a multi-hosted application can be tested using the host!() method. If your application has logic that depends on the subdomain, you can switch back and forth with lines like host!(blog.railsprescriptions.com).

Note that unlike controller tests, integration tests do not allow you to pass session values into the request methods. You can set the initial state of the session directly via the session() method, which returns a hash—so session[:user_id] = 3. You should only use this mechanism for

1. The implementation of redirect?() is status/100 == 3. I'm torn as to whether that is clever or or not.

setting the initial state of the session data. During the actual test, you should let the session be implicitly set by the controller actions being tested.

13.3 Simulating Multi-Part Interaction

The general structure of an integration test is pretty simple: make a bunch of controller calls, and validate that everything has worked out as planned.

The key to successful integration testing is to realize that integration tests have a different purpose than the detailed tests that already exist (or will exist) in your controller and model tests. Integration tests are for verifying end-to-end behavior of the application as a whole, not for testing the inner workings of your code. Attempting to use integration tests to fill the role of unit tests is not only going to make it harder to write the integration tests, it's also going to mean more test breakage when you make legitimate changes to the application.

The purpose of the integration test during development is to validate that data getting passed between the different parts of a larger whole is correctly being saved and correctly being used—you're trying to avoid test gaps. The point of using an integration test as an acceptance test is to perform a high-level verification of the application's behavior. In neither case do you need great levels of detail as to the specific HTML being output or the details of the model backend—that's a job for the actual functional or unit tests. You're trying to test the overall behavior of the system, without worrying about implementation details.

So what should an integration test look like? Let's make up an example from the Huddle project, the daily scrum support tool from Chapter 3, *Writing Your First Tests*, on page 39. Let's assume that there are a number of pages from which a user can click a checkbox and add a user to a list of people to follow. At some point, the user can go to a page that lists all the people currently being followed, allowing the user to see their current status, or send a message, or something.

Create the integration test with the following command:

```
$ script/generate integration_test add_friends
```

Your skeleton test looks like this:

```
require 'test_helper'
```

```
class AddFriendsTest < ActionController::IntegrationTest
  fixtures :all

  # Replace this with your real tests.
  test "the truth" do
    assert true
  end
end
```

Assuming that all these controller methods actually exist, a first pass at the integration test might look like this (note that none of the code that will make this pass actually exists in the project right now):

```
test "add friends" do
  post "sessions/create", :login => "quentin", :password => "monkey"
  assert_equal(users(:quentin).id, session[:user_id])

  get "users/show", :id => users(:quentin).id
  xhr :post, "users/toggle_interest", :id => users(:aaron).id
  assert_equal [users(:aaron).id], session[:interest]

  get "users/show", :id => users(:old_password_holder).id
  xhr :post, "users/toggle_interest",
      :id => users(:old_password_holder).id
  assert_equal [users(:aaron).id, users(:old_password_holder).id].sort,
      session[:interest].sort

  #testing removal from the session
  xhr :post, "users/toggle_interest",
      :id => users(:old_password_holder).id
  assert_equal [users(:aaron).id], session[:interest]

  get "users/show", :id => users(:rover).id
  assert_select "div.interest" do
    assert_select div, :text => "Aaron", :count => 1
    assert_select div, :text => "Old", :count => 0
  end
end
```

The test is similar to controller tests, so the activity should be relatively clear; it's going through some site traversal, occasionally making the Ajax call back to the server to add or remove an interesting person, then checking that the session is as expected. At the end, we do just enough testing of the output to ensure that the values in the session are actually used.

In a real system, you might want to refactor this long test into some shorter methods that encapsulate a call and some of the assertions that go with it. Integration tests seem to encourage the creation of rea-

sonably elaborate mechanisms for quickly defining complicated user behavior—which when taken to their logical conclusion result in Webrat or Capybara (see Chapter 14, *Clean Up Integration Tests with Webrat*, on page 184). Here is a sample method, which takes in a user and an expected list of users already in the session. The method simulates the Ajax toggle call, and confirms that the session matches the expected list:

```
def toggle_user(user_symbol, expected_users)
  new_friend = users(user_symbol)
  xhr :post, "users/toggle_interest", :id => new_friend.id
  expected_users = initial_user_expectation.map { |sym| users(sym).id }
  assert_equal expected_users.sort, session[:interest].sort
end
```

The method would be called like this (note that the arguments are all symbol names from the fixture list that are converted to user instances in the method):

```
toggle_user(:aaron, [:aaron])
toggle_user(:old_password_holder, [:aaron, :old_password_holder])
```

Using this kind of common factoring makes a dramatic difference in how quickly you can write integration tests.

13.4 Simulating a Multi-User Interaction

Integration testing has one more trick up its sleeve. Each integration test gets an implicit session that backs all the controller calls. That's nice, but what if you want to test the integration of two or more consecutive or simultaneous sessions? This is a common issue in, say, a social-networking site where users might be communicating with each other.

Happily, it's possible to create an arbitrary number of different sessions and validate values against any of them. By calling the method open_session(), you get a separate session with its own set of instance variables that you can make requests and perform assertions on. A sample might look something like this:

```
test "user interaction" do
  aaron_session = open_session
  quentin_session = open_session
  quentin_session.post("sessions/create", :login => "quentin",
      :password => "monkey")
  quentin_session.post("messages/send", :to => users(:aaron))
  aaron_session.post("sessions/create", :login => "aaron",
```

```
      :password => "monkey")
  aaron_session.get("messages/show")
  assert_equal(1, aaron_session.assigns(:messages))
end
```

Each session is targeted by its own post() or get() messages, and its own set of instance variables. In this case, we're testing that a message created by Quentin shows up in Aaron's message queue; in a real test, you might do some slightly more detailed testing, but again, the point here is making sure that the entire flow works—testing that the show controller action displays messages properly is the work of the controller and view tests.

You can also invoke open_session() with a block, in which case the session created is the argument to the block:

```
test "user interaction" do
  open_session do |quentin_session|
    quentin_session.post("sessions/create", :login => "quentin",
        :password => "monkey")
    quentin_session.post("messages/send", :to => users(:aaron))
  end
end
```

Note that the open_session() call still returns the session object after the block is evaluated, which means the session can be stored in a variable for further calls to be made on it even after the block has closed.

The obvious disadvantage of using the block style is that multiple sessions overlapping is awkward. If there's an obvious advantage, I haven't seen it yet.

Where this gets interesting is when you want to integrate those shorter helper methods with the session—it's obviously useful to have shortcuts for interactions that multiple session objects are going to need to perform. There are some different suggestions that use various metaprogramming contortions to inject helper methods into the session object. For instance, the Rails API docs place the open_session() call inside a login() method. Helper methods defined in a separate module are injected into the session by starting the block with an extend call, something like:

```
module MyAssertionModule
  def message(to)
    post("messages/send", :to => to)
  end
end
```

```
def login(un, pw)
  open_session do |s|
    s.extend(MyAssertionModule)
    # more stuff, including actually making a login request
  end
end

test "trying logins"
  quentin_session = login("quentin", "monkey")
  aaron_session = login("aaron", "monkey")
  quentin_sesson.message(users(:aaron))
end
```

Since the open_session() returns the session, and it's the last expression in the method, the login() method returns a session, fully logged in, and ready to take more calls. What's nice about this mechanism is that the helper methods in the assertion module can use the session as the implicit self(), so calls within the helper methods can be of the form get "login" rather than session.get "login".

Still, as much as I generally love all things Ruby and metaprogrammy, this does feel a little awkward to me. I'm not going to complain if you want to do something a little more straightforward, like this:

```
def login_as(session, un, pw)
  session.post("session/new", :username => un, :password => pw)
end

def message_from(session, to)
  session.post("messages/send", :to => to)
end

test "trying logins" do
  quentin_session = open_session
  aaron_session = open_session
  login_as(quentin_session, "quentin", "monkey")
  login_as(aaron_session, "aaron", "monkey")
  message_from(quentin_session, aaron_session)
end
```

13.5 When to Use Integration Tests

The testing tools in this part of the book—Rails integration tests, Webrat, Capybara, and Cucumber—are designed to work at a level above a unit test. These tools work best when you use them to validate the interaction between already-tested unit components. Cucumber, with its natural language syntax, is ideal for cases in which a customer or

other non-developer is creating or approving acceptance test scenarios. Integration tests tend to be of more use strictly as a developer tool, although Webrat and Capybara do make the syntax for specifying interactions much easier to write and read. All three of these testing tools are also useful as "black-box" tests, in which you only interact with the application via URL request and evaluate only the application response. Black-box testing can be helpful in testing legacy code, since the tests are not dependent on the actual structure of the code.

Integration or acceptance tests should be used to cover any process in your application that has multiple steps, is based on previously created session data, or otherwise crosses multiple user actions. The trick is integrating these tests into your TDD process. Often, integration tests at this level are either written first or last relative to the TDD tests for a feature. When written first, they can act as acceptance tests, and can drive coding in a TDD fashion—a failing acceptance tests triggers a regular TDD process to make the acceptance tests past. When written last, integration tests tend not to drive new code, since they are validating the interaction between TDD-created components that should already work.

Rails integration tests offer by far the easiest way to test the behavior of your application across multiple user sessions—they offer the only easy way to simulate more than one user hitting the site in a test.

Chapter 14

Clean Up Integration Tests with Webrat

Webrat, created by Seth Fitzimmons and maintained by Brian Helmkamp, can be used to make acceptance tests—the ones that treat the application as a black box—easier to write and execute. By now, you know I can't resist a good naming story; the "RAT" in Webrat is an acronym for Ruby Acceptance Testing. When Jonas Nicklas created a library to augment Webrat as an acceptance tool, he continued the rodent theme and called his library *Capybara*.

Although frequently used in conjunction with Cucumber, Webrat and Capybara can support ordinary Rails integration tests and provide a more expressive, easy-to-read syntax. In addition, the same test can be run headless within a Rails environment and, by changing the driver, also run using a browser-based test environment such as Selenium. Capybara also allows you to use the Celerity and Culerity gems to test JavaScript scenarios without opening an actual browser window.

Webrat and Capybara are quite similar. In fact, if Internet scuttlebutt is to be believed (and really, when has the Internet ever lied), the two are likely to merge into a single project by the time you read this. For the purposes of this chapter, I'm going to treat them as though they are more or less already merged, and note differences between the two as they arise.

14.1 Installing Webrat and Capybara

Both Webrat and Capybara are test framework-agnostic, and will work inside Rails for Test::Unit/Shoulda, RSpec, and Cucumber. (Outside Rails, they will also work with Merb and Sinatra, but that's not our focus right now.)

Install either as a gem:

```
$ sudo gem install webrat
$ sudo gem install capybara
```

Both tools have a dependency on the Nokogiri XML parser. If you are on Ubuntu Linux, you may need to run this command in order for Nokogiri to install properly. Installing Capybara on Mac OS X sometimes requires the use of a library called libffi, which can be installed using MacPorts with the command sudo port install libffi.[1]

```
$ sudo apt-get install libxslt1-dev libxml2-dev.
```

Once Webrat is installed, a bit of setup code needs to be placed in your test/test_helper.rb file if you are using Webrat outside of Cucumber:

```
require "webrat"

Webrat.configure do |config|
  config.mode = :rails
end
```

Note that this code goes *outside* the TestCase class structure.

Capybara's default setup is simpler. If you are using Capybara outside of Cucumber, you need the following:

```
require 'capybara'
require 'capybara/dsl'
```

And then inside the module or class using the Capybara commands, you need an include Capybara call.

Cucumber, for its part, automatically detects which of the two libraries is installed for the application, and generates the appropriate installation and support code. You can also make that decision when generating Cucumber's initial support files.

1. Assuming, of course, that you have MacPorts installed. See http://www.macports.org.

14.2 Using the Acceptance Testing Rodents

Webrat and Capybara are relatively simple libraries with two main purposes. First, to provide a DSL for easily specifying form output and user interaction within an integration or acceptance test, and second, to provide a similarly uncomplicated way to specify the expected output of an interaction.

The signature element of the rodent libraries is their flexibility in how form and link elements are specified in the test; this flexibility is consistent across all of the core API methods listed below. In a Webrat or Capybara API method, you identify the element to be acted on with a string. That string, which is called a *field identifier* or a *locator*, can match a DOM element in one of three ways:

- The text of the associated label (or, for anchor tags, the text inside the tag)
- The DOM ID
- The form field name (or, for anchor tags, the HTML title attribute)

If the field identifier is a regular expression, Webrat will correctly find a matching field based on the regular expression in any of these attributes except the form field name. Capybara, which is much more closely tied to XPath in implementation, does not take regular expressions. The XPath backend of Capybara also means that Capybara lookups are case-sensitive; whereas Webrat lookups are not.

The following HTML snippet:

```
<label for="phone">Phone Number</label>
<input id="phone" name="user[phone]" />
```

Would be accessible from Webrat or Capybara as "Phone Number", "phone", or "user[phone]".

Another consistent element throughout these libraries is all methods that manipulate a form element via a field identifier also serve to verify the existence of the element—if the library can't find a matching form element, the test fails.

Most of your interaction with Webrat or Capybara occurs through the following nine methods:

- attach_file(field_locator, path, content_type = nil)( )

Simulates a file attachment to a multipart form. The field_locator is the field getting the file, as specified above. The path is the path to the local file, and the content_type is an optional MIME type. The Capybara version does not take a third argument.

- check(field_locator)()

  Asserts that the field matching field_locator is a checkbox, and changes the checkbox to its checked state.

- choose(field_locator)()

  Similar to check(), but for radio buttons. Any other radio buttons in the same group would then be un-chosen.

- click_button(value = nil)()

  Clicks on a submit button; if the value is passed, then uses that as a field locator to find the button. In Webrat, if there is no value then it will click the first button on the page. In Capybara, the locator value is required.

- click_link(text_or_title_or_id, options = {})()

  Clicks on an anchor link as specified by the first argument. The text_or_title_or_id argument is similar to a field locator, except it matches against the text of the anchor tag, as well as the DOM ID or HTML title attribute. One quirk: for text or title, Webrat does a substring text match, but DOM IDs must match exactly. Webrat intelligently handles   in the text by treating them as spaces, and is also smart enough to follow the Rails JavaScript that fakes HTTP verbs (but not smart enough to follow Rails link_to_remote() calls). We'll talk more about Capybara and JavaScript in a moment.

  In Webrat, you can override the HTTP verb by passing a :method option, and you can disable the Rails HTTP checking by passing :javascript => false. The similar click_link_within() takes as arguments a CSS-style selector and then the text or title, only clicking a link if there is a matching link within a DOM element matching the selector.

  Capybara has a different way to limit scope, and takes no optional arguments. Capybara also does not perform any special treatment of   arguments.

- fill_in(field_locator, options = {})()

This method has one option, and it's used in every call, as in fill_in("Email", :with => "railsprescriptions@gmail.com"). It looks for an input field or text area that matches the field locator, and sets that field's value to the :with option when the form is eventually submitted.

- save_and_open_page()

  This is a wildly helpful method used to diagnose tests that aren't working as expected. It causes the current response to be saved to a file (including any DOM changes from other methods like fill_in) and then opened in your default browser.[2] This is helpful in Cucumber tests, too.

- select(option_text, options = {})()

  Also used with an option most of the time, select("Automotive", :from => "Industry"). This method causes an HTML option with the given display text to be the selected option in its select menu. If a :from option is specified, that is used as a field locator for the select box.

- select_date(date_to_select, options = {})()

  select_datetime(time_to_select, options = {})()

  select_time(time_to_select, options = {})()

  This trio of methods, which are only in Webrat, lets you specify an entire date/time series of pickers in one go, assuming that your form uses the Rails default date and time pickers. Most JavaScript pickers give you a text field that you can manipulate via fill_in(). The :from() option is used as in select(), and these methods all share an :id_prefix option that matches a prefix specified when the date fields are created.

- uncheck(field_locator)()

  The converse of check().

- visit(url = nil, http_method = :get, data = {})()

  Creates a browser request to the specified URL, using the specified HTTP verb, and passing the key/value pairs in the data argument as parameters. Normally, you'd use this at the beginning of a test;

2. The Webrat docs say the opening only works in Mac OS X, but the docs for the gem Webrat uses (Launchy) doesn't seem to make any similar claim. See http://ramblingsonrails.com/using-webrats-save_and_open_page-on-linux for Linux instructions.

later requests would be triggered by click_button() or click_link(). In Capybara, the method signature is visit(path, attributes = {}), and the HTTP verb is preset to GET.

14.3 A Brief Example

In Section 13.3, *Simulating Multi-Part Interaction*, on page 178, we saw an example of an integration test that used the Rails core integration test features to test adding a friend to Huddle. Let's take a look at what that test might look like in a more rodent-y style. Here's the test using Webrat. There are a couple of different ways this might go:

```
Line 1  test "add friends" do
        -   visit login_path
        -   fill_in :login, :with => "quentin"
        -   fill_in :password, :with => "monkey"
      5 -   click_button :login
        -   assert_equal(users(:quentin).id, session[:user_id])
        -
        -   visit users_path(users(:quentin))
        -   click "toggle_for_aaron"
     10 -   assert_equal [users(:aaron).id], session[:interest]
        -
        -   visit users_path(users(:old_password_holder))
        -   click "Toggle"
        -   assert_equal [users(:aaron).id, users(:old_password_holder).id].sort,
     15 -       session[:interest].sort
        -
        -   visit users_path(users(:old_password_holder))
        -   click "Toggle"
        -   assert_equal [users(:aaron).id], session[:interest]
     20 -
        -   visit users_path(users(:rover))
        -   assert_select "div.interest" do
        -     assert_select div, :text => "Aaron", :count => 1
        -     assert_select div, :text => "Old", :count => 0
     25 -   end
        - end
```

There are a few things that are interesting about this test relative to the basic integration test.

This test has much more of an acceptance test style than the original test. The language has changed, moving away from application-like language such as get "users/show", :id => users(:quentin).id to the somewhat

more user-level language visit users_path(users(:quentin)) and click "Toggle".[3] Where the original integration test made server calls directly from code, this test infers server calls based on simulating form submits and user clicks. In cases like the login on lines 2-5, this makes the code more verbose. However, it makes the test a more complete exercise of the entire stack from start to finish.

Almost. Because here's a critical point: compared to the original test, this test cheats. In the original plain integration test, the toggle calls were all xhr() calls triggering an Ajax response. Webrat, however, doesn't handle Ajax (unless, as we'll see in Section 14.4, *Webrat and Ajax*, on the next page, you configure it to run much slower Selenium tests). In this case, the Webrat test is assuming that the toggle method links will still return the same values if called from something other than an Ajax link.

Capybara has a slightly different set of tools for specifying output. By default, these tools are XPath-based, but if you're like me and are more comfortable using CSS selectors, you can set CSS as the default by placing the line of code Capybara.default_selector = :css somewhere in your test, like in a setup block. The last couple of lines of the above test might look like this in Capybara:

```
within("div.interest") do
  page.has_css?("div", :text => "Aaron", :count => 1)
  page.has_no_css?("div", :text => "Old")
end
```

The CSS default selector value only applies to the within() method—the has_css() method naturally expects CSS, and the associated has_xpath() method works similarly, but with a main argument that uses XPath. In this case, within() acts the same as the outer assert_select() block; namely, it checks for the existence of the selector and limits searches inside the block to the portion of the page inside that matching tag. In Capybara, a within() is not limited to being used with the query methods, but can be placed around the interaction methods to limit those searches to a particular part of the page.

3. I can't imagine it makes much difference, but you can assume that I mean the capitalized "Toggle" to be the text label, and the lowercase "toggle_for_aaron" to be a DOM ID.

14.4 Webrat and Ajax

Webrat has a significant Achilles' heel when run from within Rails. It does not have its own JavaScript interpreter, so any JavaScript in the page you are loading, the link you are clicking, or the form you are filling will be quietly ignored. This is a limitation when trying to test an application with any kind of significant Ajax or JavaScript component.

As a partial workaround, Webrat can be used to drive tests with in-browser verification using Selenium or Watir. You must have the tool you wish to use installed, and you need a Rails environment set up for type selenium, meaning a database.yml listing and a file in config/environments. Somewhere in that environment or the test file, you need the following configuration code:

```
Webrat.configure do |config|
  config.mode = :selenium
end
```

You can set the port that Selenium listens on inside that same configure block with the method config.application_port; avoid Selenium's default port, which is 4444.

In this configuration, Webrat will run a Mongrel server and simulate browser interactions via Selenium, which allows you to test your JavaScript.

(Thanks to the Webrat Wiki: http://wiki.github.com/brynary/webrat/selenium for these instructions.)

14.5 Capybara and Ajax

Capybara has a slightly more advanced approach to Ajax testing. While it still doesn't support Ajax through the Rails default stack, it does provide drivers for two tools that can run JavaScript without requiring a browser window. Capybara can take advantage of Celerity, which is a headless browser simulator running under JRuby (see http://celerity.rubyforge.org/), or, in the very likely event you aren't using JRuby, there is Culerity (see http://github.com/langalex/culerity for information and installation details), which is designed to run in your regular test stack and spawn a JRuby/Celerity process for your JavaScript requests.

You can change the default driver in Capybara with the line Capybara.default_driver = :culerity, where the valid drivers are :celerity, :culerity, :rack_test, and :selenium. The RackTest driver is used for any Rack application that the Capybara tests are being run within, including

Rails. You can override the driver for a single test by using a line such as Capybara.current_driver = :selenium, then return to the default with Capybara.use_default_driver. This allows you to only specify the slow JavaScript drivers for those tests that actually require them. In Chapter 15, *Acceptance Testing with Cucumber*, on page 194, we'll see that Capybara also allows you to manage the drivers on a test-by-test basis in Cucumber.

Capybara has one additional trick for managing Ajax. By definition, an Ajax call is, well... asynchronous. That's great in a web page, since it means the entire page doesn't hang while waiting for the result. But it's a problem for a test, since the test doesn't wait for the result. Because the test normally wouldn't wait, assertions that might depend on the Ajax call having finished might fail because the test goes faster than the Ajax call.

Capybara gets around this by inserting wait periods into the test. In essence, any Capybara query call, such as has_css?() continues to check the page until the expected DOM elements show up or until two seconds have passed—at which point the test will throw up its virtual hands in exasperation and fail. If you have particularly slow Ajax elements and you can't speed them up for some reason, you can change the wait time with Capybara.default_wait_time = 3; the right-hand side is the timeout in seconds.

This causes a weird gotcha when testing for the removal of text. As we saw in Section 14.3, *A Brief Example*, on page 189, Capybara provides explicit negative methods, like has_no_css?(). You should always use the negative method, as opposed to something like !has_css?.[4] The reason for this is Capybara's wait behavior. Let's say you are testing that an Ajax call removes a DOM element from the page. When the negative method has_no_css?() is called and the Ajax method hasn't returned, the element may still be there, but the negative method knows to wait and check again for the removal of the element. If you erroneously use the positive method has_css?(), the method will see that the element in question is there and immediately stop waiting, since the positive method's purpose in life is to wait until something shows up. Then the bang operator would cause the entire assertion to fail. The lesson is that Capybara provides negative methods for a reason, and you should use them.

4. In RSpec, you should avoid should_not have_css.

There's one other minor gotcha when using Rails as a driver: Capybara doesn't like an anchor tag with just a hash href, as in <a href="#">, so you'll need to avoid that in your Rails code.

14.6 Why Use the Rodents?

In some sense, it's difficult to provide standalone examples of Webrat and Capybara, as both tools have seen most of their usage inside Cucumber tests. Used with the core Rails tools, though, they can make the Rails integration tests behave more like acceptance tests. If Cucumber is not an option for practical or aesthetic reasons, adding either of these libraries can give you some of the acceptance test benefit.

At the moment, I tend to start a new project with Capybara, but there isn't enough difference between the two right now that I'd feel the need to convert already existing Webrat code. Of course, by the time you read this, the tools might have merged, in which case you should definitely use the one left standing.

Having looked at what core Rails offers for acceptance and integration testing, it's time to move on to a library that is specifically designed for acceptance testing: namely, Cucumber.

Chapter 15

Acceptance Testing with Cucumber

Cucumber is a tool for writing acceptance tests in plain language (I almost said in plain English, but that's too limiting—Cucumber speaks a lot of different languages, including LOLcat). It can be a great way to do acceptance testing simply and clearly, especially when you want a non-developer client to sign off on your acceptance tests. On the other hand, Cucumber can also be a tempting way to spin your wheels while feeling like you are accomplishing something. There's a sweet spot, and we'll take a look at where that is in this chapter.

15.1 Getting Started with Cucumber

Cucumber is distributed as a gem. As I write this, the stable gem is 0.7.3. You also want to install either Webrat or Capybara, which Cucumber will use to define common web-browser simulation tasks (see Chapter 14, *Clean Up Integration Tests with Webrat*, on page 184). Like RSpec, Cucumber has both a core gem and a separate gem for the Rails-specific parts.

```
$ sudo gem install cucumber
$ sudo gem install cucumber-rails
```

In Rails 2.3.x, the Cucumber gem can be specified in your environment.rb file or the test.rb specific environment file. Then a rake gems:install and a rake gems:unpack:dependencies puts Cucumber in the vendor/gems directory.

If you are using Rails 3 and Bundler, the Cucumber gems go in your Gemfile like this—you can substitute Webrat for Capybara if you'd like:

```
group :test do
  gem 'capybara'
  gem 'database_cleaner'
  gem 'cucumber-rails'
end
```

In either case, expect Cucumber to bring in a number of other gem dependencies. The GitHub wiki page for Cucumber at http://wiki.github.com/aslakhellesoy/cucumber is the best source for complete documentation and project status.

To start using Cucumber for Rails testing, you need to generate some files. Run the following command from the root directory of your application. In Rails 2.3.x, the command looks like this:

```
$ script/generate cucumber --testunit --capybara
        create  config/cucumber.yml
        create  config/environments/cucumber.rb
        create  script/cucumber
        exists  features/step_definitions
        create  features/step_definitions/web_steps.rb
        exists  features/support
        create  features/support/paths.rb
        create  features/support/env.rb
        exists  lib/tasks
        create  lib/tasks/cucumber.rake
```

In Rails 3, that first command is:

```
script/rails generate cucumber:skeleton --testunit --capybara
```

The options after the word cucumber allow Cucumber to generate support files based on the other tools you are using. I've decided to use the Test::Unit version of the Huddle code, and also chosen Capybara. Unsurprisingly, the other supported options are --rspec, and --webrat. However, the choice of tools should not affect the Cucumber experience overall. (Capybara, though, has a greater ability to test JavaScript actions through Cucumber.)

The generator creates a features directory, into which your actual Cucumber features go, as well as two subdirectories. One of the subdirectories contains some setup code; the other will be where you put your Cucumber step definitions, about which more in a moment. You also get a Rake file and a cucumber script, which lets you run Cucumber features from the command line, and can be used to run just one Cucumber feature file, among other tricks. You also get a separate Rails environment for

Cucumber that can have its own separate features, just like the default environment choices such as development or test. If you look at the database.yml file, you'll see that it has been augmented with a Cucumber environment entry, making it at least theoretically possible to run your Cucumber features simultaneously with other tests and not have the two tests trip over each other. The cucumber.yml file contains some setup settings that need not concern us at the moment.

15.2 Writing Cucumber Features

What Cucumber allows you to do is write acceptance tests for new features in a lightly structured natural-language format, then convert those tests into executable Ruby code that can be evaluated for correctness. We'll talk more about how this might affect your workflow in a little bit. First, let's go through an example.

Right now, the Huddle application doesn't have much of a user interface. There's no way to associate a user or a user's status reports with a project, for example. So let's create one. From the Huddle root directory, run the following command:

```
$ script/generate feature users_and_projects
  exists  features/step_definitions
  create  features/manage_users_and_projects.feature
  create  features/step_definitions/users_and_projects_steps.rb
```

In Rails 3, the initial command is script/rails generate cucumber:feature.

The Cucumber generator gives you a feature file where you put the actual Cucumber code[1] and a Ruby file where you put the step definitions that bridge the gap between Cucumber and your actual Rails application. There's also some boilerplate code, which is useful if you are trying to do acceptance test for basic CRUD functionality. (We aren't, so I'm deleting it all.) There's related boilerplate in the step definition file, that's going away as well.

Instead, I'm going to describe the feature I do want. A Cucumber feature starts with a header:

Download cucumber/features/manage_users_and_projects.feature

```
Feature: Add users to project
  In order to make this program even minimally useable
```

1. Technically, Cucumber is the entire system. The feature language is called Gherkin (an outstandingly fun word to say over and over again).

```
  Pretty much everybody on the planet
  wants to be able to add users to a project
```

This is strictly for the humans. The only Cucumber requirement in version 0.7 and up is that the first real line of the file ("real" line means not a comment and not a tag) must start off with the keyword Feature. Earlier versions of Cucumber have no constraints on the header.

After the header, there's an optional background section.

Download cucumber/features/manage_users_and_projects.feature

```
Background:
  Given a project named "Conquer The World"
  And the following users
    | login| email             | password| password_confirmation|
    | alpha| alpha@example.com| alpha1  | alpha1               |
    | beta | beta@example.com | beta12  | beta12               |
```

The Background line is indented in-line with the In order to lines, meaning that it's two spaces in from the left side. Statements within the background are indented a further two spaces—Cucumber uses the indentation to infer structure, similar to Python or YAML. The Background statements are analogous to a startup block and are evaluated before each scenario is run. We'll walk through the details of the statements in the background in just a moment.

An individual unit of a Cucumber feature is called a *scenario*. I've defined two. The first goes to the edit page display for a project:

Download cucumber/features/manage_users_and_projects.feature

```
Scenario: See user display on edit page
  Given that user "alpha" is a member of the project
  When I visit the edit page for the project
  Then I should see 2 checkboxes
  And the "alpha" checkbox should be checked
  And the "beta" checkbox should not be checked
```

And the second goes to what happens when the edit form is actually submitted:

Download cucumber/features/manage_users_and_projects.feature

```
Scenario: See users in project edit
  Given I am on the edit page for "Conquer The World"
  When I check "alpha"
  And I press "Update"
  Then I am taken to the show page for "Conquer The World"
  And I should see "alpha@example.com"
  And I should not see "beta@example.com"
```

A Cucumber scenario has a basic structure with three parts: Given, which indicates a preconditions to the action; When, indicating a user action that changes the state of the application; and Then, verifying the result of the state change. A line beginning with And belongs to its most immediate predecessor clause. For the most part, though, the distinction between Given, When, and Then is for the humans; Cucumber does not require the steps to be in a particular order, nor is the Given/When/Then header used to determine which step definition is matched.

Obviously, the items in the Background clause are all expected to be Given. As for the rest of each clause, that's pretty much free-form; it's the step definitions that give those structure. Oh—the thing that looks like a table in the Background clause. Guess what? It's a table, and that's the preferred method of defining a set of data for a Cucumber feature, rather than using fixtures or factories.

As it happens, this feature can be executed right now, using the command rake cucumber. At the moment, however, we must comment out the before_filter :require_user line in app/controllers/application_controller.rb. (We'll see how to handle logins from Cucumber in Section 15.6, *Login and Session Issues with Cucumber*, on page 210.) It's not going to do much, though.

Each scenario gets run step by step; if your terminal accepts it, the results are color-coded for success, failure, skipped due to an earlier failure, and undefined. Here's part of the output. Here is enough of the output to get the gist:

```
$ rake cucumber
UUUUUUU--UU---U--

2 scenarios (2 undefined)
15 steps (7 skipped, 8 undefined)
0m0.022s
```

At the end, you get a summary. The "15" steps is a count of the steps in the background section as part of each test: the two scenarios define 11 steps, then the two background steps are run twice, giving a total of 15 steps. After that, you get some handy boilerplate descriptions of the steps that are undefined, suitable for pasting directly into your step definition page.

```
You can implement step definitions for undefined steps
with these snippets:
```

Running Cucumber

The cucumber-rails plugin provides three separate rake tasks to run cucumber: rake cucumber (also aliased as rake cucumber:ok) is the default. When first called, it will run the entire suite, and make a note of which scenarios failed. If there are known failing scenarios, subsequent runs of rake cucumber will automatically limited to the failing scenarios until they pass. This behavior is supposed to discourage you from adding scenarios to a test suite that is already failing, but in practice it an be surprising and counter-intuitive.

You can bypass this behavior by using the rake cucumber:all task, which will run all scenarios; or by running rake cucumber:wip, which runs all scenarios tagged with @wip. The cucumber:wip task also has the same behavior as a Cucumber command-line option also called wip; namely, that it expects all the tests to fail—otherwise they wouldn't be "work in progress." The cucumber:wip task also fails if there are more than three scenarios tagged as wip.

You can also use the cucumber command-line script directly for more options, including the ability to specify tagged scenarios to run. More on that in Section 15.7, *Annotating Cucumber Features with Tags*, on page 210. The behavior of the rake tasks can be tweaked by changing parameters that are stored in the file config/cucumber.yml.

```
Given /^a project named "([^\"]*)"$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end

Given /^the following users$/ do |table|
  # table is a Cucumber::Ast::Table
  pending # express the regexp above with the code you wish you had
end


«many other undefined steps»
```

At this point, it's time to tell Cucumber what each of these steps should actually do.

15.3 Writing Cucumber Step Definitions

Step definitions need to be created for each undefined step. Cucumber does its level best to create useful step definition shells for undefined steps when run. We can paste these shells directly into the features/step_definitions/users_and_projects_steps.rb file. At least as a start.

As you can see from the sample, each step definition starts with a regular expression that defines what steps match it. When a matching step is found, the block attached to that step definition is executed. Groups in the regular expression which are, as always, marked by parenthesis, are passed as arguments to the block, which enables a single step definition to match many different steps. Since we want this step to actually create a project, we need to tell Cucumber how to accomplish that creation:

Download cucumber/features/step_definitions/users_and_projects_steps.rb

```
Given /^a project named "(.*)"$/ do |project_name|
  @project = Project.create!(:name => project_name)
end
```

This step definition matches the step Given a project named "Conquer The World" The regular expression group notation causes Conquer The World to be the argument to the block and, eventually, the name of a new project added to the database. Assigning that project to @project allows the project to be accessed from other steps, though you do need to be careful with instance variables. Creating too many instance variables makes steps more interdependent, and as a result, it's more challenging to reuse steps in multiple scenarios.

The way I like to work when using Cucumber is to go one step at a time. Define the step, then add any regular tests and code needed to make the step pass. (At least, that's how I like to work with Cucumber this week. It's very flexible and my work flow changes.) In other words, a failing line in a Cucumber scenario triggers a regular Rails test in the same way that a failing Rails test triggers application code. Sometimes this means going back to Cucumber and changing the scenarios around some, especially when setting up the available data.

This step already passes. Let's move on. The next step is And the following users. The step definition is:

Download cucumber/features/step_definitions/users_and_projects_steps.rb

```
Given /^the following users$/ do |user_data|
  User.create!(user_data.hashes)
```

```
end
```

Starting with maybe the most obvious point, even though the step is actually introduced in the scenario with And, since it's actually part of the Given clause, we describe it with a Given step definition.

The body of this definition shows how to use the table data. The data comes in as a custom Cucumber object, calling hashes() on the object converts the object to an Array of hashes, with keys corresponding to the first row of the table, and the values corresponding to each remaining row in turn. With the table that was written in the feature, this means that two users are created with the logins alpha and beta. This step passes already as well.

Moving on to the actual scenario, we start with one more given: Given that user "alpha" is a member of the project. This is another relatively easy one that passes without much further work:

Download cucumber/features/step_definitions/users_and_projects_steps.rb

```
Given /^that user "(.*)" is a member of the project$/ do |login|
  User.find_by_login(login).projects << @project
end
```

To make this work we need to add has_and_belongs_to_many() lines to both Project and User. In app/models/project.rb, do this:

```
class Project < ActiveRecord::Base

  has_many :status_reports
  has_and_belongs_to_many :users
```

And in app/models/user.rb, this:

```
class User < ActiveRecord::Base
  has_and_belongs_to_many :projects
```

Okay, we've got one more that will pass before we get to the hard stuff:

Download cucumber/features/step_definitions/users_and_projects_steps.rb

```
When /^I visit the edit page for the project$/ do
  visit("/projects/#{@project.id}/edit")
end
```

In this case, I'm using the visit() method, provided by Capybara to simulate a browser call to the given URL. You can read more about Capybara in Chapter 14, *Clean Up Integration Tests with Webrat*, on page 184; for the moment, all we need to know is that Cucumber lets us call Capybara functions to simulate user actions. This step also passes as-is.

If you are at all inclined to be skeptical about Cucumber's general awesomeness, right about now you may be wondering, "What's the point of all this silliness?" So far, we've written some formal-sounding natural language stuff, and some Ruby that has served merely to confirm things about our application that we already know, at the cost of something like a page of code.

True, so far—although to some extent this is a side effect of the fact that we're adding Cucumber tests to an application that already exists. In much the same way that adding tests to an existing legacy application has some cost, adding the first Cucumber tests to an existing non-Cucumber'd program has some cost. In essence, we're paying off some of the technical debt accrued by Huddle for not having acceptance tests. However, it is easier to add Cucumber tests to an existing program than unit tests—since Cucumber works only at the level of user input and output, any crazy or messed-up code structures in the application code don't really affect it.

As for the eventual benefit, bear with me for a little bit. I'll make that case after we complete the successful scenarios.

15.4 Making Step Definitions Pass

We're going to write the next three step definitions together, then make them all pass. The first step definition to write is for the step Then I should see 2 checkboxes. The definition of the step looks for checkboxes in the HTML using Capybara's page.has_css?() method, which is, for our purposes, effectively equivalent to assert_select():

Download cucumber/features/step_definitions/users_and_projects_steps.rb

```
Then /^I should see (.*) checkboxes$/ do |checkbox_count|
  page.has_css?("input[type = checkbox][id *= user]",
      :count => checkbox_count.to_i)
end
```

The next step definition to write is for the step Then the "alpha" checkbox should be checked. As it happens, this step is already defined by Cucumber as part of the web_steps.rb file:

Download cucumber/features/step_definitions/web_steps.rb

```
Then /^the "([^\"]*)" checkbox(?: within "([^\"]*)")? should be checked$/ do |label,...*TRUNC*
  with_scope(selector) do
    if defined?(Spec::Rails::Matchers)
      find_field(label)['checked'].should == 'checked'
    else
```

```
      assert_equal 'checked', field_labeled(label)['checked']
    end
  end
end
```

After that comes the step Then the "beta checkbox should not be checked". That one is also defined by Cucumber.

Download cucumber/features/step_definitions/web_steps.rb

```
Then /^the "([^\"]*)" checkbox(?: within "([^\"]*)")? should not be checked$/ do |la...*TRUNC*
  with_scope(selector) do
    if defined?(Spec::Rails::Matchers)
      find_field(label)['checked'].should_not == 'checked'
    else
      assert_not_equal 'checked', field_labeled(label)['checked']
    end
  end
end
```

Both of these steps are defined similarly, using Capybara to find the desired checkbox in the DOM, then using either RSpec matchers or Test::Unit assertions to verify that the checkbox is or is not checked.

With the step definitions for our scenario complete, and two of them not passing, we finally have a direction for our code. We need to add the checkboxes to the edit view for the project.

Adding the checkboxes to the edit screen probably does not require me to change the controller code. What with this being the Cucumber chapter and all, I'm feeling bold, so let's go to directly to the view.

I inserted this right below the name field in the file app/views/project/edit.html.erb. I am under no illusion that this view code will scale beyond about 10 users.

Download cucumber/app/views/projects/edit.html.erb

```
<h2>Users In Project</h2>
<table>
  <% User.all.each do |user| %>
    <tr>
      <td>
        <%= check_box_tag "users_in_project[]", user.id,
            user.projects.exists?(@project.id),
            :id => dom_id(user, :checkbox) %>
      </td>
      <td>
        <label for="<%= dom_id(user, :checkbox) %>">
          <%= h user.login %>
        </label>
```

Does Cucumber Replace Tests?

There's an ongoing debate among Rails testing people over whether Cucumber is best used in addition to the TDD/BDD tests you would already be writing, or whether Cucumber should be used to replace some of those tests, particularly the controller and view tests. One side says that the Cucumber test already covers the code added by a controller test. The other side says that, although the same code is touched by each test, the tests have different purposes, and are both useful. In essence, the question is whether writing unit or functional tests have value beyond merely verifying correctness, since the Cucumber test already verifies that. Cucumber's value in end-to-end testing and keeping development focused does not replace the exploration, design, or code-quality benefits of a regular TDD process.

The goal of Cucumber is to test end-to-end from a user perspective. As such, Cucumber is not the place to test internals of the program that only tangentially show up to users. Internal logic of models should be tested in unit tests. Best practice suggests that controllers shouldn't have much logic of their own, which means you might not need controller tests—not because the Cucumber tests replace them, but because there isn't much controller logic to test.

That said, I do often find Cucumber to be an easier way to specify view tests than the view test facilities provided by Test::Unit or RSpec. As a result, I find myself basically moving the view tests I would normally have written to Cucumber.

The answer to whether Cucumber replaces regular tests is: sometimes. Cucumber does not eliminate the need for the benefits of TDD in regular testing. However, Cucumber can be used as an easy way to write TDD tests for user-facing code. The point of Cucumber is decidedly not to get bogged down in whether you need extra tests or not. The goal of Cucumber is to write better code and focus your development efforts. Try using Cucumber a few different ways and see what works best for you.

```
      </td>
    </tr>

  <% end %>
</table>
```

So, for each user in the database, we add a table row with a checkbox and user name. The label tag is important here, because that's what the Cucumber/Capybara steps are looking for.

At this point, the scenario passes. Yay, us! Now let's get to work on the next one.

15.5 The Edit Scenario: Specifying Paths

Since it's been pages and pages since we've seen the second scenario, I'll re-run it here:

Download cucumber/features/manage_users_and_projects.feature

```
Scenario: See users in project edit
  Given I am on the edit page for "Conquer The World"
  When I check "alpha"
  And I press "Update"
  Then I am taken to the show page for "Conquer The World"
  And I should see "alpha@example.com"
  And I should not see "beta@example.com"
```

This scenario currently fails right off the bat with Given I am on the edit page for "Conquer The World". Hmm. Given I am on is a pre-existing web step, but the rest of step assumes something that can be converted to a URL.

Let's try this:

```
Given /I am on the edit page for "(.*)"/ do |project_name|
  @project = Project.find_by_name(project_name)
  visit("/projects/#{@project.id}/edit")
end
```

Seems reasonable, if very similar to the When statement I wrote a couple of paragraphs back. The problem, though, is that Cucumber doesn't like it:

```
Ambiguous match of "I am on the edit page for "Conquer The World"":

      features/step_definitions/web_steps.rb:18:in `/^I am on (.+)$/'
      features/step_definitions/users_and_projects_steps.rb:15:
          in `/I am on the edit page for "(.*)"/'
```

And that answers the question of what Cucumber does if there's a step that matches two definitions.

There's another way to specify the path. The definition for the default Cucumber step /I am on (.*)/ defers to a method called path_to(), which is passed the regular expression group as an argument. That method is defined in the file features/support/paths.rb as follows:

```
def path_to(page_name)
  case page_name

  when /the home\s?page/
    '/'

  # Add more mappings here.
  # Here is an example that pulls values out of the Regexp:
  #
  #   when /^(.*)'s profile page$/i
  #     user_profile_path(User.find_by_login($1))

  else
    raise "Can't find mapping from \"#{page_name}\" to a path.\n" +
      "Now, go and add a mapping in #{__FILE__}"
  end
end
```

See that part where it helpfully says "add a mapping"? What we want to do here is add more where clauses that return some kind of Rails URL-like object. So delete the Given clause we just wrote, and add the following When clause to the path_to() method in features/support/paths.rb:

Download cucumber/features/support/paths.rb

```
when /edit page for "(.*)"/
  @project = Project.find_by_name($1)
  edit_project_path(@project)
```

I want to walk this out step by step, because it can take a couple of tries to figure it out:

1. Cucumber sees the step Given I am on the edit page for "Conquer The World".
2. The step is matched to the existing web step Given /^I am on (.+)$/.
3. The web step takes as an argument the grouped text: the edit page for "Conquer The World" and passes it to the path_to() method.
4. The path_to() method matches the regular expression we just added, /edit page for "(.*)"/.

Route Support

One problem with the path_to() method is that it requires you to separately enter paths before they can be used by Cucumber. This feels like duplication. Solomon White posted this snippet at http://onrails.org/articles/2010/04/06/cucumber-meet-routes as a replacement for that duplication (I've edited it slightly). It infers a RESTful route from the text. This code replaces the else clause of the path_to() method:

```
else
  begin
    page_name =~ /the (.*) page/
    path_components = $1.split(/\s+/) << 'path'
    self.send(path_components.join('_').to_sym)
  rescue Object => e
    raise "Can't find mapping from \"#{page_name}\" to a path.\n" +
          "Now, go and add a mapping in #{__FILE__}"
  end
end
```

Using this snippet, something like "the story page" would convert to story_path, while "the edit user page" would send the route edit_user_path.

5. The grouped text in this regex is used to convert that expression to the RESTful edit path for the associated project. The $1 is a Ruby global for the first match group of the most recently matched regular expression. While that's a Perlism that I don't normally use in my Ruby code, it's the easiest way to get at the regular expression match data that's in the when clause.

There are two ways of looking at this mechanism. On the plus side, it's a very flexible way to allow a path to be specified in a meaningful plain language way. On the other hand, it's got a certain fog-on-fog quality, and there's a lot of indirection.

At this point the step should pass, and the next two steps, When I check "alpha" and And I press "Update", should also pass because they are web steps that are only dependent on the existence of the form elements we created for the first scenario. They have the effect of mimicking the user actions of checking a checkbox and submitting the edit form with one user checked.

The fourth step, Then I am taken to the show page for "Conquer The World", needs to be defined. The intent of this step is to validate that the form submission takes the user to the page intended. The step definition looks like this:

Download cucumber/features/step_definitions/users_and_projects_steps.rb

```
Then /^I am taken to (.*)$/ do |path|
  assert(current_url.ends_with?(path_to(path)))
end
```

This definition is dependent on another clause in the path_to() method, very similar to the last one:

Download cucumber/features/support/paths.rb

```
when /show page for "(.*)"/
  @project = Project.find_by_name($1)
  project_path(@project)
```

The step definition is comparing the URL for the current page, which will be http://www.example.com/projects/1 to the URL path output from path_1, which will be projects/1.

Now we're at the meat of the scenario. The last two steps, And I should see "alpha@example.com" and And I should not see "beta@example.com" are web steps that search for specific text in the body output. In this case, we're assuming that the eventual project show page will include the emails of the users on the project—and not include the emails of users who aren't in the project.

You have to be careful here: the Cucumber test is explicitly not testing that the user has actually been added to the project in the database. You could write such a test, but it is considered better practice to manage that at the controller and model test level and keep Cucumber at the level of user interaction. That's fine, but it's also true that we could make the Cucumber test pass by just including the text string in the output. Or, more insidiously, by just passing the form submission data to the view without saving it.

My point here is not that Cucumber is bad—it's not. It's more to say that Cucumber is only part of your nutritious testing breakfast. In this case, we are going to change the controller, so we should step down to controller tests. Put this controller test in test/functional/projects_controller_test.rb:

Download cucumber/test/functional/projects_controller_test.rb

```
test "should update with users" do
  set_current_project(:huddle)
```

```
  put :update, :id => projects(:huddle).id,
      :users_in_project => [users(:quentin).id]
  huddle = Project.find_by_name("Huddle")
  assert_equal [users(:quentin).id], huddle.user_ids
end
```

This confirms that sending user IDs to the update will be converted to users in the actual project. The case where there are no users being passed is actually being taken care of by the already existing update test. To pass the controller test, the following goes into update() in app/controllers/projects_controller.rb:

Download cucumber/app/controllers/projects_controller.rb

```
def update
  @project = Project.find(params[:id])
  @users = begin User.find(params[:users_in_project]) rescue [] end
  @project.users = @users
  respond_to do |format|
    if @project.update_attributes(params[:project])
      flash[:notice] = 'Project was successfully updated.'
      format.html { redirect_to(@project) }
      format.xml  { head :ok }
    else
      format.html { render :action => "edit" }
      format.xml  { render :xml => @project.errors,
          :status => :unprocessable_entity }
    end
  end
end
```

That passes the controller test, and puts the data in the database. But to pass the Cucumber test, the data needs to get into the view. Please add the following to the beginning of the app/views/projects/show.html.erb file:

Download cucumber/app/views/projects/show.html.erb

```
<h2>Users for <%= @project.name %></h2>

<table border="0" width="">
  <% @project.users.each do |user| %>
    <tr>
                <td><%= h user.login %></td>
      <td><%= h user.email %></td>
    </tr>
  <% end %>
</table>
```

I will grant that's not anything like a beautiful design. But Cucumber doesn't care, and happily passes. All green; the feature is complete, for some definition of complete.

15.6 Login and Session Issues with Cucumber

There's a big issue that we've kind of glossed over so far: authentication. In a real application, we'd probably need to be logged in to view and edit projects. From a controller test, simulating a login is easy—we have direct access to the controller and session, so we can add the fake user directly to the fake session.

From Cucumber, we don't have access to the controller or the session, so we need to login by simulating user actions. Something like this works if we are using Authlogic:

```
Given /^the user successfully logs into Huddle$/ do
  @user = User.create!(:user,
      :login => "the_login",
      :password => "password",
      :password_confirmation => "password")
  visit "/user_session/new"
  fill_in("User Name", :with => @user.login)
  fill_in("Password", :with => "testing")
  click_button("Submit")
end
```

This step definition creates a user as an instance variable, then travels to the Authlogic login screen and simulates filling in and submitting the login form with Webrat or Capybara. If we wanted a specific user name, it'd be easy enough to change the step definition to take one in. On the plus side, this is largely boilerplate and can be applied to most projects with little or no modification.

Anything session-based will have a similar issue, and will need to be generated by simulating user actions. Again, this indicates that Cucumber isn't going to solve all your testing problems. Just some of them.

15.7 Annotating Cucumber Features with Tags

Cucumber test suites, especially on large sites, tend to be slow. Even with the recent speed improvements in parsing the feature files, the end-to-end testing is still slow relative to unit tests. In order to speed up the development test/code loop, it's handy to be able to run a limited

subset of the Cucumber features so that you can focus on the scenarios that are actually under development.

In Cucumber, you can separate scenarios or features into groups by using *tags*. To define a tag, put the tag in the line above the scenario declaration, preceded by the @ symbol:

```
@userpage
Scenario: See user display on edit page
  Given that user "alpha" is a member of the project
  When I visit the edit page for the project
  Then I should see 2 checkboxes
```

If you want to have multiple tags apply to the same scenario, the tags all go on the same line with a space between them. You can then use the tags by specifying them from the Cucumber command line. For example, to only run scenarios with the userpage tag, use this command:

```
cucumber --tags @userpage
```

You can run all the scenarios that *don't* have a particular tag by prefacing the tag with a tilde:

```
cucumber --tags ~@userpage
```

If you want to make things more complicated, you can logically combine tags. If you include multiple --tags options, they act as a logical and. To run scenarios that are both @userpage and @complete, use this command:

```
cucumber --tags @userpage --tags @complete
```

For logical or behavior, include multiple tags in the same option, separated by commas. To run @userpage or @reporting scenarios, you'd do this:

```
cucumber --tags @userpage,@reporting
```

Also, you can specify a limit to the number of scenarios that can be run with a given tag. Running Cucumber will fail if there are the number of tagged tests exceeds the limit:

```
cucumber --tags @userpage:3
```

This behavior is used by the default cucumber:wip task, but I'm not sure how useful it is in general.

Cucumber gives privileged status to the @wip tag, which stands for "Work In Progress". The @wip tag is special because Cucumber gives

you a default rake task rake cucumber:wip, which only runs scenarios that have @wip tags.

If you are using Capybara, then the Capybara/Cucumber combination uses the special tag @javascript to indicate a scenario that should be run using Capybara's JavaScript driver, which is usually Selenium or Culerity. The JavaScript driver is used instead of the default headless stack. This behavior is not dependent on the command-line options; the JavaScript driver will be applied to all appropriately tagged scenarios. Naturally, you'd expect those scenarios to run more slowly.

15.8 Implicit Versus Explicit Cucumber Tests

After writing Cucumber tests for a while, issues of styling and structure start to become apparent. One of the first questions is whether to put specific details of the scenario in the Cucumber feature or in the step definition.

For example, a scenario that requires differentiating a newly created user from an experienced user could be written in any of the following ways (and others, but these are enough to make the point):

```
Given a user named "Noel Rappin" who joined the site 2 years ago

Given a pre-existing user named "Noel Rappin"

Given a pre-existing user
```

The topmost version is the most explicit; the step definition would take the data in the Cucumber step and use that directly. The bottom most version is completely implicit; the step definition will need to create a user and creation time on its own. The middle one, obviously, is in the middle.

There are advantages to each way. The implicit view leads to simpler, more natural-sounding Cucumber scenarios, and is generally going to be easier for a non-technical client to use. However, the step definitions are more complex and need to manage more implicit data, leading to a greater possibility of error in the step definitions. It's also less likely that you'll be able to reuse implicit definitions. An explicit Cucumber step leads to simpler step definitions, but it can be hard to distinguish the salient details of the Cucumber step. For example, a client who saw a line of code similar to the explicit line here wondered what was special about "2 years ago".

Toward Cucumber Style

The key to whether you have a good or a bad experience with Cucumber is how well you write the step definitions. Here are some guidelines that I've found useful:

- Let the scenario have the natural language that feels right for your context, and smooth things out in the step definition.
- Where possible, keep the step definitions simple; they absolutely need to be 100% accurate.
- I find it better to have multiple simple steps than to have one step that performs tricky regular expression acrobatics.
- Keep the When and Then steps at the level of the user, rather than the database.
- As with other tests, verifying what isn't there is as important as verifying what is.
- Cucumber is not the place to sweat implementation details, save that for the regular tests.
- Avoid tautological tests—it's pretty easy for the indirections of the step definitions to mask an always-passing step.

There isn't a particular right answer here; it's something to keep in mind as you write your Cucumber tests.

15.9 Is Cucumber Good for You?

For a long time, Cucumber and its predecessor StoryRunner were on my list of interesting projects to get around to someday, but it didn't seem tremendously practical. One of the things that made me turn the corner and start using it was the realization that a lot of my objections to Cucumber—writing extra code, the fact that the tests might not always be accurate, that kind of thing—were similar to the kinds of objections I'm always hearing about Test-Driven Development in general. That gave me pause, and I decided to make a concerted effort to try and figure Cucumber out.

It turned out that the analogy to regular TDD continued to hold as I used Cucumber. The startup costs turned out to generally be less than I feared, but the benefit of specifying end-to-end behavior of the application turned out to be high. Somewhat surprisingly, I found that my coding sessions were more focused when I used Cucumber, and I had a much better sense of what needed to be completed and what needed to be done next.

What I wound up with is a double loop, where a failing Cucumber test triggers a regular unit test, and a failing unit test triggers code. It's not a perfect analogy—unit tests have design value and can cover areas of the program that are difficult to reach from Cucumber—but it's a good way to think of Cucumber's place.

If you are dealing with a client or non-programming customer, the value of an acceptance test that is both natural language and executable is outstanding. My only caution at the moment would be that even though there isn't much structure in Cucumber tests, there still is some. We tend to use the Given/When/Then structure for user stories even without Cucumber, but my first attempts at executing some of them in Cucumber still required some translation. For example, the story definitions tended to do too much in one scenario. Be prepared to go back and forth and rewrite your Cucumber steps a little bit over time, for clarity and as new features become apparent. Also, try and keep your step namespace clean; on large projects, finding the steps can become a problem.

Where is the sweet spot for using Cucumber? Cucumber's strengths are in its ability to easily specify and document the end-to-end behavior of your application. It makes an excellent blueprint, both in the sense of describing what needs to be done and in the sense of allowing you to plan the direction of development. It makes an excellent compliment to a typical TDD/BDD process, taking a big picture role that smaller unit tests can't really play. Where you can get in trouble with Cucumber is by trying to move it to areas where it's not really strong, such as attempting to manage model or implementation details. Keep Cucumber in its appropriate place. If you make sure you aren't forever fiddling with the details of the steps, and use it to drive your regular tests the same way you use tests to drive code, Cucumber will improve your code.

Chapter 16

Browser Testing with Selenium

Part VI

Testing Your Tests

Chapter 17

Using Rcov to Measure Test Coverage

Test coverage is the most commonly used numerical metric for evaluating the quality of tests. Coverage measures the amount of application code that is actually executed over the course of running a test suite. At the end of the day, this is usually expressed as a percentage, as in, "My project has 85% test coverage;" or, in other words, "85% of my application code is actually touched when the tests run."

In the right circumstances, you can get into a nice little argument about how meaningful test coverage is, but most people agree on the following points:

1. The absence of code coverage indicates a potential problem—a part of your code is not being tested at all. How big a problem that is depends on the context.
2. The existence of code coverage, in and of itself, means very little. The coverage metric says nothing about whether the application code being executed is actually being verified at all, let alone being verified completely.
3. That said, in conjunction with good testing practices, a coverage run might say something useful about your test suite and its reach. To put it another way, if you are using good TDD practices, you probably are getting good test coverage, but coverage by itself doesn't indicate that you are using good TDD practices.

17.1 85% of What?

Did you see where I tried to put one by you right there in the first paragraph of this section? When I referred to a project having "85%" test coverage. Leading to the natural follow-up question, "85% of what, exactly?" Good question. While the idea of code coverage may seem intuitively clear, nothing is simple once the Computer Scientists get done with it.

Wikipedia currently lists no less than seven different ways to measure coverage, ranging from the theoretically useful to the downright goofy. But really, there are only two measures that are both easy enough to calculate and useful enough to have any currency at all. *Line coverage* is a measure of the percentage of actual, textual lines of code in your application that are executed by your test suite. The alternative, *branch coverage*, attempts to measure each branch of your code and counts each branch equally. (Spoiler alert: Rcov measures line coverage.) For example, say you have a method like the following:

```
def code_coverage_example_silly_method
  if something_good
    tell_all_my_friends
  else
    ignore_it
    run_in_circles
    scream_and_shout
    jump_up_and_down
    run_in_circles_some_more
    whine_all_day
  end
end
```

If you only had a test for the case where something_good == true—and don't we all wish that was the only case we ever had to test—then a line coverage estimate would say you had covered 4 of the 12 lines, for 33% coverage. Specifically, the tests would cover the def itself, the if statement, the tell_all_my_friends, and the end at the end of the method. A branch coverage estimate, on the other hand, would determine that this method has two branches (the if clause and the else clause) and one of them is covered, for 50% branch coverage.

In theory, branch coverage provides a "truthier" picture of your coverage. In the above method, all six lines in the else clause are essentially one unit; they are either all tested or all not. It makes very little sense for the length of each branch to have an effect on how that branch is weighted for calculating code coverage. On the other hand, the practi-

cal difference between the two measures is usually not all that great on real code, and line coverage is about a jillion times easier to calculate. Also, 100% is 100% either way. I submit that if you are deeply concerned about the difference between branch and code coverage in your app, you have bigger problems than your code coverage.

Still, there are some odd artifacts from Rcov's use of line coverage. The most potentially annoying is for fans of single-line if statements or post-statement if clauses, as in:

```
return nil if x
if x then 3 else 2 end
```

Rcov will happily mark the entire line as covered, whether or not the if statement is actually tested in both true and false states. Again, this is more an annoyance than a life-or-death concern, and I absolutely would not recommend adjusting your code style to accommodate Rcov, but it is something to keep an eye on.

17.2 Installing Rcov

Like so many things in life, installing Rcov is at least 10% more complicated than it should be. You download the gem normally, but then need to install a native extension that allows Rcov to run quickly. Without the extension, Rcov pretty much runs in geologic time—instrumenting for code coverage is a slow process.

As I write this the current, live official version of the Rcov gem is a fork of the original gem maintained by Relevance, a leading Rails consulting shop. As far as http://www.rubygems.org is concerned, the Relevance version is now the canonical version of Rcov. To install the the gem, do this:

```
$ sudo gem install rcov
```

If you have not been able to download a binary, once Rcov has been installed as a gem, you must compile the native binary that allows Rcov to run quickly enough to actually be useful. Those of you who are not Windows users need to navigate in the command terminal to the rcov gem directory (the exact directory will depend on your system. For a native Mac OS X installation, it's /Library/Ruby/Gems/1.8/gems/rcov-VERSION) and run:

```
$ sudo ruby setup.rb
```

Windows users aren't officially supported (at least as of the note that I see dated February 28, 2010); however, you may be able to install Rcov via the Windows development kit at http://www.rubyinstaller.org. Good luck.

Windows users can still create a binary from the original pre-relevance version of Rcov, by grabbing the binary file rcovert.so from http://eigenclass.org/static/rcov/binary-win32/rcovrt.so and placing the file in the Ruby extension directory, which if you used the standard one-click installer, is probably c:\ruby\lib\ruby\site_ruby\1.8\i386-mswin32\.

At this point, you might want to try a simple command like rcov --version to make sure that everything is installed correctly.

17.3 Rcov and Rails

Don't worry, installation fans, we're not done yet. There's still a separate plugin for integrating Rcov with Rails. Luckily, this one is pretty easy:

```
$ ./script/plugin install http://svn.codahale.com/rails_rcov
```

This gives you a series of Rake tasks. Specifically, each rake:test task you have defined gets a rake test:rcov and a rake test:clobber_rcov, such as rake test:functionals:rcov and rake test:functionals:clobber_rcov. The global test task is test:test:rcov. The Rcov task runs the associated tests using Rcov, resulting in an Rcov report. This report is placed in a coverage directory at the top level of your Rails app. The clobber task clears out the report data from that directory. You can also run coverage on a specific test file by including the file as the TEST option, as in:

```
$ rake rcov TEST=file_to_test.rb
```

There are some things to be aware of. First, you really don't want those coverage files in your source control, so be sure to use the appropriate ignore mechanism to keep that coverage directory out of the source control tree. Second, the rails_rcov plugin depends on the existence of Rcov. This means that your staging and production environments either need to have Rcov installed or need to have the plugin removed, which you can do as part of your Capistrano deploy script, perhaps.

Finally and most importantly, Rcov completely ignores view files that aren't "real" Ruby, such as ERb and Haml. While I suppose in the abstract I'd kind of like to see my view coverage, I have to admit that in practice my attitude is more along the lines of "one less thing to worry about today." Ideally, you'll be putting any complicated logic in

the model or helper, where it can be tested and the test coverage can be measured.

17.4 Rcov Output

When you run Rcov from one of the Rails rake tasks, you get output in your console, as well as a full HTML output report. (In the next section, we'll talk about how to tweak that output.)

The console output is simply a list of touched files, and will show up after the normal Test::Unit output. Here's an edited look at the text:

```
+-------------------------------------------------+-------+-------+--------
|                File                             | Lines |  LOC  |  COV   |
+------------------------------------------ -----+-------+-------+--------+
|app/controllers/application_controller.rb        |    75 |    50 |  86.0% |
|app/controllers/projects_controller.rb           |    87 |    61 |  80.3% |
|app/controllers/status_reports_controller.rb     |    91 |    65 |  80.0% |
|app/controllers/user_sessions_controller.rb      |    25 |    20 |  60.0% |
|app/helpers/application_helper.rb                |     3 |     2 | 100.0% |
+-------------------------------------------------+-------+-------+--------+
|Total                                            | 78429 | 41952 |  51.2% |
+-------------------------------------------------+-------+-------+--------+
```

The columns are filename, total lines of text in the file, actual lines of code in the file (meaning, lines that are not blank and are not comments), and the percentage of lines of code that are covered. The last line of the output is a sum for the project as a whole.

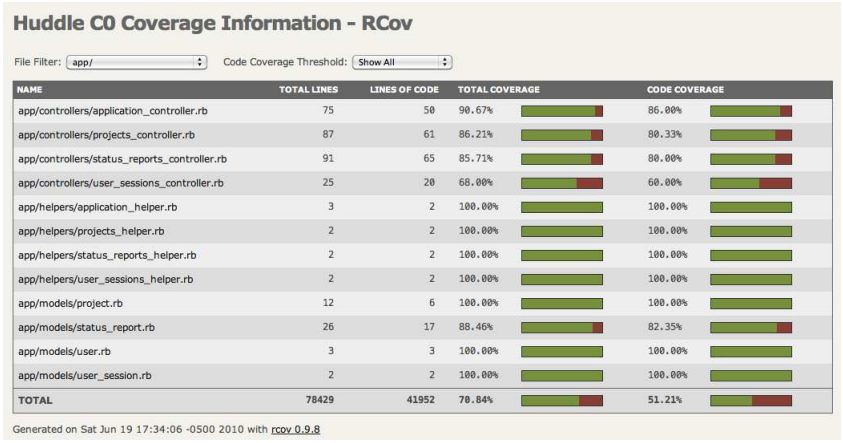
You'll note, especially if you have an unusual Ruby installation, that the default rake task will include seemingly zillions of gem, plugin, and library files that you don't really care about, which are also getting included in the total, making it less useful than it might be. The next section discusses how to get those lines out of your reports to make them more meaningful.

Inside the coverage directory, Rcov has created an entire directory for your coverage report. If you open the index.html file in that directory, you will see something like Figure 17.1, on the following page—I've filtered it to the app directory to make the output more meaningful. Depending on your setup, you might need to limit yourself to just one of the Rcov tests, such as rake test:units:rcov, in order to get a meaningful index file.

At the top of the page, you'll see pulldowns that do some JavaScript filtering of the code based on top-level directory or a threshold value for coverage, only showing files below a specified value. Both of these filters

The Invisible Class Gotcha

Since the Rcov output contains only files that are touched by the test suite, a file that is completely untouched will not show up with 0% coverage—it just won't show up, period. As a result, your coverage score will be artificially high, and you may not see the problem. What with Rails making it pretty easy to get every file in your app at least minimally touched by a test, it's rare to get bitten by this problem. But it does happen—well, it's happened to me—so I thought I'd mention it.



Huddle C0 Coverage Information - RCov

File Filter: app/ Code Coverage Threshold: Show All

NAME	TOTAL LINES	LINES OF CODE	TOTAL COVERAGE	CODE COVERAGE
app/controllers/application_controller.rb	75	50	90.67%	86.00%
app/controllers/projects_controller.rb	87	61	86.21%	80.33%
app/controllers/status_reports_controller.rb	91	65	85.71%	80.00%
app/controllers/user_sessions_controller.rb	25	20	68.00%	60.00%
app/helpers/application_helper.rb	3	2	100.00%	100.00%
app/helpers/projects_helper.rb	2	2	100.00%	100.00%
app/helpers/status_reports_helper.rb	2	2	100.00%	100.00%
app/helpers/user_sessions_helper.rb	2	2	100.00%	100.00%
app/models/project.rb	12	6	100.00%	100.00%
app/models/status_report.rb	26	17	88.46%	82.35%
app/models/user.rb	3	3	100.00%	100.00%
app/models/user_session.rb	2	2	100.00%	100.00%
TOTAL	78429	41952	70.84%	51.21%

Generated on Sat Jun 19 17:34:06 -0500 2010 with rcov 0.9.8

Figure 17.1: Rcov Index Page

are new in the Relevance Rcov, and are greatly welcome. The columns here are exactly the same as the console output with the addition of Total Coverage, which is the percentage of lines covered, counting both code and non-code lines of text.

What does it mean to cover a comment, blank line, or other non-executable line? Rcov claims to infer coverage of non-executed lines by attaching them to the lines nearest them, so an `end` is considered to have been executed if its associated block or if statement is covered, and a comment is generally attached to the first executable line of code after the comment. In practice, the code percentage in the last column is going to be the most useful.

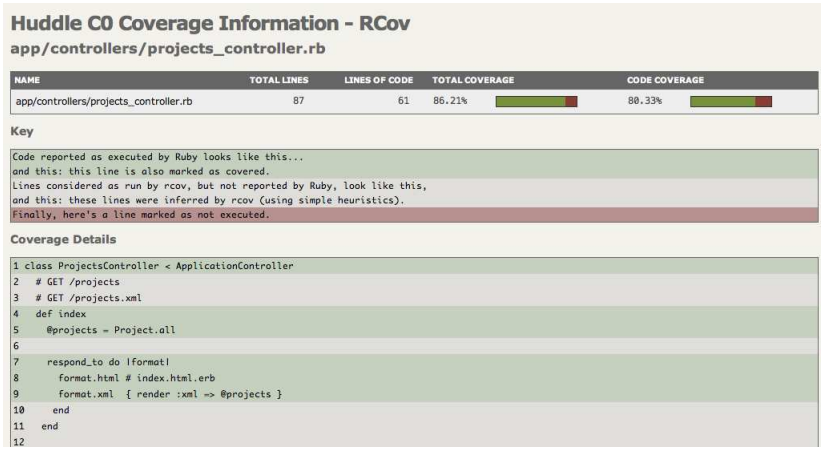


Figure 17.2: Rcov File Page

Clicking on any of the filenames takes you to an individual report for that file, as shown Figure 17.2.

By default, the individual report shows various shades of green for covered code (a different green for non-executable lines that have been inferred by Rcov), and various shades of red for uncovered lines—these are the ones that might need more testing.

17.5 Command-Line Rcov

You can run Rcov directy from the command line without Rake. By way of comparison, the system command generated by the default Rake Rcov task looks like this:

```
$ rcov -o "/Users/noel/Projects/tasker/coverage/test" -T
-x "rubygems/*,rcov*" --rails  -I"lib:test"
"/Library/Ruby/Gems/1.8/gems/rake-0.8.7/lib/rake/rake_test_loader.rb"
test_file_1.rb test_file_2.rb
```

The command is all on one line, and has a list of actual files at the end after the options. Although normal usage is for those executed files to be tests, they can be any Ruby script.

An important note: any of Rcov's command-line options can be passed to the Rake task by including them as a string in an RCOVOPTS command line variable, as in:

```
$ rake rcov RCOVOPTS="--only-uncovered --sort coverage"
```

Here's a brief guide to the most useful Rcov command-line options. In the next section, I'll show you how to roll your own Rake tasks to make the Rcov report more useful. As with most Unix command-line utilities, many of these options have a single-dash shortcut and a double-dash full name. This is not a complete list.

--aggregate FILE
: Aggregate the current data with existing data from a previous run that is in the associated Rcov data file.

--annotate -a
: Generate annotated source code.

--comments
: Mark all comments as covered (the default is the opposite, --no-comments).

--exclude PATTERNS --x
: Do not generate coverage info for files matching the list of comma-separated regular expression patterns.

--html
: Generate HTML output. The opposite is --no-html.

--include DIR:DIR -I DIR:DIR
: A colon-separated list of paths to be included in the load path.

--include-file PATTERNS -i PATTERNS
: Only generate coverage info for files matching the given set of comma-separate regular expression patterns. The inverse of --exclude.

--no-color -n
: Generates output that is colorblind-safe.

--only-uncovered
: A synonym for threshold 100.

--output DIR -o DIR
: The directory to which the HTML output reports will be saved.

--rails
: Equivalent to -x config/,environment/,vendor/.

--save FILE
: Saves raw coverage info to a file. Use with --text-coverage-diff.

--sort OPTION
: Sorts files in the output by the option. The option is name, loc, or coverage.

--sort-reverse
: Reverses the list of sorted files.

--spec-only
: Only count code that is covered from within an RSpec spec. In normal RSpec usage, this is redundant.

--text-coverage-diff FILE -D FILE
: Compares this coverage run with a previous run that was saved using --save.

--test-unit-only
: Only count code is covered from within a Test::Unit test. In normal Rails usage, this is redundant.

--text-report -T
: Place a detailed report in standard output (default).

--text-summary -t
: Place a summary in standard output.

–threshold VALUE
: Only outputs data for files with coverage below the value, as in --threshold 95.

--xrefs
: Generates a cross-referenced report showing where methods are called from.

If you need to pass options to the program being run by Rcov, not just Rcov itself, you do that by placing those options after a double-dash in the output, as in:

```
$ rcov --only-uncovered script.rb -- --opt-to-script
```

17.6 RCov and RSpec and Cucumber

RSpec and Cucumber use a similar mechanism for integrating with RCov. In both cases, the recommended way to run coverage of your Cucumber feature set is by also by creating a new Cucumber Rake task in the rake file created by the tool. For RSpec, that's lib/tasks/rspec.rake; for Cucumber, it's lib/tasks/cucumber.rake. (In RSpec 1.3.x, a spec:rcov

task is predefined; however, this does not appear to be the case in the RSpec 2.0 beta as I write this. RSpec 2.0 provides a simple mechanism to add Rcov support to any Rake task you define. The example below is for RSpec 2.0 beta.) The key is to set a task variable rcov to true inside the task you create. In RSpec, that might look like this:

```
RSpec::Core::RakeTask.new(:rcov => "db:test:prepare") do |t|
  t.pattern = "./spec/**/*_spec.rb"
  t.rcov = true
end
```

In Cucumber, the task looks like this one, which uses other options from the default Cucumber rake:ok task:

```
Cucumber::Rake::Task.new({:coverage => 'db:test:prepare'}) do |t|
  t.binary = vendored_cucumber_bin
  t.fork = true
  t.rcov = true
  t.rcov_opts = %w(--rails --exclude \/Library\/Ruby\/)
end
```

The rcov_opts variable, which works in both RSpec and Cucumber, takes command-line options that would be sent to Rcov. The options are expected to be in an array. If no options are set, the Cucumber default is %w{--rails --exclude osx\/objc,gems\/}. To my knowledge, RSpec does not have a default. If those options are good for you, you can just append to the list using something like t.rcov_opts << %w(aggregate data\/coverage.data). In RSpec 1.3.x, Rcov options can be placed in spec/rcov.opts, one option to a line, similar to the way that spec.opts works.

17.7 Rcov Tricks

Personally, I find the default Rcov tasks kind of inflexible and irritating, which is why I wrote my own. The main goal of this was to provide two different views of my test coverage, the first being just the model and helper tests against the model and helper code only, and the second aggregating that report with the result of the controller test against the entire app directory. I also wanted some better control over the output.

The simpler part of this is just a Rake task to delete the last round of data, to be invoked as rake test:coverage:clean. I'm hoping you are basically familiar with writing Rake tasks, because explaining all of Rake is way out of scope. (See http://rake.rubyforge.org for full Rake docs, or http://jasonseifer.com/2010/04/06/rake-tutorial for a good tutorial by Jason Seifer.)

```
namespace :test do
  namespace :coverage do
    desc "Delete aggregate coverage data."
    task :clean do
      rm_rf "tmp/functional"
      rm_rf "tmp/unit"
      rm_f "tmp/coverage.data"
    end
  end
end
```

That done, it's time to define the actual rake test:coverage task. What I'm doing here is defining a blank task that is dependent on the cleanup task, then defining two tasks, one for each coverage run. Each of these tasks is dynamically made an antecedent of the main rake test:coverage task. I'm helped greatly here by the Rcov::RcovTask class defined by Rcov. Here's the code (this is not the only or even necessarily the best way to write this task):

```
namespace :test
  task :coverage => "test:coverage:clean"
  %w(unit functional).each do |target|
    namespace :coverage do
      Rcov::RcovTask.new("cov_#{target}") do |t|
        t.libs << "test"
        t.test_files = FileList["test/#{target}/**/*_test.rb"]
        t.verbose = true
        t.rcov_opts << '--rails --aggregate data/coverage.data'
        t.rcov_opts << '--exclude app/controllers/' if target == "unit"
        t.rcov_opts << '--exclude db/'
        t.rcov_opts << '--exclude lib/authenticated'
        t.rcov_opts << '--exclude vendor/plugins'
        t.rcov_opts << '--exclude /Library/Ruby/'
        t.output_dir = "data/#{target}"
      end
    end
    task :coverage => "test:coverage:cov_#{target}"
  end
end
```

The Rcov::RcovTask.new() method takes an initializing block; in that block, we set up any options to the task that we want. The class defines a number of attributes to help. Just in this example, you can see libs, which contains directories to be added to the load path, test_files, which takes the list of test files to be run, verbose, which specifies output verbosity, output_dir, which is the directory where Rcov writes to, and then rcov_opts, which takes any other arguments you want.

Note that the two tasks differ only in their test directories and in that the unit test task excludes the controller directory. Rcov options being used include the --aggregate option, which allows the data from the unit test to be rolled into the controller run to create a combined coverage report the second time through.

This is what I use to measure coverage on my projects. Season to taste, and try it on your own.

17.8 How Much Coverage Is Enough

How valuable is code coverage? The inimitable Jay Fields, who has probably forgotten more about testing than I ever knew, compares code coverage to money—in that having a lot of it doesn't necessarily mean you are happy, but having none pretty much sucks.

It's not hard for a Rails project to run at 100% coverage, but there's having coverage for the purpose of getting the magic 100 number, and there's having coverage as a result of an actual test-driven development project, plus some occasional corner filling. The magic number on its own is probably useless; however, going through the process is quite valuable.

If you are executing the process correctly, your coverage should be 100% or pretty close as a natural side effect. If you find yourself writing an awful lot of tests after the fact just to get the number up, it's a hint that you could stand to improve your TDD process.

Chapter 18

Beyond Coverage: What Makes Good Tests?

One of the great things about the Ruby and Rails community is the extent to which they have accepted the idea that Test-Driven Design is a good process for developing software. That's a huge accomplishment, and a genuine difference between the Rails community and other programming groups that I have been a part of. We've spent a lot of time in this book on the mechanics of Ruby on Rails testing, how the tools work, how to get them running, and how to manage the basic TDD process. In other words, we've been talking about the craft of testing. Now, I'd like to talk about the art of testing: what makes a good test, how to balance testing priorities, how to troubleshoot, and in the next chapter, how to test when you already have a pile of bad code.

I'd like to see the discussion of testing in the Ruby community move to the next step, which is how to improve the quality of the tests you write. I see two versions of debate about testing in the community, both of which are interesting at times, but neither of which is the discussion I want to have at the moment.

- *TDD/BDD naming debate*. I've been testing for long enough that I still need to catch myself from calling it *Test-First Programming*. So I don't have a lot of patience when I refer to TDD, only to be greeted with an eye-roll and "Don't you mean BDD?" response. We probably all have better things to do then argue over Scrabble tiles.
- *Tools you should never use*. There have been a lot of posts over the last few months—some insightful, some less so—on the general

theme of "Why You Should [Never/Always] Use Cucumber." Tool choice is a useful discussion, but almost any of the popular tools in Rails can be used effectively, so this debate doesn't necessarily give much guidance as to how to test well.

There's a more fundamental debate lurking behind both of these discussions: a debate over what makes a test or a suite of tests effective and useful over the life of your application. To date, there's really only one commonly accepted objective metric of test quality—coverage. As we saw in Chapter 17, *Using Rcov to Measure Test Coverage*, on page 217, though, coverage is a flawed measure of test quality.

For the purposes of this discussion, we'll use a more subjective metric: that a good test saves time and effort over the long term, while a poor test costs time and effort. Using my own test experiences, I've focused on five qualities that tend to make a test successful by this metric. The absence of these qualities, on the other hand, is often a sign that the test could be a problem in the future.

18.1 The Five Habits of Highly Successful Tests

The best, if most general piece of advice I can give about the style and structure of automated tests is this: *remember your tests are also code*. Also, remember your tests are code that don't have tests. Your code is verified by your tests, but your tests are verified by nothing. So having your tests be as clear and manageable as possible is the only way to keep your tests honest and keep them going.[1]

A successful test has the following five features:

- Independence
- Repeatability
- Clarity
- Conciseness
- Robustness

1. That said, in practice I'm slightly more willing to allow duplication in tests in the name of readability. But only slightly.

Independence

A test is *independent* if the test does not depend on any external tests or data to run. An independent test suite gives the same results no matter what order the tests are run, and also tends to limit the scope of test failures to only those tests that cover the buggy method. In contrast, a change in one part of an application with a very dependent test suite could trigger failures throughout your tests. A clear sign that your tests are not independent is if you have test failures that only happen when the test suite is run in a particular order—in fully independent tests, the order in which they are run should not matter.

The biggest impediment to independence in the test suite itself is the use of *global data*. If the application is poorly designed, it may be difficult or impossible to make the tests fully independent of one another, but that's not exactly our lookout at the moment. Rails fixtures are not the only possible cause of global data in a Rails test suite, but they are a really common cause. Somewhat less common in a Rails context is using a tool or third-party library in a setup and not tearing it down. For example, the Flexmock mock object tool needs to be explicitly torn down between tests, as does the Timecop time-freezing gem.

Other than the use of fixtures, most Rails developers know to steer clear of global data in general, not just in a test suite, and for the same reason—code that has strange, hard-to-trace dependencies. One reason factory tools are preferable to fixtures is that they result in tests that have better independence.

Repeatability

A test is *repeatable* if running the same test multiple times gives the same result. Which is to say, a test is repeatable if running the same test multiple times gives the same result.[2] The hallmark of a test suite that lacks repeatability is intermittent test failure.

Two classic causes of repeatability problems are time and date testing and random numbers. In both cases, the issue is that your test data changes from test to test. The date and time have a nasty habit of continuing to get higher, while random data tends to stubbornly insist on being random.

2. Sorry, couldn't resist. If it's any consolation, the joke also didn't get a laugh when I did it as part of an actual talk.

The problems with the two types of data are slightly different. Dates and times tend to lead to intermittent failures when certain magic time boundaries are crossed. Random numbers, in contrast, make it somewhat difficult to test both the randomness of the number and that the randomly generated number is used properly in whatever calculation requires it.

The basic order of attack is similar for both cases, and applies to any constantly changing dataset. The goal is a combination of encapsulation and mocking. Encapsulation generally involves creating a service object that wraps the changing functionality. By wrapping the functionality, you make it easier to stub or mock the output values, providing the consistency you need for testing. You might, for example, create a RandomService class that wraps Ruby's rand() method and, critically, provides a way for you to preset a stream of output values either by using an existing mock package or by giving the service object a way to use a predefined value stream. Once you have verified that the random service class is random with its own unit tests, the service class can be stubbed in any other test to provide oxymoronic consistent random behavior.

The exact mix of encapsulation and mocking varies. Timecop, for example, stubs the time and date classes with no encapsulation. That said, nearly every time I talk about Timecop in a public forum, some audience member is sure to point out that creating a time service is a superior solution.

Clarity

A test is *clear* if its purpose is immediately understandable. Clarity in testing has two components. The first is the standard sort of readability that applies to tests as it applies to any code. The second is the kind of clarity that describes how the test fits into the larger test suite. Every test should have a point, meaning it should test something different from the other tests, and that purpose should be easy to discern from reading the test.

Fixtures are a test-specific issue that can lead to poor clarity; specifically, the way fixtures tend to create to "magic" results. To wit:

```
test "the sum should be 37" do
  assert_equal(37, User.all_total_points)
end
```

Where does the 37 come from? Well, if you were to peek into the user fixture file of this fake example, you'd see that somehow the totals of the total points of all the users in that file add to 37. The test passes. Yay?

The two most relevant problems for the current discussion are the magic literal, 37, which comes from nowhere, and the fact that the name of the test is utterly opaque about whether this is a test for the main-line case, a test for a common error condition, or a test that only exists because the coder was bored and thought it would be fun. Combine these problems, and it quickly becomes next to impossible to fix the test a few months later when a change to the User class or the fixture file breaks it.

Naming obviously helps with the latter problem. Factory tools have their place solving clarity issues, as well. Since the defaults for a factory definition are pre-set, the definition of an object created in the test can be limited to only the attributes that are actually important to test behaving as expected. Showing those attributes in the test is an important clue toward the programmer intent. Rewriting the test with a little more clarity might result in this:

```
test "total points should round to the nearest integer" do
  User.make(:points => 32.1)
  User.make(:points => 5.3)
  assert_equal(37, User.all_total_points)
end
```

It's not poetry, but at the very least, an interested reader now knows where that pesky 37 comes from, and where the test fits in the grand scheme of things. The reader might then have a better chance of fixing the test if something breaks. The test is also more independent, since it no longer relies on global fixtures—making it less likely to break.

We'll talk more about long tests in the next section, but as far as clarity goes, long tests tend to muddy the water and make it hard to identify the critical parts of the test. Basically, the guideline is that tests are code, and for the most part, the same principles that would guide refactoring and cleaning up code apply. This is especially true of the rule that states, "A method should only do one thing," which here means splitting up test setups into semantically meaningful parts, as well as keeping each test focused on on particular goal.

On the other hand, if you can't write clean tests, consider the possibility that it is the code's fault, and you need to do some redesign. If it's

hard to set up a clean test, that often indicates the code has too many internal dependencies.

There's an old programming aphorism that since debugging is more complicated than coding, if you've written code that is as complicated as you can make it, then you are by definition not skilled enough to debug it. Because tests don't have their own tests and need to be correct, this aphorism suggests that you should keep your tests simple, so as to give yourself some cognitive room to understand them. In particular, this guideline argues against clever tricks to reduce duplication among multiple tests that share a similar structure. If you find yourself starting to meta-program to generate multiple tests in your suite, you're probably going to find that complexity working against you at some point. You never want to be in a position to have to decide whether a bug is in your test or in the code. Well, you'll be in that position at some point, but it's an easier place to be if the test side is relatively simple.

Conciseness

A test is *concise* if it uses the minimum amount of code and objects to achieve its goal. Concise and clear are sometimes in conflict, as in the above example, where the clear version is a couple of lines longer than the original version. Most of the time, I'd say clear beats concise—we're not playing code golf here. Conciseness is useful only to the extent that it makes writing and maintaining the test suite easier.

Conciseness often involves writing the minimal amount of tests or creating the minimal amount of objects to test a feature. In addition to being clearer, concise tests will run faster, which is a big deal when you are running your test suite dozens of times a day. A slow test suite is a pain in the neck in all kinds of ways, obvious and subtle, and one of the best ways to prevent a slow suite is not to write slow tests.

To put this another way... how many objects do you need to create to test a sort? A simple sort can be tested with two objects, though I often use three because the difference between the initial input and the sorted input is easier to see in the test. (As an aside, if you are testing a sort, be sure to declare the items in a different order than the eventual sort; otherwise, it's hard to trigger a failure from the test.) Creating any more objects is unnecessary and makes the test slower to write and run.

To look at the issue of conciseness in another way, let's say you have a feature in which a user is given a different title based on some kind of point count; a user with less than 500 points is a novice, 501-1000 is an apprentice, 1001-2000 is a journeyman, 2001-5000 is a guru, and 5001 and up is a super Jedi rock star. How many assertions do you need to test that functionality?

In this case, there's a legitimate possibility of difference of opinion. I'd test the following cases—setup obviously is being handwaved here, and in practice I'd probably do separate single assertion tests. Also, in practice I'd be writing code after each new assertion.

```
def assert_user_level(points, level)
  User.make(:points => points)
  assert_equal(level, user.level)
end

def test_user_point_level
  assert_user_level(1, "novice")
  assert_user_level(501, "apprentice")
  assert_user_level(1001, "journeyman")
  assert_user_level(2001, "guru")
  assert_user_level(5001, "super jedi rock star")
  assert_user_level(0, "novice")
  assert_user_level(500, "novice")
  assert_user_level(nil, "novice")
end
```

That works out to one assertion for the start of each new level, plus an assertion for the special cases 0 and nil, and an assertion at the end of a level to assure that I don't have an off by one bug. That's a total of eight assertions. Given the way this code would probably be implemented, as a case statement with the while clauses using ranges, I don't feel the need to test the end condition of more than one field, nor do I feel the need to test every point value in a range. (Don't laugh, I've seen tests that would have effectively looped over every integer in the range and tested all of them. Unsurprisingly, that was on a Java project.)

Robustness

A test is *robust* if it actually tests the logic as intended. That is, the test passes when the underlying code is correct, and fails when the underlying code is wrong. Seems simple enough, but we've already seen cases in this book of tests that miss the mark.

A frequent cause of brittle tests is targeting the assertions of the test at surface features that might change even if the underlying logic stays

the same. The classic example along these lines is view testing, in which you base the assertion on the actual creative text on the page that will frequently change, even though the basic logic stays the same. Like so:

```
test "the view should show the project section" do
  get :dashboard
  assert_select("h2", :text => "My Projects")
end
```

It seems a perfectly valid test (or, if you are using Cucumber for integration testing, a perfectly valid Cucumber step definition)—right up until somebody decides that "My Projects" is a lame header and decides to go with "My Happy Fun-Time Projects." And breaks your test. You are often better served by testing something that slightly insulated from surface changes, like a DOM ID.

```
test "the view should show the project section" do
  get :dashboard
  assert_select("h2#projects")
end
```

The basic issue here is not limited to view testing. There are areas of model testing in which testing to a surface feature might be brittle in the face of trivial changes to the model. (As opposed to tests that are brittle in the face of changes to the test data itself, which we've already discussed.) For example, the test in the last section with the novice levels is actually dependent on the specific values of the level boundaries. You might want to make the test more robust with something like this:

```
def assert_user_level(points, level)
  User.make(:points => points)
  assert_equal(level, user.level)
end

def test_user_point_level
  assert_user_level(User::NOVICE_BOUND + 1, "novice")
  assert_user_level(User::APPRENTICE_BOUND + 1, "apprentice")
  # And so on...
end
```

You must be cautious at this point, because the other side of robustness is not just a test that brittlely fails when the logic is good, but a test that stubbornly passes even if the underlying code is bad—a tautology, in other words. The above test isn't a tautology, but you can see how it might easily get there.

Speaking of tautologies, mock objects have their own special robustness issues. As discussed in Chapter 7, *Using Mock Objects*, on page 105,

it's easy to create a tautology by using a mock object. It's also easy to create a brittle test by virtue of the fact that a mock object often creates a hard expectation of exactly what methods will be called on the mock object. If you add an unexpected method call to the code being tested, then you can get mock object failures simply because an unexpected method has been called. I've had changes to a login filter cause literally hundreds of test failures because mock users going through the login filter bounced off the new call. One workaround, depending on your mock package, is to use something like Mocha's mock_everything() method, which automatically returns nil for any unexpected method call without triggering an error.

18.2 Troubleshooting

Dot, dot, dot, dot, dot—tests are passing, looks like it's time for lunch—dot, dot, dot, dot, F. F? F? But the code works. I know it does. I think it does. Why is my test failing?

One of the most frustrating moments in the life of a TDD developer is when a test is failing and it's not clear why, as opposed to the more normal case in which the test fails as expected. Here's a grab bag of tips, tricks, hints, and thoughts to get us all through that difficult time.

Look for What Changed

This may be the most obvious piece of advice in the history of ever, but it's worth repeating, mantra-like, when confronted with a bad bug:

When a formerly-passing test fails, it means something changed.

It may be in the code, or the system, or the test. But it's probably not sunspots, and it's probably not evil spirits possessing your MacBook.[3]

Looking through recent changes can help you figure out what the cause of the failure is. Git's bisect tool does this automatically; or you can just look through recent changes in your source control viewer of choice. If the test was passing at one time, there's a good chance the answer is in there somewhere.

This is a great argument in favor of committing to your source control very, very frequently (especially when you are using git and can do local commits), so that your changes are very granular.

3. Unless you are living in a Charles Stross novel. Or programming in Perl.

Remember, a change to a fixture file can cause test failures all over the place—removing a model without deleting the fixture file will create errors in every test. Also, if you have a database migration, rake will automatically load the new structure to the test database, but autotest does not do so by default.

Isolate the Failure

When looking at a small number of failing tests, it's helpful to be able to run just those tests. Autotest is outstanding for this, since it runs the failing tests over and over until they pass. Cucumber's default behavior also re-runs failing tests. This is especially helpful if you have a number of failing tests that are not in the same test class.

The little code and terminal snippet at http://gist.github.com/101130 is very helpful for quickly running one class at a time, which is almost like isolating a failing test, or at least close enough to be useful. Depending on your IDE and test framework of choice, you may also be able to run individual tests from the IDE.

Isolating tests makes the tests run faster when you are focused on just a few tests, and also makes any diagnostics you insert easier to interpret.

Here are two valuable tips that I've learned from listening to and reading Kent Beck:

Back out your entire most recent change since your last passing test and start over. This works best if you work in very small increments, but it gets you out of the "I know I typed something wrong but I just can't see it" nightmare.

Replace all the expressions in the method under test with literals—if that passes, put the expressions back one by one until you find the culprit.

Diagnostics

I'm not a big fan of using stop-and-step debuggers, graphical or command line. I've used them when I've been in an IDE, but I've never really used the Rails command-line debugger. I haven't found that to be a great experience compared to having tests in which I can make assertions.

Normally, to diagnose what's going on in a test, I either add additional assertions in the test or have the code print information to the console

or log. If I diagnose via assertions, generally I'm able to test the values of variables in more detail.

For some reason, I see a lot of people using Ruby's puts() method to write to the console—I recommend p(), which calls inspect() on the object before printing, and generally results in more informative output. As a matter of course, I put require pp in my test_helper.rb file, which allows me to use pp() to get pretty-printed output, which is nice for nested data structures. Also, y() gives a YAML representation of the output—very readable for ActiveRecord objects. The gem awesome_print provides the ap method for an extremely readable output of complex data structures.

```
>> x = {1 => ['a', 'b'], 2 => 'c'}
>> puts x
1ab2c

>> p x
{1=>["a", "b"], 2=>"c"}

>> pp x
{1=>["a", "b"], 2=>"c"}

>> y x
---
1:
- a
- b
2: c

>> require 'ap'
=> true
>> ap x
{
    1 => [
        [0] "a",
        [1] "b"
    ],
    2 => "c"
}
```

I've been known to bury print statements all over the place—controllers, Rails itself (often educational). This is especially true if I have Autotest limiting the test-run to the one failing test. Just remember to take the print statements out when you are done. Adding additional assertions to the test can also act as a substitute for print statements.

If you need to get to the log, most frequently because you are running Passenger and trying to get output from the actual running app, you can write to the log from anywhere in your Rails app using Rails.logger.error(). You can substitute any of the other log levels for error, but why bother?

If you are using Cucumber or Webrat/Capybara, Webrat and Capybara have the save_and_open_page() method, which is outstandingly helpful. It takes the current Webrat DOM, including any changes you've made using Webrat form methods, saves the page to a temp file, and opens it in your default browser. The resulting page may be missing some images, and you can't really follow links, but you can see what page you got, and it's much easier to inspect the source in the browser using Firebug or the Safari 4.0/Chrome web inspector, then from print response.body.

Clear Your Head

Take a walk. Force your pair to solve the problem. Get a cup of coffee (actually, I hate coffee; get a Diet Coke). Take a nap. All these clear-your-head steps really do work. Try to explain the problem to somebody else—often, the act of explaining the issue helps identify something you overlooked.

It's tempting to comment-out the offending test; then your suite passes, and all seems well. That's generally a bad idea, although sometimes a major refactoring can genuinely make tests obsolete.

18.3 From Greenfield to Legacy

All of these style issues are helpful in planning and executing your tests. Like most testing advice, though, they have one thing in common—they are much easier to apply to a new application then to a pile of untested code that already exists. In the next chapter, we'll explore the special challenges that come from trying to add TDD to a project that has gone without for far too long.

Chapter 19

Testing a Legacy Application

Chapter 20

Test and Application Performance

Chapter 21

Using Autotest as a Test Runner

Part VII

Appendices

Appendix A

Sample Application Setup

Most of the test examples in this book are based off of a simple Rails Application called Huddle, which supports Agile teams by allowing them to enter their daily status scrums online. Although the sample code for the Huddle application is contained in the code samples for each chapter, I know that many people like to create the entire example from scratch.

A.1 Basic Rails

This appendix lists the steps I took to create a simple Rails application with Authlogic for user authentication. The installation instructions for other third-party plugins and gems are contained in the chapter where the extension is used.

You won't need anything for this setup other than Ruby, Rails, and SQLite3. Let's start with the basics.

```
% rails huddle
«many lines of response»
% cd huddle
% rake db:create:all
(in /Users/noel/Documents/pragprog/nrtest/Book/code/huddle)
```

I'm using the gem installation of Rails because I expect to have multiple copies of Huddle, and don't want to have multiple copies of Rails in my repository. That doesn't apply to you, though. If you are running a Rails 2.3.x application and if you want to have Rails in your vendor/rails directory, run the following command:

```
% rake rails:freeze:edge RELEASE=2.3.5
```

Alternately, you can download the Rails release from GitHub at http://github.com/rails/rails/tree/master and place it in vendor/rails.

In Rails 3.x, gem management, including that of the Rails gems, is handled by bundler. The Gemfile in your newly created application contains a command starting with gem 'rails' which ties your application to a specific Rails version. There is also a commented out command to use if you want the cutting edge version of Rails. We're not going to worry about the finer points of bundler here, see http://gembundler.com for more details.

A.2 Authlogic

Adding in Authlogic takes a few steps, I'm largely taking these from the Authlogic setup tutorial at http://github.com/binarylogic/authlogic_example/tree/master. The instructions for Rails 2.3.x are:

```
% sudo gem install authlogic
Successfully installed authlogic-2.1.1
1 gem installed
```

Then, add the following line to config/environment.rb:

Download huddle/config/environment.rb

```
config.gem "authlogic"
```

In Rails 3.x, gems are managed in bundler. You need to add the following line to your Gemfile to get the Authlogic gem included.

gem "authlogic", :git => "git://github.com/binarylogic/authlogic.git"

Now, we need to generate the Authlogic standard models and controllers. I like Authlogic a lot, but it does have a few quasi-boilerplate steps that need to be done to take the Authlogic skeleton and implement it in an application. I don't want to belabor this, because this book isn't *Rails Authlogic Prescriptions* and all the code is in the Authlogic tutorial I just referenced, but there is some code that needs to be added to the various files.

First up, the UserSession model. This class is specially generated by Authlogic and is fine as is.

```
% script/generate session user_session
      exists  app/models/
      create  app/models/user_session.rb
```

Next up, the User model. This uses a standard Rails model generator, so both the User model itself and the create_users migration will need some extra work.

```
% script/generate model user
      exists  app/models/
      exists  test/unit/
      exists  test/fixtures/
      create  app/models/user.rb
      create  test/unit/user_test.rb
      create  test/fixtures/users.yml
      create  db/migrate
      create  db/migrate/20090824191508_create_users.rb
```

Inside the migration, we'll put the full list of Authlogic columns, as recommended in the Authlogic tutorial. It's way overkill for this test app, but I'm not worried about that right now.

Download huddle/db/migrate/20090824191508_create_users.rb

```
class CreateUsers < ActiveRecord::Migration
  def self.up
    create_table :users do |t|
      t.timestamps
      t.string :login, :null => false
      t.string :crypted_password, :null => false
      t.string :password_salt, :null => false
      t.string :persistence_token, :null => false
      t.integer :login_count, :default => 0, :null => false
      t.datetime :last_request_at
      t.datetime :last_login_at
      t.datetime :current_login_at
      t.string :last_login_ip
      t.string :current_login_ip
    end

    add_index :users, :login
    add_index :users, :persistence_token
    add_index :users, :last_request_at
  end

  def self.down
    drop_table :users
  end
end
```

We also need to run the migration:

```
% rake db:migrate
(in /Users/noel/Documents/pragprog/nrtest/Book/code/huddle)
==  CreateUsers: migrating ====================================================
-- create_table(:users)
```

```
   -> 0.0215s
-- add_index(:users, :login)
   -> 0.0008s
-- add_index(:users, :persistence_token)
   -> 0.0005s
-- add_index(:users, :last_request_at)
   -> 0.0004s
==  CreateUsers: migrated (0.0242s) ============================================
```

All the User class itself needs as a call to the acts_as_authentic() method.

Download huddle/app/models/user.rb

```
class User < ActiveRecord::Base
  acts_as_authentic
end
```

Next up is the UserSession controller. This one needs a fair bit of code after the Rails generator runs. In a production app, we'd obviously be tweaking this code quite a lot. For our present needs, the tutorial defaults are fine.

```
% script/generate controller user_sessions
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/user_sessions
      exists  test/functional/
      create  test/unit/helpers/
      create  app/controllers/user_sessions_controller.rb
      create  test/functional/user_sessions_controller_test.rb
      create  app/helpers/user_sessions_helper.rb
      create  test/unit/helpers/user_sessions_helper_test.rb
```

The app/controllers/user_sessions_controller.rb looks like this – this is a slight edit from the boilerplate to adjust for security issues in the tutorial:

Download huddle/app/controllers/user_sessions_controller.rb

```
class UserSessionsController < ApplicationController
  skip_before_filter :require_user, :only => [:new, :create]

  def new
    @user_session = UserSession.new
  end

  def create
    @user_session = UserSession.new(params[:user_session])
    if @user_session.save
      flash[:notice] = "Login successful!"
      redirect_back_or_default projects_url
    else
      render :action => :new
    end
```

```
  end

  def destroy
    current_user_session.destroy
    flash[:notice] = "Logout successful!"
    redirect_back_or_default new_user_session_url
  end
end
```

Note that I've slid in a reference to projects_path here – that won't actually work until a page or so in the future, when the project resource is created.

And exactly one of those controller actions needs a view, which goes in app/views/user_sessions/new.html.erb. Again, it's pretty basic.

Download huddle/app/views/user_sessions/new.html.erb

```
<h1>Login</h1>

<% form_for @user_session, :url => user_session_path do |f| %>
  <%= f.error_messages %>
  <%= f.label :login %><br />
  <%= f.text_field :login %><br />
  <br />
  <%= f.label :password %><br />
  <%= f.password_field :password %><br />
  <br />
  <%= f.check_box :remember_me %><%= f.label :remember_me %><br />
  <br />
  <%= f.submit "Login" %>
<% end %>
```

Finally, there's a change to the route file in config/routes.rb.

Download huddle/config/routes.rb

```
map.resource :user_session
map.root :controller => "user_sessions", :action => "new"
```

There's actually some code that will go into the ApplicationController, which is used for very basic authentication control.

Download huddle/app/controllers/application_controller.rb

```
helper_method :current_user_session, :current_user
private

def current_user_session
  return @current_user_session if defined?(@current_user_session)
  @current_user_session = UserSession.find
end
```

```
def current_user
  return @current_user if defined?(@current_user)
  @current_user = current_user_session && current_user_session.record
end

def require_user
  unless current_user
    store_location
    flash[:notice] = "You must be logged in to access this page"
    redirect_to new_user_session_url
    return false
  end
end

def require_no_user
  if current_user
    store_location
    flash[:notice] = "You must be logged out to access this page"
    redirect_to account_url
    return false
  end
end

def store_location
  session[:return_to] = request.request_uri
end

def redirect_back_or_default(default)
  redirect_to(session[:return_to] || default)
  session[:return_to] = nil
end
```

That completes the steps in the Authlogic tutorial to get the basic application running. And if you think I put the steps here partially so that I'll always know where I can look them up, well, you've got me pegged.

A.3 Huddle's Data Models

With the Authlogic boilerplate done, we just need to create the basic data structures used for Huddle. We already have a user model and user session controller generated by Authlogic. We'll need a model for the status reports and one for the project. Right now, a Project is just a string:

```
% script/generate scaffold project name:string
```

A StatusReport has references to Project and User, plus text fields for the "what I did yesterday" and "what I'm going to do today" reports. I've also added an explicit date for the status, separate from the created_at field.

```
% script/generate scaffold status_report project:references  \
  user:references yesterday:text today:text status_date:date
```

Projects and users have a many-to-many relationship, so you'll need a join table for that:

```
% script/generate migration project_user_join
```

The migration looks like this:

Download huddle/db/migrate/20090825045451_project_user_join.rb

```
class ProjectUserJoin < ActiveRecord::Migration
  def self.up
    create_table :projects_users, :force => true, :id => false do |t|
      t.references :project
      t.references :user
      t.timestamps
    end
  end

  def self.down
    drop_table :projects_users
  end
end
```

Then run rake db:migrate one more time.

A.4 First Tests

That's a long way to just to get started. Sorry, it's the kind of set up you have to get out of the way to do any kind of example complex enough to be useful.

It's worth pointing out, though, that we already have some tests. Running the default rake testing task gives output like this. (Output slightly truncated):

```
% rake
«ruby command»
Started
...
Finished in 0.023659 seconds.

3 tests, 3 assertions, 0 failures, 0 errors

«ruby command»
Started
..................
Finished in 0.354562 seconds.
```

```
17 tests, 26 assertions, 0 failures, 0 errors
```

The first batch is the unit tests – Rails creates a dummy test for each model. The bottom tests are the functional tests. Rails creates a basic suite for each controller scaffold that covers the successful cases of each scaffold method. (The failure cases can easily be covered with mock objects). The sample Authlogic tutorial does provide tests for the UserSessionsController, which I have quietly inserted into the test build.

Congratulations, the setup is out of the way, you can now rejoin the rest of the book, already in progress.

Appendix B

Bibliography

[Mes07] Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, Reading, MA, 2007.

Index

More Books go here...

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)