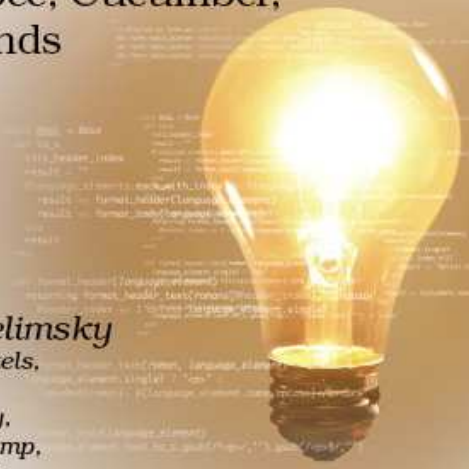
The Pragmatic Programmers

The RSpec Book

Behaviour Driven Development with RSpec, Cucumber, and Friends



David Chelimsky
with Dave Astels, Zach Dennis, Aslak Hellesøy, Bryan Helmkamp, and Dan North

Edited by Jacquelyn Carter

The Facets of Ruby Series

The book you're reading is still under development. As part of our Beta book program, we're releasing this copy well before we normally would. That way you'll be 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!

Throughout this process you'll be able to download updated PDFs 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. In the meantime, we'd appreciate you sending us your feedback on this book at http://pragprog.com/titles/achbd/errata, or by using the links at the bottom of each page.

Thank you for being part of the Pragmatic community!

► **Andy & Dave**

The RSpec Book

Behaviour Driven Development with RSpec, Cucumber, and Friends

David Chelimsky
Dave Astels
Zach Dennis
Aslak Hellesøy
Bryan Helmkamp
Dan North

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: 1-934356-37-9

ISBN-13: 978-1-934356-37-1

Printed on acid-free paper.

B4.0 printing, April 13, 2009

Version: 2009-4-13

Contents

Important Information for Beta Readers

Welcome to The RSpec Beta Book!

RSpec, Cucumber, and Webrat are all under regular development with frequent releases. The fact that the maintainers of these libraries are also authors of this book means that *you* get to learn about the latest features. In fact, some of the features you'll learn about are so new, they have not even been released yet!

For installation instructions for all the library code you need, please see http://wiki.github.com/dchelimsky/rspec/code-for-the-rspec-book-beta.

Please report any problems you run into installing these gems to The RSpec Book Forum at http://forums.pragprog.com/forums/95, and any other sorts of errata to http://www.pragprog.com/titles/achbd/errata.

Thank you *so much* for participating in our beta program. The feedback we've already received has been invaluable, and is making this a better book for everbody.

Changes

Beta 4.0—April 13, 2009

This new release includes a number of improvements per suggestions submitted by readers,[1] as well as two exciting new chapters:

Rails Controllers

Continuing inward on our outside-in journey, this chapter explores how (and *when*) to write controller specs. We also introduce approaches to dealing with some controller-specific spec'ing challenges like filters, global behaviour defined in ApplicationController, and sending email. See Chapter 23, *Rails Controllers*, on page 269.

Extending RSpec

This chapter introduces techniques for extending RSpec to cater to domain-specific needs. Covered topics include custom example group classes, custom matchers (including an exciting new matcher definition DSL), macros and custom formatters. Whether customizing RSpec for your own app, or in order to ship domain-specific spec'ing extensions with the libraries you're releasing, this chapter is filled with really useful information that will help you make your specs easier to write *and* read. See Chapter 15, *Extending RSpec*, on page 177.

1. http://www.pragprog.com/titles/achbd/errata

Preface

Coming soon ....

Part I

Getting Started with RSpec and Cucumber

Chapter 1

Introduction

Behaviour Driven Development began its journey as an attempt to better understand and explain the process of Test Driven Development. Dan North had observed that developers he was coaching were having a tough time relating to TDD as a design tool and came to the conclusion that it had a lot to do with the word *test*.

Dave Astels took that to the next step in his seminal article, A New Look at Test Driven Development,[1] in which Dave suggested that even some experienced TDD'ers were not getting all the benefit from TDD that they could be getting.

To put this into perspective, perhaps a brief exploration of Test Driven Development is in order.

1.1 Test Driven Development: Where it All Started

Test Driven Development is a developer practice that involves writing tests before writing the code being tested. Begin by writing a very small test for code that does not yet exist. Run the test and, naturally, it fails. Now write just enough code to make that test pass. No more.

Once the test passes, observe the resulting design and refactor [2] to remove any duplication you see. It is natural at this point to judge the design as too simple to handle all of the responsibilities this code will have.

1. http://techblog.daveastels.com/2005/07/05/a-new-look-at-test-driven-development/
2. Refactoring: improving the design of code without changing its behaviour. From Martin Fowler's *Refactoring* [FBB+99]

Joe Asks...

But what if "the testers" is me?

Not all project teams have a separate tester role. On teams that don't, the notion of pushing off the responsibility of testing practices to other people doesn't really fly. In cases like this, it's still helpful to separate testing practices from TDD.

When you're *wearing your TDD hat,* focus on red/green/refactor, design and documentation. Don't think about testing. Once you've developed a body of code, put on your *tester hat,* and think about all the things that could go wrong. This is where you add all the crazy edge cases, using exploratory testing to weed out the nasty bugs hiding in the cracks, documenting them as you discover them with more code examples.

Instead of adding more code, document the next responsibility in the form of the next test. Run it, watch it fail, write just enough code to get it to pass, review the design and remove duplication. Now add the next test, watch it fail, get it to pass, refactor, etc, etc, etc.

Emergent Design

As the code base gradually increases in size, more and more attention is consumed by the refactoring step. The design is constantly evolving and under constant review, though it is not pre-determined. This process is known as *emergent design*, and is one of the most significant by-products of Test Driven Development.

This is not a testing practice at all. Instead, the goal of TDD is to deliver high quality code to testers, but it is the testers who are responsible for testing practices (see the *Joe Asks...* on this page).

And this is where the *Test* in TDD becomes a problem. Specifically, it is the idea of *Unit Testing* that often leads new TDD'ers to verifying things like making sure that a register() method stores a Registration in a Registry's registrations collection, and that collection is specifically an Array.

This sort of detail in a test creates a dependency in the test on the internal structure of the object being tested. This dependency means that if

other requirements guide us to changing the Array to a Hash, this test will fail, even though the behaviour of the object hasn't changed. This brittleness can make test suites much more expensive to maintain, and is the primary reason for test suites to become ignored and, ultimately, discarded.

So if testing internals of an object is counter-productive in the long run, what should we focus on when we write these tests first?

1.2 Behaviour Driven Development: The Next Step

The problem with testing an object's internal structure is that we're testing what an object *is* instead of what it *does*. What an object *does* is significantly more important.

Think of this at the application level. When is the last time you had a conversation with a business analyst who said "when a customer places an order, the order should be stored in an ANSI-compliant relational database"? More likely, he said something like "when a customer places an order, it should be stored in *the database*." And by *the database* he was using a generic metaphor for some sort of persistent storage mechanism.

Of course you may have a more technically savvy business analyst who actually understands the technical differences and implications of ANSI-compliance and relational databases vs object and document databases, etc. But he probably doesn't care about which one you choose as much as whether the person who processes orders can recall that data in order to do his job.

At the object level, the fact that a Registry uses an Array instead of a Hash or some other data structure to store registrations is not important. What is important is that you can ask a Registry to store a registration and you can retrieve that registration later. Whether we're specifying applications or objects, the real value lies in the *behaviour*, not the *structural details*.

1.3 RSpec

RSpec was created by Steven Baker in 2005, inspired by Dave's aforementioned article. One of Dave's suggestions was that with languages like Smalltalk and Ruby, we could more freely explore new frameworks that could encourage focus on behaviour.

While the syntactic details have evolved since Steven's original version of RSpec, the basic premise remains. We use RSpec to write executable examples of the expected behaviour of a small bit of code in a controlled context. Here's how that might look:

```
describe MovieList do
  context "when first created" do
    it "should be empty" do
      movie_list = MovieList.new
      movie_list.should be_empty
    end
  end
end
```

The it() method creates an *example* of the behaviour of a MovieList, with the *context* being that the MovieList was just created. The expression movie_list.should be_empty should be self-explanatory. Just read it out loud. You'll see how be_empty() interacts with the movie_list in Section 11.3, *Predicate Matchers*, on page 140.

Running this code in a shell with the spec command yields the following specification:

```
MovieList when first created
- should be empty
```

Add some more contexts and examples, and the resulting output looks even more like a specification for a MovieList object:

```
MovieList when first created
- should be empty

MovieList with 1 item
- should not be empty
- should include that item
```

Of course, we're talking about the specification of an object, not necessarily a whole system. You *could* specify application behaviour with RSpec. Many do. Ideally, however, for specifying application behaviour, we want something that communicates in broader strokes. And for that, we use Cucumber.

1.4 Cucumber

Cucumber is a BDD tool that reads plain text descriptions of application features with example scenarios, and uses the scenario steps to automate interaction with the code being developed. For example:

Cucumber Seeds

Even before we had started exploring structures and syntax for RSpec, Dan North had been exploring a completely different model for a BDD tool. He wanted to document and drive behaviour in a simplified language that could be easily understood by customers, developers, testers, business analysts, etc, etc. The early result of that exploration was the JBehave library, which is still in active use and development.

Dan ported JBehave to Ruby as RBehave, and we merged it into RSpec as the Story Runner. It only supported scenarios written in Ruby at first, but we later added support for plain text, opening up a whole new world of expressiveness and access. But as new possibilities were revealed, so were limitations.

In the spring of 2008, Aslak Hellesøy set out to rewrite RSpec's Story Runner with a real grammar defined with Nathan Sobo's Treetop library. Aslak dubbed it Cucumber at the suggestion of his fiancée, Patricia Carrier, thinking it would be a short-lived working title until it was merged back into RSpec. Little did either of them know that Cucumber would develop a life of its own.

```
Line 1  Feature: pay bill on-line
  -       In order to reduce the time I spend paying bills
  -       As a bank customer with a checking account
  -       I want to pay my bills on-line
    5
  -       Scenario: pay a bill
  -         Given checking account with $50
  -         And a payee named Acme
  -         And an Acme bill for $37
   10       When I pay the Acme bill
  -         Then I should have $13 remaining in my checking account
  -         And the payment of $37 to Acme should be listed in Recent Payments
```

Plain text scenarios like this one are parsed and treated as real code, providing invaluable benefits like backtraces that emanate from the plain text steps, and support for multiple languages.

Everything up to and including the Scenario declaration on line 6 is treated as documentation (not executable). The subsequent lines are steps in the scenario. In the next chapter, you'll be writing *step definitions* in Ruby. These step definitions interact with the code being developed, and are invoked by Cucumber as it reads in the scenario.

Don't worry if that doesn't make perfect sense to you just yet. For right now it's only important to understand that both RSpec and Cucumber allow us to specify the behaviour of code with examples that are programmatically tied to the system. The details will become clear as you read on.

1.5 The BDD Cycle

We typically use Cucumber to describe behaviour of the application from the outside and RSpec to describe the behaviour of its component parts.[3] If you've ever done TDD before, you're probably familiar with the red/green/refactor cycle. With the addition of a higher level tool like Cucumber, we'll actually have two concentric red/green/refactor cycles, as depicted in Figure 1.1, on the following page.

Both cycles involve taking small steps and listening to the feedback you get from the tools. We start with a failing step (red) in Cucumber (the outer cycle). To get that step to pass, we'll drop down to RSpec (the inner cycle) and drive out the underlying code at a granular level (red/green/refactor).

At each green point in the RSpec cycle, we'll check the Cucumber cycle. If it is still red, the resulting feedback should guide us to the next action in the RSpec cycle. If it is green, we can jump out to Cucumber, refactor if appropriate, and then repeat the cycle by writing a new failing Cucumber step.

This will all become clear as you read through these chapters.

In the tutorial that follows, we'll be using a number of features in Cucumber and RSpec. In most cases we'll only touch the surface of a feature, covering just enough to be able to use it as needed for this project, with references to other places in the book that you can go to learn more of the detail and philosophy behind each feature.

So now it's time to grab some coffee, clear your head, leave your preconceptions at the door and get ready to get your BDD on. See you in the next chapter, in which we'll begin to drive out a command line version of the classic logic game, Mastermind.

3. Although we use Cucumber to focus on high level behaviour and RSpec on more granular aspects of behaviour, each can be used for either purpose.

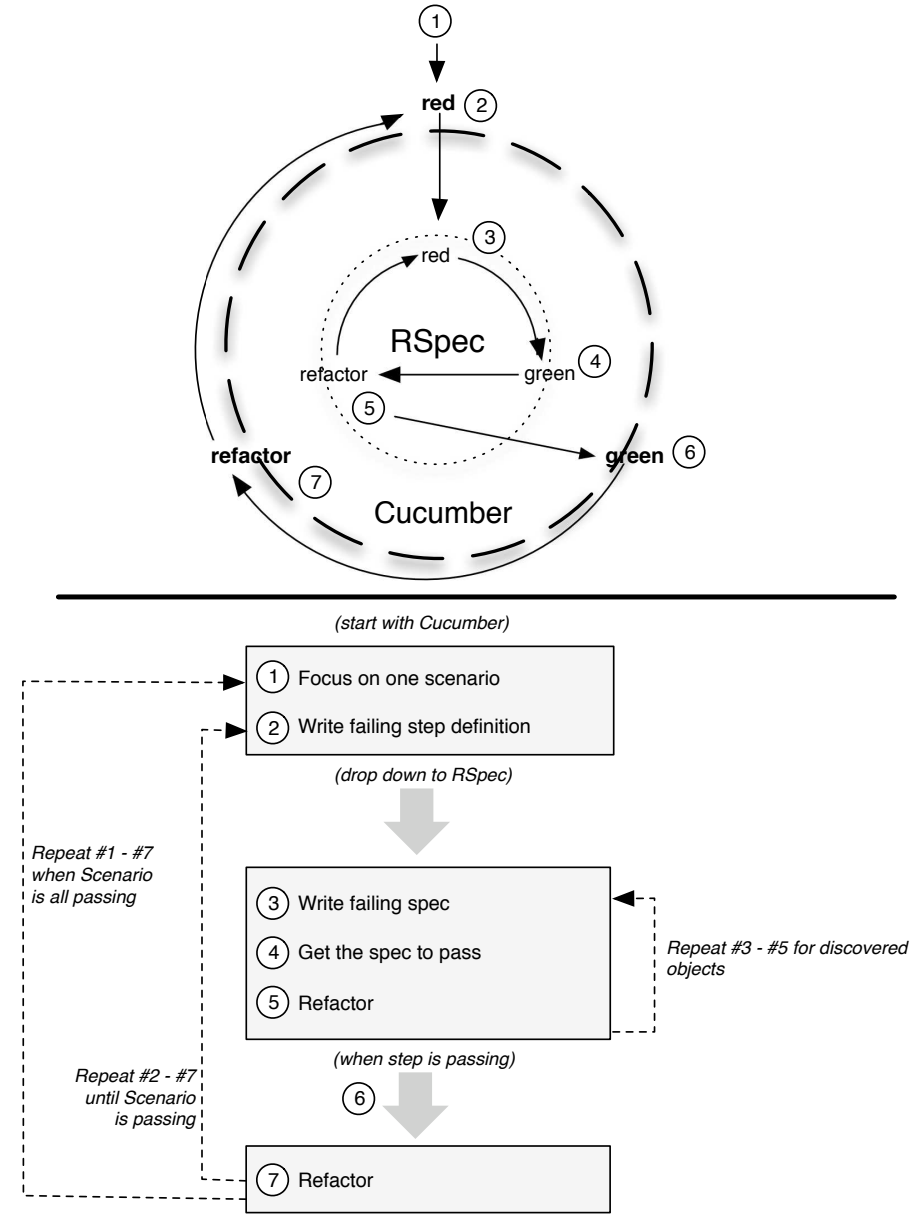


Figure 1.1: The BDD Cycle

Chapter 2

Describing Features with Cucumber

To get started with RSpec and Cucumber, we're going to write a simple command line version of the classic board game, Mastermind.

The game of Mastermind involves two players: the *code-maker* and the *code-breaker*. The job of the code-breaker is to deduce a secret code made up of four colored pegs chosen by the code-maker. The pegs come in six different colors: B=Black, C=Cyan, G=Green, R=Red, Y=Yellow, W=White.

The game is usually played on a board that looks like the one depicted in Figure 2.1, on the next page. The code-maker chooses a secret code and places pegs in the row on the left, which gets covered from view of the code-breaker. The code-breaker gets some number of chances (typically ten) to break the code. In each turn, the code-breaker takes a guess at the code, placing 4 of the colored pegs in a row. The code-maker then *marks* the guess using smaller black and white *marker* pegs.

A black marker indicates that one of the colored pegs in the guess is the right color and in the right position, but does not reveal which one. A white marker indicates that one of the pegs in the guess is of a color which is in the solution (again without revealing which one), but is in the wrong position. For example, if the score is 2 black pegs and 1 white, then we know that the guess has three colored pegs that are part of the code and two of them are actually in the right positions.

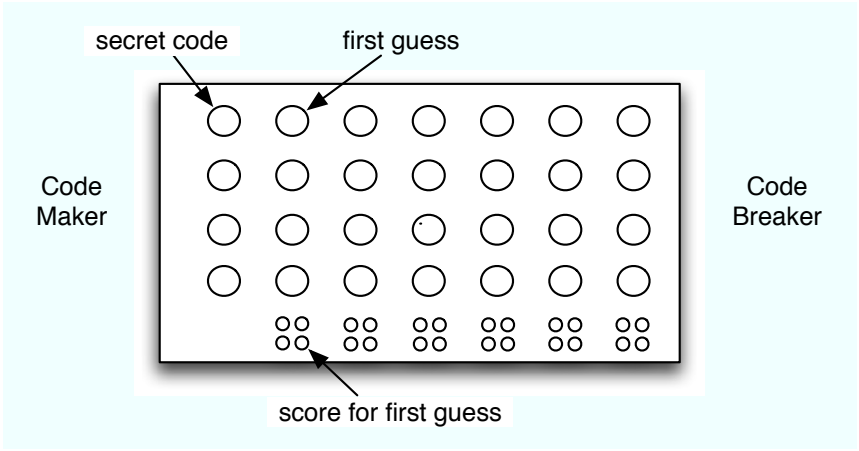


Figure 2.1: Mastermind

2.1 Selecting Stories for the First Iteration

We're going to develop the Mastermind game in short iterations using automated scenarios and code examples to drive out the code. To get started we'll need an initial set of User Stories from which to pick our first iteration. Here are some titles to get us started:

- Code-breaker starts game
- Code-breaker submits guess
- Code-breaker wins game
- Code-breaker loses game
- Code-breaker plays again

Note how each of these indicate the *code-breaker* role. We like to express stories in terms of a specific role (not just a generic *user*) because that impacts how we think about the requirement and why we're implementing code to satisfy it.

Let's start with the "Code-breaker starts game" and "Code-breaker submits guess" stories for the first iteration. We'll need a narrative for each story, and then some scenarios that we'll automate with Cucumber, and we're going to need a place to put them. Let's go ahead and get the project set up.

Focus on the Role

I heard Mike Cohn, author of *User Stories Applied* (Coh04), talk about focusing on the role when writing User Stories at the Agile 2006 Conference. The example he gave was that of an airline reservation system, pointing out that the regular business traveler booking a flight wants very different things from such a system than the occasional vacation traveler.

Think about that for a minute. Imagine yourself in these two different roles and the different sorts of details you would want from such a system based on your goals. For starters, the business traveler might want to maintain a profile of regular itineraries, while the vacationer might be more interested in finding package deals that include hotel and car at a discount.

Focusing on this distinction is a very powerful tool in getting down to the details of the features required of a system.

Figure 2.2, on the following page, shows the conventional layout with bin, features, lib, and spec directories at the root of the project. lib/mastermind.rb will be responsible for requiring the source files in the lib/mastermind directory.

features/support/env.rb and spec/spec_helper.rb will each be responsible for requiring lib/mastermind.rb, ensuring that the necessary source files are loaded when executing Cucumber scenarios and RSpec code examples. We'll talk about features/step_definitions after we've written out a couple of scenarios.

Following convention, we'll build a parallel structure below lib/mastermind and spec/mastermind. For example, in this chapter we'll describe the behaviour of Game in spec/mastermind/game_spec.rb and we'll put its class definition in lib/mastermind/game.rb. But before we get there, we need to write out some scenarios.

2.2 Deriving Features from Stories

If you're familiar with Cucumber's predecessor, RSpec's Story Runner (which, itself, succeeded RBehave), you may have seen scenarios organized by User Stories in Story files in a stories directory. We had some

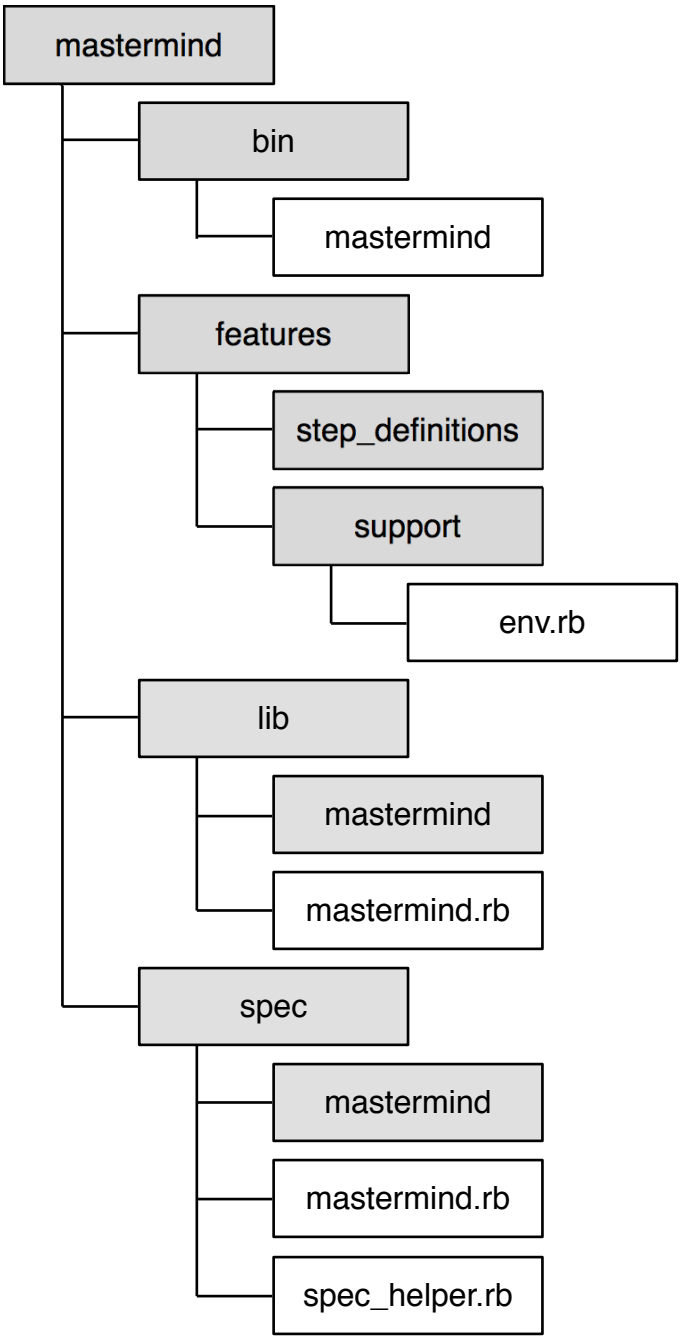


Figure 2.2: Project Structure

Joe Asks...

Shouldn't we avoid a 1-to-1 mapping?

Perhaps you've heard that a 1-to-1 mapping between objects and their specs is a BDD no-no. There is some truth to this, but the devil is in the details.

We want to avoid a strict adherence to a structure in which every object has a single example group, and every method has a single code example. That sort of structure leads to long examples that take an object through many phases, setting expectations at several stopping points in each example. Examples like these are difficult to write to begin with, and much more difficult to understand and debug later.

A 1-to-1 mapping of spec-file to subject-file, however, is not only perfectly fine, it is actually beneficial. It makes it easier to understand where to find the specs for code you might be looking at. It also makes it easier for tools to automate shortcuts like the one in The RSpec TextMate bundle, which switches between spec-file and subject-file with CTRL-SHIFT-DOWN.

debate about this within the BDD community and when Cucumber came around, it took the side of *Stories In, Features Out*.

The idea is that we use stories for planning and estimation, and we talk about which stories we're going to do in which iterations. But once we deliver working code it is more natural to talk about code in terms of features rather than stories. And since stories in later iterations can lead to enhancements of existing features, it's much easier to keep things organized by feature and just add new scenarios to the existing features.

Cucumber features have three parts to them: a title, a brief narrative, and an arbitrary number of scenarios which serve as our acceptance criteria.

Here's what the title and narrative for the "code-breaker starts game" feature might look like:

Download mm/01/features/codebreaker_starts_game.feature

```
Feature: code-breaker starts game
```

```
As a code-breaker
I want to start a game
So that I can break the code
```

The title is just enough to remind us who the feature is for, the code-breaker, and what the feature is about, starting a game. Although the narrative is free-form, we generally follow a variation of the Connextra format described in the (as yet) unwritten *sec.narrative*.

With that narrative, we have a slightly better understanding of what we want to do with the system, but how will we know when we've started the game? How will we know when we've satisfied this requirement? How will we know when we're done?

2.3 Automating Acceptance Criteria

To answer these questions, we'll add *acceptance criteria* to the feature. Imagine that you sit down to play mastermind, you fire up a shell, and type the mastermind command. How do you know it started? Perhaps it says something like "Welcome to Mastermind!" And then, so you know what to do next, it probably says something like "Enter a guess:"

That will be the acceptance criteria for this feature. To express that with Cucumber, modify features/codebreaker_starts_game.feature so it reads like this:

Download mm/01/features/codebreaker_starts_game.feature

```
Feature: code-breaker starts game
  As a code-breaker
  I want to start a game
  So that I can break the code
  Scenario: start game
    Given I am not yet playing
    When I start a new game
    Then the game should say "Welcome to Mastermind!"
    And the game should say "Enter guess:"
```

The Scenario: keyword is followed by a string and then a series of steps. Each step begins with any of five keywords: *Given*, *When*, *Then*, *And* and *But*.

Given steps represent the state of the environment before an event. *When* steps represent the event. *Then* steps represent the expected outcomes.

And and *But* steps take on the quality of the previous step. In the start game scenario, the *And* step is a second *Then*, a second expected outcome. If we wanted to expect that the game says "Welcome to Mastermind!", but not "What is your quest?", we would add a *But* step saying *But the game should not say "What is your quest?"*, which would be treated as a *Then*.

See how the Given and When steps in this scenario both use the first person? We choose the first person form because it makes the narrative feel more compelling. Given x, when *I* y, then *I* should see a message saying "z." This helps to keep the focus on how *I* would use the system if *I* were in a given role (the code breaker).

"Given I am not yet playing" expresses the context in which the subsequent steps will be executed. "When I start a new game" is the event or action that occurs because *I* did something. The Thens are the expected outcomes—what we expect to happen after the When.

To run the feature and see the result, cd to the mastermind directory in a command shell and run cucumber features -n.[1] You should see output like this:

```
Feature: code-breaker starts game
  As a code-breaker
  I want to start a game
  So that I can break the code

  Scenario: start game
    Given I am not yet playing
    When I start a new game
    Then the game should say "Welcome to Mastermind!"
    And the game should say "Enter guess:"

1 scenario
4 undefined steps

You can implement step definitions for missing steps with these snippets:

Given /^I am not yet playing$/ do
  pending
end

When /^I start a new game$/ do
```

1. This, of course, assumes that you've already installed the cucumber gem. If you haven't, simply gem install cucumber (with sudo for some environments). And while you're at it, go ahead and gem install rspec and you'll have everything you need to get through this chapter.

```
  pending
end

Then /^the game should say "Welcome to Mastermind!"$/ do
  pending
end

Then /^the game should say "Enter guess:"$/ do
  pending
end
```

In addition to printing out the title, narrative, and steps in the scenario, a summary tells us that each of the four steps are pending definition. This is followed by code snippets for each pending step. Not only do we know what to do next, we even have a little help getting started with writing step definitions, which we'll explain next.

2.4 Steps and Step Definitions

Now that we have some pending steps, we need to write *step definitions* for them. If you think of the steps in scenarios as method calls, then step definitions are like method definitions. In Ruby, when you call a method that is not defined, you get a NoMethodError. In Cucumber, you get notification of a pending step, which you can think of as an undefined step.

The first pending step is *Given I am not yet playing*, and Cucumber gave us this snippet to get that started:

```
Given /^I am not yet playing$/ do
end
```

Go ahead and create a mastermind.rb file in features/step_definitions/ and add that snippet to it. Now run cucumber features -n from the project root, and you'll see the following output:

```
Feature: code-breaker starts game
  As a code-breaker
  I want to start a game
  So that I can break the code

  Scenario: start game
    Given I am not yet playing
    When I start a new game
    Then the game should say "Welcome to Mastermind!"
    And the game should say "Enter guess:"

1 scenario
```

```
3 undefined steps
1 passed step

You can implement step definitions for missing steps with these snippets:

When /^I start a new game$/ do
  pending
end

Then /^the game should say "Welcome to Mastermind!"$/ do
  pending
end

Then /^the game should say "Enter guess:"$/ do
  pending
end
```

Now we have 1 passing step and 3 steps pending implementation. So what just happened? When Cucumber parses a feature, it tries to match all of the steps in scenarios with step definitions written in Ruby. Steps are defined by calling any of three methods provided by Cucumber: Given(), When(), or Then(). In this case, we called the Given() method, passing it a Regexp and a block.

When Cucumber sees a step definition, it stores the block in a hash-like structure with the Regexp as its key. Then, for each step in a feature file, it searches for a Regexp that matches the step, and executes the block stored with that Regexp as its key.

In our case, when Cucumber sees the *Given I am not yet playing* step in the scenario, it strips off the *Given* and looks for a Regexp that matches the string I am not yet playing. At this point we only have one step definition, and its Regexp is /^I am not yet playing$/, so Cucumber executes the associated block from the step definition.

Of course, since there is nothing in the block yet, there is nothing that can go wrong, so the step is considered passing. As it turns out, that's exactly what we want in this case. We don't actually want *Given I am not yet playing* to do anything. We just want it in the scenario to provide context for the subsequent steps, but we're going to leave the associated block empty.

The *When* is where the action is. We need to create a new game and then start it. Here's what that might look like:

Download mm/02/features/step_definitions/mastermind.rb

```
When /^I start a new game$/ do
```

The Code You Wish You Had

In my early days at Object Mentor I attended a TDD class being taught by James Grenning. As he was talking about refactoring a Long Method, he wrote a statement that called a method that didn't exist yet, saying something like "start by writing the code you wish you had."

This was a galvanizing moment for me.

It is common to write the code we wish we had doing TDD. Perhaps we send a message from the code example to an object that does not have a corresponding method. We let the Ruby interpreter tell us that the method does not exist (red), and then implement that method (green).

Doing the same thing within application code, calling the code we wish we had in one module from another module, was a different matter. It was as though an arbitrary boundary was somehow lifted and suddenly all of the code was my personal servant, ready and willing to bend to my will. It didn't matter whether we were starting in a test, or in the code being tested. What mattered was that we started from the view of the code that was going to use the new code we were about to write.

Over the years this has permeated my daily practice. It is very, very liberating, and results consistently in more usable APIs than I would have come up with starting with the object receiving the message.

In retrospect, this also aligns closely with the Outside-In philosophy of BDD, perhaps taking it a step further. If the goal is to provide great APIs then the best place to design them is from their consumers.

```
  Mastermind::Game.new.start
end
```

At this point we don't have any application code, so we're just writing *the code we wish we had*. We want to keep it simple, and this is about as simple as it can get.

Now let's move on to the *Thens*.

Download mm/02/features/step_definitions/mastermind.rb

```
Then /^the game should say "Welcome to Mastermind!"$/ do
```

```
end

Then /^the game should say "Enter guess:"$/ do
end
```

They are both pretty much the same except for the strings, and since we're dealing with regular expressions we can generalize them into a single definition like this:

Download mm/03/features/step_definitions/mastermind.rb

```
Then /^the game should say "(.*)"$/ do |message|
end
```

This step definition will handle both the Then and And steps in the scenario, passing whatever is captured to the block as the message parameter.

As for what to write in the blocks, we need to have some way of knowing what message was returned when we sent start() to the Game object so we can specify that it matches the value of the block argument. Here's one way to handle this:

Download mm/04/features/step_definitions/mastermind.rb

```
When /^I start a new game$/ do
  game = Mastermind::Game.new
  @message = game.start
end

Then /^the game should say "(.*)"$/ do |message|
  @message.should == message
end
```

Here we store the return value of @game.start in a variable named @message in the *When* step definition. The code in the *Then* step borrows from RSpec to set an expectation about the value of the @message variable. You'll read all about RSpec's expectations in Chapter 11, *Expectations*, on page 128, and learn how to make them available to Cucumber step definitions later in the chapter.

In this case, we expect that @message should equal (using Ruby's ==() method) the message from the step in the scenario. This is RSpec's way of setting expectations about equality.

But there's a problem with this setup. Can you see what it is? Take a look at the story we're working on again. How many times do we invoke *Then the game should say: ....*? Twice in this scenario! We are expecting

two different messages to appear when we start the game. Time to make a design decision.

We need another approach. We could have the start() method return an array of messages and expect the array to contain the message we're interested in. That might look like this:

Download mm/05/features/step_definitions/mastermind.rb

```
When /^I start a new game$/ do
  game = Mastermind::Game.new
  @messages = game.start
end

Then /^the game should say "(.*)"$/ do |message|
  @messages.should include(message)
end
```

That could work, but let's take a step back for a second. How are we going to invoke this? What's the outermost layer of our system going to be? It's going to be a Ruby script. And we're going to want to keep that as lightweight as possible. Here's what it might have to look like if we went with this approach:

Download mm/05/bin/mastermind

```
#!/usr/bin/env ruby
$LOAD_PATH.push File.join(File.dirname(__FILE__), "/../lib")
require 'mastermind'

game = Mastermind::Game.new
messages = game.start
messages.each { puts message }
```

If we return an array of messages, then the script needs to take on some of the responsiblity of what to display, and when to display it. Among other problems, this is a violation of the Single Responsibility Principle [Mar02].

One solution to that violation would be to have the script hand the game STDOUT and have it post messages to that. The game wouldn't need to know it was STDOUT. It would just need to know it was something it could send messages to. If we did that, the mastermind script might look like this instead:

Download mm/06/bin/mastermind

```
#!/usr/bin/env ruby
$LOAD_PATH.push File.join(File.dirname(__FILE__), "/../lib")
require 'mastermind'
```

```
game = Mastermind::Game.new(STDOUT)
game.start
```

Much better! If you're a *nix user, go ahead and copy that into bin/mastermind, and chmod 755 bin/mastermind so you'll be able to execute it later.

Windows users should copy all but the very first line into bin/mastermind, and also add bin/mastermind.bat with the following:

Download mm/06/bin/mastermind.bat

```
@"ruby.exe" "%~dpn0" %*
```

The next question is how to express this design decision in the step definition? We don't want to use STDOUT because Cucumber is using STDOUT to report results when we run the scenarios. We *do* want something that shares an interface with STDOUT so that the Game object won't know the difference.

This is one of those occasions in which Ruby provides a solution that is so simple, it's difficult to stop yourself from chuckling. STDOUT is an instance of IO. The StringIO object is very much like an IO object. We can use one of those, have it store the messages and set our expectations on it like so:

Download mm/06/features/step_definitions/mastermind.rb

```
When /^I start a new game$/ do
  @messenger = StringIO.new
  game = Mastermind::Game.new(@messenger)
  game.start
end

Then /^the game should say "(.*)"$/ do |message|
  @messenger.string.split("\n").should include(message)
end
```

That's a tiny bit more complex, but it's still straightforward.

Now that we've implemented the code in the step definitions, let's run the *code-breaker starts game* feature and see what we've got. Go back to the command shell and run cucumber features -n again and you should see output like this:

```
Feature: code-breaker starts game
  As a code-breaker
  I want to start a game
  So that I can break the code

  Scenario: start game
    Given I am not yet playing
```

```
    When I start a new game
      uninitialized constant Mastermind (NameError)
      ./06/features/step_definitions/mastermind.rb:7:in
          `/^I start a new game$/'
      06/features/codebreaker_starts_game.feature:10:in
          `When I start a new game'
    Then the game should say "Welcome to Mastermind!"
    And the game should say "Enter guess:"

1 scenario
1 failed step
2 skipped steps
1 passed step
```

There is no application code yet, so we're getting a NameError on Mastermind. Take a look at the backtrace.

The first line of the backtrace is from features/step_definitions/mastermind.rb, the file with the step definitions in it. It's a Ruby file, and we'd expect that to show up. But check out the second line. It's from features/codebreaker_starts_game.feature, a file written in plain text.[2]

We also see that one step passed (the Given), one failed (the When), and two are skipped (the Thens). When a step fails, all of the subsequent steps are skipped because whether they pass or fail is not necessarily meaningful, as the state is not what you expect it to be.

This is useful feedback, but we can get even more by running the feature without the -n flag; just run cucumber features.[3] You should see output like this:

```
Feature: code-breaker starts game
  As a code-breaker
  I want to start a game
  So that I can break the code

  Scenario: start game
       # 06/features/codebreaker_starts_game.feature:8
    Given I am not yet playing
       # 06/features/step_definitions/mastermind.rb:1
    When I start a new game
       # 06/features/step_definitions/mastermind.rb:5
      uninitialized constant Mastermind (NameError)
      ./06/features/step_definitions/mastermind.rb:7:in
        `/^I start a new game$/'
```

2. Cucumber can do this because it is built on top of *Treetop*, a library for building grammars in Ruby.
3. Run cucumber --help to see a full listing of Cucumber's command line options

```
      06/features/codebreaker_starts_game.feature:10:in
        `When I start a new game'
    Then the game should say "Welcome to Mastermind!"
        # 06/features/step_definitions/mastermind.rb:11
    And the game should say "Enter guess:"
        # 06/features/step_definitions/mastermind.rb:11

1 scenario
1 failed step
2 skipped steps
1 passed step
```

Now we see file names and line numbers for the scenario (from the feature file), and each step (from the step definition file). This makes it quite easy to see where to go when things go wrong.

It would be most tempting at this point to simply define the Mastermind::Game class with a stub implementation of the start() method, but let's stop and reassess.

2.5 What We Just Did

At this point we've made our way through the second step in the concentric cycles described in Section 1.5, *The BDD Cycle*, on page 18: we now have a failing cucumber step. And we've also laid quite a bit of foundation.

We've set up the development environment for the Mastermind game, with the standard directories for Cucumber and RSpec. We expressed the first feature from the outside using Cucumber, with automatable acceptance criteria using the simple language of Given/When/Then.

So far we've been describing things from the outside with Cucumber. In the next chapter we'll begin to work our way from the Outside-In, using RSpec to drive out behaviour of individual objects.

Chapter 3

Describing Code with RSpec

In the last chapter, we introduced and used Cucumber to describe the behaviour of our Mastermind game from the outside, at the application level. We wrote a Cucumber feature with a scenario and step definitions that will handle the steps in the scenario, but we're getting an error. The code in a step definition is trying to interact with a Mastermind::Game object, but there is no application code to support this yet.

In this chapter we're going to use RSpec to *describe* behaviour at a much more granular level: the expected behaviour of instances of the Game class.

To get going, create a file named game_spec.rb in spec/mastermind/ and add the following code:

Download mm/06/spec/mastermind/game_spec.rb

```
module Mastermind
  describe Game do
  end
end
```

The describe() method hooks into RSpec's API, and it returns a Spec::ExampleGroup, which is, as it suggests, a group of examples—examples of the expected behaviour of an object. If you're accustomed to xUnit tools like Test::Unit, you can think of an ExampleGroup as being akin to a TestCase.

Open up a shell and cd to the mastermind directory and run the game_spec.rb file with the spec command, [1] like this:

```
spec spec/mastermind/game_spec.rb
```

1. The spec command is installed when you install the rspec gem.

The resulting output should include "uninitialized constant Mastermind::Game (NameError)" To fix that we need to do a few things. First, add a file named game.rb with the following code in the lib/mastermind directory:

Download mm/07/lib/mastermind/game.rb

```
module Mastermind
  class Game
  end
end
```

Then we require that file from lib/mastermind.rb:

Download mm/07/lib/mastermind.rb

```
require 'mastermind/game'
```

Next, the spec helper needs to add the lib directory to the load path and then require mastermind and spec:

Download mm/07/spec/spec_helper.rb

```
$: << File.join(File.dirname(__FILE__), "/../lib")
require 'spec'
require 'mastermind'
```

Lastly we need to require spec_helper.rb from game_spec.rb, which should then look like this:

Download mm/07/spec/mastermind/game_spec.rb

```
require File.join(File.dirname(__FILE__), "/../spec_helper")

module Mastermind
  describe Game do
  end
end
```

Now run game_spec.rb with the spec command again. You should see output like this:

```
Finished in 0.001545 seconds

0 examples, 0 failures
```

This tells us that everything is hooked up correctly and we can move on. To see where we are in relation to our feature, add the lib directory to the load path and require mastermind in features/support/env.rb:

Download mm/07/features/support/env.rb

```
$: << File.join(File.dirname(__FILE__), "/../../lib")
```

```
require 'mastermind'
```

Running features/codebreaker_starts_game.feature gives us a different error now:

```
Feature: code-breaker starts game
  As a code-breaker
  I want to start a game
  So that I can break the code

  Scenario: start game
         # 07/features/codebreaker_starts_game.feature:8
    Given I am not yet playing
         # 07/features/step_definitions/mastermind.rb:1
    When I start a new game
         # 07/features/step_definitions/mastermind.rb:5
      wrong number of arguments (1 for 0) (ArgumentError)
      ./07/features/step_definitions/mastermind.rb:7:in `initialize'
      ./07/features/step_definitions/mastermind.rb:7:in `new'
      ./07/features/step_definitions/mastermind.rb:7:in
          `/^I start a new game$/'
      07/features/codebreaker_starts_game.feature:10:in
          `When I start a new game'
    Then the game should say "Welcome to Mastermind!"
         # 07/features/step_definitions/mastermind.rb:11
    And the game should say "Enter guess:"
         # 07/features/step_definitions/mastermind.rb:11

1 scenario
1 failed step
2 skipped steps
1 passed step
```

We're getting an ArgumentError instead of a NameError.

This tells us two things: first, the feature is hooked up to the correct code; second, the Game needs to handle the messenger argument to the initialize method.

The process we're about to go through is the Red-Green-Refactor cycle straight out of Test-Driven Development. The idea is that you write a failing example (red), write only enough code to make the example pass (green), and then remove any unwanted duplication (refactor).

This is a process not unlike music or dancing. You get into a groove and it moves very, very quickly. To strive for that feeling, we're going to go through these steps in rapid succession with very little discussion between each step.

3.1 Red: Start With a Failing Code Example

In game_spec.rb, we want to do what we've done in the feature: specify that when we start the game, it sends the right messages to the messenger. Start by modifying game_spec.rb as follows:

Download mm/08/spec/mastermind/game_spec.rb

```
require File.join(File.dirname(__FILE__), "/../spec_helper")

module Mastermind
  describe Game do
    context "starting up" do
      it "should send a welcome message" do
        messenger.should_receive(:puts).with("Welcome to Mastermind!")
        game.start
      end
    end
  end
end
```

We *describe* the behaviour of a game object in a specific *context*: the game is just starting up. We start with the smallest amount of code we can write to express the intent of the example. The example expresses an expectation that a game, when starting up, should send a welcome message.

The expectation is expressed using RSpec's built-in mock framework, which is designed to *speak* like English: the *messenger* object *should receive* the puts() message with the string literal "Welcome to Mastermind!" We'll cover the mock framework in detail later on in Chapter 12, *Mocking in RSpec*, on page 151, but for now we just need to recognize that we have this expectation, and that we want some feedback if it is not met.

The example needs a few more things to be complete, but by starting with an expression of intent, we spend more time describing exactly what we want and less time thinking about how to set things up for an imaginary example. At this point, we're going to run the file and let the feedback we get push us in the right direction. If you run the file with the spec command you'll see output like this:

```
F

1)
NameError in 'Mastermind::Game starting up should send a welcome message'
undefined local variable or method `messenger' for \
              #<Spec::Example::ExampleGroup::Subclass_1::Subclass_2:0x112fc18>
```

```
01/08/spec/mastermind/game_spec.rb:7:
01/08/spec/mastermind/game_spec.rb:4:

Finished in 0.00866 seconds

1 example, 1 failure
```

And voila! We have *red*, a failing example. Sometimes failures are logical failures, sometimes errors. In this case, we have an error. Regardless, once we have red, we want to get to green.

3.2 Green: Get the Example To Pass

The error we got is a NameError on messenger. Observing this feedback, we want to add a messenger object. Since we're using the should_receive() method from the mock framework, we can just create a stock mock object using the mock() method.

Download mm/09/spec/mastermind/game_spec.rb

```
require File.join(File.dirname(__FILE__), "/../spec_helper")

module Mastermind
  describe Game do
    context "starting up" do
      it "should send a welcome message" do
        messenger = mock("messenger")
        messenger.should_receive(:puts).with("Welcome to Mastermind!")
        game.start
      end
    end
  end
end
```

The mock() method creates an instance of Spec::Mocks::Mock, which will behave however we program it to. As you'll see a bit later this chapter, the string passed to the mock() method is used for messages when an expectation fails.

Run game_spec.rb again and you should see similar output, but this time with a NameError on game. Following the feedback from RSpec, add the game object:

Download mm/10/spec/mastermind/game_spec.rb

```
require File.join(File.dirname(__FILE__), "/../spec_helper")

module Mastermind
  describe Game do
```

```
    context "starting up" do
      it "should send a welcome message" do
        messenger = mock("messenger")
        game = Game.new(messenger)
        messenger.should_receive(:puts).with("Welcome to Mastermind!")
        game.start
      end
    end
  end
end
```

Now run the spec again and you should see this:

```
F

1)
ArgumentError in 'Mastermind::Game starting up should send a welcome message'
wrong number of arguments (1 for 0)
01/10/spec/mastermind/game_spec.rb:8:in `initialize'
01/10/spec/mastermind/game_spec.rb:8:in `new'
01/10/spec/mastermind/game_spec.rb:8:
01/10/spec/mastermind/game_spec.rb:4:

Finished in 0.008357 seconds

1 example, 1 failure
```

This time we get an argument error indicating that Game#initialize() needs to accept the messenger object. Go ahead and skip on over to lib/mastermind/game.rb and add an initialize() method as follows:

Download mm/11/lib/mastermind/game.rb

```
module Mastermind
  class Game
    def initialize(messenger)
    end
  end
end
```

Run game_spec.rb and you should see this error:

```
F

1)
NoMethodError in 'Mastermind::Game starting up should send a welcome message'
undefined method `start' for #<Mastermind::Game:0x5d8540>
01/11/spec/mastermind/game_spec.rb:10:
01/11/spec/mastermind/game_spec.rb:4:

Finished in 0.008641 seconds

1 example, 1 failure
```

Time to add a start() method:

Download mm/12/lib/mastermind/game.rb

```
module Mastermind
  class Game
    def initialize(messenger)
    end

    def start
    end
  end
end
```

Now run game_spec.rb again and instead of an error, we get our first logical failure.

```
F

1)
Spec::Mocks::MockExpectationError in \
                  'Mastermind::Game starting up should send a welcome message'
Mock 'messenger' expected :puts with \
                  ("Welcome to Mastermind!") once, but received it 0 times
01/12/spec/mastermind/game_spec.rb:9:
01/12/spec/mastermind/game_spec.rb:4:

Finished in 0.009561 seconds

1 example, 1 failure
```

The expectation that the welcome message is received by the messenger is not being met. To resolve this, we just need to store the messenger in an instance variable and send it the puts() message from the start() method in the game object:

Download mm/13/lib/mastermind/game.rb

```
module Mastermind
  class Game
    def initialize(messenger)
      @messenger = messenger
    end

    def start
      @messenger.puts "Welcome to Mastermind!"
    end
  end
end
```

Now run the file and you should see this glorious output:

```
.
```

```
Finished in 0.008373 seconds

1 example, 0 failures
```

Try running it with the format option:

```
spec spec/mastermind/game_spec.rb --format specdoc


Mastermind::Game starting up
- should send a welcome message

Finished in 0.008202 seconds

1 example, 0 failures
```

The specdoc format lists all of the examples with all of the text descriptions you include. Assuming that your monitor has more colors than this book, you can also add the --color option to see passing examples in green and failing examples in red.

At this point we move to the third part of the cycle, refactoring to remove duplication. Sometimes, however, there is really not any duplication to remove. This seems one of those cases, so we're done with this cycle. But before we start another cycle, let's see what impact we've had on the feature thus far.

Go ahead and run the feature file with the cucumber command. The output should look like this now:

```
Feature: code-breaker starts game
  As a code-breaker
  I want to start a game
  So that I can break the code

  Scenario: start game
          # 13/features/codebreaker_starts_game.feature:8
    Given I am not yet playing
          # 13/features/step_definitions/mastermind.rb:1
    When I start a new game
          # 13/features/step_definitions/mastermind.rb:5
    Then the game should say "Welcome to Mastermind!"
          # 13/features/step_definitions/mastermind.rb:11
      undefined method `include' for #<Object:0x19a3778>
          (NoMethodError)
      ./13/features/step_definitions/mastermind.rb:12:in
          `/^the game should say "(.*)"$/'
      13/features/codebreaker_starts_game.feature:11:in
          `Then the game should say "Welcome to Mastermind!"'
    And the game should say "Enter guess:"
```

```
          # 13/features/step_definitions/mastermind.rb:11

1 scenario
1 failed step
1 skipped step
2 passed steps
```

The missing include() method is an RSpec expectation matcher method, so we need to require the RSpec expectations library:

Download mm/14/features/support/env.rb

```
$: << File.join(File.dirname(__FILE__), "/../../lib")
require 'spec/expectations'
require 'mastermind'
```

Run the feature again, and:

```
Feature: code-breaker starts game
  As a code-breaker
  I want to start a game
  So that I can break the code

  Scenario: start game
          # 14/features/codebreaker_starts_game.feature:8
    Given I am not yet playing
          # 14/features/step_definitions/mastermind.rb:1
    When I start a new game
          # 14/features/step_definitions/mastermind.rb:5
    Then the game should say "Welcome to Mastermind!"
          # 14/features/step_definitions/mastermind.rb:11
    And the game should say "Enter guess:"
          # 14/features/step_definitions/mastermind.rb:11
      expected ["Welcome to Mastermind!"] to include "Enter guess:"
          (Spec::Expectations::ExpectationNotMetError)
      ./14/features/step_definitions/mastermind.rb:12:in
          `/^the game should say "(.*)"$/'
      14/features/codebreaker_starts_game.feature:12:in
          `And the game should say "Enter guess:"'

1 scenario
1 failed step
3 passed steps
```

Progress! Now one of the two *Thens* is passing, so it looks like we're about halfway done with this feature. Actually we're quite a bit more than halfway done, because, as you'll soon see, all of the pieces are already in place for the rest.

The next failing step is the next thing to work on: "And the game should say: Enter guess:" Go ahead and add an example for this behaviour to game_spec.rb. Start with the last two lines, like this:

Download mm/14/spec/mastermind/game_spec.rb

```
it "should prompt for the first guess" do
  messenger.should_receive(:puts).with("Enter guess:")
  game.start
end
```

Then run the examples and let the feedback guide you through each step like we did in the first example. The example should end up looking like this:

Download mm/16/spec/mastermind/game_spec.rb

```
it "should prompt for the first guess" do
  messenger = mock("messenger")
  game = Game.new(messenger)
  messenger.should_receive(:puts).with("Enter guess:")
  game.start
end
```

And the feedback should end up looking like this:

```
.F

1)
Spec::Mocks::MockExpectationError in \
              'Mastermind::Game starting up should prompt for the first guess'
Mock 'messenger' expected :puts with ("Enter guess:") \
              but received it with ("Welcome to Mastermind!")
./01/16/spec/mastermind/../../lib/mastermind/game.rb:8:in `start'
01/16/spec/mastermind/game_spec.rb:17:
01/16/spec/mastermind/game_spec.rb:4:

Finished in 0.008954 seconds

2 examples, 1 failure
```

It looks like we need to send the messenger puts() with “Enter guess:” So head back to game.rb and modify it as follows:

Download mm/17/lib/mastermind/game.rb

```
def start
  @messenger.puts "Welcome to Mastermind!"
  @messenger.puts "Enter guess:"
end
```

Now run game_spec.rb:

```
FF

1)
Spec::Mocks::MockExpectationError in \
              'Mastermind::Game starting up should send a welcome message'
```

```
Mock 'messenger' expected :puts with \
             ("Welcome to Mastermind!") but received it with ("Enter guess:")
./01/17/spec/mastermind/../../lib/mastermind/game.rb:10:in `start'
01/17/spec/mastermind/game_spec.rb:10:
01/17/spec/mastermind/game_spec.rb:4:

2)
Spec::Mocks::MockExpectationError in \
             'Mastermind::Game starting up should prompt for the first guess'
Mock 'messenger' expected :puts with ("Enter guess:") \
             but received it with ("Welcome to Mastermind!")
./01/17/spec/mastermind/../../lib/mastermind/game.rb:9:in `start'
01/17/spec/mastermind/game_spec.rb:17:
01/17/spec/mastermind/game_spec.rb:4:

Finished in 0.009537 seconds

2 examples, 2 failures
```

And *ta da!* Now not only is the second example still failing, but the first example is *failing now as well*! Who'da thunk? This may seem a bit confusing if you've never worked with mock objects and message expectations before, but mock objects are like computers. They are extraordinarily obedient, but they are not all that clever. By default, mocks will expect exactly what you tell them to expect, nothing more and nothing less.

We've told the mock in the first example to expect puts() with "Welcome to Mastermind!" and we've satisfied that requirement, but we've only told it to expect "Welcome to Mastermind!" It doesn't know anything about "Enter guess:"

Similarly, the mock in the second example expects "Enter guess:" but the first message it gets is "Welcome to Mastermind!"

We could combine these two into a single example, but we like to follow the guideline of "one expectation per example." The rationale here is that if there are two expectations in an example that should both fail given the implementation at that moment, we'll only see the first failure. No sooner do we meet that expectation than we discover that we haven't met the second expectation. If they live in separate examples, then they'll both fail, and that will provide us with more accurate information than if only one of them is failing.

We could also try to break the messages up into different steps, but we've already defined how we want to talk to the game object. So how can we resolve this?

There are a couple of ways we can go about it, but the simplest way is to tell the mock messenger to only listen for the messages we tell it to expect, and ignore any other messages. This is based on the *Null Object* design pattern [MRB97], and is supported by RSpec's mock framework with the as_null_object() method:

Download mm/18/spec/mastermind/game_spec.rb

```
require File.join(File.dirname(__FILE__), "/../spec_helper")

module Mastermind
  describe Game do
    context "starting up" do
      it "should send a welcome message" do
        messenger = mock("messenger").as_null_object
        game = Game.new(messenger)
        messenger.should_receive(:puts).with("Welcome to Mastermind!")
        game.start
      end

      it "should prompt for the first guess" do
        messenger = mock("messenger").as_null_object
        game = Game.new(messenger)
        messenger.should_receive(:puts).with("Enter guess:")
        game.start
      end
    end
  end
end
```

Hey, there's a fair amount of duplication here. When you observe duplication while you're in the middle of the *red* part of Red-Green-Refactor, it's best to just take note of it and plan to address it once you get to green.

Now go ahead and run game_spec.rb with --format specdoc:

```
Mastermind::Game starting up
- should send a welcome message
- should prompt for the first guess

Finished in 0.008966 seconds

2 examples, 0 failures
```

Good news. Both examples are now passing. Now that we have green, it's time to refactor!

3.3 Refactor to Remove Duplication

In the preface to his seminal book on *Refactoring* [FBB+99], Martin Fowler writes: "Refactoring is the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure."

How do we know that we're not changing behaviour? We run the examples between every change. If they pass, we've refactored successfully. If any fail, we know that the very last change we made caused a problem and we either quickly recognize and address the problem, or rollback that step to get back to green and try again.

Fowler talks about changing the designs of systems, but on a more granular scale, we want to refactor to eliminate duplication in the implementation *and* examples. Looking back at game_spec.rb, we can see that the first two lines of each example are identical. Perhaps you noticed this earlier, but we prefer to refactor *in the green* rather than *in the red*. Also, you may recall that we wrote the last two lines of each example first because they expressed intent.

In this case we have a very clear break between what is context and what is behaviour, so let's take advantage of that and move the context to a block that is executed before each of the examples. Not in the least coincidentally, RSpec calls the method before() and it takes a symbol to indicate before *what*.

As with moving from red to green, we're going to go through this one step at a time. The steps are very small, but they happen in rapid succession, running the examples between each step. I'm not going to show you the output after each step, but you should be running the examples between each step to make sure that each change is not affecting the behaviour.

First, change messenger, in the first example only, to an instance variable and move it to a before() method.

Download mm/19/spec/mastermind/game_spec.rb

```
require File.join(File.dirname(__FILE__), "/../spec_helper")

module Mastermind
  describe Game do
    context "starting up" do
      before(:each) do
        @messenger = mock("messenger").as_null_object
      end
```

```
      it "should send a welcome message" do
        game = Game.new(@messenger)
        @messenger.should_receive(:puts).with("Welcome to Mastermind!")
        game.start
      end

      it "should prompt for the first guess" do
        messenger = mock("messenger").as_null_object
        game = Game.new(messenger)
        messenger.should_receive(:puts).with("Enter guess:")
        game.start
      end
    end
  end
end
```

Run the examples to make sure they pass. If they don't, make sure that you've changed all the references to messenger in before(:each) and the first example from local variables to instance variables.

Now do the same thing with the game object.

Download mm/20/spec/mastermind/game_spec.rb

```
require File.join(File.dirname(__FILE__), "/../spec_helper")

module Mastermind
  describe Game do
    context "starting up" do
      before(:each) do
        @messenger = mock("messenger").as_null_object
        @game = Game.new(@messenger)
      end

      it "should send a welcome message" do
        @messenger.should_receive(:puts).with("Welcome to Mastermind!")
        @game.start
      end

      it "should prompt for the first guess" do
        messenger = mock("messenger").as_null_object
        game = Game.new(messenger)
        messenger.should_receive(:puts).with("Enter guess:")
        game.start
      end
    end
  end
end
```

Again, run the examples and make sure they pass. The last step is to remove the first two lines of the second example and reference the

instance variables.

Download mm/21/spec/mastermind/game_spec.rb

```
require File.join(File.dirname(__FILE__), "/../spec_helper")

module Mastermind
  describe Game do
    context "starting up" do
      before(:each) do
        @messenger = mock("messenger").as_null_object
        @game = Game.new(@messenger)
      end

      it "should send a welcome message" do
        @messenger.should_receive(:puts).with("Welcome to Mastermind!")
        @game.start
      end

      it "should prompt for the first guess" do
        @messenger.should_receive(:puts).with("Enter guess:")
        @game.start
      end
    end
  end
end
```

Now that's a bit cleaner, don't you think? As noted earlier, the code before(:each) example sets up the context and the code in each example is restricted to that which expresses intent.

Now run the feature again:

```
Feature: code-breaker starts game
  As a code-breaker
  I want to start a game
  So that I can break the code

  Scenario: start game
        # 21/features/codebreaker_starts_game.feature:8
    Given I am not yet playing
        # 21/features/step_definitions/mastermind.rb:1
    When I start a new game
        # 21/features/step_definitions/mastermind.rb:5
    Then the game should say "Welcome to Mastermind!"
        # 21/features/step_definitions/mastermind.rb:11
    And the game should say "Enter guess:"
        # 21/features/step_definitions/mastermind.rb:11

1 scenario
4 passed steps
```

And voila! We now have our first passing code examples and our first passing feature. There were a lot of steps to get there, but in practice this all really takes just a few minutes, even with all the wiring and require statements.

We've also set up quite a bit of infrastructure. You'll see, as we move along, that there is less and less new material needed to add more features, code examples and application code. It just builds gradually on what we've already developed.

In the last chapter, we created the bin/mastermind script (bin/mastermind.bat if you're on Windows) that we use to run the mastermind game. Go ahead and fire up a shell and run the script and you'll see the following output:

```
Welcome to Mastermind!
Enter guess:
```

Now look at that! Who knew that all this code was actually going to start to make something work? Of course, our Mastermind game just says hello and then climbs back in its cave, so we've got a way to go before you'll want to show this off to all your friends.

In the next chapter, we'll start to get down to the real fun, submitting guesses and having the game score them. By the end of the next chapter, you'll actually be able to play the game! But before we move on, let's review what we've done thus far.

3.4 What We Just Did

We started this chapter with a failing step in a Cucumber scenario. This was our cue to jump from the outer circle (Cucumber) to the inner circle (RSpec) of the BDD cycle.

We then followed the familiar TDD Red/Green/Refactor cycle using RSpec. Once we had a passing code example we re-ran the Cucumber scenario. We saw that we had gotten our first Then step to pass, but there was one more that was failing, so we jumped back down to RSpec, went through another Red/Green/Refactor cycle, and now the whole scenario was passing.

This is the BDD cycle. Driving development from the Outside-In, starting with business facing scenarios in Cucumber and working our way inward to the underlying objects with RSpec.

The material in the next chapter, submitting guesses, is going to present some interesting challenges. It will expose you to some really cool features in Cucumber, as well as some thought provoking discussion about the relationship between Cucumber scenarios and RSpec code examples. So take a few minutes break, drink up that brain juice, and meet me at the top of the next chapter.

Chapter 4

Adding New Features

Welcome back! We left off with the Mastermind game inviting us to guess the secret code, but then leaving us hanging at the command line. The next feature we're going to tackle is submitting a guess, and getting feedback from the Mastermind game as to how close the guess is to breaking the secret code. Here's how it should work, expressed as a Cucumber feature and narrative:

```
Feature: code-breaker submits guess

  The code-breaker submits a guess of four colored
  pegs. The mastermind game marks the guess with black
  and white "marker" pegs.

  For each peg in the guess that matches color
  and position of a peg in the secret code, the
  mark includes one black peg. For each additional
  peg in the guess that matches the color but not
  the position of a color in the secret code, a
  white peg is added to the mark.
```

For acceptance criteria, we'll need a scenario with all four colors correct and in the correct positions, like this.

```
Scenario: all correct colors in the correct positions
  Given the secret code is r g c y
  When I guess r g c y
  Then the mark should be bbbb
```

Then we'll need another with all four colors correct, but only two in the correct postions.

```
Scenario: all correct colors with two in the correct position
  Given the secret code is r g c y
  When I guess r g y c
```

```
Then the mark should be bbww
```

Then we'll need one with only one in the correct position, then none. Then we'll need more scenarios for three correct colors with three in the correct position, two in the correct position, one, and then none. Then the same for two correct colors, one correct color and no correct colors at all. That's a *lot* of scenarios, all with the same format.

We're not going to want to write that over and over again because it will be hard to read. We need a strategy to DRY[1] things up here, without the sort of indirection that would be typical in code but would make the scenarios difficult to read.

4.1 Scenario Outlines

As you'll read in the (as yet) unwritten *chp.cucumber*, Cucumber offers a number of different strategies for keeping scenarios DRY, each targeted at reducing different sorts of duplication that crop up when writing suites of features and scenarios. In this case, we have a number of scenarios that will have a common format with different input and output data.

To solve this problem, Cucumber lets us define a single *Scenario Outline* and then provide tables of input data and expected output. Here's how you define a Scenario Outline:

```
Scenario Outline: submit guess
  Given the secret code is <code>
  When I guess <guess>
  Then the mark should be <mark>
```

This looks a lot like the scenario declarations we wrote for the *codebreaker starts game* feature, with two subtle differences:

- Scenario Outline instead of Scenario
- Variable data placeholders in <angle brackets>

The words in angle brackets are placeholders for variable data that we'll provide in a tabular format, inspired by FIT (see the sidebar on the following page).

1. The Don't Repeat Yourself principle (DRY) from *The Pragmatic Programmer* [HT00] says that every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

FIT

Ward Cunningham's Framework for Integration Test, or FIT, parses display tables in rich documents written with Microsoft Word or HTML, sends the contents of table cells to the system in development, and compares the results from the system to expected values in the table.*

This allows teams who were already using tools like Word for requirements documentation to turn those documents into executable acceptance tests by specifying expected outputs resulting from prescribed inputs. This works especially well when the acceptance criteria are naturally expressed in a table.

Cucumber's Scenario Outlines and Scenario Tables provide a FIT-inspired tabular format for expressing repetitive scenarios like those in our "submit guess" feature, while maintaining the Given, When, and Then language of BDD.

*. See http://fit.c2.com/ for more information about FIT.

Tabular Data

Here is the first of several tables we'll add, supplying data for scenarios in which all four colors are correct:

```
Scenarios: all colors correct
  | code    | guess   | mark |
  | r g y c | r g y c | bbbb |
  | r g y c | r g c y | bbww |
  | r g y c | y r g c | bwww |
  | r g y c | c r g y | wwww |
```

The *Scenarios* keyword indicates that what follows are rows of example data. The first row contains column headers that align with the placeholders in the scenario outline. Each subsequent row represents a single scenario.

Following convention, we've named the columns using the same names that are in angle brackets in the scenario outline, but the placeholders and columns are bound by position, not name.

The <code> variable in the *Given* step is assigned the value r g y c, from the first column in the first data row (after the headers). It's just as though we wrote Given the secret code is r g y c.

The <guess> in the *When* step gets r g y c from the second column, and the <mark> in the *Then* step gets bbbb.

With the Scenario Outline and this first table, we've expressed four scenarios that would have taken sixteen lines in only ten. We've also reduced duplication and created very readable executable documentation in the process. Cucumber lets us supply as many groups of Scenarios as we want, supporting a very natural way to group like scenarios. Here's the resulting feature with thirteen scenarios expressed in a mere twenty five lines (beginning with the Scenario Outline):

Download mm/23/features/codebreaker_submits_guess.feature

```
Feature: code-breaker submits guess

  The code-breaker submits a guess of four colored
  pegs. The mastermind game marks the guess with black
  and white "marker" pegs.

  For each peg in the guess that matches color
  and position of a peg in the secret code, the
  mark includes one black peg. For each additional
  peg in the guess that matches the color but not
  the position of a color in the secret code, a
  white peg is added to the mark.

  Scenario Outline: submit guess
    Given the secret code is <code>
    When I guess <guess>
    Then the mark should be <mark>

  Scenarios: all colors correct
    | code    | guess   | mark |
    | r g y c | r g y c | bbbb |
    | r g y c | r g c y | bbww |
    | r g y c | y r g c | bwww |
    | r g y c | c r g y | wwww |

  Scenarios: 3 colors correct
    | code    | guess   | mark |
    | r g y c | w g y c | bbb  |
    | r g y c | w r y c | bbw  |
    | r g y c | w r g c | bww  |
    | r g y c | w r g y | www  |

  Scenarios: 2 colors correct
    | code    | guess   | mark |
    | r g y c | w g w c | bb   |
    | r g y c | w r w c | bw   |
    | r g y c | g w c w | ww   |
```

```
Scenarios: 1 color correct
  | code    | guess   | mark |
  | r g y c | r w w w | b    |
  | r g y c | w w r w | w    |
```

Now that is clean and clear! It provides a wealth of information in a very concise format. Copy that text into features/codebreaker_submits_guess.feature and run it with this command:

```
cucumber --require features features/codebreaker_submits_guess.feature
```

You should get output that looks exactly like the text in the file, plus the source locations for each step and code snippets for each undefined step.

Step Definitions

Step definitions for Scenario Outlines and Tables are just like the step definitions we learned about in Chapter 2, *Describing Features with Cucumber*, on page 20. We'll still provide regular expressions that capture input data, and a block of code that interacts with the subject code.

Copy the first snippet into features/step_definitions/mastermind.rb and modify it as follows:

Download mm/27/features/step_definitions/mastermind.rb

```
Given /^the secret code is (. . . .)$/ do |code|
  @messenger = StringIO.new
  game = Mastermind::Game.new(@messenger)
  game.start(code.split)
end
```

The Regexp captures a group of four characters separated by spaces at the end of the line. This will capture the code (r c g y, for example), and pass it to the body of the step definition. The first two lines of the body should look familiar, as they are just like the *I start a new game* step. Then the last line passes the code from the match group as an array.

Now run cucumber -r features features/codebreaker_submits_guess.feature again and you'll see output including this:

```
Scenarios: all colors correct
  | code    | guess   | mark |
  | r g y c | r g y c | bbbb |
    wrong number of arguments (1 for 0) (ArgumentError)
    ./27/features/step_definitions/mastermind.rb:8:in
        `start'
```

```
./27/features/step_definitions/mastermind.rb:8:in
    `/^the secret code is (. . . .)$/'
27/features/codebreaker_submits_guess.feature:21:in
    `Given the secret code is r g y c'
```

You should see the ArgumentError for every scenario. This is actually good news, because the error tells us that everything is wired up correctly, and we now know what we have to do next: get the start() method on Game to accept the code as an argument.

4.2 Responding to Change

At this point, all of the RSpec code examples are passing, but we've got failing Cucumber scenarios. We're *in the meantime*, so to speak, where changing requirements from the outside are rendering our requirements on the inside incorrect.

To resolve this situation, we could use brute force and simply add a secret code to the calls to the start() method in Mastermind::Game, and then modify the method to accept an argument, and then run the examples. That might actually work out OK in this case because there is so little going on in our system. It doesn't take long, however, for a system to grow complex enough that this brute force approach results in long periods of time without all of the examples passing.

The Disciplined Approach

The disciplined approach to this situation is to incrementally add examples that specify the new behaviour, and follow the red/green/refactor cycle to evolve the code to support the new *and the old* behaviour at the same time. Once the new requirements are satisfied, we'll remove support for the old behaviour.

Take a look at the first example that calls the start() method:

Download mm/23/spec/mastermind/game_spec.rb

```
it "should send a welcome message" do
  @messenger.should_receive(:puts).with("Welcome to Mastermind!")
  @game.start
end
```

Leaving that in place, add a copy of it in which the only difference is that we're passing the code to the start() method. And let's identify the difference in the docstring passed to it(), so if we get disrupted, we'll be able to look back at this and see what's going on.

Download mm/24/spec/mastermind/game_spec.rb

```
it "should send a welcome message (passing the code to start)" do
  @messenger.should_receive(:puts).with("Welcome to Mastermind!")
  @game.start(%w[r g y c])
end
```

Run game_spec.rb and you should see one failure:

```
Mastermind::Game
  starting up
    should send a welcome message
    should send a welcome message (passing the code to start) (FAILED - 1)
    should prompt for the first guess

1)
ArgumentError in 'Mastermind::Game starting up should send a welcome message (passin...*TRUNC*
wrong number of arguments (1 for 0)
./011/spec/mastermind/game_spec.rb:18:in `start'
./011/spec/mastermind/game_spec.rb:18:
```

The additional context information we added to the docstring helps us to see that the failing example is the new one we just added. To get that to pass without failing the others, we'll add an optional argument to the start() method:

Download mm/25/lib/mastermind/game.rb

```
def start(code=nil)
  @messenger.puts "Welcome to Mastermind!"
  @messenger.puts "Enter guess:"
end
```

If you run game_spec.rb again, you should see that all three examples are passing. With that simple change, the start() method is supporting both the old and the new requirements.

Finding the Way Home

At this point there are a couple of different paths to our destination. Because the code supports 0 or 1 argument, we don't need to copy the other example that was there before this refactoring. We can just modify the example to pass the code to the start() method, run the examples, and watch them pass. Go ahead and do that yourself.

So now we're left with two versions of the first example. We'd like to remove the old version, but before we do, we can do a sort of reverse-sanity-check-refactoring. Modify the start method such that the code argument is no longer optional:

Download mm/26/lib/mastermind/game.rb

```
def start(code)
  @messenger.puts "Welcome to Mastermind!"
  @messenger.puts "Enter guess:"
end
```

Now run the examples and you should see one failure:

```
Mastermind::Game
  starting up
    should send a welcome message (FAILED - 1)
    should send a welcome message (passing the code to start)
    should prompt for the first guess

1)
ArgumentError in 'Mastermind::Game starting up should send a welcome message'
wrong number of arguments (0 for 1)
./013/spec/mastermind/game_spec.rb:13:in `start'
./013/spec/mastermind/game_spec.rb:13:
```

The one that is failing is the only example left that is still calling start() with no code, and it's a near duplicate of the passing example that says (passing the code to start). We can now safely remove the failing example, and remove the parenthetical context from the passing example, leaving us with this:

Download mm/27/spec/mastermind/game_spec.rb

```
it "should send a welcome message" do
  @messenger.should_receive(:puts).with("Welcome to Mastermind!")
  @game.start(%w[r c g y])
end

it "should prompt for the first guess" do
  @messenger.should_receive(:puts).with("Enter guess:")
  @game.start(%w[r c g y])
end
```

Download mm/27/lib/mastermind/game.rb

```
def start(code)
  @messenger.puts "Welcome to Mastermind!"
  @messenger.puts "Enter guess:"
end
```

We've reached our short term destination. The route here was circuitous, but we did things with discipline, taking small steps, with a green bar only moments away at every step. Again, in this particular case, brute force *may* be OK, but it's very useful to have the skill to approach this in a disciplined way on the occasions in which brute force leads you astray.

A Small Change Goes a Long Way

Now let's see what impact that had on the scenarios. Run the feature again with cucumber -r features features/codebreaker_submits_guess.feature, and you should see output including:

```
13 scenarios
26 undefined steps
13 passed steps
```

There are no failures now, and we still have 26 steps undefined, but we now have 13 passing steps: the Given steps in each scenario. Remember, each row in the tables represents a separate scenario. Until we get to the point where the failures are logical failures, as opposed to runtime errors due to structural discrepancies, a small change is likely to impact all of the scenarios at once.

The remaining undefined steps are the When steps that actually submit the guess, and the Then steps that set the expectation that the game should mark the guess. The details of these steps vary across all the different scenarios depending on the content of the guess and the expected mark, but they all have the same structure. So even though you see many code snippets for the undefined steps, we'll really only need two.

The first one is the When step: the action step. Using the snippet provided by Cucumber, and filling in the code block, here is the implementation of this When step, along with some necessary modifications to the Given step.

Download mm/29/features/step_definitions/mastermind.rb

```
Given /^the secret code is (. . . .)$/ do |code|
  @messenger = StringIO.new
  @game = Mastermind::Game.new(@messenger)
  @game.start(code.split)
end

When /^I guess (. . . .)$/ do |code|
  @game.guess(code.split)
end
```

We need to make @game an instance variable so it can be shared across the two steps. Now run this feature with the cucumber command and you should see output like this:

```
Scenarios: all colors correct
  | code    | guess   | mark |
  | r g y c | r g y c | bbbb |
```

```
undefined method `guess' for #<Mastermind::Game:0x1805f38
    @messenger=#<StringIO:0x1805f60>> (NoMethodError)
./29/features/step_definitions/mastermind.rb:12:in
    `/^I guess (. . . .)$/'
29/features/codebreaker_submits_guess.feature:21:in
    `When I guess r g y c'
```

You should actually see many similar messages. They all tell us that we need to implement a guess() method on Mastermind::Game.

Loosening the Leash

Earlier, when the Given step told us we needed to add an argument to start(), we already had examples in place that used that method, so we refactored the method definition in the examples in order to drive that change from the code examples.

This situation is a bit different. We'd like to create examples in RSpec before writing any code in the Game class, but we haven't implemented any steps in Cucumber that tell us how that method should behave. Right now, we only need it to exist. We could, in theory, write an example like this:

```
it "should have a guess method" do
  game = Game.new
  game.guess(%w[r y c g])
end
```

Absent the existence of the guess() method, this example would fail, and we'd be able to happily move on knowing that an example drove the method into existence. But BDD is about *behaviour*, not structure. We want to *use* methods in code examples, not specify that they exist. So how can we proceed and stay true to the cycle without adding a silly example like the one above?

The answer is that we can't. And, furthermore, *it's OK!* We're going to sin a little bit here and just add the method to Game without a spec.

Download mm/30/lib/mastermind/game.rb

```
def guess(guess)
end
```

Scary, huh? Well, relax. We're only moments away from validating the existence of this method, and it *is*, after all, being driven into existence by a failing example. It's just an example that lives up at a higher level.

Now go ahead and run the feature, and you should see some different feedback:

```
13 scenarios
13 undefined steps
26 passed steps
```

Again there are no failures, but now there are only 13 steps undefined. These are the Then steps. Go ahead and implement the step definition in features/step_definitions/mastermind.rb like this:

Download mm/31/features/step_definitions/mastermind.rb

```
Then /^the mark should be (.*)$/ do |mark|
  @messenger.string.split("\n").should include(mark)
end
```

Look familiar? This is just like the Then step from earlier scenarios: the game should say "(.*)". We're getting the string from the messenger, splitting it into an array, and expecting the mark to be one of the array elements.

Run the feature and you should see feedback like this:

```
Scenarios: all colors correct
  | code    | guess   | mark |
  | r g y c | r g y c | bbbb |
    expected ["Welcome to Mastermind!", "Enter guess:"]
        to include "bbbb" (Spec::Expectations::ExpectationNotMetError)
    ./31/features/step_definitions/mastermind.rb:28:in
        `/^the mark should be (.*)$/'
    31/features/codebreaker_submits_guess.feature:21:in
        `Then the mark should be bbbb'
```

Fantastic! Now all 13 scenarios are failing on the Then step. Again, this is good news. We know everything is wired up correctly, and we now have all of the step definitions we need. Now it's time to drill down to RSpec and drive out the solution with isolated examples.

4.3 The Simplest Thing

Looking back at the scenarios, we'll build a code example around the first one that is failing: all four colors correct and all in the correct positions. Open up spec/mastermind/game_spec.rb and add the following code inside the describe Game block, after the context "starting up" block.

Download mm/32/spec/mastermind/game_spec.rb

```
Line 1  context "marking a guess" do
    -     context "with all 4 colors correct in the correct places" do
    -       it "should mark the guess with bbbb" do
    -         messenger = mock("messenger").as_null_object
    5         game = Game.new(messenger)
    -         game.start(%w[r g y c])
```

```
-
-       messenger.should_receive(:puts).with("bbbb")
-
10      game.guess(%w[r g y c])
-     end
-   end
- end
```

We're nesting the "with all 4 colors correct in the correct places" context inside the "marking a guess" context. Then the code example "should mark the guess with bbbb."

The first three lines in the example provide the givens for this example. The *when* is actually at the end of the example on line 10. The *then* is expressed as a message expectation (mock), so it has to appear before the action.

After starting the game, this example sets an expectation that the messenger should receive the puts() message with the string "bbbb" as the result of calling game.guess() with the same secret code passed to the start() method.

Running this example produces the following output:

```
Mastermind::Game
  starting up
    should send a welcome message
    should prompt for the first guess
  marking a guess
    with all 4 colors correct in the correct places
      should mark the guess with bbbb (FAILED - 1)

1)
Spec::Mocks::MockExpectationError in 'Mastermind::Game marking a guess with all 4 co...*TRUNC*
Mock 'messenger' expected :puts with ("bbbb") once, but received it 0 times
./07/spec/mastermind/game_spec.rb:30:

Finished in 0.002338 seconds

3 examples, 1 failure
```

The example is failing, as we would expect. To get it to pass, we'll do *the simplest thing that could possibly work*.

One guideline that comes directly from Test Driven Development is to write only enough code to pass the one failing example, and no more. To do this, we write the simplest thing that we can to get the example to pass. In this case, we'll simply call puts("bbbb") on the messenger from the guess() method.

Download mm/33/lib/mastermind/game.rb

```
def guess(guess)
  @messenger.puts "bbbb"
end
```

That's all we need to get the example to pass:

```
Mastermind::Game
  starting up
    should send a welcome message
    should prompt for the first guess
  marking a guess
    with all 4 colors correct in the correct places
      should mark the guess with bbbb

Finished in 0.001845 seconds

3 examples, 0 failures
```

If this idea is new to you, and you're experiencing that gnawing feeling that one only gets when knowingly committing a sin, you have our full support. Your instinct is correct. This code is naive, and will tell the user that he's broken the secret code every time he submits a guess. Obviously, this does not solve the problem that we *know* we need to solve. But there are two sides to every story.

Express the problem before solving it

If code is the solution to a problem, and naive code is passing all of its examples, then the problem has not been properly expressed in the examples. Yes, there is something missing here, but it is missing in the examples, not just the code. We'll be writing more examples, and enhancing this code to get them to pass very soon. Promise.

Now run the feature again with the cucumber command, and we get some new feedback:

```
Scenarios: all colors correct
  | code    | guess   | mark |
  | r g y c | r g y c | bbbb |
  | r g y c | r g c y | bbww |
    expected ["Welcome to Mastermind!", "Enter guess:", "bbbb"] to include "bbww" (S...*TRUNC*
    ./08/features/step_definitions/mastermind.rb:28:in `/^the mark should be (.*)$/'
    08/features/codebreaker_submits_guess.feature:22:in `Then the mark should be bbw...*TRUNC*
```

The first scenario is passing now, but the second one is still failing. *Now* we'll add a new code example based on the next failing scenario. That wasn't too long, was it?

This time we'll submit a guess that has all the right colors, but only two in the correct positions.

Download mm/34/spec/mastermind/game_spec.rb

```
context "with all 4 colors correct and 2 in the correct places" do
  it "should mark the guess with bbww" do
    messenger = mock("messenger").as_null_object
    game = Game.new(messenger)
    game.start(%w[r g y c])

    messenger.should_receive(:puts).with("bbww")

    game.guess(%w[r g c y])
  end
end
```

Run that, and watch it fail:

```
Mastermind::Game
  starting up
    should send a welcome message
    should prompt for the first guess
  marking a guess
    with all 4 colors correct in the correct places
      should mark the guess with bbbb
    with all 4 colors correct and 2 in the correct places
      should mark the guess with bbww (FAILED - 1)

1)
Spec::Mocks::MockExpectationError in 'Mastermind::Game marking a guess with all 4 co...*TRUNC*
Mock 'messenger' expected :puts with ("bbww") but received it with ("bbbb")
./09/spec/mastermind/game_spec.rb:41:

Finished in 0.002539 seconds

4 examples, 1 failure
```

The trick now is to get this example to pass without causing the previous one to fail. One solution would be to start with an empty string and append a "b" for each peg that matches the color of the peg in the same position in the code, and a "w" for each peg that matches the color of a peg in a different position in the code. Once we've looked at all four pegs in the guess, we'll just puts() the string via the messenger.

To do that, we'll need to store the code in an instance variable in the start() method. Then we can access it in the guess() method.

Download mm/35/lib/mastermind/game.rb

```
def start(code)
  @code = code
```

```
  @messenger.puts "Welcome to Mastermind!"
  @messenger.puts "Enter guess:"
end

def guess(guess)
  result = ""
  guess.each_with_index do |peg, index|
    if @code[index] == peg
      result << "b"
    elsif @code.include?(peg)
      result << "w"
    end
  end
  @messenger.puts result
end
```

There are many ways we could express this, and this one may or may not be your favorite. If it's not, go with this for the duration of the tutorial, and then feel free to refactor later.

At this point all of the examples pass:

```
Mastermind::Game
  starting up
    should send a welcome message
    should prompt for the first guess
  marking a guess
    with all 4 colors correct in the correct places
      should mark the guess with bbbb
    with all 4 colors correct and 2 in the correct places
      should mark the guess with bbww

Finished in 0.002077 seconds

4 examples, 0 failures
```

Observe the impact on the scenario

Every time we get to a green bar in the code examples, it's a good idea to run the scenarios we're working on to see what impact we've had. In this case, we've not only gotten all of the examples to pass, but we've also made some progress on the scenarios. The second scenario is passing, but the third one is failing. Look closely at the failure message:

```
Scenarios: all colors correct
  | code    | guess   | mark |
  | r g y c | r g y c | bbbb |
  | r g y c | r g c y | bbww |
  | r g y c | y r g c | bwww |
    expected ["Welcome to Mastermind!", "Enter guess:", "wwwb"]
        to include "bwww" (Spec::Expectations::ExpectationNotMetError)
```

```
    ./35/features/step_definitions/mastermind.rb:28:in
        `/^the mark should be (.*)$/'
    35/features/codebreaker_submits_guess.feature:23:in
        `Then the mark should be bwww'
```

We're expecting "bwww", but we got "wwwb" instead. The implementation is returning the correct collection of letters, but in the wrong order. That's because the one peg that is in the correct position is the last peg that is evaluated. Let's create one more example based on this scenario:

Download mm/36/spec/mastermind/game_spec.rb

```
context "with all 4 colors correct and 1 in the correct place" do
  it "should mark the guess with bwww" do
    messenger = mock("messenger").as_null_object
    game = Game.new(messenger)
    game.start(%w[r g y c])

    messenger.should_receive(:puts).with("bwww")

    game.guess(%w[y r g c])
  end
end
```

Now run it and watch it fail:

```
Mastermind::Game
  starting up
    should send a welcome message
    should prompt for the first guess
  marking a guess
    with all 4 colors correct in the correct places
      should mark the guess with bbbb
    with all 4 colors correct and 2 in the correct places
      should mark the guess with bbww
    with all 4 colors correct and 1 in the correct place
      should mark the guess with bwww (FAILED - 1)

1)
Spec::Mocks::MockExpectationError in 'Mastermind::Game marking a guess with all 4 co...*TRUNC*
Mock 'messenger' expected :puts with ("bwww") but received it with ("wwwb")
./11/spec/mastermind/game_spec.rb:52:

Finished in 0.007508 seconds

5 examples, 1 failure
```

To get this example to pass, we'll need to put all the "b"s at the beginning of the mark. Since Ruby appends to Strings and to Arrays using the << method, we can just change the string to an array, sort it and join its elements:

Download mm/37/lib/mastermind/game.rb

```
def guess(guess)
  result = []
  guess.each_with_index do |peg, index|
    if @code[index] == peg
      result << "b"
    elsif @code.include?(peg)
      result << "w"
    end
  end
  @messenger.puts result.sort.join
end
```

With that subtle change, all of the code examples pass:

```
Mastermind::Game
  starting up
    should send a welcome message
    should prompt for the first guess
  marking a guess
    with all 4 colors correct in the correct places
      should mark the guess with bbbb
    with all 4 colors correct and 2 in the correct places
      should mark the guess with bbww
    with all 4 colors correct and 1 in the correct place
      should mark the guess with bwww

Finished in 0.002428 seconds

5 examples, 0 failures
```

As it turns out, that was all we needed to get all of the scenarios to pass as well:

```
13 scenarios
39 passed steps
```

4.4 Examples are Code Too

With all of the scenarios passing, now is a great time to review the code and sniff out some code smell. We want to look at all the code that we've written, including step definitions and code examples.

Refactoring step definitions

Here's the full content of features/step_definitions/mastermind.rb.

Download mm/37/features/step_definitions/mastermind.rb

```
Given /^I am not yet playing$/ do
end
```

```
Given /^the secret code is (. . . .)$/ do |code|
  @messenger = StringIO.new
  @game = Mastermind::Game.new(@messenger)
  @game.start(code.split)
end

When /^I guess (. . . .)$/ do |code|
  @game.guess(code.split)
end

When /^I start a new game$/ do
  @messenger = StringIO.new
  game = Mastermind::Game.new(@messenger)
  game.start(%w[r g y c])
end

Then /^the game should say "(.*)"$/ do |message|
  @messenger.string.split("\n").should include(message)
end

Then /^the mark should be (.*)$/ do |mark|
  @messenger.string.split("\n").should include(mark)
end
```

There's not only a fair amount of duplication here, but the code in the step definitions is not always as expressive as it could be. We can improve this quite a bit by extracting helper methods from the step definitions. After a few minutes refactoring, here's where we ended up:

Download mm/38/features/step_definitions/mastermind.rb

```
def messenger
  @messenger ||= StringIO.new
end

def game
  @game ||= Mastermind::Game.new(messenger)
end

def messages_should_include(message)
  messenger.string.split("\n").should include(message)
end

Given /^I am not yet playing$/ do
end

Given /^the secret code is (. . . .)$/ do |code|
  game.start(code.split)
end
```

```
When /^I guess (. . . .)$/ do |code|
  game.guess(code.split)
end

When /^I start a new game$/ do
  game.start(%w[r g y c])
end

Then /^the game should say "(.*)"$/ do |message|
  messages_should_include(message)
end

Then /^the mark should be (.*)$/ do |mark|
  messages_should_include(mark)
end
```

The messenger() and game() methods use a common Ruby idiom of initializing an instance variable if it does not exist, and returning the value of that variable like a standard accessor. The messages_should_include() method is an expressive wrapper for the expectation that is repeated in the Then steps.

In addition to making things more expressive, they are also more DRY. If we decide later to introduce a Messenger type, we'll only have to change that in one place.

Refactoring code examples

Now let's look at game_spec.rb:

Download mm/37/spec/mastermind/game_spec.rb

```
Line 1  require File.join(File.dirname(__FILE__), "/../spec_helper")
-
-   module Mastermind
-     describe Game do
5      context "starting up" do
-        before(:each) do
-          @messenger = mock("messenger").as_null_object
-          @game = Game.new(@messenger)
-        end
10
-       it "should send a welcome message" do
-         @messenger.should_receive(:puts).with("Welcome to Mastermind!")
-         @game.start(%w[r g y c])
-       end
15
-       it "should prompt for the first guess" do
-         @messenger.should_receive(:puts).with("Enter guess:")
-         @game.start(%w[r g y c])
-       end
```

```
20      end
-
-       context "marking a guess" do
-         context "with all 4 colors correct in the correct places" do
-           it "should mark the guess with bbbb" do
25            messenger = mock("messenger").as_null_object
-             game = Game.new(messenger)
-             game.start(%w[r g y c])
-
-             messenger.should_receive(:puts).with("bbbb")
30
-             game.guess(%w[r g y c])
-           end
-         end
-         context "with all 4 colors correct and 2 in the correct places" do
35         it "should mark the guess with bbww" do
-           messenger = mock("messenger").as_null_object
-           game = Game.new(messenger)
-           game.start(%w[r g y c])
-
40         messenger.should_receive(:puts).with("bbww")
-
-           game.guess(%w[r g c y])
-         end
-       end
45       context "with all 4 colors correct and 1 in the correct place" do
-         it "should mark the guess with bwww" do
-           messenger = mock("messenger").as_null_object
-           game = Game.new(messenger)
-           game.start(%w[r g y c])
50
-           messenger.should_receive(:puts).with("bwww")
-
-           game.guess(%w[y r g c])
-         end
55     end
-    end
-  end
```

The starting up context on line 5 includes a before() block that initializes the @messenger and @game instance variables. Looking further down, all of the examples in the marking a guess context use similar local variables.

We could have caught duplication earlier if we had stopped to look at the full listing. You may have already noticed it. In fact you may have already refactored it! But in case you haven't, here's what we did:

Download mm/38/spec/mastermind/game_spec.rb

```
Line 1  require File.join(File.dirname(__FILE__), "/../spec_helper")
-
-   module Mastermind
-     describe Game do
5      before(:each) do
-        @messenger = mock("messenger").as_null_object
-        @game = Game.new(@messenger)
-      end
-
10     context "starting up" do
-        it "should send a welcome message" do
-          @messenger.should_receive(:puts).with("Welcome to Mastermind!")
-          @game.start(%w[r g y c])
-        end
15
-        it "should prompt for the first guess" do
-          @messenger.should_receive(:puts).with("Enter guess:")
-          @game.start(%w[r g y c])
-        end
20     end
-
-      context "marking a guess" do
-        context "with all 4 colors correct in the correct places" do
-          it "should mark the guess with bbbb" do
25          @game.start(%w[r g y c])
-           @messenger.should_receive(:puts).with("bbbb")
-           @game.guess(%w[r g y c])
-         end
-       end
30     context "with all 4 colors correct and 2 in the correct places" do
-        it "should mark the guess with bbww" do
-          @game.start(%w[r g y c])
-          @messenger.should_receive(:puts).with("bbww")
-          @game.guess(%w[r g c y])
35      end
-      end
-      context "with all 4 colors correct and 1 in the correct place" do
-        it "should mark the guess with bwww" do
-          @game.start(%w[r g y c])
40        @messenger.should_receive(:puts).with("bwww")
-          @game.guess(%w[y r g c])
-        end
-      end
-    end
45  end
-  end
```

We moved the before(each) block to line 5, inside the outer-most describe() block. This makes it available to all of the nested groups, and therefore every example in this file.

Looking back at the three examples in the marking a guess context, they all look pretty much the same except for the values. It is *very* tempting to try to DRY this up, and there are a number of ways we could do it, but none of the ways we tried ended up as simple, expressive, localized, and clear as they are right now.

Remember that the code examples are, well, *code examples*! We want them to communicate on a few different levels. We want the docstrings we pass to the describe() and it() methods to tell a high level story of what it means to be a particular object. We want the code in the examples to show us how to use that object.

Now that we have enough working code to start interacting with the mastermind game, this would be a good time to start doing some *exploratory testing* to make sure the marking algorithm is complete. Here's a hint: it's not.

4.5 Exploratory Testing

Exploratory testing is exactly what it sounds like: testing through exploration of the app. Don't be fooled by the seeming haphazard nature of this. Google "exploratory testing" and you'll find upwards of 785,000 results. Testing is a craft of its own, rich with literature, experience, and community. But the details of exploratory testing are outside the scope of this book, and simply "exploring the app" will suffice for our needs.

Now that the mastermind game can mark a guess for us, we just need a minor adjustment to bin/mastermind and we can begin interacting with the game. Here's the script for *nix users:

Download mm/38/bin/mastermind

```
#!/usr/bin/env ruby
$LOAD_PATH.push File.join(File.dirname(__FILE__), "/../lib")
require 'mastermind'

game = Mastermind::Game.new(STDOUT)
game.start(%w[r g y c])
while guess = gets
  game.guess guess.split
end
```

Windows users use the same script without the first line, and also add bin/mastermind.bat with the following:

Download mm/37/bin/mastermind.bat

```
@"ruby.exe" "%~dpn0" %*
```

Clearly the game won't be too much fun because it's got the same code every time, but at least at this point you can show your friends. You know, the ones who know about CTRL-C, but don't know about cat.

Perhaps you're wondering why we'd want to do exploratory testing if we've already tested the app. Well, we haven't. Remember, that BDD is a design practice, not a testing practice. We're using executable examples of how we want the application to behave. But just as Big Design Up Front fails to allow for discovery of features and designs that naturally emerge through iterative development, so does driving out behaviour with examples fail to unearth all of the corner cases that we'll naturally discover by simply using the software.

As you explore the Mastermind game, try to find the flaws in the marking algorithm. You'll know what the not-so-secret code is, so try different inputs and see what happens. What happens when you guess rgyb, with no spaces? What happens when you use the same colors more than once in the secret code? In the guess? What happens when you type characters other than the prescribed r, g, y, c, b and w?

As you're doing this, flaws will appear for a variety of reasons. Perhaps there are missing scenarios or code examples. Some flaws may stem from naive design choices. The reasons for these flaws are not important. What *is* important is that the investment we've made to get this far has been very, very small compared to an exhaustive up-front requirements gathering process. An interactive session with working software is worth a thousand meetings.

4.6 What We Just Did

We began this chapter with a pre-existing set of passing scenarios, passing code examples, and working code. We then set out to extend the code to handle the next feature, submitting a guess to the system and having the system mark that guess.

In introducing this feature, we met some interesting challenges and learned some new techniques, including:

- Scenario outlines and tables to express like scenarios in a clean, readable way.

- Refactoring examples and code together when requirements change, so that we can embrace change *and* keep the bar green.
- Allowing some duplication between scenarios and code examples in order to maintain quick fault isolation and good object design.
- Doing the simplest thing to get a failing code example to pass, and channeling the urge to enhance the code into enhancing its specifications first.
- Refactoring step definitions and code examples, keeping in mind that expressiveness and localization are key factors in keeping them maintainable.
- Exploratory testing to weed out bugs and edge cases that we might not think of until we're actually interacting with the software.

Chapter 5

Evolving Existing Features

Welcome back! How was the exploratory testing? Did you discover anything missing from the marking algorithm? There are certainly edge cases that we missed. As you discovered each one, how did it make you feel? Did you feel frustrated that we had done so much work to produce something so incomplete? Or did you feel happy to make these discoveries before showing your friends? Or perhaps you *did* show your friends, and felt embarrassed when they discovered the bugs for you!

Now imagine that we had attempted to think through all of the edge cases before. We might have gotten further along than we did. This isn't a very complex problem compared to many, but the likelihood is that we'd not have thought of them all. And the companion likelihood is that we would have done pretty much the same amount of exploratory testing that we ended up doing. We just would have caught fewer faults.

So now that we *have* discovered some faults, let's feed that learning back into the process.

5.1 Adding New Scenarios

We're now at the beginning of a new iteration, and it's time to apply the lessons learned from exploratory testing. One flaw that emerged was that the marking algorithm does not handle duplicates properly when duplicate pegs in the guess match a single peg in the secret code.

Here's a Cucumber scenario that should help to clarify this:

Download mm/39/features/codebreaker_submits_guess.feature

```
Scenarios: dups in guess match color in code
  | code    | guess   | mark |
```

```
  | r y g c | r y g g | bbb  |
```

In this scenario, the first three pegs in the guess match the first three pegs in the code, so they each get a black marker peg. The fourth peg in the guess is the same color as the third peg. Since the third peg already earned a black mark for matching the third peg in the guess, the fourth peg should not get any mark at all.

Add this scenario to codebreaker_submits_guess.feature and run the feature with cucumber. Here's the output:

```
| r y g c | r y g g | bbb  |
  expected ["Welcome to Mastermind!", "Enter guess:", "bbbw"]
      to include "bbb" (Spec::Expectations::ExpectationNotMetError)
```

As you can see, the game is giving a black mark for the third peg in the guess, *and* a white mark for the fourth!

Here's another scenario we didn't account for, but learned about through exploratory testing:

Download mm/39/features/codebreaker_submits_guess.feature

```
Scenarios: dups in guess match color in code
  | code    | guess   | mark |
  | r y g c | r y g g | bbb  |
  | r y g c | r y c c | bbb  |
```

The second scenario is similar to the first, but this time the third and fourth pegs in the guess match the *fourth* peg in the secret code instead of the third. In this case, we want the game to give a black marker for the fourth peg in the guess and ignore the third. Here's what we get now:

```
| r y g c | r y c c | bbb  |
  expected ["Welcome to Mastermind!", "Enter guess:", "bbbw"]
      to include "bbb" (Spec::Expectations::ExpectationNotMetError)
```

This is the same result as the first scenario.

Here's a third scenario along the same lines, with a subtle difference. This time neither of the duplicate pegs are in the right position in the guess:

Download mm/39/features/codebreaker_submits_guess.feature

```
Scenarios: dups in guess match color in code
  | code    | guess   | mark |
  | r y g c | r y g g | bbb  |
  | r y g c | r y c c | bbb  |
  | r y g c | g y r g | bww  |
```

The first green peg in the guess gets a white mark because it matches a color in the code, but in the wrong position. The second peg in the guess, yellow, matches the code in color and position, so it gets a black mark. One black, one white.

The third peg in the guess, red, matches the first peg in the code, so it gets a white mark. One black, two white.

Finally, the last peg in the guess is green. While it matches the color of the third peg in the code, that peg was already accounted for by the first green peg in the guess. So this last peg in the guess does not get a mark at all. This leaves us with a mark of bww. At least that's what we expect. When we run the feature with this third new scenario, we see this in the output:

```
| r y g c | g y r g | bww  |
  expected ["Welcome to Mastermind!", "Enter guess:", "bwww"]
      to include "bww" (Spec::Expectations::ExpectationNotMetError)
```

5.2 Managing Increasing Complexity

We now have three brand new failing scenarios. Before this chapter we've been blithely cruising down the happy path, but we are now faced with sinister edge cases. And these new scenarios bring us to a whole new level of complexity in the marking algorithm.

Not only do we need to account for each peg as we see it, but we have to keep track of what we've already marked so that we don't mark against the same peg in the code twice. And to make things even more complex, when we hit a peg that we want to give a white mark, we need to see if it might earn a black mark later on!

OK. Settle down. We can do this. And not only can we do this, but we can do this calmly, rationally, and *safely*, by refactoring the existing design step by step.

The three new scenarios are failing due to logical errors, not syntax or structural errors. We don't need any new step definitions at this point. The ones we have are working just fine. We just need to improve the code, so let's head right to RSpec. Again, we'll start with an example mirroring the first failing Cucumber scenario:

Download mm/39/spec/mastermind/game_spec.rb

```
context "with duplicates in the guess that match a peg in the code" do
  context "by color and position" do
```

Lessons learned while coaching

In March, 2009, after we had released the second beta of this book, and before this chapter was written, I had the good fortune to run the first ever Chicago Ruby Brigade Coding Dojo. Given that I was working on this tutorial, I used the Mastermind marking algorithm as the problem to solve.

Much to everybody's joy, about twenty minutes into it, all of the scenarios (from the previous chapters) were passing. Much to everybody's dismay, that was all I had prepared for the session. So we decided to explore these more complex scenarios, and an interesting thing happened.

The first few scenarios had allowed for a very gradual increase in complexity. But when it came to these new scenarios, the leap was so severe that it was disorienting. We all, including me, lost sight of the task at hand, forgot the principles we were trying to put into practice, and launched into a theoretical algorithm problem-solving session.

The step by step approach described in this chapter is the antidote for that sort of blockage.

```
    it "should add a single b to the mark" do
      @game.start(%w[r y g c])
      @messenger.should_receive(:puts).with("bbb")
      @game.guess(%w[r y g g])
    end
  end
end
```

Which produces this failure:

```
1)
Spec::Mocks::MockExpectationError in 'Mastermind::Game \
  marking a guess with duplicates in the guess that \
  match a peg in the code by color and position should \
  add a single b to the mark'
Mock 'messenger' expected :puts with ("bbb") but received it with (["bbbw"])
./39/spec/mastermind/game_spec.rb:49:

Finished in 0.003503 seconds

6 examples, 1 failure
```

To get this to pass, we need to make sure that we don't add a white

mark for the last peg in the guess. One way we could do this is to keep track of each position in the secret code as we iterate through the guess. We would begin by changing result to an array of four nils. If the peg at guess[0] matches the peg at @code[0], then we replace result[0] with a b.

If not, then we check the code to see if the peg is anywhere in the code at all using @code.include?(peg). If it *is* in the code, then we discover its position with @code.index(peg). Here's where it gets interesting. We then ask the result if it still has nil in that position. If not, then we know it's already been marked! If so, then we replace it with a w.

Once we've done all that, we get rid of any remaining nils before sorting and joining the contents of the array to generate the mark.

Now this seems like a good plan, but the current design won't support it very easily. We'll need to do some refactoring *before* we can add this new capability. But there's a catch. We have a failing example right now, so we'd be refactoring *in the red*.

5.3 Refactoring In the Green

Here's a great guideline to follow: only refactor when all of your examples are passing. We call this refactoring *in the green*, because we rely on the green bar of passing examples to know that we're preserving behaviour as we go.

We just added a failing example. It's failing because the design doesn't support the new expected behaviour, not because a change to the code introduced a bug. We can therefore safely remove the example, refactor in the green, and then restore the example when we're done. But where do we put it in the meantime?

Many folks just comment the example out. *Do not do this!* There is little more unnerving than discovering an example that was commented out months ago and forgotten about. RSpec offers a better solution in the form of the pending() method.

Pending

To temporarily disable an example, but do so in such a way that you won't lose sight of it, add pending() to the example, like this:

Download mm/40/spec/mastermind/game_spec.rb

```
context "with duplicates in the guess that match a peg in the code" do
```

```
  context "by color and position" do
    it "should add a single b to the mark" do
      pending()
      @game.start(%w[r y g c])
      @messenger.should_receive(:puts).with("bbb")
      @game.guess(%w[r y g g])
    end
  end
end
```

When you run the specs you'll see something like this at the bottom of the output:

```
Pending:

Mastermind::Game marking a guess with duplicates in the \
  guess that match a peg in the code by color and position \
  should add a single b to the mark (TODO)
./spec/mastermind/game_spec.rb:48

6 examples, 0 failures, 1 pending
```

Now every time you run the examples, you'll be reminded that you have a pending example waiting for your attention. There are actually three different ways to use pending(). You can read about the others in detail in Section 10.2, *Pending Examples*, on page 113.

Refactor towards the new design

Our goal now is to change the design such that it passes the same code examples and scenarios, but better positions us to satisfy the new requirements. For the next little while, we're going to zip through changing this code very quickly in very small steps. Be sure to run the examples between every step. The reason we can move this quickly is that the green bar tells us that we're safe after every change we make. We'll discuss this more at the end of the chapter. Ready? Set? Go!

Step one:

Download mm/41/lib/mastermind/game.rb

```
Line 1  def guess(guess)
-         result = []
-         temp = [nil,nil,nil,nil]
-         guess.each_with_index do |peg, index|
5           if @code[index] == peg
-             result << "b"
-             temp[index] = "b"
-           elsif @code.include?(peg)
-             result << "w"
```

```
10        temp[index] = "w"
-       end
-     end
-     @messenger.puts result.sort.join
- end
```

Here we add a new array named temp on line 3, and then assign it “b” and “w” values at the current index on lines 7 and 10. This is preparatory work. Nothing in the calculation has changed yet.

Step two (did you run the specs first?):

Download mm/42/lib/mastermind/game.rb

```
Line 1  def guess(guess)
-       result = []
-       temp = [nil,nil,nil,nil]
-       guess.each_with_index do |peg, index|
5         if @code[index] == peg
-           result << "b"
-           temp[index] = "b"
-         elsif @code.include?(peg)
-           result << "w"
10          temp[index] = "w"
-         end
-       end
-       @messenger.puts temp.sort.join
- end
```

Here we simply replace the result with temp on line 13. Run the specs. Still passing!

Step 3: remove all the references to result.

Download mm/43/lib/mastermind/game.rb

```
def guess(guess)
  temp = [nil,nil,nil,nil]
  guess.each_with_index do |peg, index|
    if @code[index] == peg
      temp[index] = "b"
    elsif @code.include?(peg)
      temp[index] = "w"
    end
  end
  @messenger.puts temp.sort.join
end
```

Run the specs. Still passing. Step 4: change temp to result.

Download mm/44/lib/mastermind/game.rb

```
def guess(guess)
  result = [nil,nil,nil,nil]
```

```
  guess.each_with_index do |peg, index|
    if @code[index] == peg
      result[index] = "b"
    elsif @code.include?(peg)
      result[index] = "w"
    end
  end
  @messenger.puts result.sort.join
end
```

Run the specs. Still passing. Now reinstate the pending example by removing or commenting the pending() declaration and run them again. Same failure as before, but we've made progress! We're now interacting with the structure we want internally, and we haven't caused any of the earlier examples to fail.

Check the scenarios

It turns out, however, that all of the earlier code examples involve guessing all four colors in the secret code. The Cucumber scenarios had examples with three, two, one and even no colors correct, so let's run codebreaker_submits_guess.feature and see where we are.

```
Scenarios: 3 colors correct
  | code    | guess   | mark |
  | r g y c | w g y c | bbb  |
    undefined method `<=>' for nil:NilClass (NoMethodError)
    ./44/features/support/../../lib/mastermind/game.rb:23:in `sort'
    ./44/features/support/../../lib/mastermind/game.rb:23:in `guess'
    ./44/features/step_definitions/mastermind.rb:21:in
        `/^I guess (. . . .)$/'
    44/features/codebreaker_submits_guess.feature:28:in
        `When I guess w g y c'
```

There are actually quite a few scenarios failing like this, and they all report the same problem: undefined method `<=>' for nil:NilClass. This is happening because we're sorting all the elements in the array, but there are still going to be nils in there unless we guess all four colors.

Heading back to RSpec, restore the pending() declaration to the example of duplicates, and let's add a new example that matches this first scenario that we just broke:

Download mm/45/spec/mastermind/game_spec.rb

```
context "with three colors correct in the correct places" do
  it "should mark the guess with bbb" do
    @game.start(%w[r g y c])
    @messenger.should_receive(:puts).with("bbb")
    @game.guess(%w[r g y w])
```

```
  end
end
```

Run the examples and you'll see this in the output:

```
1)
ArgumentError in 'Mastermind::Game marking a guess with three colors \
  correct in the correct places should mark the guess with bbb'
comparison of String with nil failed
./45/spec/mastermind/../../lib/mastermind/game.rb:23:in `sort'
./45/spec/mastermind/../../lib/mastermind/game.rb:23:in `guess'
./45/spec/mastermind/game_spec.rb:49:
```

The comparison of String with nil failed message shows that the example fails in the way we expected. And the fix is simple:

Download mm/46/lib/mastermind/game.rb

```
Line 1  def guess(guess)
        -   result = [nil,nil,nil,nil]
        -   guess.each_with_index do |peg, index|
        -     if @code[index] == peg
      5         result[index] = "b"
        -     elsif @code.include?(peg)
        -       result[index] = "w"
        -     end
        -   end
     10     @messenger.puts result.compact.sort.join
        - end
```

We just compact the array before sorting and joining on line 10.

Now run the scenarios again, and you'll find that we have successfully refactored the design in order to better adapt to the new requirement. We know that the refactoring was successful because all of the previously passing scenarios are still passing.

But don't stop to bask in the glory just yet. We're not quite finished. While the previously passing scenarios and code examples are all passing, all three new scenarios are still failing. And if you get rid of the pending() declaration, you'll see that the new example is still failing.

Slide in the new change

Now that we've refactored towards the structure we want, and we've used the Cucumber scenarios to help us prove that out, it's time to add the logic that we discussed earlier. There are two things we still need to do:

- Put white marks in the position of the matching peg in the secret code instead of the current index.

- Check to see that the peg we're about to give a white mark for has not already been given a black mark.

Step one:

Download mm/47/lib/mastermind/game.rb

```
def guess(guess)
  result = [nil,nil,nil,nil]
  guess.each_with_index do |peg, index|
    if @code[index] == peg
      result[index] = "b"
    elsif @code.include?(peg)
      result[@code.index(peg)] = "w"
    end
  end
  @messenger.puts result.compact.sort.join
end
```

Run the specs and they all pass. Now remove the pending() declaration and run them again. You'll see that the example is still failing, but now it's failing differently. Instead of getting bbbw, we get bbw. Can you see why this is happening? See if you can before reading the next paragraph.

No, seriously.

The guess() method marks the first three pegs with a b, but then it replaces the third b with a w when it evaluates the fourth peg in the guess. So now we're only one step away from getting this to pass. And here is that step:

Download mm/48/lib/mastermind/game.rb

```
Line 1  def guess(guess)
-         result = [nil,nil,nil,nil]
-         guess.each_with_index do |peg, index|
-           if @code[index] == peg
5             result[index] = "b"
-           elsif @code.include?(peg)
-             result[@code.index(peg)] ||= "w"
-           end
-         end
10        @messenger.puts result.compact.sort.join
-       end
```

We simply change the assignment on line 7 to a conditional assignment using ||=. For those new to Ruby, this will only assign the value if it is currently nil or false.

Now run the specs. They all pass. Now run the feature. All the scenarios are passing.

5.4 What we just did

In this chapter we added new scenarios to an existing feature. We modified the code in baby steps, refactoring in the green nearly the whole time. This sort of approach can seem exhausting at first. We only changed a few lines of code, why not just change them all at once and be done with it?

The truth is that most experienced TDD practitioners will stray from this granularity and make changes in larger strokes, but they have the wisdom to recognize quickly whether those changes are heading them in the right direction. Once they do, they typically back up to the last known good state, run the examples and watch them pass, and then start again, but this time with smaller steps. But you've got to learn to walk before you can run.

You also learned how to temporarily disable an example with the pending() method. This is one of the many tools that RSpec includes to make the process of driving out code with examples a more pleasurable and productive one.

In the next and last chapter of this tutorial, we'll take the next step in making the Mastermind game real by driving out a random code generator with Cucumber and RSpec. Before we move on though, were there any other defects in the marking algorithm that you found in your own exploratory testing? If so, add some new scenarios to reflect them. If they fail, drive out the changes on your own, trying to stay in the green as much as possible.

Chapter 6

Random Expectations

Coming soon ...

Part II

Behaviour Driven Development

Chapter 7

The Case for BDD

Most of the software we write will never get used. It's nothing personal—it's just that as an industry we are not very good at giving people what they want. It turns out that the underlying reason for this is that traditional software methods are set up to fail—they actually work against us. Heroic individuals deliver software *in spite of* their development process rather than because of it. In this chapter we look at how and why projects fail, and shine a spotlight on some of the challenges facing Agile development.

7.1 How traditional projects fail

Traditional projects fail for all sorts of reasons. A good way to identify the different failure modes is to ask your project manager what keeps them up at night. (It's nice to do this from time to time anyway—it helps their self-esteem.) It is likely your project manager will come up with a list of fears similar to ours:

Delivering late or over budget

We estimate, we plan, we have every contingency down to the nth degree and then much to our disappointment real life happens. When we slip the first date no-one minds too much. I mean, it will only be a couple of weeks. If it goes on for long enough—slipping week by week and month by month—enough people will have left and joined that we can finally put the project out of its misery. Eighteen months to two years is usually enough. This is software that doesn't matter.

Delivering the wrong thing

Most of us use software that was delivered late and over budget—on our desktops, in our mobile phones, in our offices and homes. In fact we have become used to systems that update themselves with bugfixes and new features in the form of service packs and system updates, or websites that grow new features over time. But none of us use software that doesn't solve the problem we have.

It is surprising how much project management effort is spent looking after the schedule or budget when late software is infinitely more useful than irrelevant software.

So how does this happen? Maybe the requirements changed after we agreed to them because the business moved on. Perhaps they weren't clear enough in the first place. It might be that we delivered what the business asked for rather than what they needed. In any case we put a load of effort into delivering the project, within budget and on time, but it turns out no-one will actually get any benefit from it. This is software that doesn't matter.

Unstable in production

Hooray! The project came in on time and on budget, the users looked at it and decided they like it, so we put it into production. The problem is it crashes twice a day. We think it's a memory thing, or a configuration thing, or a clustering thing, or an infrastructure thing, or—but who are we kidding? We don't really know what's causing it except that it's rather embarrassing and it's costing us a lot of money. If only we'd spent more time testing it. People will use this once and then give up when it keeps crashing. This is software that doesn't matter.

Costly to maintain

There are a number of things we don't need to consider if we are writing disposable software. Maintainability is one of them. However if we expect to follow Release 1 with a Release 2, Release 3, or even a 2009 Professional Super Cow Power Edition then we can easily paint ourselves into a corner by not considering downstream developers.

For a start they probably weren't involved in the early releases and aren't privy to the decisions and conversations that led to the current design. If the code isn't obvious they will struggle to understand it. Similarly if the design isn't obvious—if there is lots of coupling or unnecessary redundancy, if lots of chunks were copied and pasted and

changed slightly—then they will struggle to work out the implications of the changes they make, which is a surefire way to introduce regression defects.

Over time the rate at which they can introduce new features will diminish until they end up spending more of their time tracking down unexpected regressions and unpicking spaghetti code than actually getting work done. At some point the software will cost more to improve than the revenue it can generate. This is software that doesn't matter.

7.2 Why traditional projects fail

Most of these failure modes happen with smart people trying to do good work. For the most part software people are diligent and well-intentioned, as are the stakeholders they are delivering to, which makes it especially sad when we see the inevitable "blame-storming" that follows in the wake of another failed delivery. It also makes it unlikely that project failures are the results of incompetence or inability—there must be another reason.

Perhaps these types of failure are an inevitable outcome of the approach we have been taking—the traditional or waterfall method of software delivery. No matter how smart or well-intentioned people are, things are set up for them to fail, and it is only by superhuman efforts that software gets delivered at all.

How traditional projects work

Most software projects go through the familiar sequence of Planning, Analysis, Design, Code, Test, Deploy. Your process may have different names but the basic activities in each phase will be fairly consistent. (We are assuming some sort of business justification has already happened, although even that isn't always the case.)

We start with the *Planning phase*: how many people, how long for, what resources will they need, basically how much will it cost to deliver this project and how soon will we see anything?

Then we move into an *Analysis phase*. This is where we articulate in detail the problem we are trying to solve, ideally without prescribing how it should be solved, although this is almost never the case.

Then we have a *Design phase*. This is where we think about how we can use a computer system to solve the problem we articulated in Analy-

sis. During this phase we think about design and architecture, large- and small-scale technical decisions, the various standards around the organization, and we gradually decompose the problem into manageable chunks for which we can produce functional specifications.

It appears to be a point of pride that—unless they are following a published methodology like *Prince 2*—each organization will have its own custom TLAs[1] to describe the documents that form the inputs and outputs of the Analysis and Design phases.

Now we move onto the *Coding phase*, where we write the software that is going to solve the problem, according to the specifications that came out of the Design phase. A common assumption by the program board at this stage is that it's all plain sailing from here because all the hard thinking has been done. This isn't as mean as it sounds—what they are saying is we have now made the activities of programming and testing relatively low risk because we did so much upfront planning, analysis and design. This is why so many organizations think it's ok to have their programming and testing carried out by offshore, third party vendors.

Now because we are responsible adults we have a *Testing phase* where we test the software to make sure it does what it was supposed to do. This phase contains activities with names like *User Acceptance Testing* or *Performance Testing* to emphasise that we are getting closer to the users now and the final delivery.

Eventually we reach the *Deployment phase* where we deploy the application into production. With a suitable level of fanfare the new software glides into production and starts making us money!

All these phases are necessary. You can't start solving a problem you haven't articulated; you can't start implementing a solution you haven't described; you can't test software that doesn't exist and you can't (or at least shouldn't) deploy software that hasn't been tested. Of course in reality you can do any of these things but it usually ends in tears.

How traditional projects really work

We have delivered projects in pretty much this way since we first started writing computer systems. There have been various attempts at improving the process and making it more efficient and less error-prone, using

1. TLA is a three-letter acronym meaning Three-Letter Acronym, and so is itself a TLA.

documents for formalised hand-offs, creating templates for the documents that make up those hand-offs, assembling review committees for the templates for the documents, establishing standards and formalised accreditation for the review committees . . . . You can certainly see where the effort has gone.

The reason for all this ceremony around hand-offs, reviews, and suchlike is that the later in the software delivery lifecycle we detect a defect—or introduce a change—the more expensive it is to put right. And not just a little more—in fact empirical evidence over the years has shown that it is exponentially more expensive the later you find out. With this in mind it makes sense to front-load the process. We want to make sure we have thought through all the possible outcomes and covered all the angles early on so we aren't surprised by "unknown unknowns" late in the day.

There are also, of course, the questions of accountability and responsibility when things do inevitably go wrong. In an organization with a traditional blame culture each group needs to be able to demonstrate that it wasn't their fault: the analysts, the architects, the programmers, testers, operations team and ultimately the project manager. They do that by getting a group of people to sign a declaration that an artifact—a project plan, a requirements document, a functional specification, some code—meets the appropriate level of assurance. If anything goes wrong now, it *must* be because of human error (i.e. incompetence, and more importantly *someone else's* incompetence) later in the process.

But this isn't the whole story. However diligent we are at each of the development phases, anyone who has delivered software in a traditional way will attest to the amount of work that happens "under the radar."

The program team signs off the project plan, resplendent in its detail, dependencies, resource models, and Gantt charts. Then the analysts start getting to grips with the detail of the problem and say things like: "hmm, this seems to be more involved than we thought. We'd better re-plan, this is going to be a biggie."

Then the architects start working on their functional specifications, which uncover a number of questions and ambiguities about the requirements. How does this data relate to that screen? What happens if this message isn't received by that other system? Sometimes the analysts can immediately answer the question but more often it means we need more analysis and hence more time from the analysts. Better update

that plan. And get it signed off. And sign off the new, enhanced requirements document.

You can see how this coordination cost can rapidly mount up. Of course it really kicks off during the testing phase. When the tester raises a defect, the programmer throws his hands in the air and says he did what was in the functional spec, the architect blames the business analyst, and so on right back up the chain. It's easy to see where this exponential cost comes from.

As this back-and-forth becomes more of a burden, we become more afraid of making changes, which means people do work outside of the process and documents get out of sync with one another and with the software itself. Testing gets squeezed, people work late into the night, and the release itself is usually characterized by wailing and gnashing of teeth, bloodshot eyes, and multiple failed attempts at decyphering the instructions in the release notes.

This is compounded by the fact that people typically work on one phase of a project and then move on, so by the time the tester is pointing out the defects the business analyst has long since joined a different project and is no longer available.

If you ask experienced software delivery folks why they run a project like that, front-loading it with all the planning and analysis, then getting into the detailed design and programming, and only really integrating and testing it at the end, they will gaze into the distance, looking older than their years, and patiently explain that this is to mitigate against the exponential cost of change. This *top-down* approach seems the only sensible way to hedge against the possibility of discovering a defect late in the day.

A self-fulfilling prophecy

To recap, projects become exponentially more expensive to change the further we get into them, due to the cumulative effect of keeping all the project artifacts in sync, so we front-load the process with lots of risk-mitigating planning, analysis and design activities to reduce the likelihood of rework.

Now, how many of these artifacts—the project plan, the requirements specification, the high- and low-level design documents, the software itself—existed before the project began? That's right, *exactly none!* So all that effort—that exponentially increasing effort—occurs *because we*

Bottom-up and Top-down development

In the 1990s, recognising that project failures were a fact of life, a different approach started to emerge, led by the technologists—the architects and programmers—rather than the process people. Instead of worrying about nailing down all the requirements they decided that *reuse* was the ultimate objective and set about constructing technical frameworks and component libraries that would do everything. That way the analysts would simply be pulling together existing business components (an Account Holder, an Account, an Order) and wiring them up to do new and clever things. They just needed to design comprehensive enough versions of all these business concepts and come up with sophisticated enough frameworks.

They called this approach *bottom-up* in contrast to the top-down method. They eventually discovered there is no such 'ultimate Order implementation' that would be all-encompassing, because we model orders differently in different contexts. This isn't accidental: there are some contexts in which certain things about an order are significant and others simply aren't. (Do I care about the contents of the order or just its value? Is currency significant? What are the business rules for audit and compliance? What about non-repudiation—do I need to prove who placed the order?) If we had an Order implementation that could cope with every eventuality it would be completely unworkable.

run projects the way we do! So now we have a chicken-and-egg situation—or a *reinforcing loop* in Systems Thinking terminology. The irony of the traditional project approach is that *the process itself causes the exponential cost of change!*

When we ask our project managers how they know about this exponential cost of change they tell us it is "through experience." They have seen enough projects in enough situations going through the same pain. Our industry's response to this has traditionally been to become better at reinforcing the loop rather than trying something that might break the cycle altogether. However software development is still a very young industry, so where did this cost curve come from in the first place?

Digging a little deeper, it turns out the curve originates in civil engineer-

ing. It makes sense that you might want to spend a lot of time in the design phases of a bridge or a ship. You wouldn't want to get two thirds of the way through building a hospital only to have someone point out it should be 20 metres to the left. Once the reinforced concrete pillars are sunk and the cast iron infrastructure is in place, things become very expensive to put right!

However, these rules only apply to software development *because we let them!* Software is, well, soft. It is supposed to be the part that's easy to change, and with the right approach and some decent tooling it can be very malleable. So by using the metaphor of civil engineering and equating software with steel and concrete, we've done ourselves a disservice.

7.3 Redefining the problem

It's not all doom and gloom though. There are many teams out there delivering projects on time, within budget, and delighting their stakeholders, and they manage to do it again and again. It's not easy. It takes discipline and dedication, and relies on a high degree of communication and collaboration, but it is possible. People who work like this tend to agree it is also a lot of fun!

Behaviour-driven development is one of a number of *Agile* methodologies. Specifically it is a *second generation* Agile methodology, building on the work of the really smart guys. Let's look at how these Agile methods came about and how they address traditional project risks, then we can see how BDD allows us to concentrate on writing software that matters.

A brief history of Agile

Since we first started delivering software as projects there has been a small but persistent community of software professionals asking themselves the same questions. Why do so many software projects fail? Why are we so *consistently* bad at delivering software? Why does it seem to happen more on larger projects with bigger teams? And can anything be done about it?

Independently they developed a series of lightweight methodologies whose focus was on delivering working software to users, rather than producing reams of documents or staging ceremonial reviews to show how

robust their processes were. They found they could cut through a lot of organizational red tape just by putting everyone in the same room.

Now this was a self-selecting group of people. Their irreverence for the status quo wasn't to everyone's taste, and the various practitioners found themselves described as "cowboys" or "amateurs" by the established order. How could they estimate the work without carrying out a comprehensive *function point analysis* first?[2] What madness was it to start programming without a complete set of detailed design specifications? Hadn't they heard of the exponential cost of change?

Then in early 2001 a few of these practitioners got together in a cabin in Snowbird, Utah, recognising that their various methodologies were more similar than they were different. They used different terminology and day-to-day practices but the underlying values were largely the same. They realized they could reach a much wider audience with a consistent brand—albeit with their own flavours to differentiate themselves—and they settled on the word "Agile."

They produced a short manifesto describing their common position. You might well have seen it before but it is worth reproducing here because it describes the common ground so perfectly.[3]

The Agile Manifesto

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions *over processes and tools*
Working software *over comprehensive documentation*
Customer collaboration *over contract negotiation*
Responding to change *over following a plan*

That is, while there is value in the things on the right, we value the things on the left more.

The Agile Manifesto is empirical—it's based on real experience: "We are uncovering better ways … *by doing it.*" Also notice that it doesn't *dismiss* traditional ideas like documentation and contracts—a criticism often levelled at Agile methods—but rather it expresses a preference for

2. Function point analysis is a measure of design complexity developed in the 1970s by IBM for the purpose of estimating software projects. You don't want to know how it works.
3. You can find the Agile Manifesto online at http://agilemanifesto.org

something different: something lighter weight and more directly relevant to the customer or stakeholder.

Following the manifesto's publication in 2001, the term "Agile" has been taken up across the industry and has started to become mainstream, with many blue chip organizations and large technology consultancies professing at least some Agile expertise. Whether this is due to them understanding and valuing the underlying principles of Agile or simply a marketing ruse to dress up the same old tired processes is left as an exercise for the reader.

How Agile methods address project risks

The authors of the manifesto go further than just the few lines quoted above. They also documented the principles underpinning their thinking. Central to these is a desire to "deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale."

Imagine for a moment you could do this, namely delivering production-quality software *every two weeks* to your stakeholders, on your current project, in your current organization, with your current team, starting tomorrow. How would this address the traditional delivery risks we outlined at the start of the chapter?

No longer delivering late or over budget

Since we are delivering the system in tiny, one- or two-week *iterations* or mini-projects, using a small, fixed-size team, it is easy to calculate our project budget: it is simply the burn rate of the team times the number of weeks, plus some hardware and licenses.

Provided we start with a reasonable guess at the overall size of the project, that is how much we are prepared to invest in solving the business problem in the first place, and we prioritise the features appropriately, then the team can deliver the really important stuff in the early iterations. (Remember, we are delivering by feature not by module.) So as we get towards the point when the money runs out, we should by definition be working on lower priority features. Also we can measure how much we actually produce in each iteration, known as our *velocity* or *throughput*, and use this to predict when we are really likely to finish.

If, as we approach the deadline, the stakeholders are still having new ideas for features and seeing great things happening, they may choose to fund the project for a further few iterations. Conversely they may

decide *before the deadline* that enough of the functionality has been delivered that they want to finish up early and get a release out. This is another option they have.

No longer delivering the wrong thing

We are delivering working software to the stakeholders every two weeks (say), which means we are delivering demonstrable features. We don't have a two week "database schema iteration" or "middleware iteration."

After each iteration we can demonstrate the new features to the stakeholders and they can make any tweaks or correct any misunderstandings while the work is still fresh in the development team's mind. These regular, small-scale micro-corrections ensure that we don't end up several months down the line with software that simply doesn't do what the stakeholders wanted.

To kick off the next iteration we can get together with the stakeholders to reassess the priorities of the features in case anything has changed since last time.[4] This means any new ideas or suggestions can get scheduled, and the corresponding amount of work can be descoped (or extra time added.)

No longer unstable in production

We are delivering every iteration, which means we have to get good at building and deploying the application. In fact we rely heavily on process automation to manage this for us. It is not uncommon for an experienced Agile team to produce over 100 good software builds every week.

In this context, releasing to production or testing hardware can be considered just another build to just another environment. Application servers are automatically configured and initialized; database schemas are automatically updated; code is automatically built, assembled and deployed over the wire; all manner of tests are automatically executed to ensure the system is behaving as expected.

In fact in an Agile environment, the relationship between the development team and the downstream operations and DBA folks is often much healthier and more supportive.

4. In practice the planning session often follows directly after the showcase for the previous iteration.

No longer costly to maintain

This last one is one of the biggest tangible benefits of an Agile process. After their first iteration the team is effectively in maintenance mode. They are adding features to a system that "works" so they have to be very careful.

Assuming they can solve the issues of safely changing existing code so as not to introduce regression defects, their working practices should be exactly the same as downstream support developers. It is not uncommon for an Agile development team to be working on several versions of an application simultaneously, adding features to the new version, providing early live support to a recently-released version, and providing bug fixing support to an older production version (because we still make mistakes, and the world still moves on!)

7.4 The cost of going Agile

So this is great news! By rethinking the way we approach project delivery we've managed to comprehensively address all our traditional project risks. Instead of seeing a project as a linear sequence of activities that ends up with a big delivery, we find things work better if we deliver frequently in short iterations. So why isn't everyone doing this?

The obvious but unpopular answer is: *because it's really hard!* Or rather, it's really hard to do well. Delivering production quality software week after week takes a lot of discipline and practice. For all their systemic faults, traditional software processes cause you to focus on certain aspects of a system at certain times. In an Agile process the training wheels come off and the responsibility now lies with you. That autonomy comes at a cost!

If we want to deliver working software frequently—as often as every week on many projects—there are a number of new problems we need to solve. Luckily Agile has been around for long enough that we have an answer to many of these problems, or at least understand them well enough to have an opinion about them. Let's look at some of the challenges of Agile, then we will see how BDD addresses them.

Outcome-based planning

The only thing we really know at the beginning of a project is that we don't know very much and that what we do know is subject to change. Much like steering a car, we know the rough direction but we don't

know every detailed nuance of the journey, such as exactly when we will turn the steering wheel or by how many degrees. We need to find a way to estimate the cost of delivering a project amongst all this uncertainty and accept that the fine details of the requirements are bound to change, and that that's ok.

Streaming requirements

If we want to deliver a few features every week or two we have to start describing requirements in a way that supports this. The traditional requirements process tends to be document-based, where the business analyst takes on the role of author and produces a few kilos of requirements.

Instead of this batch delivery of requirements we need to come up with a way to describe features that we can feed into a more streamlined delivery process.

Evolving design

In a traditional process the senior techies would come up with The Design (with audible capitals, most likely based on The Standards). Before we were allowed to start coding they would have produced high level designs, detailed designs and probably class diagrams describing every interaction. Each stage of this would be signed off. In an Agile world the design needs to flex and grow as we learn more about the problem and as the solution takes shape. This requires rethinking the process of software design.

Changing existing code

Traditional programming is like building little blocks for later assembly. We write a module and then put it to one side while we write the next one, and so on until all the modules are written. Then we bring all the modules together in a (usually painful) process called Integration. An Agile process requires us to keep revisiting the same code as we evolve it to do new things.

Because we take a feature-wise approach to delivery rather than a module-wise one, we will often need to add new behavour to existing code. This isn't because we got it "wrong" the first time, but because the code is currently exactly fit for purpose, and we need the application to do more now. *Refactoring*, the technique of restructuring code without changing its observable behaviour, is probably the place where

most advances have been made in terms of tool support and automation, especially with statically-typed languages like Java and C#.

Frequent code integration

Integrating code ahead of a testing cycle is a thankless and fraught task. All the individual modules "work"—just not together! Imagine doing this every single month? Or every week? What about potentially several times *every day*? This is the frequency of integration an iterative process demands: frequent enough that it is known as *continuous integration*.

Continual regression testing

Whenever we add a new feature it might affect many parts of the codebase. We are doing feature-wise development so different parts of the codebase are evolving at different rates, depending on the kind of feature we are implementing. When we have a single feature the system is easy to test. When we add the one hundredth feature we suddenly have to regression test the previous ninety-nine. Imagine when we add the two hundredth feature—or the one thousandth! We need to get really good at regression testing otherwise we will become ever slower at adding features to our application.

Frequent production releases

This is one of the hardest challenges of Agile software delivery, because it involves co-ordination with the downstream operations team. Things are suddenly outside of the team's control. All the other aspects: streaming requirements, changing design and code, frequent integration and regression testing, are behaviours we can adopt ourselves.

Getting software into formally-controlled environments puts us at odds with the corporate governance structures. But if we can't get into production frequently, there is arguably little value in all the other stuff. It may still be useful for the team's benefit, but software doesn't start making money until it's in production. Remember, we want to be writing software that matters!

Co-located team

To make this all work you can't afford for a developer to be waiting around for her manager to talk to someone else's manager to get permission for her to talk to them. The turnaround is just too slow. There

are organisational and cultural changes that need to happen in order to shorten the feedback cycles to minutes rather than days or weeks.

The kind of interactions we require involve the whole team sitting together, or at least as near one another as possible. It simply isn't effective to have the programmers in one office, the project managers in another and the testers elsewhere, whether along the corridor or in a different continent.

7.5 What have we learned?

There are a number of different ways in which traditional software projects fail, and these failures are intrinsic to the way the projects are run. The result of "process improvement" on traditional projects is simply to reinforce these failure modes and ironically make them even more likely.

An analysis of this approach to running software projects leads back to the exponential cost curve that originated in the world of civil engineering, where things are made of steel and concrete. Being aware of this, a number of individuals in the IT industry had been spending some time wondering what software delivery might look like if they ignored the constraints of thinking like civil engineers.

A group of software practitioners have suggested that taking an iterative, collaborative approach to software delivery could systemically eliminate the traditional risks that project managers worry about. They call this approach Agile.

It isn't all plain sailing, however, and adopting an Agile approach introduces a host of new challenges itself. There is no free lunch!

In the next chapter we look at how BDD responds to these challenges and allows us to concentrate on writing software that matters.

Chapter 8

Writing Software that Matters

Coming soon ...

Chapter 9

Mock Objects

Coming soon ...

Part III

RSpec

Chapter 10

Code Examples

In this part of the book, we'll explore the details of RSpec's built-in expectations, mock objects framework, command line tools, IDE integration, extension points, and even show you how to integrate RSpec with Test::Unit so that you can take advantage of the myriad extensions that are written for both frameworks.

Our goal is to make Test Driven Development a more joyful and productive experience with tools that elevate the design and documentation aspects of TDD to first class citizenship. Here are some words you'll need to know as we reach for that goal:

subject code The code whose behaviour we are specifying with RSpec.

expectation An expression of how the subject code is expected to behave. You'll read about state based expectations in Chapter 11, *Expectations*, on page 128, and interaction expectations in Chapter 12, *Mocking in RSpec*, on page 151.

code example An executable example of how the subject code can be used, and its expected behaviour (expressed with expectations) in a given context. In BDD, we write the code examples before the subject code they document.

The *example* terminology started with Brian Marick, whose website is even named http://exampler.com. Using "example" instead of "test" reminds us that the writing them is a design and documentation practice, even though once they are written and the code is developed against them they become regression tests.

example group A group of code examples.

Familiar structure, new nomenclature

If you already have some experience with Test::Unit or similar tools in other languages and/or TDD, the words we're using here map directly to words you're already familiar with:

- *Assertion* becomes *Expectation*.
- *Test Method* becomes *Code Example*.
- *Test Case* becomes *Example Group*.

In addition to finding these new names used throughout this book, you'll find them in RSpec's code base as well.

spec, a.k.a. **spec file** A file that contains one or more example groups.

In this chapter you'll learn how to organize executable *code examples* in *example groups* in a number of different ways, run arbitrary bits of code before and after each example, and even share examples across groups.

10.1 Describe It!

RSpec provides a Domain Specific Language for specifying the behaviour of objects. It embraces the metaphor of describing behaviour the way we might express it if we were talking to a customer, or another developer. A snippet of such a conversation might look like this:

You: *Describe a new account*

Somebody else: *It should have a balance of zero*

Here's that same conversation expressed in RSpec:

```
describe "A new Account" do
  it "should have a balance of 0" do
    account = Account.new
    account.balance.should == Money.new(0, :USD)
  end
end
```

We use the describe() method to define an example group. The string we pass to it represents the facet of the system that we want to describe (a new account). The block holds the code examples that make up that group.

Joe Asks...

Where are the objects?

The declarative style we use to create code examples in example groups is designed to keep you focused on documenting the expected behaviour of an application.

While this works quite well for many, there are some who find themselves distracted by the opacity of this style. If you fall in the latter category, or if you are looking to write custom extensions,* you may want to know what the underlying objects are.

The describe() method creates a subclass of Spec::Example::ExampleGroup. The it() method defines a method on that class, which represents a code example.

While we don't recommend it, it is possible to write code examples in example groups using classes and methods. Here is the new account example expressed that way:

```
class NewAccount < Spec::Example::ExampleGroup
  def should_have_a_balance_of_zero
    account = Account.new
    account.balance.should == Money.new(0, :USD)
  end
end
```

RSpec interprets any method that begins with "should_" to be a code example.

*. See Section 15.2, *Custom Example Groups*, on page 179 to learn about writing custom example group classes.

The it() method defines a code example. The string passed to it describes the specific behaviour we're interested in specifying about that facet (should have a balance of zero). The block holds the example code that exercises the subject code and sets expectations about its behaviour.

Using strings like this instead of legal Ruby class names and method names provides a lot of flexibility. Here's an example from RSpec's own code examples:

```
it "should match when value < (target + delta)" do
  be_close(5.0, 0.5).matches?(5.49).should be_true
end
```

This is an example of the behaviour of code, so the intended audience is

someone who can read code. In Test::Unit, we might name the method test_should_match_when_value_is_less_than_target_plus_delta, which is pretty readable, but the ability to use non-alpha-numeric characters makes the name of this example more expressive.[1]

To get a better sense of how you can unleash this expressiveness, let's take a closer look at the describe() and it() methods.

The describe() method

The describe() method can take an arbitrary number of arguments and a block, and returns a subclass of Spec::Example::ExampleGroup.[2] We generally only use one or two arguments, which represent the facet of behaviour that we wish to describe. They might describe an object, perhaps in a pre-defined state, or perhaps a subset of the behaviour we can expect from that object. Let's look at a few examples, with the output they produce so we can get an idea of how the arguments relate to each other.

```
describe "A User" { ... }
=> A User

describe User { ... }
=> User

describe User, "with no roles assigned" { ... }
=> User with no roles assigned

describe User, "should require password length between 5 and 40" { ... }
=> User should require password length between 5 and 40
```

The first argument can be either a reference to a Class or Module, or a String. The second argument is optional, and should be a String. Using the class/module for the first argument provides an interesting benefit: when we wrap the ExampleGroup in a module, we'll see that module's name in the output. For example, if User is in the Authentication module, we could do something like this:

```
module Authentication
  describe User, "with no roles assigned" do
```

The resulting report would look like this:

1. activesupport-2.2 introduced support for test "a string" do...end syntax, so you can get the basic benefit of strings out of the box in rails-2.2 or later.
2. As you'll see later in Chapter 15, *Extending RSpec*, on page 177, you can coerce the describe() method to return your own custom ExampleGroup subclass.

```
Authentication::User with no roles assigned
```

So by wrapping the ExampleGroup in a Module, we see the fully qualified name Authentication::User, followed by the contents of the second argument. Together, they form a descriptive string, and we get the fully qualified name for free. This is a nice way to help RSpec help *us* to understand where things live as we're looking at the output.

You can also nest example groups, which can be a very nice way of expressing things in both input and output. For example, we can nest the input like this:

```
describe User do
  describe "with no roles assigned" do
    it "should not be allowed to view protected content" do
```

This produces output like this:

```
User with no roles assigned
- should not be allowed to view protected content
```

Or, with the --nested flag on the command line, the output looks like this:

```
User
  with no roles assigned
    should not be allowed to view protected content
```

To understand this example better, let's explore describe()'s yang, the it() method.

What's it() all about?

Similar to describe(), the it() method takes a single String, an optional Hash and an optional block. The String should be a sentence that, when prefixed with "it," represents the detail that will be expressed in code within the block. Here's an example specifying a stack:

```
describe Stack do
  before(:each) do
    @stack = Stack.new
    @stack.push :item
  end

  describe "#peek" do
    it "should return the top element" do
      @stack.peek.should == :item
    end

    it "should not remove the top element" do
      @stack.peek
```

```
      @stack.size.should == 1
    end
  end

  describe "#pop" do
    it "should return the top element" do
      @stack.pop.should == :item
    end

    it "should remove the top element" do
      @stack.pop
      @stack.size.should == 0
    end
  end
end
```

This is also exploiting RSpec's nested example groups feature to group the examples of pop() separately from the examples of peek().

When run with the --format nested command line option, this would produce the following output.

```
Stack
  #peek
    should return the top element
    should not remove the top element
  #pop
    should return the top element
    should remove the top element
```

Looks a bit like a specification, doesn't it? In fact, if we reword the example names without the word "should" in them, we can get output that looks even more like documentation:

```
Stack
  #peek
    returns the top element
    does not remove the top element
  #pop
    returns the top element
    removes the top element
```

What? No "should?" Remember, the goal here is readable sentences. "Should" was the tool that Dan used to get people writing sentences, but is not itself essential to the goal.

The ability to pass free text to the it() method allows us to name and organize examples in meaningful ways. As with describe(), the String can even include punctuation. This is a good thing, especially when

we're dealing with code-level concepts in which symbols have important meaning that can help us to understand the intent of the example.

10.2 Pending Examples

In *Test Driven Development: By Example* [Bec02], Kent Beck suggests keeping a list of tests that you have yet to write for the object you're working on, crossing items off the list as you get tests passing, and adding new tests to the list as you think of them.

With RSpec, you can do this right in the code by calling the it() method with no block. Let's say that we're in the middle of describing the behaviour of a Newspaper:

```
describe Newspaper do
  it "should be black" do
    Newspaper.new.colors.should include('black')
  end

  it "should be white" do
    Newspaper.new.colors.should include('white')
  end

  it "should be read all over"
end
```

RSpec will consider the example with no block to be pending. Running these examples produces the following output

```
Newspaper
- should be black
- should be white
- should be read all over (PENDING: Not Yet Implemented)

Pending:
Newspaper should be read all over (Not Yet Implemented)
  Called from newspaper.rb:20

Finished in 0.006682 seconds

3 examples, 0 failures, 1 pending
```

As you add code to existing pending examples and add new ones, each time you run all the examples RSpec will remind you how many pending examples you have, so you always know how close you are to being done!

Another case for marking an example pending is when you're in the middle of driving out an object, you've got some examples passing and you add a new failing example. You look at the code, see the change you want to make and realize that the design really doesn't support what you want to do to make this example pass.

There are a couple of different paths people choose at this juncture. One is to comment out the failing example so you can refactor *in the green*, and then uncomment the example and continue on. This works great until you're interrupted in the middle of this near the end of the day on Friday, and 3 months later you look back at that file and find examples you commented out three months ago.

Instead of commenting the example out, you can mark it pending like this:

```
describe "onion rings" do
  it "should not be mixed with french fries" do
    pending "cleaning out the fryer"
    fryer_with(:onion_rings).should_not include(:french_fry)
  end
end
```

In this case, even though the example block gets executed, it stops execution on the line with the pending() declaration. The subsequent code is not run, there is no failure, and the example is listed as pending in the output, so it stays on your radar. When you've finished refactoring you can remove the pending declaration to execute the code example as normal. This is, clearly, much better than commenting out failing examples and having them get lost in the shuffle.

The third way to indicate a pending example can be quite helpful in handling bug reports. Let's say you get a bug report and the reporter is kind enough to provide a failing example. Or you create a failing example yourself to prove the bug exists. You don't plan to fix it this minute, but you want to keep the code handy. Rather than commenting the code, you could use the pending() method to keep the failing example from being executed.

You can also, however, wrap the example code in a block and pass that to the pending method, like this:

```
describe "an empty array" do
  it "should be empty" do
    pending("bug report 18976") do
      [].should be_empty
    end
```

```
  end
end
```

When RSpec encounters this block it actually executes the block. If the block fails or raises an error, RSpec proceeds as with any other pending example.

If, however, the code executes without incident, RSpec raises a PendingExampleFixedError, letting you know that you've got an example that is pending for no reason:

```
an empty array
- should be empty (ERROR - 1)

1)
'an empty array should be empty' FIXED
Expected pending 'bug report 18976' to fail. No Error was raised.
pending_fixed.rb:6:
pending_fixed.rb:4:

Finished in 0.007687 seconds

1 example, 1 failure
```

The next step is to remove the pending wrapper, and re-run the examples with your formerly-pending, newly-passing example added to the total of passing examples.

So now you know three ways to identify pending examples, each of which can be helpful in your process in different ways:

- add pending examples as you think of new examples that you want to write
- disable examples without losing track of them (rather than commenting them out)
- wrap failing examples when you want to be notified that changes to the system cause them to pass

So now that you know how to postpone writing examples, let's talk about what happens when you actually write some!

10.3 Before and After

If we were developing a Stack, we'd want to describe how a Stack behaves when it is empty, almost empty, almost full, and full. And we'd

want to describe how the push(), pop(), and peek() methods behave under each of those conditions.

If we multiply the 4 states by the 3 methods, we're going to be describing 12 different scenarios that we'll want to group together by either state or method. We'll talk about grouping by method in the (as yet) unwritten *sec.organizingExamples*. Right now, let's talk about grouping things by *Initial State*, using RSpec's before() method.

before(:each)

To group examples by initial state, or *context*, RSpec provides a before() method that can run either one time before :all the examples in an example group or once before :each of the examples. In general, it's better to use before(:each) because that re-creates the context before each example and keeps state from leaking from example to example. Here's how this might look for the Stack examples:

Download describeit/stack.rb

```
describe Stack, "when empty" do
  before(:each) do
    @stack = Stack.new
  end
end

describe Stack, "when almost empty (with one element)" do
  before(:each) do
    @stack = Stack.new
    @stack.push 1
  end
end

describe Stack, "when almost full (with one element less than capacity)" do
  before(:each) do
    @stack = Stack.new
    (1..9).each { |n| @stack.push n }
  end
end

describe Stack, "when full" do
  before(:each) do
    @stack = Stack.new
    (1..10).each { |n| @stack.push n }
  end
end
```

As we add examples to each of these example groups, the code in the block passed to before(:each) will be executed before each example is

executed, putting the environment in the same known starting state before each example in that group.

before(:all)

In addition to before(:each), we can also say before(:all). This gets run once and only once in its own instance of Object, but its instance variables get copied to each instance in which the examples are run. A word of caution in using this: in general, we want to have each example run in complete isolation from one another. As soon as we start sharing state across examples, unexpected things begin to happen.

Consider a stack. The pop() method removes the top item from a stack, which means that the second example that uses the same stack *instance* is starting off with a stack that has one less item than in the before(:all) block. When that example fails, this fact is going to make it more challenging to understand the failure.

Even if it seems to you that sharing state won't be a problem right now in any given example, this is sure to change over time. Problems created by sharing state across examples are notoriously difficult to find. If we have to be debugging at all, the last thing we want to be debugging is the examples.

So what is before(:all) actually good for? One example might be opening a network connection of some sort. Generally, this is something we wouldn't be doing in the isolated examples that RSpec is really aimed at. If we're using RSpec to drive higher level examples, however, then this might be a good case for using before(:all).

after(:each)

Following the execution of each example, before(:each)'s counterpart after(:each) is executed. This is rarely necessary because each example runs in its own scope and the instance variables consequently go out of scope after each example.

There are cases, however, when after(:each) can be quite useful. If you're dealing with a system that maintains some global state that you want to modify just for one example, a common idiom for this is to set aside the global state in an instance variable in before(:each) and then restore it in after(:each), like this:

```
before(:each) do
  @original_global_value = $some_global_value
  $some_global_value = temporary_value
```

```
end

after(:each) do
  $some_global_value = @original_global_value
end
```

after(:each) is guaranteed to run after each example, even if there are failures or errors in any before blocks or examples, so this is a safe approach to restoring global state.

after(:all)

We can also define some code to be executed after(:all) of the examples in an example group. This is even more rare than after(:each), but there are cases in which it is justified. Examples include closing down browsers, closing database connections, closing sockets, etc. Basically, any resources that we want to ensure get shut down, but not after every example.

So we've now explored before and after :each and before and after :all. These methods are very useful in helping to organize our examples by removing duplication—not just for the sake of removing duplication but with the express purpose of improving clarity and thereby making the examples easier to understand.

But sometimes we want to share things across a wider scope. The next two sections will address that problem by introducing Helper Methods and Shared Examples.

10.4 Helper Methods

Another approach to cleaning up our examples is to use Helper Methods that we define right in the example group, which are then accessible from all of the examples in that group. Imagine that we have several examples in one example group, and at one point in each example we need to perform some action that is somewhat verbose.

```
describe Thing do
  it "should do something when ok" do
    thing = Thing.new
    thing.set_status('ok')
    thing.do_fancy_stuff(1, true, :move => 'left', :obstacles => nil)
    ...
  end

  it "should do something else when not so good" do
```

```
    thing = Thing.new
    thing.set_status('not so good')
    thing.do_fancy_stuff(1, true, :move => 'left', :obstacles => nil)
    ...
  end
end
```

Both examples need to create a new Thing and assign it a status. This can be extracted out to a helper like this:

```
describe Thing do
  def create_thing(options)
    thing = Thing.new
    thing.set_status(options[:status])
    thing
  end

  it "should do something when ok" do
    thing = create_thing(:status => 'ok')
    thing.do_fancy_stuff(1, true, :move => 'left', :obstacles => nil)
    ...
  end

  it "should do something else when not so good" do
    thing = create_thing(:status => 'not so good')
    thing.do_fancy_stuff(1, true, :move => 'left', :obstacles => nil)
    ...
  end
end
```

One idiom you can apply to clean this up even more is to yield self from initializers in your objects. Assuming that Thing's initialize() method does this, and set_status() does as well, you can write the above like this:

```
describe Thing do
  def given_thing_with(options)
    yield Thing.new do |thing|
      thing.set_status(options[:status])
    end
  end

  it "should do something when ok" do
    given_thing_with(:status => 'ok') do |thing|
      thing.do_fancy_stuff(1, true, :move => 'left', :obstacles => nil)
      ...
    end
  end

  it "should do something else when not so good" do
    given_thing_with(:status => 'not so good') do |thing|
      thing.do_fancy_stuff(1, true, :move => 'left', :obstacles => nil)
      ...
```

```
    end
  end
end
```

Obviously, this is a matter of personal taste, but you can see that this cleans things up nicely, reducing the noise level in each of the examples. Of course, with almost all benefits come drawbacks. In this case, the drawback is that we have to look elsewhere to understand the meaning of given_thing_with. This sort of indirection can make understanding failures quite painful when overused.

A good guideline to follow is to keep things consistent within each code base. If all of the code examples in your system look like the one above, even your new team mates who might not be familiar with these idioms will quickly learn and adapt. If there is only one example like this in the entire codebase, then that might be a bit more confusing. So as you strive to keep things clean, be sure to keep them consistent as well.

Sharing Helper Methods

If we have helper methods that we wish to share across example groups, we can define them in one or more modules and then include the modules in the example groups we want to have access to them.

```
module UserExampleHelpers
  def create_valid_user
    User.new(:email => 'e@mail.com', :password => 'shhhhh')
  end

  def create_invalid_user
    User.new(:password => 'shhhhh')
  end
end

describe User do
  include UserExampleHelpers

  it "does something when it is valid" do
    user = create_valid_user
    # do stuff
  end

  it "does something when it is not valid" do
    user = create_invalid_user
    # do stuff
  end
```

If we have a module of helper methods that we'd like available in all of our example groups, we can include the module in the configuration

(see Section 15.1, *Global Configuration*, on page 177 for more information):

```
Spec::Runner.configure do |config|
  config.include(UserExampleHelpers)
end
```

So now that we can share helper methods across example groups, how about sharing *examples*?

10.5 Shared Examples

When we have a situation in which more than one class should behave in *exactly* the same way, we can use a shared example group to describe it once, and then include that example group in other example groups. We declare a shared example group with the shared_examples_for() method.

```
shared_examples_for "Any Pizza" do
  it "should taste really good" do
    @pizza.should taste_really_good
  end
  it "should be available by the slice" do
    @pizza.should be_available_by_the_slice
  end
end
```

Once a shared example group is declared, we can include it in other example groups with the it_should_behave_like() method.

```
describe "New York style thin crust pizza" do
  it_should_behave_like "Any Pizza"

  before(:each) do
    @pizza = Pizza.new(:region => 'New York', :style => 'thin crust')
  end

  it "should have a really great sauce" do
    @pizza.should have_a_really_great_sauce
  end
end

describe "Chicago style stuffed pizza" do
  it_should_behave_like "Any Pizza"

  before(:each) do
    @pizza = Pizza.new(:region => 'Chicago', :style => 'stuffed')
  end

  it "should have a ton of cheese" do
    @pizza.should have_a_ton_of_cheese
```

```
  end
end
```

which produces:

```
New York style thin crust pizza
- should taste really good
- should be available by the slice
- should have a really great sauce

Chicago style stuffed pizza
- should taste really good
- should be available by the slice
- should have a ton of cheese
```

This report does not include "Any Pizza", but the "Any Pizza" examples, "should taste really good" and "should be available by the slice" do appear in both of the other example groups. Also, @pizza is referenced in the shared examples before they get included in the others. Here's why that works. At runtime, the shared examples are stored in a collection and then copied into each example group that uses them. They aren't actually executed until the example group that uses them gets executed, but that happens after before(:each) happens.

This example also hints at a couple of other features that RSpec brings us to help make the examples as expressive as possible: Custom Expectation Matchers and Arbitrary Predicate Matchers. These will be explained in detail in later chapters, so if you haven't skipped ahead to read about them yet, consider yourself teased.

Sharing Examples in a Module

In addition to share_examples_for() and it_should_behave_like(), you can also use the share_as method, which assigns the group to a constant so you can include it using Ruby's include method, like this:

```
share_as :AnyPizza do
  ...
end

describe "New York style thin crust pizza" do
  include AnyPizza
  ...
end

describe "Chicago style stuffed pizza" do
  include AnyPizza
  ...
end
```

This leads to the same result as share_examples_for() and it_should_behave_like(), but allows you to use the familiar Ruby syntax instead.

Even with both of these approaches, shared examples are very limited in nature. Because the examples are run in the same scope in which they are included, the only way to share state between them and other examples in the including group is through instance variables. You can't just pass state to the group via the it_should_behave_like method.

Because of this constraint, shared examples are really only useful for a limited set of circumstances. When you want something more robust, we recommend that you create custom macros, which we'll discuss at length in Chapter 15, *Extending RSpec*, on page 177.

10.6 Nested Example Groups

Nesting example groups is a great way to organize your examples within one spec. Here's a simple example:

```
describe "outer" do
  describe "inner" do
  end
end
```

As we discussed earlier in this chapter, the outer group is a subclass of ExampleGroup. In this example, the inner group is a *subclass of the outer group*. This means that any helper methods and/or before and after declarations, included modules, etc declared in the outer group are available in the inner group.

If you declare before and after blocks in both the inner and outer groups, they'll be run as follows:

1. outer before
2. inner before
3. example
4. inner after
5. outer after

To demonstrate this, copy this into a ruby file:

```
describe "outer" do
  before(:each) { puts "first" }
  describe "inner" do
```

```
    before(:each) { puts "second" }
    it { puts "third"}
    after(:each) { puts "fourth" }
  end
  after(:each) { puts "fifth" }
end
```

If you run that with the spec command, you should see output like this:

```
first
second
third
fourth
fifth
```

Because they are all run in the context of the same object, you can share state across the before blocks and examples. This allows you to do a progressive setup. For example, let's say you want to express a given in the outer group, an event (or *when*) in the inner group, and the expected outcome in the examples themselves. You could do something like this:

```
describe Stack do
  before(:each) do
    @stack = Stack.new(:capacity => 10)
  end
  describe "when full" do
    before(:each) do
      (1..10).each {|n| @stack.push n}
    end
    describe "when it receives push" do
      it "should raise an error" do
        lambda { @stack.push 11 }.should raise_error(StackOverflowError)
      end
    end
  end
  describe "when almost full (one less than capacity)"
    before(:each) do
      (1..9).each {|n| @stack.push n}
    end
    describe "when it receives push" do
      it "should be full" do
        @stack.push 10
        @stack.should be_full
      end
    end
  end
end
```

Now, I can imagine some of you thinking "w00t! Now *that* is DRY!" while others think "Oh my god, it's so complicated!" I, personally, sit in the

latter camp, and tend to avoid structures like this, as they can make it very difficult to understand failures. But in the end you have to find what works for you, and this structure is one option that is available to you. Handle with care.

I *do*, however, use nested example groups all the time. I just tend to use them to organize concepts rather than build up state. So I'd probably write the example above like this:

```
describe Stack do
  describe "when full" do
    before(:each) do
      @stack = Stack.new(:capacity => 10)
      (1..10).each {|n| @stack.push n}
    end
    describe "when it receives push" do
      it "should raise an error" do
        lambda { @stack.push 11 }.should raise_error(StackOverflowError)
      end
    end
  end
  describe "when almost full (one less than capacity)"
    before(:each) do
      @stack = Stack.new(:capacity => 10)
      (1..9).each {|n| @stack.push n}
    end
    describe "when it receives push" do
      it "should be full" do
        @stack.push 10
        @stack.should be_full
      end
    end
  end
end
```

In fact, there are many who argue that you should never use the before blocks to build up context at all. Here's the same example:

```
describe Stack do
  describe "when full" do
    describe "when it receives push" do
      it "should raise an error" do
        stack = Stack.new(:capacity => 10)
        (1..10).each {|n| stack.push n}
        lambda { stack.push 11 }.should raise_error(StackOverflowError)
      end
    end
  end
  describe "when almost full (one less than capacity)"
    describe "when it receives push" do
      it "should be full" do
```

```
        stack = Stack.new(:capacity => 10)
        (1..9).each {|n| stack.push n}
        stack.push 10
        stack.should be_full
      end
    end
  end
end
```

Now this is probably the most readable of all three examples. The nested describe blocks provide documentation and conceptual cohesion, and each example contains all of the code it needs. The great thing about this approach is that if you have a failure in one of these examples, you don't have to look anywhere else to understand it. It's all right there.

On the flip side, this is the least DRY of all three examples. If we change the Stack's constructor, we'll have to change it in two places here, and many more in a complete example. So you need to balance these concerns. Sadly, there's no one true way. And if there were, we'd all be looking for new careers, so let's be glad for the absence of the silver bullet.

What you've learned

In this chapter we covered quite a bit about the approach RSpec takes to structuring and organizing executable code examples. You learned that you can:

- Define an example group using the describe() method
- Define an example using the it() method
- Identify an example as *pending* by either omiting the block or using the pending() method inside the block
- Share state across examples using the before() method
- Define helper methods within an example group that are available to each example in that group
- Share examples across multiple groups
- Nest example groups for cohesive organization

But what about the stuff that goes inside the examples? We've used a couple of expectations in this chapter but we haven't really discussed them. The next chapters will address these lower level details, as well

as introduce some of the peripheral tooling that is available to help you nurture your inner BDD child and evolve into a BDD ninja.

Chapter 11

Expectations

A major goal of BDD is *getting the words right*. We're trying to derive language, practices, and processes that support communication between all members of a team, regardless of each person's understanding of things technical. This is why we like to use non-technical words like *Given*, *When* and *Then*.

We also like to talk about *expectations* instead of *assertions*. The dictionary defines the verb "to assert" as "to state a fact or belief confidently and forcefully." This is something we do in a courtroom. We assert that it was *Miss Peacock* in the *kitchen* with a *rope* because that's what we believe to be true.

In executable code examples, we are describing an expectation of what *should* happen rather than what *will* happen, so we choose the word *should*.[1] Having chosen "should", we have another problem to solve: where do we put it? Consider the following assertion from test/unit.

```
assert_equal 5, result
```

In this example, assert_equal() accepts the expected value followed by the actual value. Now read that aloud: "Assert equal five result." A little bit cryptic, no? So now do what we normally do when reading code out loud and insert the missing words: "Assert that five equals the result." That's a bit better, but now that we're speaking in English, we see another problem. We don't really want to "assert that five equals result." We want to "assert that the result equals five!" The arguments are backwards!

1. See the (as yet) unwritten *chp.writingSoftwareThatMatters* for more on the motivations behind *should*.

RSpec addresses the resulting confusion by exploiting Ruby's meta-programming facilities to provide a syntax that speaks the way we do. What we want to say is that "the result should equal five." Here's how we say it in English:

```
the result should equal 5
```

And here's how we say it in RSpec:

```
result.should equal(5)
```

Read that out loud. In fact, climb up on the roof and cry out to the whole town!!! Satisfying, isn't it?

This is an example of an RSpec *expectation*, a statement which expresses that at a specific point in the execution of a code example, some thing should be in some state. Here are some other expectations that come with RSpec:

```
message.should match(/on Sunday/)
team.should have(11).players
lambda { do_something_risky }.should raise_error(
  RuntimeError, "sometimes risks pay off ... but not this time"
)
```

Don't worry about understanding them fully right now. In this chapter you'll learn about all of RSpec's built-in expectations. You'll also learn about the simple framework that RSpec uses to express expectations, which you can then use to extend RSpec with your own domain-specific expectations. With little effort, you'll be able to express things like:

```
judge.should disqualify(participant)
registration.should notify_applicant("person@domain.com", /Dear Person/)
```

Custom expectations like these can make your examples far more readable and feel more like descriptions of behaviour than tests. Of course, don't forget to balance readability with clarity of purpose. If an example with notify_applicant() fails, you'll want to understand the implications of that failure without having to go study a custom matcher. Always consider your team-mates when creating constructs like this, and strive for consistency within any code base (including its code examples).

With the proper balance, you'll find that this makes it much easier to understand what the examples are describing when looking back at them days, weeks, or even months later. Easier understanding saves time, and saving time saves money. This can help to reduce the cost of change later on in the life of an application. This is what Agile is all about.

To better understand RSpec's expectations, let's get familiar with their different parts. We'll start off by taking a closer look at the should() and should_not() methods, followed by a detailed discussion of various types of *expression matchers*. As you'll see, RSpec supports expression matchers for common operations that you might expect, like equality, and some more unusual expressions as well.

11.1 should and should_not

RSpec achieves a high level of expressiveness and readability by exploiting open classes in Ruby to add the methods should() and should_not() to the Object class, and consequently every object in the system. Both methods accept either an *expression matcher* or a Ruby expression using a specific subset of Ruby operators. An Expression Matcher is an object that does exactly what its name suggests: it matches an expression.

Let's take a look at an example using the Equal Expression Matcher, which you can access through the method equal(expected). This is one of the many expression matchers that ships with RSpec.

```
result.should equal(5)
```

Seems simple enough, doesn't it? Well let's take a closer look. First, let's add parentheses as a visual aid:

```
result.should(equal(5))
```

Now take a look at Figure 11.1, on the next page.

When the Ruby interpreter encounters this line, it begins by evaluating equal(5), which returns a new instance of the Equal class, initialized with the value 5. This object is the expression matcher we use for this Expectation. This instance of Equal is then passed to result.should.

Next, should() calls matcher.matches?(self). Here matcher is the instance of Equal we just passed to should() and self is the result object. Because should() is added to every object, it can be ANY object. Similarly, the matcher can be ANY object that responds to matches?(target). This is a beautiful example of how dynamic languages make it so much easier to write truly Object Oriented code.

If matches?(self) returns true, then the Expectation is considered met, and execution moves on to the next line in the example. If it returns

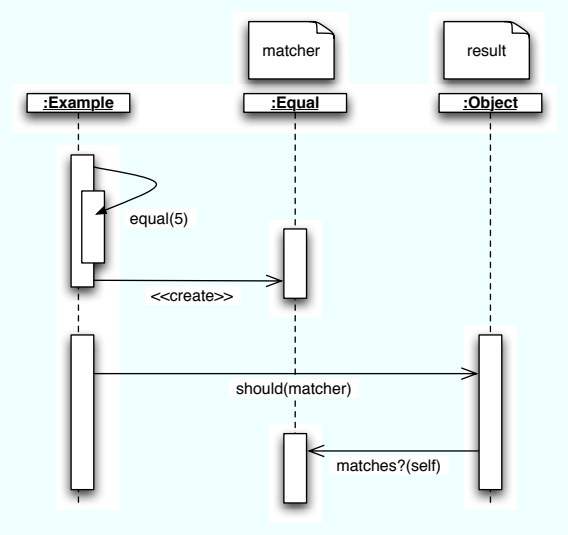


Figure 11.1: Should/Matcher Interaction Diagram

false, then an ExpectationNotMetException is raised with a message returned by matcher.failure_message().

should_not() works the opposite way. If matches?(self) returns false, then the Expectation is considered met and execution moves on to the next line in the example. If it returns true, then an ExpectationNotMetException is raised with a message returned by matcher.negative_failure_message. Note that should() uses failure_message, while should_not() uses negative_failure_message, allowing the Matcher to provide meaningful messages in either situation. Clear, meaningful feedback is one of RSpec's primary goals.

The should() and should_not() methods can also take any of several operators such as == and =~. You can read more about those in Section 11.5, *Operator Expressions*, on page 145. Right now, let's take a closer look at RSpec's built-in matchers.

11.2 Built-In Matchers

RSpec ships with several built-in matchers with obvious names that you can use in your examples. In addition to equal(expected), others include:

Matchers

The idea of a *Matcher* is not unique to RSpec. In fact, when I first pointed out to Dan North that we were using these, I referred to them as *Expectations*. Given Dan's penchant for "Getting The words Right", he corrected me, gently, saying that "while should eat_cheese is an Expectation, the eat_cheese part is a Matcher", citing jMock2 (http://jmock.org) and Hamcrest (http://code.google.com/p/hamcrest/) as examples.

jMock and Hamcrest are "A Lightweight Mock Object Library" and a "library of matchers for building test expressions," respectively, and it turns out that jMock2 actually uses Hamcrest's matchers as Mock Argument Constraints. Seeing that inspired me to have RSpec share matchers across Spec::Expectations and Spec::Mocks as well. Since they are serving as both Mock Argument Constraint Matchers and Expectation Matchers, we'll refer to them henceforth as expression matchers.

```
include(item)
respond_to(message)
raise_error(type)
```

By themselves, they seem a bit odd, but in context they make a bit more sense:

```
prime_numbers.should_not include(8)
list.should respond_to(:length)
lambda { Object.new.explode! }.should raise_error(NameError)
```

We'll cover each of RSpec's built-in matchers, starting with those related to equality.

Equality: Object Equivalence and Object Identity

Although we're focused on behaviour, many of the expectations we want to set are about the state of the environment after some event occurs. The two most common ways of dealing with post-event state are to specify that an object should have values that match our expectations (object equivalence) and to specify that an object *is* the very same object we are expecting (object identity).

Most xUnit frameworks support something like assert_equal to mean that two objects are equivalent and assert_same to mean that two objects are really the same object (object identity). This comes from languages

like Java, in which there are really only two constructs that deal with equality: the == operator, which, in Java, means the two references point to the same object in memory, and the equals method, which defaults to the same meaning as ==, but is normally overridden to mean equivalence.

Note that you have to do a mental mapping with assertEqual and assertSame. In Java, assertEqual means equal, assertSame means ==. This is OK in languages with only two equality constructs, but Ruby is bit more complex than that. Ruby has four constructs that deal with equality.

```
a == b
a === b
a.eql?(b)
a.equal?(b)
```

Each of these has different semantics, sometimes differing further in different contexts, and can be quite confusing.[2] So rather than forcing you to make a mental mapping from expectations to the methods they represent, RSpec lets you express the exact method you mean to express.

```
a.should == b
a.should === b
a.should eql(b)
a.should equal(b)
```

The most common of these is should ==, as the majority of the time we're concerned with value equality, not object identity. Here are some examples:

```
(3 * 5).should == 15

person = Person.new(:given_name => "Yukihiro", :family_name => "Matsumoto")
person.full_name.should == "Yukihiro Matsumoto"
person.nickname.should == "Matz"
```

In these examples, we're only interested in the correct values. Sometimes, however, we'll want to specify that an object is the exact object that we're expecting.

```
person = Person.create!(:name => "David")
Person.find_by_name("David").should equal(person)
```

Note that this puts a tighter constraint on the value returned by find_by_name(), that it must be the exact same object as the one returned by create!().

2. See http://www.ruby-doc.org/core/classes/Object.html#M001057 for the official documentation about equality in Ruby.

While this may be appropriate when expecting some sort of caching behaviour, the tighter the constraint, the more brittle the expectation. If caching is not a real requirement in this example, then saying Person.find_by_name("David").should == person is good enough and means that this example is less likely to fail later when things get refactored.

Floating Point Calculations

Floating point math can be a pain in the neck when it comes to setting expectations about the results of a calculation. And there's little more frustrating than seeing "expected 5.25, got 5.251" in a failure message, especially when you're only looking for two decimal places of precision.

To solve this problem, RSpec offers a be_close matcher that accepts an expected value *and* an acceptable delta. So if you're looking for precision of two decimal places, you can say:

```
result.should be_close(5.25, 0.005)
```

This will pass as long as the given value is within .005 of 5.25.

Multiline Text

Imagine developing an object that generates a statement. You could have one big example that compares the entire generated statement to an expected statement. Something like this:

```
expected = File.open('expected_statement.txt','r') do |f|
  f.read
end
account.statement.should == expected
```

This approach of reading in a file that contains text that has been reviewed and approved, and then comparing generated results to that text, is known as the "Golden Master" technique and is described in detail in J.B. Rainsberger's *JUnit Recipes* [Rai04].

This serves very well as a high level code example, but when we want more granular examples, this can sometimes feel a bit like brute force, and it can make it harder to isolate a problem when the wheels fall off.

Also, there are times that we don't really care about the entire string, just a subset of it. Sometimes we only care that it is formatted a specfic way, but don't care about the details. Sometimes we care about a few details but not the format.

In any of these cases we can expect a matching regular expression using either of the following patterns:

```
result.should match(/this expression/)
result.should =~ /this expression/
```

In the statement example, we might do something like this:

```
statement.should =~ /Total Due: \$37\.42/m
```

One benefit of this approach is that each example is, by itself, less brittle, less prone to fail due to unrelated changes. RSpec's own code examples are filled with expectations like this related to error messages, where we want to specify certain things are in place but don't want the expectations to fail due to some inconsequential changes to formatting.

Arrays and Hashes

As is the case with text, sometimes we want to set expectations about an entire Array or Hash, and sometimes just a subset. Because RSpec delegates == to Ruby, we can use that any time we want to expect an entire Array or Hash, with semantics we should all be familiar with.

```
[1,2,3].should == [1,2,3]
[1,2,3].should_not == [1,2,3,4]
{'this' => 'hash'}.should == {'this' => 'hash'}
{'this' => 'hash'}.should_not == {'that' => 'hash'}
```

But sometimes we just want to expect that 2 is in the Array [1,2,3]. To support that, RSpec includes an include() method that invokes a matcher that will do just that:

```
[1,2,3].should include(2)
{'a' => 1, 'b' => 2}.should include('b' => 2)
{'a' => 1}.should_not include('a' => 2)
```

Sometimes we don't need that much detail, and we just want to expect an Array of a specific length, or a Hash with 17 key/value pairs. You could express that using the equality matchers, like this:

```
array.length.should == 37
hash.keys.length.should == 42
```

That's perfectly clear and is perfectly acceptable, but lacks the DSL feel that we get from so many of RSpec's matchers. For those of you who prefer that, we can use the have matcher, which you'll learn about in more detail later in this chapter in Section 11.4, *Have Whatever You Like*, on page 141. For an Array of players on a baseball field, you can do this:

```
team.should have(9).players_on_the_field
```

For a hash with 17 key/value pairs:

```
hash.should have(17).key_value_pairs
```

In these examples, the players_on_the_field() and key_value_pairs() methods are actually there as pure syntactic sugar, and are not even evaluated. Admittedly, some people get confused and even angered by this magic, and they have a valid argument when suggesting that this violates the principle of least surprise. So use this approach if you like the way it reads and use the more explicit and less magical, but equally effective array.length.should == 37 if that works better for you and your development team.

Ch, ch, ch, ch, changes

Ruby on Rails extends test/unit with some rails-specific assertions. One such assertion is assert_difference(), which is most commonly used to express that some event adds a record to a database table, like this:

```
assert_difference 'User.admins.count', 1 do
  User.create!(:role => "admin")
end
```

This asserts that the value of User.admins.count will increase by 1 when you execute the block. In an effort to maintain parity with the rails assertions, RSpec offers this alternative:

```
lambda {
  User.create!(:role => "admin")
}.should change{ User.admins.count }
```

You can also make that much more explicit if you want by chaining calls to by(), to() and from().

```
lambda {
  User.create!(:role => "admin")
}.should change{ User.admins.count }.by(1)

lambda {
  User.create!(:role => "admin")
}.should change{ User.admins.count }.to(1)

lambda {
  User.create!(:role => "admin")
}.should change{ User.admins.count }.from(0).to(1)
```

This does not only work with Rails. You can use it for any situation in which you want to express a side effect of some event:

```
lambda {
  seller.accept Offer.new(250_000)
}.should change{agent.commission}.by(7_500)
```

Now you could, of course, express the same thing like this:

```
agent.commission.should == 0
seller.accept Offer.new(250_000)
agent.commission.should == 7_500
```

This is pretty straightforward and might even be easier to understand at first glance. Using should change, however, does a nice job of identifying what is the event and what is the expected outcome. It also functions as a wrapper for more than one expectation if you use the from() and to() methods, as in the examples above.

So which approach should you choose? It really comes down to a matter of personal taste and style. If you're working solo, it's up to you. If you're working on a team, have a group discussion about the relative merits of each approach.

Expecting Errors

When I first started learning Ruby I was very impressed with how well the language read my mind! I learned about Arrays before I learned about Hashes, so I already knew about Ruby's iterators when I encountered a problem that involved a Hash, and I wanted to iterate through its key/value pairs. Before using ri or typing puts hash.methods, I typed hash.each_pair |k,v| just to see if it would work. Of course, it did. And I was happy.

Ruby is filled with examples of great, intuitive APIs like this, and it seems that developers who write their own code in Ruby strive for the same level of *obvious*, inspired by the beauty of the language. We all want to provide that same feeling of happiness to developers that they get just from using the Ruby language directly.

Well, if we care about making developers happy, we should also care about providing meaningful feedback when the wheels fall off. We want to provide error classes and messages that provide context that will make it easier to understand what went wrong.

Here's a great example from the Ruby library itself:

```
$ irb
irb(main):001:0> 1/0
ZeroDivisionError: divided by 0
  from (irb):1:in `/'
  from (irb):1
```

The fact that the error is named ZeroDivisionError probably tells you everything you need to know to understand what went wrong. The message

"divided by 0" reinforces that. RSpec supports the development of informative error classes and messages with the raise_error() matcher.

If a checking account has no overdraft support, then it should let us know:

```
account = Account.new 50, :dollars
lambda {
  account.withdraw 75, :dollars
}.should raise_error(
  InsufficientFundsError,
  /attempted to withdraw 75 dollars from an account with 50 dollars/
)
```

The raise_error() matcher will accept 0, 1 or 2 arguments. If you want to keep things generic, you can pass 0 arguments and the example will pass as long as any subclass of Exception is raised.

```
lambda { do_something_risky }.should raise_error
```

The first argument can be any of a String message, a Regexp that should match an actual message, or the class of the expected error.

```
lambda {
  account.withdraw 75, :dollars
}.should raise_error(
  "attempted to withdraw 75 dollars from an account with 50 dollars"
)

lambda {
  account.withdraw 75, :dollars
}.should raise_error(/attempted to withdraw 75 dollars/)

lambda {
  account.withdraw 75, :dollars
}.should raise_error(InsufficientFundsError)
```

When the first argument is an error class, it can be followed by a second argument that is either a String message or a Regexp that should match an actual message.

```
lambda {
  account.withdraw 75, :dollars
}.should raise_error(
  InsufficientFundsError,
  "attempted to withdraw 75 dollars from an account with 50 dollars"
)

lambda {
  account.withdraw 75, :dollars
}.should raise_error(
  InsufficientFundsError,
```

```
  /attempted to withdraw 75 dollars/
)
```

Which of these formats you choose depends on how specific you want to get about the type and the message. Sometimes you'll find it pragmatic to have just a few code examples that get into details about messages, while others may just specify the type. If you look through RSpec's own code examples, you'll see many that look like this:

```
lambda {
  @mock.rspec_verify
}.should raise_error(MockExpectationError)
```

Since there are plenty of other examples that specify details about the error messages raised by message expectation failures, this example only cares that a MockExpectationError is raised.

Expecting a Throw

A less often used, but very valuable construct in Ruby is the throw/catch block. Like raise() and rescue(), throw() and catch() allow you to stop execution within a given scope based on some condition. The main difference is that throw/catch expresses expected circumstances as opposed to exceptional circumstances. It is most commonly used (within its rarity) to break out of nested loops.

For example, let's say we want to know if anybody on our team has worked over 50 hours in one week in the last month. We're going to have a nested loop:

```
re_read_the_bit_about :sustainable_pace if working_too_hard?

def working_too_hard?
  weeks.each do |week|
    people.each do |person|
      return true if person.hours_for(week) > 50
    end
  end
end
```

This seems perfectly sound, but what if we want to optimize it so it short circuits as soon as working_too_hard == true? This is a perfect case for using throw/catch:

```
def working_too_hard?
  catch :working_too_hard do
    weeks.each do |week|
      people.each do |person|
        throw :working_too_hard, true if person.hours_for(week) > 50
```

```
      end
    end
  end
end
```

To set an expectation that a symbol is thrown, we wrap up the code in a proc and set the expectation on the proc:

```
lambda {
  team.working_too_hard
}.should throw_symbol(:working_too_hard)
```

Like the raise_error() matcher, the throw_symbol() matcher will accept 0, 1 or 2 arguments. If you want to keep things generic, you can pass 0 arguments and the example will pass as long as anything is thrown.

The first (optional) argument to throw_symbol() must be a Symbol, as shown in the example above.

The second argument, also optional, can be anything, and the matcher will pass only if both the symbol and the thrown object are caught. In our current example, that might look like this:

```
lambda {
  team.working_too_hard
}.should throw_symbol(:working_too_hard, true)

# or ...

lambda {
  team.working_too_hard
}.should throw_symbol(:working_too_hard, false)
```

11.3 Predicate Matchers

A Ruby predicate method is one whose name ends with a "?" and returns a boolean response. One example built right into the language is array.empty?. This is a simple, elegant construct that allows us to write code like this:

```
do_something_with(array) unless array.empty?
```

When we want to set an expectation that a predicate should return a specific result, however, the code isn't quite as pretty.

```
array.empty?.should == true
```

While that does express what we're trying to express, it doesn't read that well. What we really want to say is that the "array should be empty", right? Well, say it then!

```
array.should be_empty
```

Believe it or not, that will work as you expect. The expectation will be met and the example will pass if the array has an empty? method that returns true. If array does not respond to empty?, then we get a NoMethodError. If it does respond to empty? but returns false, then we get an ExpectationNotMetError.

This feature will work for any Ruby predicate. It will even work for predicates that accept arguments, such as:

```
user.should be_in_role("admin")
```

This will pass as long as user.in_role?("admin") returns true.

How They Work

RSpec overrides method_missing to provide this nice little bit of syntactic sugar. If the missing method begins with "be_", RSpec strips off the "be_", appends a "?", and sends the resulting message to the given object.

Taking this a step further, there are some predicates that don't read as fluidly as we might like when prefixed with "be_". instance_of?(type), for example, becomes be_instance_of. To make these a bit more readable, RSpec also looks for things prefixed with "be_a_" and "be_an_". So we also get to write be_a_kind_of(Player) or be_an_instance_of(Pitcher).

Even with all of this support for prefixing arbitrary predicates, there will still be cases in which the predicate just doesn't fit quite right. For example, you wouldn't want to say parser.should be_can_parse("some text"), would you? Well, we wouldn't want to have to say anything quite so ridiculous, so RSpec supports writing custom matchers with a simple DSL that you'll read about in Section 15.3, *Custom Matchers*, on page 182.

Up until now we've been discussing expectations about the state of an object. The object should be_in_some_state. But what about when the state we're interested in is not in the object itself, but in an object that it owns?

11.4 Have Whatever You Like

A hockey team should have 5 skaters on the ice under normal conditions. The word "character" should have 9 characters in it. Perhaps a

Hash should have a specific key. We *could* say Hash.has_key?(:foo).should be_true, but what we really want to say is Hash.should have_key(:foo).

RSpec combines expression matchers with a bit more method_missing goodness to solve these problems for us. Let's first look at RSpec's use of method_missing. Imagine that we've got a simple RequestParameters class that converts request parameters to a hash. We might have an example like this:

```
request_parameters.has_key?(:id).should == true
```

This expression makes sense, but it just doesn't read all that well. To solve this, RSpec uses method_missing to convert anything that begins with have_ to a predicate on the target object beginning with has_. In this case, we can say:

```
request_parameters.should have_key(:id)
```

In addition to the resulting code being more expressive, the feedback that we get when there is a failure is more expressive as well. The feedback from the first example would look like this:

```
expected true, got false
```

Whereas the have_key example reports this:

```
expected #has_key?(:id) to return true, got false
```

This will work for absolutely any predicate method that begins with "has_". But what about collections? We'll take a look at them next.

Owned Collections

Let's say we're writing a fantasy baseball application. When our app sends a message to the home team to take the field, we want to specify that it sends 9 players out to the field. How can we specify that? Here's one option:

```
field.players.collect {|p| p.team == home_team }.length.should == 9
```

If you're an experienced rubyist, this might make sense right away, but compare that to this expression:

```
home_team.should have(9).players_on(field)
```

Here, the object returned by have() is a matcher, which does not respond to players_on(). When it receives a message it doesn't understand (like players_on()), it delegates it to the target object, in this case the home_team.

This expression reads like a requirement and, like arbitrary predicates, encourages useful methods like players_on().

At any step, if the target object or its collection doesn't respond to the expected messages, a meaningful error gets raised. If there is no players_on method on home_team, you'll get a NoMethodError. If the result of that method doesn't respond to length or size, you'll get an error saying so. If the collection's size does not match the expected size, you'll get a failed expectation rather than an error.

Un-owned Collections

In addition to setting expectations about owned collections, there are going to be times when the object you're describing *is* itself a collection. RSpec lets us use have to express this as well:

```
collection.should have(37).items
```

In this case, items is pure syntactic sugar. What's happening to support this is safe, but a bit sneaky, so it is helpful for you to understand what is happening under the hood, lest you be surprised by any unexpected behaviour. We'll discuss the inner workings of have a bit later in this section.

Strings

Strings are collections too! Not quite like Arrays and Hashes, but they do respond to a lot of the same messages as collections do. Because Strings respond to length and size, you can also use have to expect a string of a specific length.

```
"this string".should have(11).characters
```

As in unowned collections, characters is pure syntactic sugar in this example.

Precision in Collection Expectations

In addition to being able to express an expectation that a collection should have some number of members, you can also say that it should have *exactly* that number, *at least* that number or *at most* that number:

```
day.should have_exactly(24).hours
dozen_bagels.should have_at_least(12).bagels
internet.should have_at_most(2037).killer_social_networking_apps
```

have_exactly is just an alias for have. The others should be self explanatory. These three will work for all of the applications of have described in the previous sections.

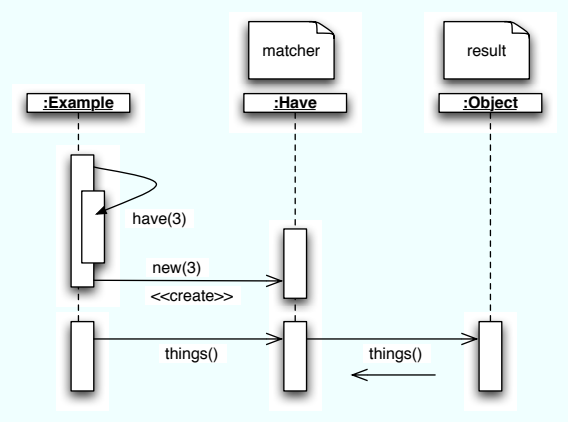


Figure 11.2: Have Matcher Sequence

How It Works

The have method can handle a few different scenarios. The object returned by have is an instance of Spec::Matchers::Have, which gets initialized with the expected number of elements in a collection. So the expression:

```
result.should have(3).things
```

is the equivalent of the expression:

```
result.should(Have.new(3).things)
```

Figure 11.2 shows how this all ties together. The first thing to get evaluated is Have.new(3), which creates a new instance of Have, initializing it with a value of 3. At this point, the Have object stores that number as the expected value.

Next, the Ruby interpreter sends things to the Have object. method_missing is then invoked because Have doesn't respond to things. Have overrides method_missing to store the message name (in this case things) for later use and then returns self. So the result of have(3).things is an instance of Have that knows the name of the collection you are looking for and how many elements should be in that collection.

The Ruby interpreter passes the result of have(3).things to should(), which, in turn, sends matches?(self) to the matcher. It's the matches? method in which all the magic happens.

First, it asks the target object (result) if it responds to the message that it stored when method_missing was invoked (things). If so, it sends that message and, assuming that the result is a collection, interrogates the result for its length or its size (whichever it responds to, checking for length first). If the object does not respond to either length or size, then you get an informative error message. Otherwise the actual length or size is compared to the expected size and the example passes or fails based the outcome of that comparison.

If the target object does not respond to the message stored in method_missing, then Have tries something else. It asks the target object if it, itself, can respond to length or size. If it will, it assumes that you are actually interested in the size of the target object, and not a collection that it owns. In this case, the message stored in method_missing is ignored and the size of the target object is compared to the expected size and, again, the example passes or fails based the outcome of that comparison.

Note that the target object can be anything that responds to length or size, not just a collection. As explained in our discussion of Strings, this allows you to express expectations like "this string".should have(11).characters.

In the event that the target object does not respond to the message stored in method_missing, length or size, then Have will go ahead and send the message to the target object and let the resulting NoMethodError bubble up to the example.

As you can see, there is a lot of magic involved. RSpec tries to cover all the things that can go wrong and give you useful messages in each case, but there are still some potential pitfalls. If you're using a custom collection in which length and size have different meanings, you might get unexpected results. But these cases are rare, and as long as you are aware of the way this all works, you should certainly take advantage of its expressiveness.

11.5 Operator Expressions

Generally, we want to be very precise about our expectations. We would want to say that "2 + 2 should equal 4," not that "2 + 2 should be greater than 3." There are exceptions to this, however. Writing a random generator for numbers between 1 and 10, we would want to make sure that 1 appears roughly 1000 in 10,000 tries. So we set some level of tolerance, say 2%, which results in something like "count for 1's should be greater than or equal to 980 and less than or equal to 1020."

An example like that might look like this:

```
it "should generate a 1 10% of the time (plus/minus 2%)" do
  result.occurences_of(1).should be_greater_than_or_equal_to(980)
  result.occurences_of(1).should be_less_than_or_equal_to(1020)
end
```

Certainly it reads like English, but it's just a bit verbose. Wouldn't it be nice if, instead, we could use commonly understood operators like >= instead of be_greater_than_or_equal_to? As it turns out, we can!

Thanks to some magic that we get for free from the Ruby language, RSpec is able to support the following expectations using standard Ruby operators:

```
result.should == 3
result.should =~ /some regexp/
result.should be < 7
result.should be <= 7
result.should be >= 7
result.should be > 7
```

RSpec can do this because Ruby interprets these expressions like this:

```
result.should.==(3)
result.should.=~(/some regexp/)
result.should(be.<(7))
result.should(be.<=(7))
result.should(be.>=(7))
result.should(be.>(7))
```

RSpec exploits that interpretation by defining == and =~ on the object returned by should() and <, <=, >, and >= on the object returned by be.

11.6 Generated Descriptions

Sometimes we end up with a an example docstring which is nearly an exact duplication of the expectation expressed in the example. For example:

```
describe "A new chess board" do
  before(:each) do
    @board = Chess::Board.new
  end

  it "should have 32 pieces" do
    @board.should have(32).pieces
  end
end

Produces:
```

```
A new chess board
- should have 32 pieces
```

In this case, we can rely on RSpec's automatic example-name generation to produce the name you're looking for:

```
describe "A new chess board" do
  before(:each) { @board = Chess::Board.new }
  specify { @board.should have(32).pieces }
end

Produces:
A new chess board
- should have 32 pieces
```

This example uses the specify() method instead of it() because specify is more readable when there is no docstring. Both it() and specify() are actually aliases of the example() method, which creates an example.

Each of RSpec's matchers generates a description of itself, which gets passed on to the example. If the example (or it, or specify) method does not receive a docstring, it uses the last of these descriptions that it receives. In this example, there is only one: "should have 32 pieces."

It turns out that it is somewhat rare that the auto-generated names express exactly what you would want to express in the descriptive string passed to example. Our advice is to always start by writing exactly what you want to say and only resort to using the generated descriptions when you actually see that the string and the expectation line up precisely. Here's an example in which it might be more clear to leave the string in place:

```
it "should be eligible to vote at the age of 18" do
  @voter.birthdate = 18.years.ago
  @voter.should be_eligible_to_vote
end
```

Even though the auto-generated description would read "should be eligible to vote," the fact that he is 18 today is very important to the requirement being expressed. Whereas, consider this example:

```
describe RSpecUser do
  before(:each) do
    @rspec_user = RSpecUser.new
  end
  it "should be happy" do
    @rspec_user.should be_happy
  end
end
```

This Expectation would produce a string identical to the one that is being passed to it, so this is a good candidate for taking advantage of auto-generated descriptions.

11.7 Subject-ivity

The *subject* of an example is the object being described. In the happy RSpecUser example, the subject is an instance of RSpecUser, instantiated in the before block.

RSpec offers an alternative to setting up instance variables in before blocks like this, in the form of the subject() method. You can use this method in a few different ways, ranging from explicit, and consequently verbose, to implicit access which can make things more concise. First let's discuss explicit interaction with the subject.

Explicit Subject

In an example group, you can use the subject() method to define an explicit subject by passing it a block, like this:

```
describe Person do
  subject { Person.new(:birthdate => 19.years.ago) }
end
```

Then you can interact with that subject like this:

```
describe Person do
  subject { Person.new(:birthdate => 19.years.ago) }
  specify { subject.should be_eligible_to_vote }
end
```

Delegation to Subject

Once a subject is declared, the example will delegate should() and should_not() to that subject, allowing you to clean that up even more:

```
describe Person do
  subject { Person.new(:birthdate => 19.years.ago) }
  it { should be_eligible_to_vote }
end
```

Here the should() method has no explicit receiver, so it is received by the example itself. The example then calls subject() and delegates should() to it. Note that we used it() in this case, rather than specify(). Read that aloud and compare it to the previous example and you'll see why.

The previous example reads "specify subject should be eligible to vote," whereas this example reads "it should be eligible to vote." Getting more concise, yes? It turns out that, in some cases, we can make things even more concise using an implicit subject.

Implicit Subject

In the happy RSpecUser example, we created the subject by calling new on the RSpecUser class without any arguments. In cases like this, we can leave out the explicit subject declaration and RSpec will create an *implicit subject* for us:

```
describe RSpecUser do
  it { should be_happy }
end
```

Now *that* is concise! Can't get much more concise that this. Here, the subject() method used internally by the example returns a new instance of RSpecUser.

Of course this only works when all the pieces fit. The describe() method has to receive a class that can be instantiated safely without any arguments to new(), and the resulting instance has to be in the correct state.

One word of caution: seeing things so concise like this breeds a desire to make everything else concise. Be careful to *not* let the goal of keeping things concise get in the way of expressing what you really want to express. Delegating to an implicit subject takes a lot for granted, and it should only be used when all the pieces really fit, rather than coercing the pieces to fit.

Beyond Expectations

In this chapter, we've covered:

- should() and should_not()
- RSpec's built-in matchers
- Predicate matchers
- Operator expressions
- Generated descriptions
- Declaring an explicit subject()
- Using the implicit subject()

For most projects, you'll probably find that you can express what you want to using just the tools that come along with RSpec. But what about those cases where you think to yourself "if only RSpec had this one additional matcher"? We'll address that question in Chapter 15, *Extending RSpec*, on page 177, along with a number of other techniques for extending RSpec and tuning its DSL towards your specific projects.

In the meantime, there's still quite a bit more material to cover without extending things at all. In the next chapter we'll introduce you to RSpec's built-in mock objects framework, a significant key to thinking in terms of behaviour.

Chapter 12

Mocking in RSpec

Coming soon ...

Chapter 13

RSpec and Test::Unit

Are you working on a Ruby project that already uses Test::Unit? Are you considering migrating over to RSpec?

Migrating from Test::Unit to RSpec is a straightforward, but manual process. It involves a series of refactorings to your tests and, as with all refactorings, you should rerun them between each refactoring. That way if any changes you make cause things to go awry, you'll always know what caused the problem because it's the last change you made before you ran the tests.

While you're in the middle of this refactoring, your tests will look like half tests and half RSpec code examples because you'll be mixing the two styles. This is not pretty, but it's extremely important as it allows you to rerun everything after each refactoring. As you'll see, RSpec and Test::Unit are completely interoperable, but the reason for this is to make migration easier. We recommend you don't use this interoperability to leave your tests (or specs) in the hybrid state, as it will just lead to confusion later on.

The migration work essentially consists of refactoring the following Test::Unit elements to RSpec:

- class SomeClassTest < Test::Unit::TestCase becomes describe SomeClass
- def test_something becomes it "should do something descriptive"
- def setup becomes before(:each)
- def teardown becomes after(:each)
- assert_equal 4, array.length becomes array.length.should == 4

Before we jump in and start with these refactorings, let's get you set up so that you can run the tests between each refactoring using RSpec's runner.

13.1 Running Test::Unit tests with the RSpec runner

There are several ways to run tests written with Test::Unit. You can use rake to run one or more test files, run them directly with the ruby interpreter, or you can use the testrb script that comes with your Ruby distribution. We'll use the TestTask that ships with Rake for our example.

Let's start with a very minimal project that has one library file, one test file, a test_helper.rb, and a Rakefile with a TestTask defined.

Download testunit/lib/person.rb

```
class Person
  def self.unregister(person)
  end

  def initialize(first_name, last_name)
    @first_name, @last_name = first_name, last_name
  end

  def full_name
    "#{@first_name} #{@last_name}"
  end

  def initials
    "#{@first_name[0..0]}#{@last_name[0..1]}"
  end
end
```

Download testunit/test/test_helper.rb

```
$:.unshift File.join(File.dirname(__FILE__), *%w[.. lib])

require 'person'
```

Download testunit/test/person_test.rb

```
require File.join(File.dirname(__FILE__), "/test_helper.rb")
require 'test/unit'

class PersonTest < Test::Unit::TestCase

  def setup
    @person = Person.new('Dave','Astels')
  end

  def test_full_name
```

```
    assert_equal 'Dave Astels', @person.full_name
  end

  def test_initials
    assert_equal 'DA', @person.initials
  end

  def teardown
    Person.unregister(@person)
  end

end
```

Download testunit/Rakefile

```
require 'rake/testtask'
Rake::TestTask.new do |t|
  t.test_files = FileList['test/person_test.rb']
end
```

This PersonTest has a setup and teardown, one passing test and one failing test. We're including a failing test to give you a better picture of the enhanced output you get from RSpec. Go ahead and run rake test, and you should see the following output:

```
Started
.F
Finished in 0.00903 seconds.

  1) Failure:
test_initials(PersonTest) [./test/person_test.rb:15]:
<"DA"> expected but was
<"DAs">.

2 tests, 2 assertions, 1 failures, 0 errors
```

If you've been using Test::Unit this should be quite familiar to you. After the word "Started" we get a text-based progress bar with a "." for each passing test and an "F" for each failure.

The progress bar is followed by the details of each failure, including a reference to the line in the test file that contains the failed assertion, and an explanation of the failure.

Lastly we have a summary listing how many test methods were run, how many assertions were evaluated, the number of logical failures (failed assertions) and the number of errors.

To get started transforming the PersonTest to a Person spec, add an RSpec Rake task that will run the same tests:

Download testunit/Rakefile

```
require 'rubygems'
require 'spec/rake/spectask'

Spec::Rake::SpecTask.new do |t|
  t.ruby_opts = ['-r test/unit']
  t.spec_files = FileList['test/person_test.rb']
end
```

When RSpec gets loaded, it checks whether Test::Unit has been loaded and, if it has, enables the bridge between RSpec and Test::Unit that supports running tests with RSpec. By passing -r test/unit to the Ruby interpreter, Test::Unit will be loaded before RSpec.

For now no other changes are needed, so go ahead and run the tests with rake spec and you should see output like this:

```
.F

1)
Test::Unit::AssertionFailedError in 'PersonTest test_initials'
<"DA"> expected but was
<"DAs">.
./test/person_test.rb:15:in `test_initials'

Finished in 0.028264 seconds

2 examples, 1 failure
```

At this point, RSpec's output is almost identical to that which we get from Test::Unit, but the summary is different. It sums up *code examples* instead of tests, and it doesn't discriminate between logical failures and execution errors. If something goes wrong it's gotta get fixed. It doesn't really matter if it's a failure or an error, and we'll know all we need to know as soon as we look closely at the detailed messages.

Enabling RSpec's Test::Unit bridge from Rake is an easy way to start when you want to get all your tests running through RSpec, but if you want to run individual test cases from an editor like TextMate, or straight from the command line using the ruby command, you'll need to modify the require 'test/unit' statements wherever they appear.

If you're using rspec-1.2 or later, change require 'test/unit' to require 'spec/test/unit'. With rspec-1.1.12 or earlier, use require 'spec/interop/test'. In either case, you may also need to require 'rubygems' first. Here's what you'll end up with:

Generating RSpec HTML reports from Test::Unit tests

You saw how easy it was to make RSpec run your Test::Unit tests. Once you've successfully done that, try to output a HTML report for your tests. Just add --format html:result.html to RSpec's command line.

If you're using Rake to run your tests it's just a matter of adding the following line inside your SpecTask:

```
t.spec_opts = ['--format', 'html:result.html']
```

Then just open up result.html in a browser and enjoy the view!

Download testunit/test/person_test_with_rspec_required.rb

```
require File.join(File.dirname(__FILE__), "/test_helper.rb")
require 'rubygems'
require 'spec/test/unit'
```

Once you have done this, you no longer need the -r test/unit in the Rakefile, so go ahead and remove it:

Download testunit/Rakefile

```
Spec::Rake::SpecTask.new do |t|
  t.spec_files = FileList['test/person_test.rb']
end
```

Now run the test again with rake spec and you should get the same output:

```
.F

1)
Test::Unit::AssertionFailedError in 'PersonTest test_initials'
<"DA"> expected but was
<"DAs">.
test/person_test_with_rspec_required.rb:19:in `test_initials'
test/person_test_with_rspec_required.rb:22:

Finished in 0.016624 seconds

2 examples, 1 failure
```

That's all it takes to run Test::Unit tests with RSpec. And with that, you've also taken the first step towards migrating to RSpec. You can now start to refactor the tests themselves, and after every refactoring

you'll be able to run all the tests to ensure that your refactorings are ok.

13.2 Refactoring Test::Unit Tests to RSpec Code Examples

Although you haven't seen it yet, by loading RSpec's Test::Unit bridge, we have also snuck RSpec in the back door. All of RSpec's API is now available and ready to be used within this TestCase, and the refactorings in this section will help you gradually change your tests to specs.

Describing Test::Unit::TestCases

The first step we'll take is to add a describe() declaration to the TestCase, as shown on line 6 in the code that follows:

Download testunit/test/person_test_with_describe.rb

```
Line 1  require File.join(File.dirname(__FILE__), "/test_helper.rb")
      - require 'rubygems'
      - require 'spec/test/unit'
      -
    5 class PersonTest < Test::Unit::TestCase
      -   describe('A Person')
      -
      -   def setup
      -     @person = Person.new('Dave','Astels')
   10   end
      -
      -   def test_full_name
      -     assert_equal 'Dave Astels', @person.full_name
      -   end
   15
      -   def test_initials
      -     assert_equal 'DA', @person.initials
      -   end
      -
   20   def teardown
      -     Person.unregister(@person)
      -   end
      -
      - end
```

This not only embeds intent right in the code, but it also adds documentation to the output. Go ahead and run rake spec and you should get output like this:

```
.F

1)
Test::Unit::AssertionFailedError in 'A Person test_initials'
```

```
<"DA"> expected but was
<"DAs">.
test/person_test_with_describe.rb:18:in `test_initials'
test/person_test_with_describe.rb:21:

Finished in 0.018498 seconds

2 examples, 1 failure
```

The String passed to describe() gets included in the failure message, providing more context in which to understand the failure. Of course, since we've only described the context, but haven't migrated the tests to code examples, the resulting "A Person test_initials" is a bit odd. But that's just temporary.

We can take this a step further and just use the describe() method to generate RSpec's counterpart to a TestCase, the Spec::ExampleGroup:

Download testunit/test/person_spec_with_setup_and_tests.rb

```
require File.join(File.dirname(__FILE__), "/test_helper.rb")
require 'rubygems'
require 'spec/test/unit'

describe('A Person') do

  def setup
    @person = Person.new('Dave','Astels')
  end

  def test_full_name
    assert_equal 'Dave Astels', @person.full_name
  end

  def test_initials
    assert_equal 'DA', @person.initials
  end

  def teardown
    Person.unregister(@person)
  end

end
```

This not only provides similar internal documentation, but it also reduces the noise of the creation of the class, focusing on the DSL of describing the behaviour of objects and making the code more readable. If you run rake spec you should see the same output that was generated when we added describe() to the TestCase.

So now we've got setup, teardown and test methods with assertions wrapped inside the RSpec DSL. This is the hybrid you were warned about earlier in this chapter, so let's keep working our way from the outside-in—time to get rid of those nasty tests!

test methods to examples

Now that we've replaced the concept of a TestCase with a group of examples, let's continue inward and replace the tests with examples. We create examples using the it() method within an example group. Here's what our Person examples look like in RSpec:

Download testunit/test/person_spec_with_examples.rb

```
require File.join(File.dirname(__FILE__), "/test_helper.rb")
require 'rubygems'
require 'spec/test/unit'

describe('A Person') do

  def setup
    @person = Person.new('Dave','Astels')
  end

  it "should include the first and last name in #full_name" do
    assert_equal 'Dave Astels', @person.full_name
  end

  it "should include the first and last initials in #initials" do
    assert_equal 'DA', @person.initials
  end

  def teardown
    Person.unregister(@person)
  end

end
```

Using strings passed to it() instead of method names that start with "test" provides a much more fluid alternative to expressing the intent of the example.

Running this with rake spec provides this output:

```
.F

1)
Test::Unit::AssertionFailedError in \
      'A Person should include the first and last initials in #initials'
<"DA"> expected but was
<"DAs">.
```

```
test/person_spec_with_examples.rb:17:
test/person_spec_with_examples.rb:6:

Finished in 0.020142 seconds

2 examples, 1 failure
```

Look how much more expressive that is! “A Person should include the first and last initials in #initials” actually tells you something you can tell your grandmother.

Two refactorings down, two to go. Next up, setup() and teardown().

before and after

RSpec runs the block passed to before(:each) before each example is run. This is RSpec’s replacement for Test::Unit’s test-centric setup() method.

RSpec also runs the block passed to after(:each) after each example is run, replacing Test::Unit’s teardown().

So the next step is to simply replace setup() and teardown() with before() and after():

Download testunit/test/person_spec_with_before_and_after.rb

```
require File.join(File.dirname(__FILE__), "/test_helper.rb")
require 'rubygems'
require 'spec/test/unit'

describe('A Person') do

  before(:each) do
    @person = Person.new('Dave','Astels')
  end

  it "should include the first and last name in #full_name" do
    assert_equal 'Dave Astels', @person.full_name
  end

  it "should include the first and last initials in #initials" do
    assert_equal 'DA', @person.initials
  end

  after(:each) do
    Person.unregister(@person)
  end

end
```

This time, the output from rake spec should be exactly the same as when setup() and teardown() were in place. We're almost done with this refactoring now. There's only one step left—converting assertions to RSpec expectations.

should and should_not

The last step in refactoring from tests to RSpec code examples is replacing assertions with RSpec expectations using should() and should_not().

Go ahead and replace the assert_equal with a should ==.

Download testunit/test/person_spec_with_should.rb

```
require File.join(File.dirname(__FILE__), "/test_helper.rb")
require 'rubygems'
require 'spec/test/unit'

describe('A Person') do

  before(:each) do
    @person = Person.new('Dave','Astels')
  end

  it "should include the first and last name in #full_name" do
    @person.full_name.should == 'Dave Astels'
  end

  it "should include the first and last initials in #initials" do
    @person.initials.should == 'DA'
  end

  after(:each) do
    Person.unregister(@person)
  end

end
```

This will produce the following output:

```
.F

1)
'A Person should include the first and last initials in #initials' FAILED
expected: "DA",
     got: "DAs" (using ==)
test/person_spec_with_should.rb:17:
test/person_spec_with_should.rb:6:

Finished in 0.007005 seconds

2 examples, 1 failure
```

As you see, the error messages from should failures are a little different that the assert failures. We still have one passing and one failing example, but the class name is gone. At this point, we've replaced the class name and test name with the example group string (passed to describe()) and the example string (passed to it()).

One last step

At this point it appears that the TestCase has been completely migrated over to an RSpec ExampleGroup, but appearances can be deceiving. The object returned by describe() is *still* a TestCase. You can see this by adding puts self to the describe() block:

Download testunit/test/person_spec_with_puts.rb

```
describe('A Person') do
  puts self
```

Run rake spec again and you should see Test::Unit::TestCase::Subclass_1 in the output. So now, as the final step in the conversion, remove 'test/unit' from require 'spec/test/unit', so you just have require 'spec', and run rake spec again. This time you'll see Spec::Example::ExampleGroup::Subclass_1 instead, thus completing the migration.

13.3 What We Just Did

In this chapter we showed you how to refactor from tests to specs with a series of refactorings that allow you to run all your tests/examples between each step.

We started by converting TestCase classes to RSpec example groups with the describe() method. Then the test methods became RSpec examples with it(). Next we converted the setup() and teardown() declarations to before() and after(). Lastly, we converted the Test::Unit assertions to RSpec expectations, using should() and should_not().

While the order does seem logical, you should know that you can do these refactorings in any order. In fact, there is no technical reason that you can not have test methods with RSpec expectations and RSpec code examples with assertions all living happily side by side. The aesthetic reasons for avoiding this are clear, but this does mean that you can use Test::Unit extensions in your specs. Most notable are the Test::Unit assertions that ship with Ruby on Rails, any of which can be called from within an RSpec example.

Chapter 14

Tools And Integration

In the Mastermind tutorial in Part I, you used the spec command to run specs from a command line shell. In this chapter, we'll show you a number of command line options that you may not have tried out yet, as well as how RSpec integrates with other command line tools like Rake and autotest, and GUI editors like TextMate.

14.1 The spec Command

The spec command is installed when you install the rspec gem, and provides a number of options that let you customize how RSpec works. You can print a list of all these options by asking for help:

```
spec --help
```

Most of the options have a long form using two dashes and a shorthand form using one dash. The help option, for example, can be invoked with -h in addition to --help. We recommend you use the long form if you put it in a script such as a Rakefile (for clarity) and the short form when you run it directly from the command line (for brevity).

All of the command line options are also available when you run individual spec files directly with the ruby command.

Running One Spec File

Running a single file is a snap. You can use the spec command or even just the ruby command. For example, enter the following into simple_math_spec.rb:

```
require 'rubygems'
require 'spec'
```

```
describe "simple math" do
  it "should provide a sum of two numbers" do
    (1 + 2).should == 3
  end
end
```

Now run that file with the spec command:

```
spec simple_math_spec.rb
```

You should see output like this:

```
.

Finished in 0.00621 seconds

1 example, 0 failures
```

This is RSpec's default output format, the *progress bar* format. It prints out a dot for every code example that is executed and passes (only one in this case). If an example fails, it prints an F. If an example is pending it prints a *. These dots, F's and *'s are printed after each example is run, so when you have many examples you can actually see the progress of the run, hence the name "progress bar."

After the progress bar, it prints out the time it took to run and then a summary of what was run. In this case, we ran one example and it passed, so there are no failures.

Now try running it with the ruby command instead:

```
ruby simple_math_spec.rb
```

You should see the same output. When executing individual spec files, the spec and ruby commands are somewhat interchangeable. We do, however, get some added value from the spec command when running more than just one file.

Running Several Specs at Once

Running specs directly is handy if you just want to run one single file, but in most cases you really want to run many of them in one go. The simplest way to do this is to just pass the directory containing your spec files to the spec command. So if your spec files are in the spec directory (they are, aren't they?), you can just do this:

```
spec spec
```

...or if you're in a Rails project:

```
script/spec spec
```

In either case, the spec command will load all of the spec files in the spec directory and its sub-directories. By default, the spec command only loads files ending with _spec.rb. As you'll see later in this chapter, while this pattern is the convention, you can configure RSpec to load files based on any pattern you choose.

Being able to execute the files is only the tip of the iceberg. The spec command offers quite a few options, so let's take a closer look at them.

Diff output with --diff

One of the most common expectations in code examples is that an object should match an expected value. For example, comparing two strings:

Download tools/command_line/diff_spec.rb

```
    bill.to_text.should == <<-EOF
From: MegaCorp
To: Bob Doe
Ref: 9887386
Note: Please pay imminently
EOF
```

The *here doc* defines the expected result, and it is compared to the actual result of the to_text() method. If the to_text() method returns a different string the example will fail, and if the difference is subtle it can be hard to spot. Let's assume we goofed the implementation by forgetting to add the last name and hardcoded a silly message because we were irritated and working overtime. Without the --diff option the output would be:

```
expected: "From: MegaCorp\nTo: Bob Doe\nRef: 9887386\nNote: Please pay ...
     got: "From: MegaCorp\nTo: Bob\nRef: 9887386\nNote: We want our money ...
```

It's not exactly easy to spot where the difference is. Now, let's add the --diff option to the command line and run it again. This time we'll see:

```
Diff:

@@ -1,5 +1,5 @@
 From: MegaCorp
-To: Bob
+To: Bob Doe
 Ref: 9887386
-Note: We want our money!
+Note: Please pay imminently
```

The diff format shows the difference of each line. It uses Austin Ziegler's excellent diff-lcs Ruby gem, which you can install with:

```
gem install diff-lcs
```

Diffing is useful for more than strings. If you compare two objects that are not strings, their #inspect representation will be used to create the diff.

Tweaking the output with --format

By default, RSpec will report the results to the console's standard output by printing something like ...F.....F.... followed by a backtrace for each failure. This is fine most of the time, but sometimes you'll want a more expressive form of output. RSpec has several built-in formatters that provide a variety of output formats. You can see a full list of all the built-in formatters with RSpec's --help option.

For example, the specdoc formatter can be used to print out the results as specdoc. The specdoc format is inspired from TestDox (see the sidebar).

You activate it simply by telling the spec command:

```
spec path/to/my/specs --format specdoc
```

The output will look something like the following:

```
Stack (empty)
- should be empty
- should not be full
- should add to the top when sent #push
- should complain when sent #peek
- should complain when sent #pop

Stack (with one item)
- should not be empty
- should return the top item when sent #peek
- should NOT remove the top item when sent #peek
- should return the top item when sent #pop
- should remove the top item when sent #pop
- should not be full
- should add to the top when sent #push
```

If you use nested example groups, like this:

```
describe Stack do
  context "when empty" do
    it "should be empty" do
```

Then you can use the *nested* format, like this:

TestDox

In 2003, Chris Stevenson, who was working with Aslak in ThoughtWorks at the time, created a little Java tool called TestDox (http://agiledox.sourceforge.net/). What it did was simple: It scanned Java source code with JUnit tests and produced textual documentation from it. The following Java source code...

```
public class AccountDepositTest extends TestCase {
    public void testAddsTheDepositedAmountToTheBalance() { ... }
}
```

...would produce the following text:

```
Account Deposit
- adds the deposited amount to the balance
```

It was a simplistic tool, but it had a profound effect on the teams that were introduced to it. They started publishing the TestDox reports for everyone to see, encouraging the programmers to write real sentences in their tests, lest the TestDox report should look like gibberish.

Having real sentences in their tests, the programmers started to think about behaviour, what the code should do, and the BDD snowball started to roll...

```
spec path/to/my/specs --format nested
```

and generate output like this:

```
Stack
  when empty
    should be empty
    should not be full
    should add to the top when sent #push
    should complain when sent #peek
    should complain when sent #pop
  with one item
    should not be empty
    should return the top item when sent #peek
    should NOT remove the top item when sent #peek
    should return the top item when sent #pop
    should remove the top item when sent #pop
    should not be full
    should add to the top when sent #push
```

RSpec also bundles a formatter that can output the results as HTML. You probably don't want to look at the HTML in a console, so you should

Several formatters?

RSpec lets you specify several formatters simultaneously by using several --format options on the command line. Now why would anyone want to do that? Maybe you're using a continuous integration (CI) environment to build your code on every checkin. If both you and the CI use the same Rake tasks to run RSpec, it can be convenient to have one progress formatter that goes to standard output, and one HTML formatter that goes to a file.

This way you can see the CI RSpec result in HTML and your own in your console—and share the Rake task to run your specs.

tell RSpec to output the HTML to a file:

```
spec path/to/my/specs --format html:path/to/my/report.html
```

For all of the formatters, RSpec will treat whatever comes after the colon as a file, and write the output there. Of course, you can omit the colon and the path, and redirect the output to a file with >, but using the --format flag supports output of multiple formats simultaneously to multiple files, like so:

```
spec path/to/my/specs --format progress \
                      --format nested:path/to/my/report.txt \
                      --format html:path/to/my/report.html
```

After you have done this and opened the resulting HTML file in a browser, you should see something like Figure 14.1, on the next page.

Finally, the profile formatter works just like the default progress formatter, except that it also outputs the 10 slowest examples. We really recommend using this to constantly improve the speed of your code examples and application code.

Loading extensions with --require

If you're developing your own extensions to RSpec, such as a custom --formatter or maybe even a custom --runner, you must use the --require option to load the code containing your extension.

The reason you can't do this in the spec files themselves is that when they get loaded, it's already too late to hook in an RSpec plugin, as RSpec is already running.

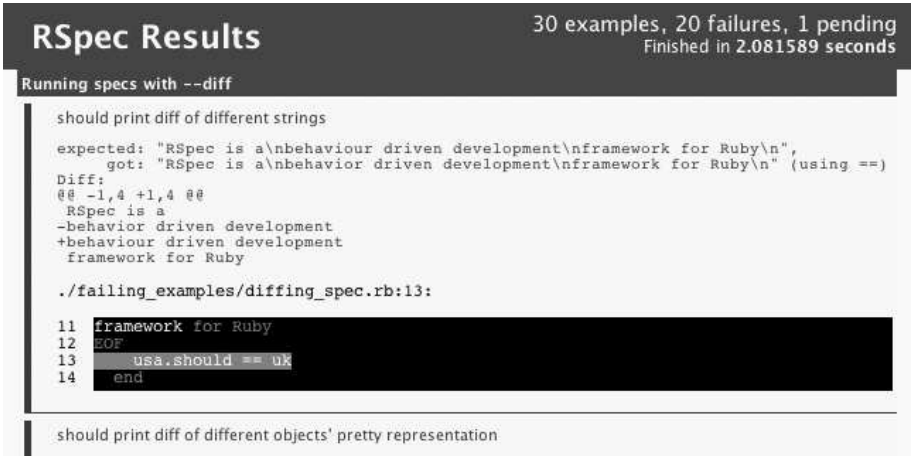


Figure 14.1: HTML Report

Getting the noise back with --backtrace

Have you ever seen a backtrace from a failing test in an xUnit tool? It usually starts with a line in your test or the code being tested, and then further down you'll see ten furlongs of stack frames from the testing tool itself. All the way to where the main thread started.

Most of the time, most of the backtrace is just noise, so with RSpec you'll only see the frames from *your* code. The entire backtrace can be useful from time to time, such as when you think you may have found a bug in RSpec, or when you just want to see the whole picture of why something is failing. You can get the full backtrace with the --backtrace flag:

```
spec spec --backtrace
```

Colorize Output with --color

If you're running the specs all the time (you are, aren't you?), it requires some focus to notice the difference between the command line output from one run and the next. One thing that can make it easier on the eyes is to colorize the output, like this:

```
spec spec --color
```

With this option, passing examples are indicated by a green dot (.), failing examples by a red F, and pending examples by a yellow asterisk

(*). Error reports for any failing examples are red.

The summary line is green if there are no pending examples and all examples pass. If there are any failures it is red. If there are no failures, but there are pending examples, it is yellow. This makes it much easier to see what's going on by just looking at the summary.

Invoke With Options Stored in a File with --options

You can store any combination of these options in a file and tell the spec command where to find it. For example, you can add this to spec/spec.opts:

```
--color
--format specdoc
```

You can list as many options as you want, with one or more words per line. As long as there is a space, tab or newline between each word, they will all be parsed and loaded. Then you can run the code examples with this command:

```
spec spec --options spec/spec.opts
```

That will invoke the options listed in the file.

Generate an Options File with --generate-options

The --generate-options option is a nice little shortcut for generating the options file referenced in the previous section. Let's say that we want to generate spec/spec.opts with --color and --format html:examples.html. Here's what the command would look like:

```
spec --generate-options spec/spec.opts \
  --color \
  --format html:examples.html
```

Then you can run the specs using the –options option:

```
spec spec --options spec/spec.opts
```

14.2 TextMate

The RSpec Development Team maintains a TextMate bundle which provides a number of useful commands and snippets. The bundle has been relatively stable for some time now, but when we add new features to RSpec, they are sometimes accompanied with an addition or a change to the TextMate bundle.

We maintain the bundle in two different locations: in the official TextMate Bundle subversion repository at http://svn.textmate.org/trunk/Bundles/RubyRSpec.tmbundle and our development source repository at http://github.com/dchelimsky/rspec-tmbundle.

We update the subversion repository with each rspec release, so if you prefer to stick with rspec releases, the official TextMate repository is a simple and clean option. Just follow the bundle maintenance instructions on the TextMate website at http://manual.macromates.com/en/bundles.

If, however, you're an early adopter who likes to keep a local copy of rspec's git repository and update it regularly to keep up with the latest changes, then you'll want to do the same with the TextMate bundle. Instructions for this can be found on the rspec-tmbundle github wiki at http://github.com/dchelimsky/rspec-tmbundle/wikis.

14.3 Autotest

Autotest is one of several tools that ship with Seattle.rb's ZenTest library. The basic premise is that you open up a shell, fire up autotest, and it monitors changes to files in specific locations. Based on its default mappings, every time you save a test file, autotest will run that test file. And every time you save a library file, autotest will run the corresponding test file.

When you install the rspec gem, it installs an autospec command, which is a thin wrapper for autotest that lets you use autotest with projects developed with RSpec.

To try this out, open up a shell and cd to the mastermind directory that you created back in Chapter 2, *Describing Features with Cucumber*, on page 20. If you use command line editors like vim or emacs, open up a second shell to the same directory, otherwise open the project in your favorite text editor.

In the first shell, type the autospec command. You should see it start up and execute a command which loads up some number of spec files and runs them. Now, go to one of the spec files and change one of the code examples so it will fail and save the file. When you do, autotest will execute just that file and report the failure to you. Note that it only runs *that* file, not all of the code example files.

Now reverse the change you just made so the example will pass and save the file again. What autotest does now is quite clever. First it runs the one file, which is the one with failures from the last run, and sees that all the examples pass. Once it sees that the previous failures are now passing, it loads up the entire suite and runs all of the examples again.

I can tell you that when I first heard about autotest, I thought it sounded really interesting, but wasn't moved by it. Then I actually tried it. All I can say is *try it*.

By default, autotest maps files in the lib directory to corresponding files in the test directory. For example, if you have a lib/account.rb file and a test/test_account.rb file, each time you save either autotest will run test/test_account.rb.

These mappings are completely configurable, so if you prefer to name your test files account_test.rb instead of test_account.rb, you can configure autotest to pay attention to files ending with _test.rb rather than starting with test_. See the ZenTest rdoc for more information about configuring these mappings.

RSpec uses standard autotest hooks to modify the autotest mappings to cater to RSpec's conventions. So if you run autospec and you modify spec/mastermind/game_spec.rb or lib/mastermind/game.rb, autotest will run spec/mastermind/game_spec.rb.

rspec-rails modifies the mappings even further, so when you save app/models/account.rb, its code examples in spec/models/account_spec.rb will be run automatically.

14.4 Heckle

Heckle is a *mutation testing* tool written by Ryan Davis and Kevin Clark. From heckle's rdoc:

Heckle is a mutation tester. It modifies your code and runs your tests to make sure they fail. The idea is that if code can be changed and your tests don't notice, either that code isn't being covered or it doesn't do anything.

To run heckle against your specs, you have to install the heckle gem, and then identify the class you want to heckle on the command line.

To heckle the Game class from the Mastermind tutorial in Part I, you would do this:

```
spec spec/mastermind/game_spec.rb --heckle Mastermind::Game
```

Depending on how far you got in the tutorial, the output looks something like this:

```
Line 1  ********************************************************************
2       ***  Mastermind::Game#start loaded with 4 possible mutations
3       ********************************************************************
4
5       4 mutations remaining...
6       3 mutations remaining...
7       2 mutations remaining...
8       1 mutations remaining...
9       No mutants survived. Cool!
```

Line 2 indicates that heckle found four opportunities to mutate the code in the start() method. Heckle prints out 4 mutations remaining..., and mutates the code. Perhaps it changes the value of an argument to the method. Perhaps it changes a conditional expression to return true or false, rather than performing a calculation.

Heckle then runs the examples against the mutated code. If the mutation *survives*, meaning there are no failures, then the examples aren't really robust enough to fully cover all of the different paths through the code. It is, therefore, a good thing if the mutation does *not* survive.

"No mutants survived", on line 9, tells us that there were failures after each mutation, so our code examples are sufficiently robust.

You can run heckle against all of the classes in a module by naming just that module. This command would run all of the specs in the spec/ directory, and heckle every class it could find in the Mastermind module.

```
spec spec --heckle Mastermind
```

You can also run heckle against a single method, like so:

```
spec spec --heckle Mastermind::Game#start
```

This would only heckle the start() method, ignoring the other methods defined in the Game class.

As of version 1.4.1, released back in 2007, heckle will only mutate instance methods, so this won't check your class methods or methods defined in a module, unless that module is included in a class that heckle can find.

14.5 Rake

Rake is a great automation tool for Ruby, and RSpec ships with custom tasks that let you use RSpec from Rake. You can use this to define one or several ways of running your examples. For example, rspec-rails ships with several different tasks:

```
rake spec             # Run all specs in spec directory (excluding plugin specs)
rake spec:controllers # Run the code examples in spec/controllers
rake spec:helpers     # Run the code examples in spec/helpers
rake spec:models      # Run the code examples in spec/models
rake spec:views       # Run the code examples in spec/views
```

This is only a partial list. To see the full list, cd into the root of any rails app you have using RSpec and type rake -T | grep "rake spec". All of these tasks are defined using the Spec::Rake::SpecTask.

Spec::Rake::SpecTask

The Spec::Rake::SpecTask class can be used in your Rakefile to define a task that lets you run your specs using Rake. The simplest way to use it is to put the following code in your Rakefile:

```
require 'spec/rake/spectask'

Spec::Rake::SpecTask.new
```

This will create a task named spec that will run all of the specs in the spec directory (relative to the directory rake is run from—typically the directory where Rakefile lives). Let's run the task from a command window:

```
rake spec
```

Now that's simple! But that's only the beginning. The SpecTask exposes a collection of useful configuration options that let you customize the way the command runs.

To begin with, you can declare any of the command line options. If you want to have the SpecTask colorize the output, for example, you would do this:

```
Spec::Rake::SpecTask.new do |t|
  t.spec_opts = ["--color"]
end
```

spec_opts takes an Array of Strings, so if you also wanted to format the output with the specdoc format, you could do this:

```
Spec::Rake::SpecTask.new do |t|
```

About Code Coverage

Code coverage is a very useful metric, but be careful, as it can be misleading. It is possible to have a suite of specs that execute 100% of your codebase without ever setting any expectations. Without expectations you'll know that the code will probably run, but you won't have any way of knowing if it behaves the way you expect it to.

So while low code coverage is a clear indicator that your specs need some work, high coverage does not necessarily indicate that everything is honky-dory.

```
  t.spec_opts = ["--color", "--format", "specdoc"]
end
```

Check the rdoc for Spec::Rake::SpecTask to see the full list of configuration options.

14.6 RCov

RCov is a code coverage tool. The idea is that you run your specs and rcov observes what code in your application is executed and what is not. It then provides a report listing all the lines of code that were never executed when you ran your specs, and a summary identifying the percentage of your codebase that is covered by specs.

There is no command line option to invoke rcov with RSpec, so you have to set up a rake task to do it. Here's an example (this would go in Rakefile):

```
require 'rake'
require 'spec/rake/spectask'

namespace :spec do
  desc "Run specs with RCov"
  Spec::Rake::SpecTask.new('rcov') do |t|
    t.spec_files = FileList['spec/**/*_spec.rb']
    t.rcov = true
    t.rcov_opts = ['--exclude', '\/Library\/Ruby']
  end
end
```

This is then invoked with rake spec:rcov and produces a report that excludes any file with /Library/Ruby as part of its path. This is useful if your library depends on other gems, because you don't want to include the code in those gems in the coverage report. See rcov's documentation for more info on the options it supports.

As you can see, RSpec's spec command offers you a lot of opportunities to customize how RSpec runs. Combine that with powerful tools like Rake, Autotest, and Heckle and you've got a great set of tools you can use to drive out code with code examples, and run metrics against your specs to make sure you've got good code coverage (with rcov) and good branch coverage (with heckle).

Chapter 15

Extending RSpec

RSpec provides a wealth of functionality out of the box, but sometimes we want to express things in more domain specific ways, or modify the output format to make it better serve as documentation for a specific audience. In this chapter, we'll explore the utilities and extension points that RSpec provides to satisfy these needs.

15.1 Global Configuration

RSpec exposes a configuration object that supports the definition of global before and after blocks, as well as hooks to include modules in examples, or extend example group classes. We can access it via the Spec::Runner module like this:

```
Spec::Runner.configure {|config| ... }
```

The config block argument is the configuration object, and it exposes the following methods:

before(scope = :each, options={}, &block) Though more commonly used, this is an alias for append_before.

append_before(scope = :each, options={}, &block) Appends the submitted block to the list of before blocks that get run by every example group. scope can be any of :each, :all, or :suite. If :each, the block is run before each example. If :all, the block is run once per group, before any examples have been run. If :suite, the block is run once before any example groups have run.

The only supported option is :type, which allows you to limit the inclusion of this before block to example groups of the specified type. For example, with rspec-rails, you might say something like:

```
config.before(:each, :type => :controller) do
  ...
end
```

This would cause the submitted block to be run before each controller example, but no other types of examples. See Section 15.2, *Custom Example Groups*, on the next page for more information.

prepend_before(scope = :each, options={}, &block) Just like append_before(), but adds the block to the beginning of the list instead of the end. This is rarely useful, as anything added to the global list is going to run before anything added in individual example groups and examples. If you're using another library that extends RSpec, however, and you really need your before block to run first, prepend_before() is your friend.

after(scope = :each, options={}, &block) Though more commonly used, this is an alias for prepend_after.

prepend_after(scope = :each, options={}, &block) Adds the submitted block to the beginning of the list of after blocks that get run by every example group. See append_before(), above, for notes about scope.

append_after(scope = :each, options={}, &block) Just like prepend_after(), but adds the block to the end of the list.

include(*modules, options={}) Includes the submitted module or modules in every example group. Methods defined in submitted modules are made available to every example.

Like the before() and after() methods, the options hash supports a :type option that lets you limit the inclusion of the module(s) to a specific type of example group.

extend(*modules, options={}) Extends every example group with the submitted module or modules. Methods defined in submitted modules are made available to every example group. This is the easiest way to make macros (see Section 15.4, *Macros*, on page 186) available to example groups.

mock_with(framework) By default, RSpec uses its own mocking framework. You can, however, choose any framework. framework can be

a Symbol or a module reference. If it's a symbol, it can be any of :rspec (default), :mocha, :flexmock, and :rr. These all reference adapters that ship with RSpec.

If you use a different mock framework, or perhaps you've written your own, you can write an adapter module for it, and then pass that module to mock_with(). See Chapter 12, *Mocking in RSpec*, on page 151 for more information about writing your own adapter.

Each of these methods supports extending example groups by including modules, extending them with modules, or adding to their lists of before and after blocks. While these are very useful ways of extending groups, sometimes we need something a bit more robust. For cases like this, we can write our own example group classes.

15.2 Custom Example Groups

In Michael Feathers' presentation at SD West 2007, API Design As If Unit Testing Mattered,[1] he suggested that API designers should not just test their own code, but they should test code that *uses their code*! He also suggested that they should ship the tools that they develop to do this with the software, so that developers using their APIs have an easy path to testing their own code.

If you've worked with Ruby on Rails' built-in testing support, you know well the result of doing this. Rails' ships with specialized subclasses of Test::Unit::TestCase that bring us domain-specific commands like get() and post(), and assertions like assert_template(). These extensions make tests for our rails apps a joy to write and a snap to read.

In the next part of the book, you'll learn about *rspec-rails*, the extension library that brings RSpec to Rails development. The rspec-rails gem ships with custom example groups that actually extend the Rails TestCase classes, providing developers with all of the utilities that ship with Rails, *plus* the additional facilities that come with RSpec.

In this section, we'll explore approaches to authoring custom example groups. Whether shipping a domain-specific spec framework with your library, or developing one for internal use, we think you'll find this quite simple and flexible.

1. http://www.objectmentor.com/resources/articles/as_if_unit_testing_mattered.pdf

Object Model

In order to understand how and when to write a custom example group class, let's explore RSpec's object model first. It is quite simple but, because it is hidden behind RSpec's DSL, it is not always easy to spot without some internal inspection. Here's an example for discussion:

```
describe Account do
  it "has a balance of zero when first created" do
    account = Account.new
    account.balance.should == Money.new(0)
  end
end
```

As you read in Section 10.1, *Describe It!*, on page 108, the describe() method creates a subclass of Spec::Example::ExampleGroup, and the it() method creates a method on that class. If you look at the code for Spec::Example::ExampleGroup, however, you'll only see this:

```
module Spec
  module Example
    class ExampleGroup
      extend  Spec::Example::ExampleGroupMethods
      include Spec::Example::ExampleMethods
    end
  end
end
```

Spec::Example::ExampleGroup is really just a wrapper for the ExampleGroupMethods and ExampleMethods modules that define the behaviour of an example group. This design lets RSpec use Spec::Example::ExampleGroup as a default example group base class, while also allowing us to choose an entirely different base class and add RSpec behaviour to it.

This is how RSpec supports interoperability with test/unit. We just reopen Test::Unit::TestCase, and add RSpec behaviour to it. Of course, in the case of test/unit it's not *quite* that simple because RSpec does some things that test/unit already does, so there is some monkey patching involved. But given that test/unit ships with Ruby 1.8, the risk of changes to test/unit impacting RSpec and, consequently, RSpec users, is very low.

So now we have three ways to write a custom example group base class. We can subclass Spec::Example::ExampleGroup, we can write our own from scratch, adding example group behaviour the same way that RSpec does in Spec::Example::ExampleGroup, or we can add the behaviour to a class from an entirely different library like test/unit, or minitest.

Registering a custom default example group class

Once we have a custom subclass, we need to tell RSpec to use it instead of its own ExampleGroup class. We do this by registering the class with RSpec's ExampleGroupFactory. Here's how we register a custom class as the default base class for example groups:

```
Spec::Example::ExampleGroupFactory.default(CustomExampleGroup)
```

This does two powerful things. First, the describe() method creates a subclass of CustomExampleGroup (in this example). Second, CustomExampleGroup is assigned to the constant, Spec::ExampleGroup, which is guaranteed to reference the default base class whether its RSpec's own Spec::Example::ExampleGroup or a custom class. If a library ships with its own default base class, end-users can still add facilities to it by simply re-opening Spec::ExampleGroup and add utilities to it, regardless of its class.

Named example group classes

Developing our own subclass is a nice first step, but sometimes we have different needs for different parts of our system. In rspec-rails, for example, we have different example groups for specifying models, controllers, views, and even helpers and routing. Each of these types of example groups has different needs.

Controller specs need methods like get() and post(), and expectations like should render_template(). Model specs don't need any of those facilities, but they do need a means of isolating database state from one example to the next.

In order to support different example group classes for different purposes within a single spec suite, RSpec's ExampleGroupFactory lets us register classes with keys to access them. Here's how rspec-rails does this with its ControllerExampleGroup:

```
Spec::Example::ExampleGroupFactory.register(:controller, self)
```

This code appears within the ControllerExampleGroup, so self is referencing that.

Once a class is registered, we can coerce RSpec into returning a subclass of the class we want in two different ways. The more obvious way is to explicitly name it in the describe() declaration, like this:

```
describe WidgetsController, :type => :controller do
  ...
end
```

When the ExampleGroupFactory receives this request (delegated from the call to describe()), it first looks to see if a :type is declared. If so, it returns a subclass of the class registered with that name.

If you're already an rspec-rails user, you very likely have not seen that options hash appended to calls to describe() before. That's because when the ExampleGroupFactory fails to find a :type key in the request, it then inspects the path to the file in which the group was declared. In the case of controller specs, they are conventionally stored in spec/controllers. The ExampleGroupFactory extracts "controllers" from that path, converts it from a string to a symbol, and looks to see if it has a class registered with :controllers as its key.

If there is no :type declaration and no subclass mapped to the path, then the ExampleGroupFactory creates a subclass of the default example group class.

Now that we have a means of separating behaviour for different needs in different example group classes, the next thing we'll talk about is how to develop custom matchers that *speak* in our domain.

15.3 Custom Matchers

RSpec's built-in matchers support most of the expectations we'd like to write in our examples out of the box. There are cases, however, in which a subtle change would allow us to express exactly what we want to say rather than *almost* exactly what we want to say. For those situations we can easily write our own custom matchers.

You're already using some of these if you're using the rspec-rails gem. render_template(), for example, is a Rails-domain-specific matcher for expecting that a specific template gets rendered by a controller action. Without that matcher, we'd write expectations such as:

```
response.rendered_template.should == "accounts/index"
```

With this custom matcher, we are able to write examples using language closer to the domain:

```
response.should render_template("accounts/index")
```

All of RSpec's built-in matchers follow a simple protocol, which we use to write our own custom matchers from scratch. We'll go over the the protocol in a bit, but first let's take a look at RSpec's Matcher DSL for defining custom matchers in just a few lines of code.

Matcher DSL

RSpec's Matcher DSL makes defining custom matchers a snap.[2] Let's say we're working on a personnel application and we want to specify that joe.should report_to(beatrice).

To get there, we'd probably start off with something like joe.reports_to?(beatrice).should be_true. That's a good start, but it presents a couple of problems. If it fails, the failure message says expected true, got false. That's accurate, but not very helpful.

Another problem is that it just doesn't read as well as it could. We really want to say joe.should report_to(beatrice). And if it fails, we want the message to tell us we were expecting an employee who reports to beatrice.

We can solve the readability and feedback problems using RSpec's Matcher DSL to generate a report_to() method, like this:

```
Spec::Matchers.define :report_to do |boss|
  match do |employee|
    employee.reports_to?(boss)
  end
end
```

The define() method on Spec::Matchers defines a report_to() method that accepts a single argument. We can then call report_to(beatrice) to create an instance of Spec::Matchers::Matcher configured with beatrice as the boss, and the match declaration stored for later evaluation.

Now when we say that joe.should report_to(beatrice), the report_to method creates a in instance of Spec::Matchers::Matcher that will call the block with joe.

The match block should return a boolean value. True indicates a match, which will pass if we use should() and fail if we use should_not(). False indicates no match, which will do the reverse: fail if we use should() and pass if we use should_not().

In the event of a failure, the matcher generates a message from its name and the expected and actual values. In this example the message would be something like this:

```
expected <Employee: Joe> to report to <Employee: Beatrice>
```

2. The matcher DSL is based on suggestions from Yehuda Katz.

The representation of the employee objects depends on how to_s() is implemented on the Employee class, but the matcher gleans "report to" from the Symbol passed to define().

In the event of a failure using should_not(), the generated message would read like this:

```
Spec::Matchers.define :report_to do |boss|
  match do |employee|
    employee.reports_to?(boss)
  end

  failure_message_for_should do |employee|
    "expected the team run by #{boss} to include #{employee}"
  end

  failure_message_for_should_not do |employee|
    "expected the team run by #{boss} to exclude #{employee}"
  end

  description do
    "expected a member of the team run by #{boss}"
  end
end
```

Those messages work generally well, but sometimes we'll want a bit of control over the failure messages. We can get that by overriding them, and the description, with blocks that returns the messages we want.

```
Spec::Matchers.define :report_to do |boss|
  match do |employee|
    employee.reports_to?(boss)
  end

  failure_message_for_should do |employee|
    "expected the team run by #{boss.inspect} to include #{employee.inspect}"
  end

  failure_message_for_should_not do |employee|
    "expected the team run by #{boss.inspect} to exclude #{employee.inspect}"
  end

  description do
    "expected a member of the team run by #{boss.inspect}"
  end
end
```

The block passed to failure_message_for_should() will be called and the result displayed in the event of a should() failure. The block passed to failure_message_for_should_not() will be called and the result displayed

in the event of a should_not() failure. The description() will be displayed when this matcher is used to generate its own description.

As with the stock matchers, RSpec's matcher DSL will probably cover 80% of the remaining 20%. Still, there are cases where you'll want even more control over certain types of things. As of this writing, for example, there is no support for passing a block to the matcher itself. RSpec's built-in change() matcher needs that ability to express expectations like this:

```
account = Account.new
lambda do
  account.deposit(Money.new(50, :USD))
end.should change{ account.balance }.by(Money.new(50, :USD))
```

We can't easily define a matcher that accepts a block with the DSL because Ruby won't let us pass one block to another without first packaging it as a Proc object. We probably could do it with some gymnastics, but in cases like this it is often simpler to just write some clear code using RSpec's Matcher Protocol.

Matcher Protocol

A matcher in RSpec is any object that responds to a specific set of messages. The simplest matchers only need to respond to these two:

matches? The should() and should_not() methods use this to decide if the expectation passes or fails. Return true for a passing expection; false for a failure.

failure_message_for_should The failure message to be used when you use should() and matches?() returns false.

Here's the report_to() matcher we used in Section 15.3, *Matcher DSL*, on page 183, written using these two methods:

```
class ReportTo
  def initialize(manager)
    @manager = manager
  end

  def matches?(employee)
    @employee = employee
    employee.reports_to?(@manager)
  end

  def failure_message_for_should
    "expected #{@employee} to report to #{@manager}"
  end
```

```
end

def report_to(manager)
  ReportTo.new(manager)
end
```

This is clearly more verbose than the Matcher DSL, as we have to define a class *and* a method. We also have to store state in order to generate the failure message, which is not necessary in the DSL because it delivers the actual and expected objects to the match and message declaration blocks. Still, if writing a matcher out this way is more expressive than using the DSL in a given circumstance, then a custom matcher from scratch is the way to go.

The following methods are also part of the protocol, supported by the should() and should_not() methods, but completely optional:

failure_message_for_should_not optional - the failure message to be used when you use should_not() and matches?() returns true.

description optional - the description to be displayed when you don't provide one for the example (i.e. it { ... } instead of it "should ... " do ... end)

does_not_match? optional - rarely needed, but on occasion it can be useful for the matcher to know if it's being called by should() or should_not(). does_not_match?() will *only* be called by should_not()

With just these few methods and the expressive support of the Ruby language, we can create some sophistocated matchers. While we recommend using the Matcher DSL first, this simple protocol offers a robust back-up plan.

15.4 Macros

Custom matchers can help us to build up domain-specific DSLs for specifying our code, but they still require a bit of repetative ceremony. In rspec-rails, for example, it is quite common to see examples like this:

```
describe Widget do
  it "requires a name" do
    widget = Widget.new
    widget.valid?
    widget.should have(1).error_on(:name)
  end
end
```

With a custom matcher, we can clean that up a bit:

```
describe Widget do
  it "requires a name" do
    widget = Widget.new
    widget.should require_attribute(:name)
  end
end
```

We can even get more terse by taking advantage of the implicit subject, which you read about in Section 11.7, *Implicit Subject*, on page 149, like this:

```
describe Widget do
  it { should require_attribute(:name) }
end
```

Now that is terse, expressive, and complete all at the same time. But for the truly common cases like this, we can do even better. In 2006, the *shoulda* library emerged as an alternative to RSpec for writing more expressive tests.[3] One of the innovations that came from shoulda was *macros* to express the common, redundant things we want to express in tests. Here's the widget example with a shoulda macro instead of a custom matcher:

```
class WidgetTest < Test::Unit::TestCase
  should_require_attributes :name
end
```

In late 2007, Rick Olsen introduced his own rspec-rails extension library named *rspec_on_rails_on_crack*.[4], which added macros to rspec-rails. In rspec_on_rails_on_crack, the widget example looks like this:

```
describe Widget do
  it_validates_presence_of Widget, :name
end
```

Macros like this are great for the things that are ubiquitous in our applications, like Rails' model validations. They're a little bit like shared example groups, which you read about in Section 10.5, *Shared Examples*, on page 121, but they are more expressive because they have unique names, and, unlike shared examples, they can accept arguments.

Macros are also quite easy to add to RSpec. Let's explore a simple example. Here is some code that you might find in a typical controller spec.

3. http://www.thoughtbot.com/projects/shoulda
4. http://github.com/technoweenie/rspec_on_rails_on_crack

```
describe ProjectsController do
  context "handling GET index" do
    it "should render the index template" do
      get :index
      controller.should render_template("index")
    end

    it "should assign @projects => Project.all" do
      Project.should_receive(:all).and_return(['this array'])
      get :index
      assigns[:projects].should == ['this array']
    end
  end
end
```

This would produce output like this:

```
ProjectsController handling GET index
- should render the index template
- should assign @projects => Project.all
```

Using macros inspired by rspec_on_rails_on_crack and shoulda, we can express the same thing at a higher level and get the same output like this:

Download extending_rspec/macro_example/spec/controllers/projects_controller_spec.rb

```
describe ProjectsController do
  get :index do
    should_render "index"
    should_assign :projects => [Project, :all]
  end
end
```

The underlying code is quite simple for the experienced Rubyist:

Download extending_rspec/macro_example/spec/spec_helper.rb

```
module ControllerMacros
  def should_render(template)
    it "should render the #{template} template" do
      do_request
      response.should render_template(template)
    end
  end

  def should_assign(hash)
    variable_name = hash.keys.first
    model, method = hash[variable_name]
    model_access_method = [model, method].join('.')
    it "should assign @#{variable_name} => #{model_access_method}" do
      expected = "the value returned by #{model_access_method}"
      model.should_receive(method).and_return(expected)
```

```
      do_request
      assigns[variable_name].should == expected
    end
  end

  def get(action)
    define_method :do_request do
      get action
    end
    yield
  end
end

Spec::Runner.configure do |config|
  config.use_transactional_fixtures = true
  config.use_instantiated_fixtures  = false
  config.fixture_path = RAILS_ROOT + '/spec/fixtures/'
  config.extend(ControllerMacros, :type => :controller)
end
```

The get() method defines a method that is used internally within the macros named do_request(), and yields to the block that contains the other macros, giving them access to the do_request() method.

The should_assign() method seems a bit complex, but it goes out of its way to provide you nice feedback so when you're writing the examples first (as I trust you are), you'll get a failure message like this:

```
expected: "the value returned by Project.all",
     got: nil (using ==)
```

We exposed these macros to controller specs by extending all controller example groups with the ControllerMacros module in the last line of the configuration. If we didn't want them in all controller specs, we could also explicitly extend individual groups inline, like this:

```
describe ProjectsController do
  extend ControllerMacros
  ...
```

At this point we've explored a number of ways to make RSpec code examples more expressive, but all of these techniques apply only to the *input*: the code we write and read in our examples. This is great if you're a developer, but part of RSpec's value-add is its ability to customize output for different audiences. We'll explore how RSpec does this and how we can customize it in the next section.

15.5 Custom Formatters

RSpec uses message formatters to generate the output you see when running a suite of specs. These formatters receive notification of events, such as when an example group is about to be run, or an individual example fails.

RSpec ships with a number of built-in formatters designed to generate plain text output, an all-purpose html formatter, and a TextMate-specific html formatter as well. You're probably already familiar with the progress bar formatter, which is the default formatter when you run the spec command with no options. Run spec --help to see a full listing of all of the built-in formatters.

If none of the built-in formatters satisfy your specific reporting needs, you can easily create a *custom formatter*. This can be very useful for building out custom spec reports for co-workers or a client. And if you happen to be an IDE developer, custom formatters are definitely your friend.

In this section, we'll review the APIs for the various parts of the puzzle that RSpec uses to write all of its built-in formatters, and anybody can use to write a custom formatter.

Formatter API

The simplest way to write a custom formatter is to subclass Spec::Runner::Formatter::BaseFormatter, which implements all of the required methods as no-ops. This allows us to implement only the methods we care about, and reduces the risk that changes in future versions of RSpec will impact the formatter.

Here is a list of all the required methods as of this writing, but be sure to look at the documentation for Spec::Runner::Formatter::BaseFormatter to ensure that you have the latest information.

initialize(options, output) When formatters are initialized, they are handed an options struct with colour and dry_run options to help determine how to format output.

The output is STDOUT by default, but can be overridden on the command line to be a filename, in which case a File object is passed to initialize().

To handle either possibility, RSpec's built-in formatters write to the output object with output << "text", which works for any IO object.

start(example_count) This is the first method that is called. example_count is the total count of examples that will be run.

example_group_started(example_group_proxy) Called as an example group is started. See Section 15.5, *ExampleGroupProxy*, on the following page for more about example_group_proxy. There is no corresponding example_group_finished message because we have not found a use for one in any of RSpec's built-in formatters.

example_started(example_proxy) Called as an example is started. See below for more about the example_proxy.

example_pending(example_proxy, message) Called when an example is pending. The example_proxy is the same object that was passed to example_started(). The message is the message passed to the pending method, or a default message generated by RSpec (see Section 5.3, *Pending*, on page 80 for more information).

example_failed(example_proxy, counter, failure) Called when an example fails. The example_proxy is the same object that was passed to example_started(). The counter indicates the sequential number of this failure within the current run. So if there are seven failures, and this is the last, counter will be the number 7. See below for more information about the failure object.

example_passed(example_proxy) Called when an example passes. The example_proxy is the same object that was passed to example_started().

start_dump() Called after all of the code examples have been executed. The next method to be called will be dump_failure() if there are any failures.

dump_failure(counter, failure) Called once for each failure in a run. counter is the sequential number of this failure, and is the same as the counter passed to example_failed() for this example. See below for more information about the failure object.

dump_summary(duration, example_count, failure_count, pending_count) Called after any calls to dump_failure(). duration is the total time it took to run the suite. example_count is the total number of examples that were run. failure_count is the number of examples that failed. pending_count is the number of examples that are pending.

dump_pending() Called after dump_summary(), and is a trigger to output messages about pending examples. It is up to the formatter

to collect information about pending examples and generate any output at this point.

close() Called once at the very end of the run, signaling the formatter to clean up any resources it still has open.

ExampleGroupProxy

An ExampleGroupProxy is a lightweight proxy for an example group. This is the object sent to the example_group_started() method, and it carries the following information that can be useful in formatters:

description This is the complete description string, built by concatenating the strings and objects passed to the describe() or context() method, and all of its parents in nested groups. For example, this code:

```
describe ParkingTicket do
  context "on my windshield"
```

would produce "ParkingTicket" when starting the outer group, and "ParkingTicket on my windshield" when starting the inner group.

nested_descriptions Similar to description, except the formatted strings for each group are not concatenated. In the ParkingTicket example, the nested_descriptions for the outer group would be ["ParkingTicket"], and the inner group would get ["ParkingTicket","on my windshield"].

This is used by RSpec's built-in nested formatter, which is invoked with --format nested on the command line.

examples An array of ExampleProxy objects for all of the examples in this group.

location The file and line number at which the proxied example group was declared. This is extracted from caller, and is therefore formatted as an individual line in a backtrace.

ExampleProxy

An ExampleProxy is a lightweight proxy for an individual example. This is the object sent to the example_started(), and then either example_passed(), example_failed(), or example_pending().

Note that the same ExampleProxy object is passed to both example_started() method *and* the appropriate method after the example is finished. This lets the formatter map the beginning and end of an example using object identity (equal?()). RSpec's profile formatter, invoked with --format

profile, uses this feature to calculate the time it takes for each example to run.

Each ExampleProxy carries the following information:

description The description string passed to the it() method or any of its aliases. This is nil when the proxy is passed to example_started(), but has a non-nil value when passed to the other example_xxx() methods. The reason is that RSpec users can write examples like this:

```
describe MyCustomFormatter do
  it { should solve_all_my_reporting_needs }
end
```

In this case there is no string passed to the it() method, so the example doesn't know its own description until the solve_all_my_reporting_needs() matcher generates it, which won't happen until the example is run.

location The file and line number at which the proxied example was declared. This is extracted from caller, and is therefore formatted as an individual line in a backtrace.

Failure

The example_failed() and dump_failure() methods are each sent a Failure object, which contains the following information:

header Header messsage for reporting this failure, including the name of the example and an indicator of the type of failure. FAILED indicates a failed expectation. FIXED indicates a pending example that passes, and no longer needs to be pending. RuntimeError indicates that a RuntimeError occured.

exception This is the actual Exception object that was raised.

Invoking A Custom Formatter

Once we've put in all of the energy to write a formatter using the APIs we've discussed, we'll probably want to start using it! Invoking a custom formatter couldn't be much simpler. We just need to require the file in which it is defined, and then add its class to the command line.

Let's say we've got a PDF formatter that generates a PDF document that we can easily ship around to colleagues. Here is the command

we'd use, assuming that it is named PdfFormatter and defined in formatters/pdf_formatter.rb:

```
spec spec --require formatters/pdf_formatter --format PdfFormatter:report.pdf
```

The structure of the --format argument is FORMAT[:WHERE]. FORMAT can be any of the built-in formatters, or the name of the class of a custom formatter. WHERE is STDOUT by default, or a filename. Either way, that's what gets submitted to the initialize method of the formatter.

15.6 What We've Learned

In this chapter we explored the utilities and extension points that RSpec provides to support extending RSpec to meet your specific needs. These include:

- **Global Configuration** lets us assign before and after blocks to every example group. We can also use it to add methods to example groups by *extending* them with custom modules, and add methods to individual examples by *including* custom modules.
- **Custom Example Group Classes** provide a logical home for custom behaviour. They are ideal for libraries that want to ship with spec'ing facilities for their end users.
- We can use **Custom Matchers** to build up a domain-specific DSL for expressing code examples.
- **Macros** also support a domain-specific DSL, but with a different flavor than the custom matchers. Because they generate code themselves, we can also use them to express groups of expectations in a single command.
- **Custom Formatters** let us control the output that RSpec provides so we can produce different spec reports for different purposes and audiences.

In practice, we find that the global configuration, custom matchers defined with the Matcher DSL, and macros tend to be the most common ways that we extend RSpec. There are already numerous matcher and macro libraries for RSpec that are targeted at Rails development. Custom formatters tend to be the domain of IDE developers that support RSpec, like NetBeans and RubyMine.

Chapter 16

Cucumber

Coming soon ...

Part IV

Behaviour Driven Rails

Chapter 17

BDD in Rails

Ruby on Rails lit the web development world on fire by putting developer happiness and productivity front and center. Concepts like convention over configuration, REST, declarative software, and the Don't Repeat Yourself principle are first class citizens in Rails, and have had a profound and widespread impact on the web developer community as a whole.

In the context of this book, the single most important concept expressed directly in Rails is that automated testing is a crucial component in the development of web applications. Rails was the first web development framework to ship with an integrated full-stack testing framework. This lowered the barrier to entry for those new to testing and, in doing so, raised the bar for the rest of us.

RSpec's extension library for Rails, rspec-rails, extends the Rails testing framework by offering separate classes for spec'ing Rails models, views, controllers and even helpers, in complete isolation from one another. All that isolation can be dangerous if not accompanied with some level of automated integration testing to make sure all the pieces work together. For that we use Cucumber and supporting tools like Webrat and Selenium.

All of these tools are great additions to any web developer's arsenal of testing tools, but, in the end, tools are tools. While RSpec and Cucumber are optimized for BDD, using them doesn't automatically mean you're *doing* BDD.

In the chapters that follow, we'll show you how to use rspec-rails in conjunction with tools like Cucumber, Webrat, and Selenium, to drive

application development from the Outside-In with a powerful BDD toolset and, much more importantly, a BDD *mindset*.

So what does that mean? What *is* the BDD mindset? And how do we apply it to developing Rails apps? To put this into some perspective, let's take a look at traditional Rails development.

17.1 Traditional Rails Development

Rails developers typically use an inside-out approach to developing applications. You design the schema and implement models first, then the controllers, and lastly the views and helpers.

This progression has you build things you *think* other parts of the system are going to need before those other parts exist. This approach moves quickly at first, but often leads to building things that won't be used in the way you imagined, or perhaps won't get used at all. When you realize that the models don't really do what you need, you'll need to either revisit what you've already built or make do. At this juncture, you might hear your conscience telling you to "do the simplest thing."

The Illusion of "simple" with Inside-Out
by Zach Dennis

I once worked on an application that had to display events to a user. Working inside-out, we had built several of the models that we felt adequately represented the application, including some custom search functionality that would find events based on a set of criteria from the user. When we got to implementing the views we realized that we needed to filter the list of events based on some additional criteria. Did the event belong to the user or to a group the user belonged to? Was the user an admin?

We had already set up the associations for events belonging to users and groups. Rather than go back and change the custom search functionality, it *seemed* simpler to take advantage of what we had already built. So instead of refactoring the model to support the additional filtering, we added those checks in the views. Afterwards we refactored the view, extracting the checks to a method in a helper module, erroneously easing our guilt over putting logic in a view. We felt good at the time about the decision. We were being pragmatic and doing the "simplest thing."

Over time, we were presented with similar situations and made similar decisions. Before long the application had all of this "simple-ness" tucked away in places where it was very difficult to re-use and work with. We did end up needing to re-use some of this logic in other parts of the

> application, but it wasn't simple any longer. Now it felt like we had to choose between the lesser of three evils: force some awkward way of accessing the logic for re-use, duplicate the logic, or go back and perform time-consuming surgery on the application to make it easier to work with.

Building from the models out to the views means writing code based on what you *think* you're going to need. Ironically, it's when you focus on the UI that you discover what is really needed from the models, and at that point there's already a bunch of supporting code developed, refactored, and well tested—ready to be used.

It turns out that you can alleviate these issues and build what you need rather than building what you think you need by working Outside-In.

17.2 Outside-In Rails Development

Outside-in Rails development is like standing the traditional inside-out approach on its head. Instead of working from the models out to views, you work from the views in toward the models.

This approach lets customer-defined acceptance criteria drive development in a much more direct way. It puts you in a better position to discover the objects and interfaces earlier on in the process and make design decisions based on real need.

The BDD cycle with Rails is the same Outside-In process you'd use with any other framework (or no framework), web, desktop, command line, or even an API. The cycle depicted in Figure 17.1, on the following page is the same cycle depicted in Figure 1.1, on page 19, but we've added some detail to help you map it to Rails.

- Start with a scenario. Make sure you have a clear understanding of the scenario and how it is expected to work, including how the UI should support a user interacting with the app (see the sidebar on page 201).
- Execute the scenario with Cucumber. This will show you which steps are pending. When you first start out most, if not all of the steps will be pending.
- Write a step definition for the first step. Execute the scenario with Cucumber and watch it fail.
- Drive out the view implementation using the red/green/refactor cycle with RSpec. You'll discover any assigned instance variables,

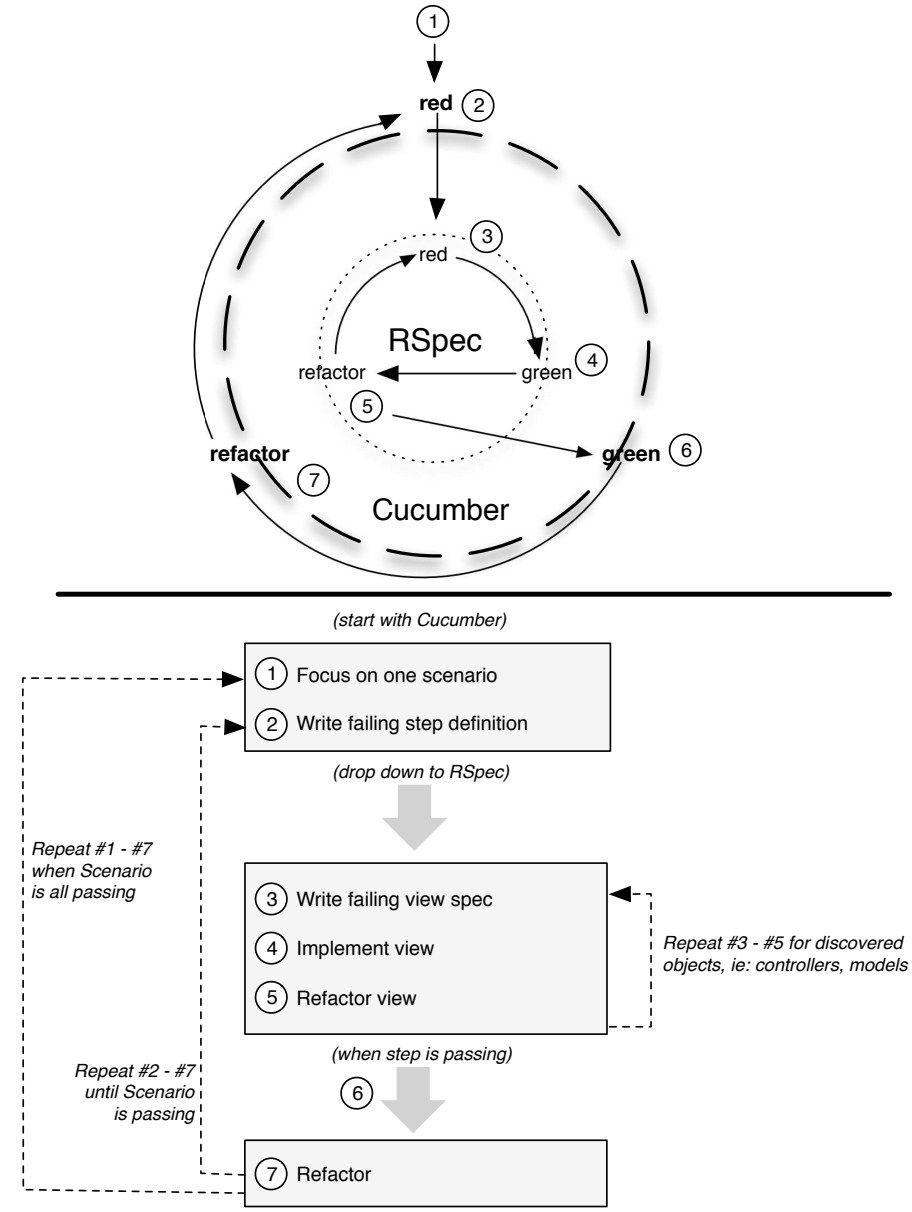


Figure 17.1: The BDD Cycle in Rails

Understanding application behaviour

How a user expects to interact with a webapp is going to influence the resulting implementation. This affects both client-side and server-side code. Failing to take this into account can lead to a design that supports the desired functionality, but with an implementation that is poorly aligned with the behaviour.

Outside-in suggests that you focus first on the outermost point of a scenario, the UI, and then work your way in. This involves communicating with the customer using visual tools like whiteboards, wireframes, screen mockups, or other forms of visual aide. And if there are designers on the team, these communications should definitely involve them.

BDD is about *writing software that matters*. And little matters more to your customer than how people will interact with the application. Understanding user interaction, and addressing that early, will go a long way towards understanding the underlying behaviour of the app.

controllers, controller actions, and models that it will need in order to do its job.

- Drive out the controller with RSpec, ensuring the proper instance variables are assigned. With the controller in place you'll know about any additional objects, models and methods that it needs to do its job.
- Drive out those objects and models with RSpec, ensuring they provide the appropriate methods that you found are needed by the view and the controller. This usually leads to generating the required migrations for fields in the database.
- Once you have implemented all of the objects and methods that you have discovered are needed, execute the scenario with Cucumber again to make sure the step is satisfied.

Once the step is passing, move on to the next unimplemented step and continue working outside-in. When a scenario is done, move on to the next scenario or find the nearest customer and have them validate that it's working as it should—then move on to the next scenario.

This is outside-in Rails development—implementing a scenario from its

outermost-point down, building what you discover is needed to make it work.

Now that you have a high level view of the outside-in process in Rails, let's get started by setting up a Rails project with the necessary tools. This will let us explore ground zero in the following chapters.

17.3 Setting up a Rails project

Setting up a Rails project for outside-in development is simply a matter of installing RSpec, Cucumber, rspec-rails, and Webrat. There are however four different installation methods that you can choose from.

The easiest installation method is the system-wide gem installation. If you're interested in packaging a specific version of Cucumber, RSpec, rspec-rails, or Webrat into your application then you'll be interested in the vendor/gems, vendor/plugins, and config.gem installation methods.

System-wide gems

Installing the necessary libraries and tools is as simple as installing an everyday rubygem. When you're working with the latest stable releases this is a great route to go:

```
> sudo gem install cucumber rspec-rails webrat
```

RSpec is a dependency of rspec-rails, so when you install rspec-rails you get RSpec for free.

Bundling in vendor/gems

Rails supports loading gems found in vendor/gems/ before loading system-wide gems. After you've installed the system-wide gems you can unpack them into vendor/gems/:

```
> cd RAILS_ROOT/vendor
> mkdir gems
> cd gems
> gem unpack cucumber
> gem unpack rspec
> gem unpack rspec-rails
> gem unpack webrat
```

This method allows you to store the gems your application relies on in version control. It also makes application development and deployment very simple since you don't have to worry about installing every single

gem required—they're bundled with the app. The only time this doesn't work is when your app requires a platform-specific gem. In that case you will have to install the gem so it compiles correctly for the target architecture.

Bundling in vendor/plugins

Rails supports loading plugins found in vendor/plugins/ before loading gems found in vendor/gems. This is a great way to stay on the edge of development:

```
> cd RAILS_ROOT
> script/plugin install --force git://github.com/aslakhellesoy/cucumber.git
> script/plugin install --force git://github.com/dchelimsky/rspec.git
> script/plugin install --force git://github.com/dchelimsky/rspec-rails.git
> script/plugin install --force git://github.com/brynary/webrat.git
```

This method also allows you to store libraries your application relies on in version control and it shares the same benefits of development and deployment as the vendor/gems installation method.

Using Rails config.gem

To take advantage of Rails' built-in gem management, we recommend that you configure the gems in config/environments/test.rb: [1]

```
config.gem 'rspec-rails', :lib => false
config.gem 'rspec', :lib => false
config.gem 'cucumber'
config.gem 'webrat'
```

Use lib => false for rspec and rspec-rails because even though we may want rails' gem configuration to help us with installing and bundling gems, we want rspec-rails' rake tasks to control when they are loaded.

After saving this file you should be able to perform any of the following commands:

```
# see what gems are required in the test environment
rake gems RAILS_ENV=test

# install required gems to your system
[sudo] rake gems:install RAILS_ENV=test

# unpack required gems from your system to the app's vendor/gems
```

1. rspec-rails and config.gem have had some conflicts in the past. If you run into any trouble with the instructions here, be sure to check http://wiki.github.com/rspec/rspec for the latest installation instructions.

```
[sudo] rake gems:unpack RAILS_ENV=test

# unpack required dependencies for your app's gems
[sudo] rake gems:unpack:dependencies RAILS_ENV=test
```

A few things to note:

- Use sudo on *nix machines if you normally use sudo to install and unpack gems.
- Put RAILS_ENV=test at the end of the line, especially if you're using sudo.
- The rake tasks installed by old versions of rspec-rails cause some trouble in this process, so delete lib/tasks/rspec.rake before executing these commands if you're upgrading.

Bootstrapping your app w/Cucumber and RSpec

Cucumber and RSpec both ship with a Rails generator in order to setup a Rails application to use them. You'll need to run these two commands to finish bootstrapping your Rails app for development with Cucumber and RSpec:

```
> script/generate cucumber
> script/generate rspec
```

We'll explore the cucumber generator in the next chapter, Chapter 18, *Cucumber with Rails*, on page 206. For now, let's take a look at what the rspec generator is doing.

```
$ ./script/generate rspec
    create  lib/tasks/rspec.rake
    create  script/autospec
    create  script/spec
    create  script/spec_server
    create  spec
    create  spec/rcov.opts
    create  spec/spec.opts
    create  spec/spec_helper.rb
```

Here's a description of each file and directory that was generated:

- lib/tasks/rspec.rake: Adds a collection of rake spec tasks to your application. These tasks offer various ways of running your specs. Run rake -T spec to find out more about these tasks.

- script/autospec: A command that provides integration with autotest in your Rails project. [2]
- script/spec: A command to run spec files directly with the version of rspec the Rails app was configured for, e.g. system-wide rspec gem, a local rspec gem in vendor/gems, or rspec installed in vendor/plugins.
- script/spec_server: A command that runs a server for running specs more quickly in your Rails app.
- spec: The directory where you place specs for your Rails app.
- spec/rcov.opts: Add options to this file that you want rcov to run with when running any of the rake spec tasks with rcov, e.g. rake spec:rcov.
- spec/spec.opts: Add options to this file that you want rspec to utilize when running any of the rake spec tasks.
- spec/spec_helper.rb: This file is used load and configure rspec. It is also where you would require and configure any additional helpers or tools that your project utilizes when running specs.

Now you and your project are ready for outside-in development.

17.4 What We Just Learned

So far we've explored what it means to do BDD in Rails using outside-in development and we've set up a Rails project with the recommended tools. In the next chapter, we'll take a look at how Cucumber and Rails can be used together to drive development from the outside. Turn the page and let's begin.

2. See Section 14.3, *Autotest*, on page 171 for more information on rspec and autotest integration

Chapter 18

Cucumber with Rails

Cucumber was created to support collaboration between project stakeholders and application developers, with the goal of developing a common understanding of requirements and providing a backdrop for discussion. The result is a set of feature files with automated scenarios that application code must pass. Once they're passing, they serve as regression tests as the development continues.

These same benefits apply when using Cucumber with Rails. In this chapter we'll look at how Cucumber integrates with a Rails project. We'll explore the variety of approaches to setting up context, triggering events and specifying outcomes as we describe the features of our web application.

18.1 Working with Cucumber in Rails

Cucumber scenarios serve as a high level description of a Rails application's behaviour within your codebase. As such, they'll replace Rails' integration tests, and serve as the outer loop in the Outside-In development cycle.

Like a good set of integration tests, they'll give you tremendous confidence to refactor your code and evolve the application in response to changing requirements. In addition, they'll document the system's behaviour and your progress in implementing it by connecting those requirements directly to Ruby code.

As we saw in Chapter 17, *BDD in Rails*, on page 197, installing Cucumber into a Rails app is simple. The last step of the installation process is to run the generator included in Cucumber.

```
$ ./script/generate cucumber
      create  features/step_definitions
      create  features/step_definitions/webrat_steps.rb
      create  features/support
      create  features/support/env.rb
      exists  lib/tasks
      create  lib/tasks/cucumber.rake
      create  script/cucumber
```

Let's take a look at these four directories and two files that were created:

- features/step_definitions: Clearly, this is where you'll put step definitions.
- features/step_definitions/webrat_steps.rb: You'll put your commonly used Webrat step definitions here. We'll learn about this file in Chapter 19, *Simulating the Browser with Webrat*, on page 220.
- features/support: This directory holds any Ruby code that needs to be loaded to run your scenarios that are *not* step definitions, like helper methods shared between step definitions.
- features/support/env.rb: Bootstraps and configures the Cucumber runner environment.
- lib/tasks/cucumber.rake: Adds the rake features task which prepares the test database and runs your application's feature suite.
- script/cucumber: The command line feature runner.

That's all you need to run Cucumber feature files for a Rails application. As we progress, we'll be adding files to three places (see Figure 18.1, on the next page):

- RAILS_ROOT/features: This is where you'll place each Cucumber *.feature file containing your scenarios.
- RAILS_ROOT/features/step_definitions: You'll add step definitions to implement the plain text scenarios here. Use one file for each domain concept, for example movie_steps.rb and checkout_steps.rb.
- RAILS_ROOT/features/support: This directory holds any supporting code or modules you extract from your step definitions as you refactor.

Now that we're all set up, let's take a look at the different approaches to creating executable scenarios for a Rails application.

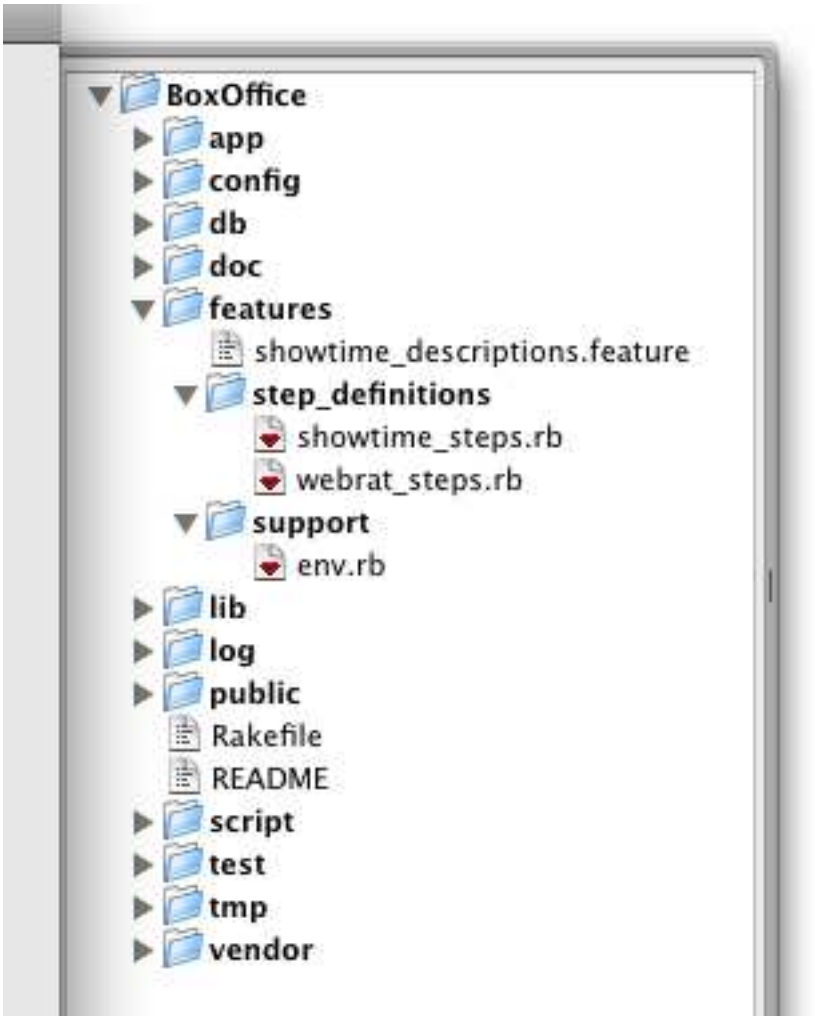


Figure 18.1: Rails Project Tree with Cucumber Features

18.2 Step Definition Styles

Step definitions connect the natural language steps from the scenarios in a feature file with the Ruby code blocks that interact directly with the system. Since Cucumber allows us to easily describe behaviour in business terms, the steps shouldn't express technical details. The same Cucumber step of "Given I'm logged in as an administrator" could apply to a CLI, client-side GUI, or Web-based application. It's in the step definitions that the rubber meets the road and code is created to interact with the application.

The first step of the Outside-In cycle is to produce a failing scenario, and to do that we'll need a step definition, but how should it be implemented? Rails applications contain many layers, from the model and the database all the way up to the web browser, and this leaves us with options and choices in how step definitions interact with an application.

We want scenarios to exercise a vertical slice through all of our code, but we also want them to run fast. The fastest scenarios are going to bypass HTTP, routing, and controllers and just talk directly to the models. The slowest ones are going to exercise everything through the web browser, giving us the fullest coverage, the most confidence, and the least desire to run them on a regular basis!

So what's a pragmatic story teller to do? I'm going to put on my consultant hat for a second and say "it depends." When building step definitions for a Rails application, we typically deal with three step definition styles for interacting with a Web-based system in order to specify its behaviour:

- *Direct Model Access*: Access the ActiveRecord models directly (like model specs) while skipping the routing, controllers, and views. This is the fastest but least integrated style. It deliberately avoids the other layers of the Rails stack.
- *Simulated Browser*: Access the entire MVC stack using Webrat, a DSL for interacting with web applications. This style provides a reliable level of integration while remaining fast enough for general use, but doesn't exercise JavaScript at all.
- *Automated Browser*: Access the entire Rails MVC stack and a real web browser by driving interactions with the Webrat API and its support for piggy-backing on Selenium. This style is fully integrated but can be slow to run and cumbersome to maintain.

Fast is better than slow, of course, and integrated is better than isolated in order to provide confidence the app works in the hands of your users once you ship it. When writing Cucumber scenarios, integration and speed are opposing forces. This conundrum is illustrated in Figure 18.2, on the following page. The balance of these forces that feels best will vary a bit from developer to developer, so I'll share my own preferences to serve as a starting point. It's not the "one true way," but it's based on my experience playing with a variety of approaches.

I use Direct Model Access in Givens to prepare the state of the system, except for logging-in or other actions that set up browser session state. Whens and Thens execute against the full Rails stack using Webrat as a Simulated Browser. This provides confidence that all of the component parts are working well together but still produces a suite that can be executed relatively quickly and without depending on a real web browser.

If there is any JavaScript or AJAX, I'll add scenarios that use the Automated Browser approach in their Whens and Thens for the *happy path* and critical less common paths. The added value we get from doing this is exercising client side code, so when no client code is necessary, there is no reason to use the browser.

Lastly, for features that produce many edge cases, it can be useful to drive a few through the Rails stack and the rest using just Direct Model Access for everything. This may seem more like a unit test, but keep in mind that scenarios are about communication, not just coverage. We want to make sure that we're writing the right code. If the customer asks for specific error messages depending on a variety of error conditions, then it's OK to go right to the model if that's the source of the message, as long as we have confidence that the relevant slice of the full stack is getting sufficient coverage from our other scenarios.

In this chapter, we'll start with the simplest style, Direct Model Access, and walk through implementing a feature. Then we'll explore using Webrat for both the Simulated Browser and Automated Browser styles in Chapter 19, *Simulating the Browser with Webrat*, on page 220.

18.3 Direct Model Access

The Direct Model Access style of step definitions is just like what you might find in Rails unit tests or RSpec model specs. They execute quickly and are immune to changes in the controller and view layers. In

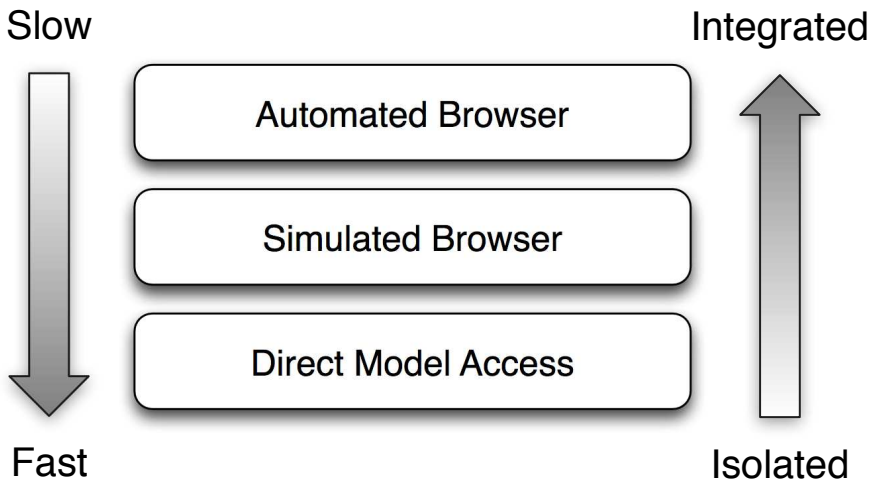


Figure 18.2: Comparing step definition styles

fact, in this DMA-only example, we won't even need to build a controller or views to get our scenarios passing.

That speed and isolation comes at a price. DMA step definitions don't provide any assurance that the application actually works (unless your users happen to fire up the application using script/console). They are also unlikely to catch bugs that a good set of model specs wouldn't have already caught.

It's not all bad, however. They still facilitate a good conversation between the customer and the developers, and will catch regressions if the logic inside the models is broken in the future. In this way, DMA step definitions are useful for exercising fine grained behaviours of a system, when driving all of them through the full stack would be too cumbersome.

To see this in action, let's look at some scenarios for a movie box office system. The customer wants the structured movie schedule data to be distilled into the best human-readable one line showtime description for display on a website. Create a feature file named showtime_descriptions.feature and add the following text to it:

Cucumber::Rails::World

Cucumber::Rails::World is the primary building block of Cucumber's support for Rails. As the bridge between the two frameworks, it provides the Rails integration testing methods within each of your scenarios.

When Cucumber's Rails support is loaded by requiring cucumber/rails/world in features/support/env.rb, instances of Cucumber::Rails::World are configured to be the World for each scenario:

```
World do
  Cucumber::Rails::World.new
end
```

Cucumber::Rails::World inherits Rails' ActionController::IntegrationTest, and makes surprisingly few modifications to the superclass behaviour. Here's how it's defined inside Cucumber:

```
class Cucumber::Rails::World < ActionController::IntegrationTest
  «code»
end
```

Each scenario will run in a newly instantiated Cucumber::Rails::World. This gives us access to all of the functionality of Rails' Integration tests and RSpec's Rails-specific matchers, including simulating requests to the application and specifying behaviour with RSpec expectations.

In the default configuration, it will also cause each scenario to run in an isolated DB transaction, just like RSpec code examples. You can disable this by removing the following line from the RAILS_ROOT/features/support/env.rb generated by Cucumber:

```
Cucumber::Rails.use_transactional_fixtures
```

If you disable per-scenario transactions, you'll have to worry about records left over from one scenario affecting the results of the next. This often leads to inadvertent and subtle ordering dependencies in your scenario build. For these reasons, we *strongly* recommend using the transactional fixtures setting.

Download cucumber_rails/01/features/showtime_descriptions.feature

```
Feature: Showtime Descriptions

  So that I can find movies that fit my schedule
  As a movie goer
  I want to see accurate and concise showtimes

  Scenario: Show minutes for times not ending with 00

    Given a movie
    When I set the showtime to 2007-10-10 at 2:15pm
    Then the showtime description should be "October 10, 2007 (2:15pm)"

  Scenario: Hide minutes for times ending with 00

    Given a movie
    When I set the showtime to 2007-10-10 at 2:00pm
    Then the showtime description should be "October 10, 2007 (2pm)"
```

If we ran that file using script/cucumber now, all the steps would be pending:

Download cucumber_rails/01/out/01.all_pending

```
./script/cucumber features/showtime_descriptions.feature

Feature: Showtime Descriptions

  So that I can find movies that fit my schedule
  As a movie goer
  I want to see accurate and concise showtimes
  Scenario: Show minutes for times not ending with 00
    Given a movie
    When I set the showtime to 2007-10-10 at 2:15pm
    Then the showtime description should be "October 10, 2007 (2:15pm)"

  Scenario: Hide minutes for times ending with 00
    Given a movie
    When I set the showtime to 2007-10-10 at 2:00pm
    Then the showtime description should be "October 10, 2007 (2pm)"


6 steps pending

You can use these snippets to implement pending steps:

Given /^a movie$/ do
end

When /^I set the showtime to 2007\-10\-10 at 2:15pm$/ do
end
```

```
Then /^the showtime description should be "October 10, 2007 \(2:15pm\)"$/ do
end

When /^I set the showtime to 2007\-10\-10 at 2:00pm$/ do
end

Then /^the showtime description should be "October 10, 2007 \(2pm\)"$/ do
end
```

Getting the First Scenario to Pass

Let's focus on the step definitions needed for the first scenario. Using the Direct Model Access style, it's easy to fill in the step definitions with the sort of code you might see in Rails' model specs.[1] We could put them in features/step_definitions/showtime_steps.rb.

Download cucumber_rails/01/features/step_definitions/showtime_steps.rb

```
Given /^a movie$/ do
  @movie = Movie.create!
end

When /^I set the showtime to 2007\-10\-10 at 2:15pm$/ do
  @movie.update_attribute(:showtime_date, Date.parse("2007-10-10"))
  @movie.update_attribute(:showtime_time, "2:15pm")
end

Then /^the showtime description should be "October 10, 2007 \(2:15pm\)"$/ do
  @movie.showtime.should == "October 10, 2007 (2:15pm)"
end
```

Since Cucumber step definitions execute in the context of a Rails environment, you can use any techniques that work in Rails unit tests or RSpec model specs. That includes creating models in the database and using RSpec's Expectations API.[2]

Just as instance variables can be created in before(:each) blocks and referenced in individual code examples, the @movie instance variable created in the Given() step is available to all subsequent steps.

Let's check how we're doing. We can use Cucumber's --scenario command line option to run the one scenario we're focused on:[3]

1. Implementing Rails model specs with RSpec is covered in depth in the (as yet) unwritten *chp.models*.
2. RSpec's mocking and stubbing API is not available, however.
3. The --scenario option was introduced in cucumber-0.1.9.

Download cucumber_rails/01/out/02.failing

```
./script/cucumber features/showtime_descriptions.feature:7

Feature: Showtime Descriptions

  So that I can find movies that fit my schedule
  As a movie goer
  I want to see accurate and concise showtimes
  Scenario: Show minutes for times not ending with 00
    Given a movie
      uninitialized constant Movie (NameError)
      /Library/Ruby/Gems/1.8/gems/activesupport-2.2.2/lib/active_support/ \
                              dependencies.rb:445:in `load_missing_constant'
      /Library/Ruby/Gems/1.8/gems/activesupport-2.2.2/lib/active_support/ \
                              dependencies.rb:77:in `const_missing'
      /Library/Ruby/Gems/1.8/gems/activesupport-2.2.2/lib/active_support/ \
                              dependencies.rb:89:in `const_missing'
      ./features/step_definitions/showtime_steps.rb:2:in `Given /^a movie$/'
      features/showtime_descriptions.feature:9:in `Given a movie'
    When I set the showtime to 2007-10-10 at 2:15pm
    Then the showtime description should be "October 10, 2007 (2:15pm)"


1 steps failed
2 steps skipped
```

Now we've got a failing scenario that will drive adding functionality to our application. The minimum amount of work to get the first scenario passing would be adding a Movie model and a Movie#showtime method to properly format the date and time. We'll do just that.

Download cucumber_rails/02/app/models/movie.rb

```
class Movie < ActiveRecord::Base

  def showtime
    "#{formatted_date} (#{formatted_time})"
  end

  def formatted_date
    showtime_date.strftime("%B %d, %Y")
  end

  def formatted_time
    showtime_time.strftime("%l:%M%p").strip.downcase
  end

end
```

Let's check the results of running our feature file again:

Joe Asks...

Where does RSpec fit into this picture?

In this example, we go straight from a Cucumber scenario to the Rails model code without any more granular code examples written in RSpec. This is really just to keep things simple and focused on Cucumber for this chapter.

We have yet to introduce you to the other styles of step definitions, or the Rails-specific RSpec contexts provided by the spec-rails library. As you learn about them in the coming chapters, you'll begin to get a feel for how all these puzzle pieces fit together, and how to balance the different tools and approaches.

```
./script/cucumber features/showtime_descriptions.feature:7

Feature: Showtime Descriptions

  So that I can find movies that fit my schedule
  As a movie goer
  I want to see accurate and concise showtimes
  Scenario: Show minutes for times not ending with 00
    Given a movie
    When I set the showtime to 2007-10-10 at 2:15pm
    Then the showtime description should be "October 10, 2007 (2:15pm)"


3 steps passed
```

That's looking much better, isn't it? This would probably be a good time to commit to your version control system. Working scenario by scenario like this, we get the benefit of ensuring we don't break previously passing scenarios as we continue to refactor and add behaviour.

Completing the Feature

Looking at the second scenario, the step definitions we need are similar to the two we've already implemented. Astute readers might be wondering if we can use Cucumbers's parameterized step definitions feature that you read about in the (as yet) unwritten *chp.cucumber*. They'd be right.

Using the power of regular expressions we can make a When() step definition that works for arbitrary times, and a Then() step definition that works for arbitrary schedules:

Download cucumber_rails/02/features/step_definitions/02/showtime_steps.rb

```
Given /^a movie$/ do
  @movie = Movie.create!
end

When /^I set the showtime to 2007\-10\-10 at (.+)$/ do |time|
  @movie.update_attribute(:showtime_date, Date.parse("2007-10-10"))
  @movie.update_attribute(:showtime_time, time)
end

Then /^the showtime description should be "(.+)"$/ do |description|
  @movie.showtime.should == description
end
```

We can replace our old When() and Then() step definitions with these new parameterized versions, and the result of running the feature file won't change. They'll also make it easier to add many scenarios about movies throughout the life of the application. As you build up a suite of reusable step definitions for your applications, you'll often find the number of new step definitions you have to write for new functionality to be surprisingly small.

With our step definitions implemented, we can turn our attention to getting the last scenario to pass. Before we implement the application code, let's check that we're seeing the failure we expect:

Download cucumber_rails/02/out/02.one_failing

```
./script/cucumber features/showtime_descriptions.feature

Feature: Showtime Descriptions

  So that I can find movies that fit my schedule
  As a movie goer
  I want to see accurate and concise showtimes
  Scenario: Show minutes for times not ending with 00
    Given a movie
    When I set the showtime to 2007-10-10 at 2:15pm
    Then the showtime description should be "October 10, 2007 (2:15pm)"

  Scenario: Hide minutes for times ending with 00
    Given a movie
    When I set the showtime to 2007-10-10 at 2:00pm
    Then the showtime description should be "October 10, 2007 (2pm)"
      expected: "October 10, 2007 (2pm)",
           got: "October 10, 2007 (2:00pm)" (using ==) \
```

```
        (Spec::Expectations::ExpectationNotMetError)
./features/step_definitions/02/showtime_steps.rb:11:in \
        `Then /^the showtime description should be "(.+)"$/'
features/showtime_descriptions.feature:17:in \
       `Then the showtime description should be "October 10, 2007 (2pm)"'


5 steps passed
1 steps failed
```

Now we can go back to our Movie model and enhance the logic of the formatted_time() method.

Download cucumber_rails/03/app/models/movie.rb

```
def formatted_time
  format_string = showtime_time.min.zero? ? "%l%p" : "%l:%M%p"
  showtime_time.strftime(format_string).strip.downcase
end
```

That should be enough to get us to green:

Download cucumber_rails/03/out/01.done

```
./script/cucumber features/showtime_descriptions.feature

Feature: Showtime Descriptions

  So that I can find movies that fit my schedule
  As a movie goer
  I want to see accurate and concise showtimes
  Scenario: Show minutes for times not ending with 00
    Given a movie
    When I set the showtime to 2007-10-10 at 2:15pm
    Then the showtime description should be "October 10, 2007 (2:15pm)"

  Scenario: Hide minutes for times ending with 00
    Given a movie
    When I set the showtime to 2007-10-10 at 2:00pm
    Then the showtime description should be "October 10, 2007 (2pm)"


6 steps passed
```

Success! We've completed our work on the "Showtime Descriptions" feature. Our passing scenarios tell us that we've written the right code, and that we're done. Before we leap into the next chapter, let's take a second to consider what we learned.

Like most important development decisions, when choosing a step definition style there are opposing forces on each side that need to be considered and balanced. Direct Model Access step definitions offer the

speed and flexibility of model specs at the cost of reduced confidence that the application is working for its users.

For most situations, it makes more sense to create a more integrated set of step definitions that ensure the Models, Views and Controllers are working together correctly, even though they will execute a bit slower. Next we'll take a look at how we can use Webrat to implement either the Simulated Browser or Automated Browser styles to do just that.

Chapter 19

Simulating the Browser with Webrat

Even though we call Rails an MVC framework, there are really more than three layers in a Rails app. In addition to model, view, and controller, we've also got a routing layer, a persistence layer (the class methods in Rails models) and a database, and we want to ensure that all of these layers work well together.

In the last chapter, we introduced Direct Model Access step definitions and used them to implement Givens, Whens and Thens. This approach can be useful to specify fine-grained Model behaviours, but running those scenarios doesn't give us any confidence that the different layers of our application are working well together.

In practice, in the rare cases we use DMA-only scenarios, it's to augment a strong backbone of coverage established by Simulated Browser scenarios exercising the full Rails stack. We covered it first because it's the simplest style, but the primary role of DMA step definitions is to help keep our Simulated and Automated Browser scenarios focused by quickly setting up repeated database state in Givens, as we'll see later in this chapter.

We consider the Simulated Browser style to be the default approach for implementing Whens and Thens for a Rails app because it strikes a good balance between speed and integration. We can count on the software actually working in the hands of our end users when we ship, and we can execute the scenarios relatively quickly as the requirements and code evolve.

If you're building an application without much JavaScript, the Simulated Browser (combined with DMA for Givens) is likely all you'll need. It's a fast, dependable alternative to in-browser testing tools like Selenium and Watir. Even when JavaScript is important to the user experience, we like to start with a set of Simulated Browser scenarios, and then add Automated Browser scenarios (which we'll cover in the (as yet) unwritten *chp.automatedBrowser*) to drive client side interactions.

If you've ever written a Rails integration test, you've probably used the Simulated Browser style of testing. In that context, methods like get_via_redirect() and post_via_redirect() build confidence because they simulate requests that exercise the full stack, but they don't make it easy to express user behaviours clearly. Throughout this chapter we'll explore how Webrat builds on this approach to help us bridge the last mile between page loads and form submissions, and the behaviour our applications provide to the real people whose lives they touch.

19.1 Writing Simulated Browser Step Definitions

Let's walk through implementing a few step definitions for a simple scenario using the Simulated Browser style. We'll be building on the web-based movie box office system from last chapter. The next requirement is that visitors should be able to browse movies by genre. Start by creating a file named browse_movies.feature in the features directory with the following content:

Download simulated_browser/01/features/browse_movies.feature

```
Feature: Browse Movies

  So that I quickly can find movies of interest
  As a movie goer
  I want to browse movies by genres

  Scenario: Add movie to genre

    Given a genre named Comedy
    When I create a movie Caddyshack in the Comedy genre
    Then Caddyshack should be in the Comedy genre
```

As usual, we'll begin by running the file with Cucumber to point us at which step definitions we need to implement:

Download simulated_browser/01/out/01.all_pending

```
./script/cucumber features/browse_movies.feature
```

```
Feature: Browse Movies

  So that I quickly can find movies of interest
  As a movie goer
  I want to browse movies by genres

  Scenario: Add movie to genre
    Given a genre named Comedy
    When I create a movie Caddyshack in the Comedy genre
    Then Caddyshack should be in the Comedy genre

1 scenario
3 undefined steps

You can implement step definitions for missing steps with these snippets:

Given /^a genre named Comedy$/ do
  pending
end

When /^I create a movie Caddyshack in the Comedy genre$/ do
  pending
end

Then /^Caddyshack should be in the Comedy genre$/ do
  pending
end
```

The "Given a genre named Comedy" step could be implemented using either DMA or the Simulated Browser style. Using a Simulated Browser would ensure that the Views and Controllers used to create Genres are working with the models properly. DMA won't go through those layers of the stack, but provides a bit more convenience, simplicity and speed.

So which style should we use? Let's imagine we already have scenarios that thoroughly exercise the interface to manage Genres using the Simulated Browser style in a manage_genres.feature file. With that coverage already in place, we can benefit from the DMA style without reducing our confidence in the application. As we add features throughout the evolution of an application, we see a pattern emerge in which we implement DMA Givens for a Model that has its own Simulated Browser scenarios elsewhere in the Cucumber suite. Here's what it might look like in this case:

Download simulated_browser/01/features/step_definitions/genre_steps.rb

```
Given /^a genre named Comedy$/ do
  @comedy = Genre.create!(:name => "Comedy")
end
```

The step passes because the Genre model and table already exist. Running the scenario again shows us that our When is the next pending step to turn our attention to:

Download simulated_browser/01/out/02.one_passing

```
./script/cucumber features/browse_movies.feature

Feature: Browse Movies

  So that I quickly can find movies of interest
  As a movie goer
  I want to browse movies by genres

  Scenario: Add movie to genre
    Given a genre named Comedy
    When I create a movie Caddyshack in the Comedy genre
    Then Caddyshack should be in the Comedy genre

1 scenario
2 undefined steps
1 passed step

You can implement step definitions for missing steps with these snippets:

When /^I create a movie Caddyshack in the Comedy genre$/ do
  pending
end

Then /^Caddyshack should be in the Comedy genre$/ do
  pending
end
```

The wireframe for the Add Movie screen shown in Figure 19.1, on the following page, shows that a user will need to provide a movie's title, release year, and genres to add it to the system. Since our When step specifies the main *action* of the scenario, we'll use the Simulated Browser to drive this interaction through the full Rails stack.

Before we look at how Webrat can help us with this, let's see what Rails provides out-of-the-box. If you were to implement the "When I create a movie Caddyshack in the Comedy genre" step with the Rails integration testing API, you might end up with something like the following:

Download simulated_browser/misc/features/step_definitions/movie_steps.rb

```
When /^I create a movie Caddyshack in the Comedy genre$/ do
  get_via_redirect movies_path
  assert_select "a[href=?]", new_movie_path, "Add Movie"

  get_via_redirect new_movie_path
```

Figure 19.1: Creating a Movie with a form

```
  assert_select "form[action=?][method=post]", movies_path do
    assert_select "input[name=?][type=text]", "movie[title]"
    assert_select "select[name=?]", "movie[release_year]"
    assert_select "input[name=?][type=checkbox][value=?]", "genres[]", @comedy.id
  end

  post_via_redirect movies_path, :genres => [@comedy.id], :movie =>
    { :title => "Caddyshack", :release_year => "1980" }

  assert_response :success
end
```

This gets the job done, but a lot of implementation details like HTTP methods, form input names, and URLs have crept up into our step definition. These sorts of details will change through the lifespan of an application and can make scenarios quite brittle. We could mitigate some of that risk by extracting helper methods for specifying forms and posts that might appear in multiple scenarios, but that still leaves a

significant issue.

With the generated HTML being specified separately from the post, it is entirely possible to assert_select "input[name=?]", "movie[name]" and then post to movies_path, :movie => { :title => "Caddyshack"}. This specifies that the form displays an input for movie[name], but then the step posts movie[title]. If the form is incorrectly displaying a movie[name] field, this step will pass, but the application will not work correctly.

Luckily, there's a better way that makes the Simulated Browser approach not only viable but enjoyable to work with.

Like the Rails integration testing API, Webrat works like a fast, invisible browser. It builds on that functionality by providing a simple, expressive DSL for manipulating a web application. We can use Webrat to describe the same interaction at a high level, using language that is similar to how you might explain using the application to a non-technical friend:

Download simulated_browser/01/features/step_definitions/movie_steps.rb

```
When /^I create a movie Caddyshack in the Comedy genre$/ do
  visit movies_path
  click_link "Add Movie"
  fill_in "Title", :with => "Caddyshack"
  select "1980", :from => "Release Year"
  check "Comedy"
  click_button "Save"
end
```

That certainly *feels* a lot better than the first version. Notice how Webrat let us focus on exactly the details an end user would experience and didn't force us to worry about how it will be built. The sole implementation detail remaining is using the movies_path() route as an entry point.

In addition to being more expressive, Webrat also delivers on the promise of catching regressions without the false positives described earlier. Don't worry about the details of how this works just yet. That will become clear throughout the rest of this chapter.

By re-running the scenario with Cucumber, it will show us where to start implementing:

Download simulated_browser/01/out/03.one_failing

```
./script/cucumber features/browse_movies.feature

Feature: Browse Movies
```

```
  So that I quickly can find movies of interest
  As a movie goer
  I want to browse movies by genres

  Scenario: Add movie to genre
    Given a genre named Comedy
    When I create a movie Caddyshack in the Comedy genre
      undefined method `movies_path' for #<ActionController::Integration::Session:0x...*TRUNC*
      /Users/bhelmkamp/p/book/Book/code/simulated_browser/01/vendor/plugins/rspec/li...*TRUNC*
      ./features/step_definitions/movie_steps.rb:2:in `/^I create a movie Caddyshack...*TRUNC*
      features/browse_movies.feature:10:in `When I create a movie Caddyshack in the ...*TRUNC*
    Then Caddyshack should be in the Comedy genre

1 scenario
1 failed step
1 undefined step
1 passed step

You can implement step definitions for missing steps with these snippets:

Then /^Caddyshack should be in the Comedy genre$/ do
  pending
end
```

At this point, the outside-in development process described in Section 17.2, *Outside-In Rails Development*, on page 199 would lead us through the steps necessary to get that feature completed. The upcoming chapters dive into the specifics of spec'ing views, controllers and models, so to keep this example focused on the Simulated Browser style, we'll leave that as an exercise for you.

To get this step passing we'll need a MoviesController and an app/views/movies/new.html.erb view that displays a form with a Title text field, a Release Year drop-down, and a list of checkboxes for each genre. Once we've built those, re-running our scenario will show us we've got one step left:

Download simulated_browser/02/out/01.one_pending

```
./script/cucumber features/browse_movies.feature

Feature: Browse Movies

  So that I quickly can find movies of interest
  As a movie goer
  I want to browse movies by genres

  Scenario: Add movie to genre
    Given a genre named Comedy
    When I create a movie Caddyshack in the Comedy genre
    Then Caddyshack should be in the Comedy genre
```

```
1 scenario
1 undefined step
2 passed steps

You can implement step definitions for missing steps with these snippets:

Then /^Caddyshack should be in the Comedy genre$/ do
  pending
end
```

To browse movies by genre, a site visitor would click over to the Comedy page, which displays one movie entitled Caddyshack. The Webrat step definition for our Then reflects this:

Download simulated_browser/03/features/step_definitions/movie_steps.rb

```
Then /^Caddyshack should be in the Comedy genre$/ do
  visit genres_path
  click_link "Comedy"
  response.should contain("1 movie")
  response.should contain("Caddyshack")
end
```

To get this to pass, we need to make sure we've got a GenresController with an index and a "show" view displaying a list of movies in the genre. Also, we'll need to go back to the MoviesController and get it to collaborate with the models to persist the movie and its genres correctly.

Again, in practice we'd drop down to isolated code examples with RSpec to drive the design and implementation of our objects. A few cycles of Red, Green, Refactor later and we should be all set:

Download simulated_browser/03/out/01.all_passing

```
./script/cucumber features/browse_movies.feature

Feature: Browse Movies

  So that I quickly can find movies of interest
  As a movie goer
  I want to browse movies by genres

  Scenario: Add movie to genre
    Given a genre named Comedy
      Could not find table 'genres' (ActiveRecord::StatementInvalid)
      ./features/step_definitions/genre_steps.rb:2:in `/^a genre named Comedy$/'
      features/browse_movies.feature:9:in `Given a genre named Comedy'
    When I create a movie Caddyshack in the Comedy genre
    Then Caddyshack should be in the Comedy genre

1 scenario
```

```
1 failed step
2 skipped steps
```

Great. The passing scenario is telling us we're done. By leveraging the DMA style for Givens and combining it with the Simulated Browser style with Webrat for Whens and Thens, we've reached a good balance of expressive specification, speed and coverage. We can read the scenario to understand what we should expect from the application, and we can be quite confident that it will work for our users when we ship it.

Throughout the rest of the chapter, we'll dive into the details of Webrat's features and how they work. Let's start by looking at how Webrat lets you navigate from page to page in your application.

19.2 Navigating to Pages

Just as a user can't click any links or submit any forms until he has typed a URL into his browser's address bar and requested a web page, Webrat can't manipulate a page until you've given it a place to start. The visit() method lets you open a page of your application.

Inside each scenario, visit() must be called before any other Webrat methods. Usually you'll call it with a routing helper, like we did in our When step definition from the previous section:

```
When /^I create a movie Caddyshack in the Comedy genre$/ do
  visit movies_path
  # ...
end
```

Behind the scenes, Webrat leverages Rails' integration testing functionality to simulate GET requests, and layers browser-like behaviour on top. Like other Webrat methods that issue requests, it looks at the response code returned to figure out what to do next:

Successful (200-299) or Bad Request (400-499) Webrat stores the response so that subsequent methods can fill out forms, click links, or inspect its content.

Redirection (300-399) If the redirect is to a URL within the domain of the application, Webrat issues a new request for the destination specified by the redirect, preserving HTTP headers and including a proper Location header. If the redirect is external, Webrat saves it as the response for later inspection but won't follow it.

Server Error (500-599) Webrat raises a Webrat::PageLoadError. If you want to specify that making a request produces an error, you can use RSpec's raise_error() to catch it.

Clicking links

Once you've opened a page of your application using visit(), you'll often want to navigate to other pages. Rather than using visit() to load each URL in succession, it's convenient to simulate clicking links to jump from page to page.

click_link() lets you identify a link by its text and follows it by making a request for the URL it points to. To navigate to the URL in the href=, wherever that may be, of a link with the text "Comedy" we wrote:

```
Then /^Caddyshack should be in the Comedy genre$/ do
  # ...
  click_link "Comedy"
  # ...
end
```

click_link() can lead to a more natural flow in your step definitions and has the advantage that your step definitions are less bound to your routing scheme. On the other hand, each page load takes a little bit of time, so to keep your scenarios running quickly you'll want to avoid navigating through many pages of the site that aren't directly related to what you're testing. Instead, you could pick an entry point for visit() closer to the area of the application you're concerned with.

In addition to clicking links based on the text between the <a> tags, Webrat can locate links by their id= and title= values. For example, if we have the following HTML:

```
<a href="/" title="Example.com Home" id="home_link">
  Back to homepage
</a>
```

Then the following step definitions would all be equivalent:

```
When /^I click to go back to the homepage$/ do
  # Clicking the link by its title
  click_link "Example.com Home"
end

When /^I click to go back to the homepage$/ do
  # Clicking the link by its id
  click_link "home_link"
end
```

```
When /^I click to go back to the homepage$/ do
  # Clicking the link by its text
  click_link "Back to homepage"
end
```

click_link() has rudimentary support for handling JavaScript links generated by Rails' link_to() for non-GET HTTP requests. Since it can't actually run any JavaScript, it relies on matching the onclick= value with regular expressions. This functionality, though limited, can be useful when dealing with RESTful Rails applications that aren't implemented with obtrusive JavaScript techniques.

Let's say the box office application requires that a moderator approves movie listings before they are visible on the site. Here's how you might express that with Webrat:

```
When /^I approve the listing$/ do
  click_link "Approve"
end
```

And here's the likely implementation:

```
<%= link_to "Approve", approve_movie_path(movie), :method => :put %>
```

When clicked, the link would generate a PUT request to the approve_movie_path. You can disable this functionality by passing the :javascript => false option to click_link():

```
When /^I approve the listing$/ do
  click_link "Approve", :javascript => false
end
```

Instead of sending a PUT request, this tells Webrat to issue a GET request as if the JavaScript were not present. This can be useful when you want to specify the app works correctly for users without JavaScript enabled.

Now that we're comfortable navigating to pages within our application, we can take a look at how to use Webrat to submit forms.

19.3 Manipulating Forms

Once we've reached a page of interest, we'll want to trigger actions before we can specify outcomes. In the context of a web-based application, that usually translates to filling out and submitting forms. Let's take a look at Webrat's methods to do that. They'll serve as the bread and butter of most of our When step definitions.

webrat_steps.rb

You might be looking at the step definitions used throughout this chapter and wondering if you'll be forced to write step definitions for every When and Then step in each of your app's scenarios. After all, maintaining separate step definitions for both "When I click the Save button" and "When I click the Delete button" (and more) would get tedious pretty quickly.

Fortunately, Cucumber has just the feature to help us out of this: parameterized step definitions. Instead of maintaining a step definition for each button, we can write one that's reusable by wrapping the Webrat API:

```
When /^I click the "(.+)" button$/ do |button_text|
  click_button button_text
end
```

In fact, Cucumber ships with a bunch of these sort of step definitions in a webrat_steps.rb file. It was added to your project's step_definitions directory when you ran the Cucumber generator.

Be sure to take a look at what's in there. It can save you quite a bit of time as you're implementing new scenarios.

fill_in()

You'll use fill_in() to type text into *text* fields, *password* fields, and <*textarea*>s. We saw an example of this in the When step definition of our box office example:

```
When /^I create a movie Caddyshack in the Comedy genre$/ do
  # ...
  fill_in "Title", :with => "Caddyshack"
  # ...
end
```

fill_in() supports referencing form fields by id=, name= and <*label*> text. Therefore, if we've got a conventional Rails form with proper label tags like this:

```
<dl>
  <dt>
    <label for="movie_title">Title</label>
  </dt>
  <dd>
    <input type="text" name="movie[title]" id="movie_title" />
```

```
  </dd>
</dl>
```

Then all of the following would be functionally equivalent:

```
When /^I fill in the movie title Caddyshack$/ do
  # using the field's label's text
  fills_in "Title", :with => "Caddyshack"
end

When /^I fill in the movie title Caddyshack$/ do
  # using the field's id
  fills_in "movie_title", :with => "Caddyshack"
end

When /^I fill in the movie title Caddyshack$/ do
  # using the field's name
  fills_in "movie[title]", :with => "Caddyshack"
end
```

In practice, referencing fields by label text is preferred. That way we can avoid coupling our step definitions to class and field names, which are more likely to change as we evolve the application. In the above example, if we renamed the Movie class to Film, we'd have to change line 8 which uses the field id= and line 13 which uses the field name=, but line 3 would continue to work just fine. Unless otherwise noted, Webrat's other form manipulation methods support targeting fields using these three strategies.

Beyond making your step definitions easier to write and maintain, providing active form field labels is a good habit to get into for accessibility and usability.

check() and uncheck()

check() lets you click a checkbox which was not checked by default or had been previously unchecked. Here's an example:

```
When /^I create a movie Caddyshack in the Comedy genre$/ do
  # ...
  check "Comedy"
  # ...
end
```

To uncheck a checkbox that was checked by default or has been previously checked, you'd write:

```
When /^I uncheck Save as draft$/ do
  uncheck "Save as draft"
end
```

choose()

You'll use choose() to manipulate *radio* form fields. Just like a browser with a GUI, Webrat ensures only one radio button of a given group is checked at a time.

Let's say we wanted to select "Premium" from a list of plan levels on a signup page. You might write:

```
When /^I choose to create a Premium plan$/ do
  choose "Premium"
end
```

select()

You'll use select() to pick options from select drop-down boxes.

```
When /^I create a movie Caddyshack in the Comedy genre$/ do
  # ...
  select "1980"
  # ...
end
```

By default, Webrat will find the first option on the page that matches the text. This is usually fine. If you'd like to be more specific, or you have multiple selects with overlapping options, you can provide the :from option. Then, Webrat will only look for the option inside selects matching the label, name or id. For example:

```
When /^I create a movie Caddyshack in the Comedy genre$/ do
  # ...
  select "1980", :from => "Release Year"
  # ...
end
```

select_date(), select_time() and select_datetime()

When rendering a form, Rails typically exposes date and time values as a series of <*select*> fields. Each individual select doesn't get its own <*label*> so filling in a date using Webrat's select() method is a bit cumbersome:

```
When /^I select October 1, 1984 as my birthday$/ do
  select "October", :from => "birthday_2i"
  select "1", :from => "birthday_3i"
  select "1984", :from => "birthday_1i"
end
```

To ease this pain, Webrat now supports filling out conventional Rails date and time fields with the select_date(), select_time() and select_datetime()

methods. They act like a thin layer on top of select() to hide away the Rails-specific implementation details. Here's how you might use them:

```
When /^I select April 26, 1982$/ do
  # Select the month, day and year for the given date
  select_date Date.parse("April 26, 1982")
end

When /^I select 3:30pm$/ do
  # Select the hour and minute for the given time
  select_time Time.parse("3:30PM")
end

When /^I select January 23, 2004 10:30am$/ do
  # Select the month, day, year, hour and minute for the given time
  select_datetime Time.parse("January 23, 2004 10:30AM")
end
```

All three of the methods also support Strings instead of Date or Time objects, in which case they'll do the required parsing internally.

Unlike select(), they don't support the :from option because no single <*label*>, input name= or id= could identify the different <*select*> fields that need to be manipulated. Instead, to help when there are multiple date or time fields on the same page, they support an :id_prefix option used to specify the attribute name:

```
When /^I set the start time to 1pm$/ do
  select_time Time.parse("1:00PM"), :id_prefix => "start_time"
end

When /^I set the end time to 3:30pm$/ do
  select_time Time.parse("3:30PM"), :id_prefix => "end_time"
end
```

attach_file()

To simulate file uploads, Webrat provides the attach_file() method. Instead of passing a file field's value as a string, it stores an ActionController::TestUploadedFile in the params hash that acts like a Tempfile object a controller would normally receive during a multipart request.

When you use it, you'll want to save the fixture file to be uploaded somewhere in your app's source code. We usually put these in spec/fixtures. Here's how you could implement a step definition for uploading a photo:

```
When /^I attach my Vacation photo$/ do
  attach_file "Photo", Rails.root.join("spec", "fixtures", "vacation.jpg")
end
```

By default, Rails' TestUploadedFile uses the text/plainMIME type. When that's not right, you can pass in a specific MIME type as a third parameter to attach_file():

```
When /^I attach my Vacation photo$/ do
  attach_file "Photo", Rails.root.join("spec", "fixtures", "vacation.jpg"), "image/j...*TRUNC*
end
```

set_hidden_field()

Occasionally, it can be useful to manipulate the value of a hidden form field when using the Simulated Browser approach. The fill_in() method, like an app's real users, will never manipulate a hidden field, so Webrat provides a set_hidden_field() specifically for this purpose:

```
When /^I select Bob from the contact list dialog$/ do
  set_hidden_field "user_id", @bob.id
end
```

Use this method with caution. It's interacting with the application in a different way than any end user actually would, so not all of the integration confidence normally associated with the Simulate Browser style applies, but it can help in a pinch.

click_button()

After you've filled out your fields using the above methods, you'll submit the form. If there's only one *submit* button on the page, you can simply use:

```
When /^I click the button$/ do
  click_button
end
```

If you'd like to be a bit more specific or there is more than one button on the page, click_button() supports specifying the button's value=, id= or name=. Let's say you have the following HTML on your page:

```
<input type="submit" id="save_button" name="save" value="Apply Changes" />
```

There are three ways you could click it using the Webrat API:

```
When /^I click the button$/ do
  # Clicking a button by id
  click_button "save_button"
end

When /^I click the button$/ do
  # Clicking a button by the name attribute
  click_button "save"
end
```

```
When /^I click the button$/ do
  # Clicking a button by its text (value attribute)
  click_button "Apply Changes"
end
```

Just like when navigating from page to page, when Webrat submits a form it will automatically follow any redirects, and ensure the final page did not return a server error. There's no need to check the response code of the request by hand. The returned page is stored, ready to be manipulated or inspected by subsequent Webrat methods.

submit_form()

Occasionally, you might need to submit a form that doesn't have a submit button. The most common example is a select field that is enhanced with JavaScript to auto-submit its containing form. Webrat provides the submit_form() method to help in these situations. To use it, you'll need to specify the <*form*>'s id= value:

```
When /^I submit the quick navigation form$/ do
  submit_form "quick_nav"
end
```

reload()

Real browsers provide a reload button to send another request for the current page to the server. Webrat provides the reload() method to simulate this action:

```
When /^I reload the page$/ do
  reload
end
```

You might find yourself using this if you want to ensure that refreshing a page after an important form submission behaves properly. Webrat will repeat the last request, resubmitting forms and their data.

19.4 Specifying Outcomes with View Matchers

Simply by navigating from page to page and manipulating forms in Whens, you've been implicitly verifying some behaviour of your application. If a link breaks, a server error occurs, or a form field disappears, your scenario will fail. That's a lot of coverage against regressions for free. In Then steps, we're usually interested in explicitly specifying the contents of pages and Webrat provides three custom RSpec matchers to help with this.

contain()

The simplest possible specification of a page is to ensure it displays the right words. Webrat's contain() takes a bit of text and ensures it's in the response's content:

```
Then /^I should see Thank you!$/ do
  response.should contain("Thank you!")
end
```

contain() also works with regular expressions instead of strings:

```
Then /^I should see Hello$/ do
  response.should contain(/Hello/i)
end
```

You'll find you can accommodate almost all of your day to day uses of the contain() matcher with a couple of reusable step definitions from Cucumber's generated webrat_steps.rb file described in the sidebar on page 231:

```
Then /^I should see "(.+)"$/ do |text|
  response.should contain(text)
end

Then /^I should not see "(.+)"$/ do |text|
  response.should_not contain(text)
end
```

contain() will match against the HTML decoded text of the document, so if you want to ensure "Peanut butter & jelly" is on the page, you'd type just that in the string, not "Peanut butter & jelly".

have_selector()

Imagine you're building an online photo gallery. Specifying the text on the page probably isn't good enough if you're looking to make sure the photo a user uploaded is being rendered in the album view. In this case, it can be quite useful to ensure the existence of a CSS selector using Webrat's have_selector():

```
Then /^I should see the photo$/ do
  response.should have_selector("img.photo")
end
```

As you'd expect, that specifies there is at least one <*img*> element on the page with a class of *photo*. Webrat supports the full set of CSS3 selectors like the :nth-child pseudo-class,

giving it lots of flexibility. The image's src= is particularly important in this case, so we might want to check that too:

```
Then /^I should see the photo$/ do
  response.should have_selector("img.photo", :src => photo_path(@photo))
end
```

Webrat will take any keys and values specified in the options hash and translate them to requirements on the element's attributes. It's just a more readable way to do what you can do with CSS's img[src=...] syntax but saves you from having to worry about escaping strings.

Occasionally the number of elements matching a given selector is important. It's easy to imagine a scenario that describes uploading a couple photos and specifying the number of photos in the album view should increase. This is supported via the special :count option:

```
Then /^I should see the photo$/ do
  response.should have_selector("img.photo", :count => 5)
end
```

When we don't care where on the page a piece of text might be, contain() gets the job done, but in some cases the specific element the text is in may be important. A common example would be ensuring that the correct navigation tab is active. To help in these cases, Webrat provides the :content option. Here's how you use it:

```
Then /^the Messages tab should be active$/ do
  response.should have_selector("#nav li.selected", :content => "Messages")
end
```

This tells Webrat to make sure that at least one element matching the selector contains the specified string. Like contain(), the provided string is matched against the HTML decoded content so there's no need to use HTML escaped entities.

Finally, for cases when you need to get fancy, have_selector() supports nesting. If you call it with a block, the block will be passed an object representing the elements matched by the selector and within the block you can use any of Webrat's matchers. Here's how you might check that the third photo in an album is being rendered with the right image tag and caption:

```
Then /^the Vacation photo should be third in the album$/ do
  response.should have_selector("#album li:nth-child(3)") do |li|
    li.should have_selector("img", :src => photo_path(@vacation_photo))
    li.should contain("Vacation Photo")
  end
end
```

By combining the power of CSS3 selectors with a few extra features, Webrat's have_selector() should provide all you need to write expecta-

tions for the vast majority of your step definitions. For the rare cases where CSS won't cut it, let's take a look at the have_xpath() matcher, which lets you go further.

have_xpath()

When CSS just isn't powerful enough, Webrat exposes have_xpath() as a matcher of last resort. It's infinitely powerful, but due to the nature of XPath it's usually not the most expressive. Here's an example from a recent project:

```
Then /^the page should not be indexable by search engines$/ do
  response.should have_xpath(".//meta[@name = 'robots' and @content = 'noindex, nofo...*TRUNC*
  response.should_not have_xpath(".//meta[@name = 'robots' and @content = 'all']")
end
```

Under the hood, have_selector() actually works by translating CSS selectors to XPath and using the have_xpath() implementation. That means all of the have_selector() features we explored work with have_xpath() too.

This implementation strategy hints at an interesting rule about CSS and XPath: *All CSS selectors can be expressed as XPath, but not all XPath selectors can be expressed as CSS*. There are a lot of occasionally useful features XPath supports that CSS does not, like traversing up the document tree (e.g. give me all <*div*>s containing a <*p*>). While an overview of XPath is outside the scope of this book, it's a good thing to get familiar with if you find yourself wanting more power than CSS selectors can provide.

19.5 Building on the Basics

Now that we've seen how to manipulate forms and specify page content with Webrat, we'll take a look at some of Webrat's more advanced, less commonly used features. You probably won't need them day to day, but it's helpful to have a rough idea of what's available so you can recognize cases when they might come in handy.

Working Within a Scope

Sometimes targeting fields by a label isn't accurate enough. Going back to our box office example application, we might want a form where a user can add multiple genres at once. Each row of the form would have its own <*label*> for the genre name, but using Webrat's fill_in() method would always manipulate the input field in the first row.

For these cases, Webrat provides the within() method. By providing a CSS selector, you can scope all of the contained form manipulations to a subset of the page. Here's how you could fill out the second genre name field:

```
When /^I fill in Horror for the second genre name$/ do
  within "#genres li:nth-child(2)" do
    fill_in "Name", :with => "Horror"
  end
end
```

If no elements matching the CSS selector are found on the page, Webrat will immediately raise a Webrat::NotFoundError. Like most other Webrat methods, if multiple elements match, it will use the first one in the HTML source.

Locating Form Fields

When a form is rendered with pre-filled values, you may want to check that the proper values are present when the page loads. To help with this, Webrat exposes methods that return objects representing fields on the page, which include accessors for their values. Here's a simple example based on field_labeled(), which looks up input fields based on their associated <*label*>s:

```
Then /^the email address should be pre-filled$/ do
  field_labeled("Email").value.should == "robert@example.com"
end
```

Checkboxes also provide a checked?() method for convenience:

```
Then /^the Terms of Service checkbox should not be checked$/ do
  field_labeled("I agree to the Terms of Service").should_not be_checked
end
```

When <*label*>-based lookups won't work, you can use field_named() which matches against the field's name= value, or field_with_id() which matches against the field's id=:

```
Then /^the email address should be pre-filled$/ do
  field_named("user[email]").value.should == "robert@example.com"
end

Then /^the email address should be pre-filled$/ do
  field_with_id("user_email").value.should == "robert@example.com"
end
```

Dropping Down to HTTP

To keep our scenarios as expressive and maintainable as possible, we generally try to avoid tying them to implementation details. For example, our users aren't concerned with the URL of the page they end up on, just that it's showing them the right information. Building our specifications of the app's behaviour on page content rather than URLs aligns our executable specifications with our users' interactions.

For the rare cases where the lower level operation of the application is important to the customers or it's the only available option for specifying a behaviour, Webrat provides a few methods that expose these details. To check the current URL of the session after the last request (and following redirects), you can use the current_url() method:

```
Then /^the page URL should contain the album SEO keywords$/ do
  current_url.should =~ /vacation-photos/
end
```

If your application does some form of browser sniffing or you're building a REST API, you might be interested in specifying the behaviour of an app in the presence of a specific HTTP header. You can set any request header for the duration of the test with Webrat's header() method:

```
Given /^I'm browsing the site using Safari$/ do
  header "User-Agent", "Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_5_6; en-us)"
end

When /^I request the users list using API version 2.0$/ do
  header "X-API-Version", "2.0"
  visit users_path
end
```

When the MIME type should affect the behaviour of the application, you can use the http_accept() method as a shortcut to set the Accept header. It can be called with a small set of symbols that map to MIME types or a MIME type string:

```
Given /^my web browser accepts iCal content$/ do
  http_accept :ics
end

Given /^my user agent accepts MP3 content$/ do
  http_accept "audio/mpeg"
end
```

Finally, if you're going to use the HTTP protocol's built in Basic authentication mechanism, Webrat includes a basic_auth() method for setting

the HTTP_AUTHORIZATION header to the encoded combination of a username and password:

```
Given /^I am logged in as "robert" with the password "secret"$/ do
  basic_auth "robert", "secret"
end
```

When Things Go Wrong

Every once in awhile, you'll hit a point where you think a step should be passing but it's failing. It might raise a Webrat::NotFoundError about a field that's not present or complain that an expected element is missing. Before diving into your test.log or the Ruby debugger, it's good to take a look at the page as Webrat is seeing it, to check if it matches your understanding of what should be rendered.

You can use the save_and_open_page() method to capture the most recent response at any point in your scenario, and open it up in a web browser as a static HTML file on your development machine. Just drop it in before any line that seems to be misbehaving:

```
When /^I uncheck Save as draft$/ do
  save_and_open_page
  uncheck "Save as draft"
end
```

Now when you re-run the scenario, you'll be shown the page response as Webrat captured it.

19.6 Wrapping Up

Before we move on to looking at how the Automated Browser style of step definitions can be used to exercise interactions that are dependent on JavaScript, let's take a moment to consider what we've learned.

- Webrat simulates a browser by building on the functionality of the Rails integration testing API, providing an expressive language to describe manipulating a web application.
- By specifying behaviour at a high level and avoiding coupling our tests to implementation details, we can build expressive and robust step definitions that give us confidence that the full Rails stack stays working while avoiding brittle scenarios.
- Leveraging the DMA style for Givens can provide convenience, simplicity and speed without reducing confidence. We use this approach when the actions required to get to a specific database

state have already been exercised through the full Rails stack in their own Simulated Browser scenarios.

- Through the course of describing the actions in our scenarios in our When steps, Webrat implicitly ensures that requests are successful and the right links and form elements are on the page. In our Then steps we specify the outcomes from our scenarios in terms of expected text and elements using Webrat's view matchers.

Chapter 20

Automating the Browser with Webrat

Coming soon ...

Chapter 21

Rails Views

The user interface is subject to more change than just about anything else in the life of an application. These changes are driven by usability concerns, user experience considerations, design aesthetics, as well as evolving application behaviour. This makes producing simple, flexible, and easy to change views desirable and beneficial. But there is more.

Views can also have a profound impact on the design of the underlying code base. Focusing on views last commits you to a design and implementation before you have an opportunity to discover what you need and want to work with. We think that focusing on views first, letting emergent design take its course, will lead to simpler views, driving an application *design* that is better aligned with application *behaviour*.

This chapter is going to take you through the process of developing views from the outside-in, using view specs written in RSpec as the driving force.

21.1 Writing View Specs

A view spec is a collection of code examples for a particular view template. Unlike examples for POROs (plain old ruby objects), view examples are inherently state-based. We specify expectations by supplying objects and data, rendering the view, and then looking at the rendered content.

In most cases, we're interested in the semantic content as it pertains to requirements of the application, as opposed to the syntactical correctness of the markup. The main exception to this is forms, in which case

we do want to specify that form elements are rendered correctly within a form tag.

> **A Big Misconception**
> *by Zach Dennis*
>
> I routinely meet developers who dismiss the value in writing view specs. After talking with them I quickly understand why. They often think about the specs as strictly syntactical verification for page layout! This just isn't the case.
>
> Page structure changes too frequently to try to keep specs in sync with markup changes. It's just not that valuable to know that a div is inside of another div which is inside of something else—just like it's not that valuable to assert static content. Markup and static content are better handled by people, not specs.
>
> Before I started driving views with specs I noticed artifacts creeping up into the views. They'd start slowly at first, but over time they'd make it painful to change the views and awkward to extend parts of the app.
>
> In an effort to alleviate the pain these artifacts would be moved into helpers, controller actions, and model methods. This did make the views more manageable, but it didn't always make the code easier to change or understand. In many cases the problems had really just found a new home. And once the bandaid wore off the design of the app appeared worse off.
>
> I've found that driving views with specs puts me in a position to make better decisions in small repeatable increments as I quickly piece together a view. I spend far less time running into issues related to bad design, either in the views or in the rest of the app. Specs make it natural to define clean separation of what goes in a view, what goes in the controller, and what goes in the model.
>
> For me, driving views with specs influences the entire application—for the better.

Driving views through examples helps us think about what the view needs in order for it to do its job. It encourages us to write the code we wish we had to fulfill those needs. This leads to simple views that defer logic and complexity to objects with well-defined interfaces. The result is views that are flexible and adaptable in the face of the high frequency of changes of the UI.

Not every example will lead you to an "Ah-ha!" moment. Some will only lead to views that meet simple expectations, but even that has

value. After all, it is often the simple things which lead to that “Ah-ha!" moment.

So, let’s get started. You’ll want to generate a fresh Rails app with RSpec, rspec-rails, and Webrat installed. We won’t be using Cucumber in this chapter so it’s not required, but it won’t be in our way if you have it installed. Key in the following commands:

```
rails views_example
cd views_example
sudo gem install rspec rspec-rails webrat
script/generate rspec
```

We’re going to use the Webrat matchers that you learned about in Chapter 19, *Simulating the Browser with Webrat*, on page 220, so we’ll need to configure the project accordingly.

Configuring Webrat

Open up spec_helper.rb and add the following line after rspec is required, but before the Spec::Runner.configure block

```
require 'webrat'
```

Now add this line anywhere inside of the Spec::Runner.configure block. This exposes Webrat’s matchers to the view specs:

```
config.include Webrat::Matchers, :type => :views
```

Here’s an example spec_helper.rb file to show you the placement. Your spec_helper.rb file may not look identical to this:

```
# move out to code example
ENV["RAILS_ENV"] = "test"
require File.expand_path(File.dirname(__FILE__) + "/../config/environment")
require 'spec'
require 'spec/rails'

require 'webrat'

Spec::Runner.configure do |config|
  config.use_transactional_fixtures = true
  config.use_instantiated_fixtures  = false
  config.fixture_path = RAILS_ROOT + '/spec/fixtures/'

  config.include Webrat::Matchers, :type => :views
end
```

That’s it. Webrat and RSpec are ready to go. Now, on to writing view specs.

Simple Example

We're going to build a view that displays a single message, and we're going to write a spec for it first. Create a show.html.erb_spec.rb file in the RAILS_ROOT/spec/views/messages/ directory. That directory won't exist so you'll need to create it. Here's what the contents of that spec should look like:

Download rails_views/messages/01/spec/views/messages/show.html.erb_spec.rb

```
require File.expand_path(File.dirname(__FILE__) + '/../../spec_helper')

describe "messages/show.html.erb" do
  before(:each) do
    @message = stub("Message")
    assigns[:message] = @message
  end

  it "should display the text of the message" do
    @message.stub!(:text).and_return "Hello world!"
    render "messages/show.html.erb"
    response.should contain("Hello world!")
  end
end
```

The script/generate rspec command we ran earlier installed a script/spec command. Use that now to run the spec:

```
script/spec spec/views/messages/show.html.erb_spec.rb
```

You should have the following failure:

```
Missing template messages/show.html.erb
```

The template doesn't exist yet. Go ahead and add show.html.erb to the RAILS_ROOT/app/views/messages/ directory (which you'll need to make) and run the spec again.

```
expected the following element's content to include "Hello world!"
```

This time it failed because there's nothing in the show.html.erb template. Now add the following to make the example pass:

Download rails_views/messages/03/app/views/messages/show.html.erb

```
<%=h @message.text %>
```

Run the spec again. You should get one example, zero failures. And that's it for the first example. Pretty simple, right? While this example didn't do justice to the intricacies views are often composed of, it did give us just enough to start us with a foundation in which to build.

Let's break down the spec:

Download rails_views/messages/01/spec/views/messages/show.html.erb_spec.rb

```
require File.expand_path(File.dirname(__FILE__) + '/../../spec_helper')

describe "messages/show.html.erb" do
  before(:each) do
    @message = stub("Message")
    assigns[:message] = @message
  end

  it "should display the text of the message" do
    @message.stub!(:text).and_return "Hello world!"
    render "messages/show.html.erb"
    response.should contain("Hello world!")
  end
end
```

The before block gets run before every example. The first line inside it is @message = stub("Message") which assigns a stub to the @message instance variable in the spec. At this point the instance variable is not accessible to the view when it gets rendered.

The second line in the before block is assigns[:message] = @message which makes a @message instance variable available to the view. The name of the variable is determined by the symbol :message that is passed into assigns(). If that line read assigns[:foo] = @message then a @foo instance variable would be available in our view which pointed to the message stub.

Each time an example in the view spec runs a new stub will be created and assigned to a local instance variable in the spec, which would then be assigned to an instance variable accessible by the view.

Now for the code example. The first thing it does is stub out the text() method on the stub message. Setting up objects and data for the view needs to happen before the view is actually rendered.

The next line, render "messages/show.html.erb", renders the messages/show.html.erb template.

The last line, response.should contain("Hello World!"), specifies that the view displays the same text that is being returned from the text() method that we stubbed only two lines before. Isn't it easy to read the example with everything relevant being included inside the it block?

In addition to understanding the breakdown of a spec here are a few more things we can glean from what we just did:

Directory organization The directory structure for view specs mimics the directory structure found in RAILS_ROOT/app/views/. For example, specs found in spec/views/messages/ will be for view templates found in app/views/messages/.

File naming View specs are named after the template they provide examples for, with an _spec.rb appended to the filename. For example, index.html.erb would have a corresponding spec named index.html.erb_spec.rb.

Always require spec_helper.rb Every view spec will need to require the spec_helper.rb file. Otherwise you'll get errors about core rspec or rspec-rails methods not existing.

Describing view specs The outer describe() block in a view spec typically uses the path to the view minus the RAILS_ROOT/app/views/ portion. While this isn't a hard requirement, it clearly communicates what template the examples are for.

In addition to revealing some of the conventions we use with RSpec and Rails, we also used some helper methods and matchers:

assigns()

The assigns() method is used to assign instance variables in the view. It is equivalent to assigning instance variables in a controller and having Rails make them available to the view. For example:

```
assigns[:foo] = stub("foo")
assigns[:text] = "some text"
```

A call to assigns() must happen before the call to render(), and you *must* use assigns() to make something available in the view. An instance variable in a view spec will not be accessible to the view being rendered without using assigns().

render()

As its name suggests the render() method is used to render a template. It behaves just like any render call you would use in your application. Beneath the covers, it really is making a render call on a controller—just not on one of your application's controllers.

stub()

You've already learned about the stub() method in Chapter 12, *Mocking in RSpec*, on page 151. The stub used in our view spec is one and the same. In view specs you do have access to the mocking framework that RSpec is configured to use.

contain()

The contain() method is a Webrat matcher. All matchers that Webrat has to offer are available in view specs since we configured RSpec to include them. This means methods like have_selector, have_xpath, etc. are available.

Now that you've got the basics down, let's explore a little beyond.

21.2 Mocking Models

When working outside-in it's common to discover a need for a model that doesn't yet exist. Rather than switch focus to the model we can utilize the mock_model() method provided by rspec-rails to continue flushing out the view.

Mock Example

Building on the messages example we started with earlier in the chapter, we'll introduce the need for a model and continue driving the view, utilizing mock_model.

Based on the view spec conventions we learned about earlier in the chapter, we need a spec named new.html.erb_spec.rb in the spec/views/messages/ directory. Here are the contents of the spec we'll start with:

Download rails_views/messages/03/spec/views/messages/new.html.erb_spec.rb

```
require File.expand_path(File.dirname(__FILE__) + '/../../spec_helper')

describe "messages/new.html.erb" do
  before(:each) do
    @message = stub("Message")
    assigns[:message] = @message
  end

  it "should render a form to create a message" do
    render "messages/new.html.erb"
    response.should have_selector("form[method=post]", :action => messages_path) do
      |form|
      form.should have_selector("input[type=submit]")
    end
  end
end
```

Running the spec should result in a familiar failure. Go ahead and create the new.html.erb template in app/views/messages and run the spec again.

```
NameError in 'messages/new.html.erb should render a form to create a message'
undefined local variable or method `messages_path' for \
#<Spec::Rails::Example::ViewExampleGroup::Subclass_1:0
```

At this point the example is failing because there is no messages_path route. Go ahead and update the routes.rb file to produce the appropriate paths:

Download rails_views/messages/05/config/routes.rb

```
ActionController::Routing::Routes.draw do |map|
  map.resources :messages
end
```

The example will still be failing, but you should see another familiar error message: MissingTemplate. Go ahead and implement the new.html.erb template. Here's one way it could work:

Download rails_views/messages/05/app/views/messages/new.html.erb

```
<% form_for @message do |f| %>
  <%= f.submit "Save" %>
<% end %>
```

The MissingTemplate error is gone, but the spec still fails with a new error:

```
undefined method `spec_mocks_mock_path' for #<ActionView::Base:0x2216d38>
```

What's happening here is that the form_for() method used in the view expects the message to be a model, but it's a simple stub. If this has piqued your interest to discover how and what form_for relies on under the hood, by all means explore! Otherwise, we can get past this failure by using the mock_model() method.

Let's update the before block to use the mock_model() method instead of stub():

Download rails_views/messages/05/spec/views/messages/new.html.erb_spec.rb

```
before(:each) do
  @message = mock_model(Message)
  assigns[:message] = @message
end
```

Running the spec results in a failure:

```
uninitialized constant Message
```

The Message class we passed to mock_model() doesn't exist yet, so we need to create it. Go ahead and make a message.rb file in the app/models/ directory. We'll use a plain ruby class for now. We can subclass ActiveRecord::Base later, when it's time to implement the model.

Download rails_views/messages/06/app/models/message.rb

```
class Message
end
```

The example should still be failing, but for a different reason:

```
expected following output to contain a <form[method=post][action='/messages'] \
action='/messages'/> tag
```

To resolve this failure we need to know a little bit more about mock_model. Using mock_model will produce a mock which acts like an existing model (e.g. new_record?() returns false). When form_for gets an existing model it produces a form action to *update* the model, which is not what we want. We want a form which posts to *create* a new model. It turns out we can do this by telling the mocked model to act like a new record:

Download rails_views/messages/07/spec/views/messages/new.html.erb_spec.rb

```
@message = mock_model(Message).as_new_record
```

Update the before block to do that and re-run the spec. You should now have one example, zero failures.

Now that we've got the form working, let's add some input fields. We'll start with a code example that expects a text field for the message title:

Download rails_views/messages/08/spec/views/messages/new.html.erb_spec.rb

```
it "should render a text field for the message title" do
  @message.stub!(:title).and_return "the title"
  render "messages/new.html.erb"
  response.should have_selector("form") do |form|
    form.should have_selector(
      "input[type=text]",
      :name => "message[title]",
      :value => "the title"
    )
  end
end
```

Run the spec. The example we just wrote should be the only failure. Go ahead and implement the view to resolve that failure:

Download rails_views/messages/09/app/views/messages/new.html.erb

```
<% form_for @message do |f| %>
  <%= f.text_field :title %>
  <%= f.submit "Save" %>
<% end %>
```

Now when you run the spec you still have a failure, but it's coming from the example we didn't touch. Let's take a closer look at the failure

message:

```
Mock 'Message_1001' received unexpected message :title with (no args)
```

RSpec's mock objects let us know when they receive messages they don't expect. In this case, the text_field() helper in the view is asking the mock message for its title attribute, but the mock hasn't been told to expect that request, so it raises an error.

The first example doesn't care about the message title, so we don't want to have to tell the mock to expect title(). What we *can* do is tell the mocked message to act like a null object. This will let us write examples that are well focused without introducing unnecessary verbosity in other examples.

Go ahead and update the before block:

Download rails_views/messages/10/spec/views/messages/new.html.erb_spec.rb

```
before(:each) do
  @message = mock_model(Message, :null_object => true).as_new_record
  assigns[:message] = @message
end
```

Let's add one more example against the new page to make sure the form has a text area for the text of the message. Add this example to your spec:

Download rails_views/messages/10/spec/views/messages/new.html.erb_spec.rb

```
it "should render a text area for the message text" do
  @message.stub!(:text).and_return "the message"
  render "messages/new.html.erb"
  response.should have_selector("form") do |form|
    form.should have_selector(
      "textarea",
      :name => "message[text]",
      :content => "the message"
    )
  end
end
```

Go ahead and implement the example. You should end up with three examples, zero failures. This time the new field in the form didn't cause other examples to fail, thanks to making the message a null object.

Mock models that act as_null_object help to keep view specs lean and simple, allowing each example to be explicit about only the things it cares about. They also save us from unwanted side-effects being introduced in other examples.

Now let's take a look at mock_model a little more in-depth.

mock_model()

The mock_model() method sets up an RSpec mock with common ActiveRecord methods stubbed out. In its most basic form mock_model can be called with a single argument: the class you want to represent as an ActiveRecord model. The class must exist, but it doesn't have to be a subclass of ActiveRecord::Base. Here are the default stubs on a mocked model:

new_record? Returns false since mocked models represent existing records by default.

id Returns an auto-generated number to represent an existing record.

to_param Returns a string version of the id.

Just like standard mocks/stubs in RSpec, additional methods can be stubbed by passing in an additional Hash argument of method name/value pairs. For example:

```
user = mock_model(User,
  :login => "zdennis",
  :email => "zdennis@example.com"
)
```

In case you don't want your mock to represent an existing record, you can tell it to be a new record by sending it the as_new_record() message:

```
new_user = mock_model(User).as_new_record
```

This will change the default values stubbed by mock_model to the following:

new_record? Will return true just like a new ActiveRecord object.

id Will return nil just like a new ActiveRecord object.

to_param Will return nil just like a new ActiveRecord object.

We encourage you to use mock_model when the code example is clearly expressing a dependency on an ActiveRecord model. If that's not the case use the mock() or stub() methods provided by the RSpec mocking framework with a clear description for what is expected in the view.

While mock_model provides value when writing view examples, it can also be used in any kind of spec throughout a Rails app. As your application grows and your models take shape you'll find that you have an alternative, the stub_model() method which we'll explore next.

stub_model

The stub_model() method is similar to mock_model() except that it creates an actual instance of the model. This requires that the model has a corresponding table in the database.

You create a stub_model just like a mock_model: the first argument is the model to instantiate, and the second argument is a Hash of method/value pairs to stub.

```
user = stub_model(User)

user = stub_model(User,
  :login => "zdennis",
  :email => "zdennis@example.com"
)
```

Similar to mock_model, a stubbed model represents an existing record by default, and you can tell it to act like a new record with as_new_record(). In fact, stub_model is a lot like mock_model, with just a couple of exceptions.

Because stub_model creates an ActiveRecord model instance, you don't need to tell it to act as_null_object() to keep it quiet when asked for its attributes. ActiveRecord will just return nil in those cases, as long as the attribute *is* defined in the schema.

The other difference is that stub_model() prohibits the model instance from accessing the database. If it receives any database related messages, like save(), or update_attributes(), it will raise an error:

```
Spec::Rails::IllegalDataAccessException: stubbed models are not allowed to \
access the database
```

This can be a good indicator that your view is doing something it shouldn't be doing, or that the method in question should really be stubbed out in the example.

Like mock_model, stub_model can be used in any kind of spec throughout your Rails app.

21.3 Working with Partials

Rails developers work with partials every day. We introduce them to re-use part of a view, or simply break up a view into more manageable chunks. We'll explore how partials are handled in the outside-in style of driving views.

Extracting Partials for Organization

Once again we'll build on the messages example we started with earlier in this chapter. We'll start where we left off in Section 21.2, *Mock Example*, on page 251. We had just produced a new page with a form to create a simple message, and now we're going to add a sidebar component to the page to display recent messages.

Starting with the spec, open up the new.html.erb_spec.rb again, and add the following example group:

Download rails_views/messages/11/spec/views/messages/new.html.erb_spec.rb

```
it "should render recent messages" do
  assigns[:recent_messages] = [
    mock_model(Message, :text => "Message 1", :null_object => true),
    mock_model(Message, :text => "Message 2", :null_object => true)
  ]
  render "messages/new.html.erb"
  response.should have_selector(".recent_messages") do |sidebar|
    sidebar.should have_selector(".message", :content => "Message 1")
    sidebar.should have_selector(".message", :content => "Message 2")
  end
end
```

Run the spec. You should have four examples, one failure. Let's implement the view to satisfy the latest example:

Download rails_views/messages/11/app/views/messages/new.html.erb

```
<div class="sidebar">
  <ul class="recent_messages">
    <% @recent_messages.each do |message| %>
      <li class="message"><%=h message.text %></li>
    <% end %>
  </ul>
</div>

<% form_for @message do |f| %>
  <%= f.text_field :title %>
  <%= f.text_area :text %>
  <%= f.submit "Save" %>
<% end %>
```

Semantics vs Syntax

Notice that we added a *sidebar* container on the page, but it's not referenced in the spec. The spec also references elements with the classes recent_messages and message, but it does not specify that those elements should be ul and li, or that the ul is inside a div.

While we do care about the semantic details, we do *not* care about syntactic detail. This distinction is not always black and white. For example, we do care that the messages are inside a recent_messages container, and that has both semantic and syntactic implications.

Semantic HTML and view specs complement each other. They both provide meaning about what's being displayed, rather than just the markup. Even if the presentation of the recent messages changes, we can be assured that they are still being displayed. This adds to the flexibility of the view and its spec.

Getting back to the spec, there are now four examples, three of which are failing.

The example we just added is passing, but we broke the other three! That's because the view now depends on a @recent_messages instance variable that is only being set up by the new example. Even though the other examples aren't interested in @recent_messages, they still have to supply it for the view to render properly.

Because the other examples don't care what's in @recent_messages, however, we can just supply an empty array in the before(:each) block:

Download rails_views/messages/11/spec/views/messages/new.html.erb_spec.rb

```
before(:each) do
  @message = mock_model(Message, :null_object => true).as_new_record
  assigns[:message] = @message
  assigns[:recent_messages] = []
end
```

Now all four examples should be passing.

Now that we've got a sidebar, let's move it out into its own partial. Not necessarily for re-use at this point, but just for organization.

Create a messages/_sidebar.html.erb template and move the sidebar container to reside in that partial. In doing this, you should update the _sidebar.html.erb partial to rely on a local variable recent_messages rather than on an instance variable:

Download rails_views/messages/12/app/views/messages/_sidebar.html.erb

```
<div class="sidebar">
  <ul class="recent_messages">
    <% recent_messages.each do |message| %>
      <li class="message"><%=h message.text %></li>
    <% end %>
  </ul>
</div>
```

Now update the new.html.erb template to render the sidebar partial, passing in the appropriate locals:

Download rails_views/messages/12/app/views/messages/new.html.erb

```
<%= render :partial => "sidebar",
           :locals => { :recent_messages => @recent_messages } %>

<% form_for @message do |f| %>
  <%= f.text_field :title %>
  <%= f.text_area :text %>
  <%= f.submit "Save" %>
<% end %>
```

Run the spec again—four examples, zero failures.

This is how including partials works in Rails. We don't create a new view spec for the partial and the examples in the existing view spec don't change. The only thing that changes is how the view being rendered is organized. This approach works well when partials are extracted to make portions of the UI more manageable.

Let's take this example one step further. Let's re-use it in another view and find out about another technique—isolating partials.

Extracting Partials for Re-use

At this point the new page has a form to create a message and it has a sidebar to display recent messages. But how often is there a new page without a corresponding edit page? Not that often. We need a way to allow people to edit their messages. Who else is going to fix those typos?

We're going to extract the form for re-use, which is exactly like extracting the sidebar for organization, except that we'll also extract the form's spec from new.html.erb_spec.rb.

Start by creating app/views/messages/_form.html.erb, copying over the form from app/views/messages/new.html.erb, and rendering the _form partial from the new template:

Download rails_views/messages/14/app/views/messages/new.html.erb

```
<%= render :partial => "sidebar",
           :locals => { :recent_messages => @recent_messages } %>

<%= render :partial => "form", :locals => { :message => @message } %>
```

Download rails_views/messages/14/app/views/messages/_form.html.erb

```
<% form_for message do |f| %>
```

```
  <%= f.text_field :title %>
  <%= f.text_area :text %>
  <%= f.submit "Save" %>
<% end %>
```

Be sure to update the call to form_for() to utilize the message variable passed in from the locals rather than relying on the @message instance variable. Now run spec/views/messages/new.html.erb_spec.rb and it should still be passing.

Next, create spec/views/messages/_form.html.erb_spec.rb and copy over all the form related examples from spec/views/messages/new.html.erb_spec.rb. You'll need to modify them to render the partial instead of the new template, and pass in the appropriate locals:

Download rails_views/messages/14/spec/views/messages/_form.html.erb_spec.rb

```
require File.expand_path(File.dirname(__FILE__) + '/../../spec_helper')

describe "messages/_form.html.erb" do
  before(:each) do
    @message = mock_model(Message, :null_object => true)
  end

  context "when the message is a new record" do
    it "should render a form to create a message" do
     @message.as_new_record
     render "messages/_form.html.erb", :locals => { :message => @message }
     response.should have_selector("form[method=post]", :action => messages_path) do
       |form|
       form.should have_selector("input[type=submit]")
     end
    end
  end

  context "when the message is an existing record" do
    it "should render a form to update a message" do
      render "messages/_form.html.erb", :locals => { :message => @message }
      response.should have_selector(
        "form[method=post]",
        :action => message_path(@message)
      ) do |form|
        form.should have_selector("input[type=submit]")
      end
    end
  end

  it "should render a text field for the message title" do
    @message.stub!(:title).and_return "the title"
    render "messages/_form.html.erb", :locals => { :message => @message }
    response.should have_selector("form") do |form|
```

```
      form.should have_selector(
        "input[type=text]",
        :name => "message[title]",
        :value => "the title"
      )
    end
  end

  it "should render a text area for the message text" do
    @message.stub!(:text).and_return "the message"
    render "messages/_form.html.erb", :locals => { :message => @message }
    response.should have_selector("form") do |form|
      form.should have_selector(
        "textarea",
        :name => "message[text]",
        :content => "the message"
      )
    end
  end
end
```

Once you see that passing, add an example to new.html.erb_spec.rb to specify that it renders the _form partial:

Download rails_views/messages/14/spec/views/messages/new.html.erb_spec.rb

```
it "should render the messages/form" do
  template.should_receive(:render).with(
    :partial => "form",
    :locals => { :message => @message }
  ).and_return "rendered form partial"
  render "messages/new.html.erb"
  response.should contain("rendered form partial")
end
```

If you run new.html.erb_spec.rb now, you'll see four passing examples. These are the four that got copied over to _form.html.erb_spec.rb. Because those same examples are passing in the _form spec, we can safely remove them from the new spec.

Now that we're done refactoring, we can go back into spec-driving mode and add an example that expects the edit template to render the form partial. Go ahead and create a spec for edit with the following example:

Download rails_views/messages/14/spec/views/messages/edit.html.erb_spec.rb

```
it "should render the messages/form" do
  template.should_receive(:render).with(
    :partial => "form",
    :locals => { :message => @message }
  ).and_return "rendered form partial"
  render "messages/edit.html.erb"
```

```
  response.should contain("rendered form partial")
end
```

Run that spec, watch it fail, and add a edit.html.erb template that renders the partial:

Download rails_views/messages/14/app/views/messages/edit.html.erb

```
<%= render :partial => "form", :locals => { :message => @message } %>
```

Run the spec again and, voila! We're back to green.

This time we extracted a partial with the goal of re-use, not just to organize the application code. By extracting a partial with its own spec, any changes to requirements for this form will only impact this one spec and the form itself. This helps keep views and their specs very easy to maintain.

Speaking of maintaining specs, let's look at a few techniques for managing specs as they grow and repetitive patterns emerge.

21.4 Refactoring Code Examples

Over time, patterns begin to emerge and take shape in view specs. These patterns range from single statement patterns, like expecting similar content in different specs, to duplicate examples. RSpec supports two techniques for removing this sort of duplication: shared examples and custom matchers.

You learned about custom matchers in Section 15.3, *Custom Matchers*, on page 182. The shared examples are the same ones you learned about in Section 10.5, *Sharing Examples in a Module*, on page 122. Let's look at using them first.

Shared Examples

View specs may be the sweet spot for shared examples in Rails apps since the examples are easily isolated from one another and there's no need to share state across examples. You can simply override the expected assigns, or pass in the appropriate options to render().

In the last section, we extracted a form partial from the new page so we could re-use it in the edit page. You may have noticed that at the end of the process we had duplicate examples in new.html.erb_spec.rb and edit.html.erb_spec.rb for ensuring the form partial was rendered. We can

remove this duplication by using shared examples to consolidate the example to one location with a good description.

Make the spec/views/shared_examples/shared_partials_spec.rb file with the following shared example:

Download rails_views/messages/14/spec/views/shared_examples/shared_partials_spec.rb

```
shared_examples_for "a template that renders the messages/form partial" do
  it "should render the messages/form" do
    template.should_receive(:render).with(
      :partial => "form",
      :locals => { :message => @message }
    ).and_return "rendered form partial"
    render "messages/new.html.erb"
    response.should contain("rendered form partial")
  end
end
```

Right now the render call knows which template to render. We don't want it to know that, otherwise it'd be a pretty lousy shared example. We can take advantage of the render() method's implicit default for the template: if no template is submitted, it will grab the first argument passed to describe() in the including spec.[1]

Download rails_views/messages/15/spec/views/shared_examples/shared_partials_spec.rb

```
render
```

With that change, we can update new.html.erb_spec.rb to utilize the shared example. Add the following line to the spec:

Download rails_views/messages/15/spec/views/messages/new.html.erb_spec.rb

```
it_should_behave_like "a template that renders the messages/form partial"
```

Go ahead and run the spec. It should fail with a similar message:

```
Shared Example Group 'a template that renders the messages/form partial' can not \
be found (RuntimeError)
```

The spec can't find the shared example. We want shared examples to always be loaded so specs that rely on them work. Let's update spec_helper.rb to load all spec files found in any shared_examples/ directory. Append the following to your spec_helper.rb:

Download rails_views/messages/16/spec/spec_helper.rb

```
Dir[File.dirname(__FILE__) + "/**/shared_examples/**/*_spec.rb"].each do |file|
  require file
```

1. This feature was added in rspec-rails-1.2

```
end
```

Run the spec again—three examples, zero failures. Let's do a quick sanity check to make sure it's doing what we expect. Open the new page and remove the form. Now run the spec again—it should fail. Go ahead and put the form back in place, run the spec again, and watch it pass. Now we can be confident our shared example is doing what we expect.

With the shared example working let's remove the original examples which ensured the form partial was being rendered. The new.html.erb_spec.rb should look like:

Download rails_views/messages/16/spec/views/messages/new.html.erb_spec.rb

```
require File.expand_path(File.dirname(__FILE__) + '/../../spec_helper')

describe "messages/new.html.erb" do
  before(:each) do
    @message = mock_model(Message, :null_object => true).as_new_record
    assigns[:message] = @message
    assigns[:recent_messages] = []
  end

  it_should_behave_like "a template that renders the messages/form partial"

  it "should render recent messages" do
    assigns[:recent_messages] = [
      mock_model(Message, :text => "Message 1", :null_object => true),
      mock_model(Message, :text => "Message 2", :null_object => true)
    ]
    render "messages/new.html.erb"
    response.should have_selector(".recent_messages") do |sidebar|
      sidebar.should have_selector(".message", :content => "Message 1")
      sidebar.should have_selector(".message", :content => "Message 2")
    end
  end
end
```

For practice, go ahead and make the same change to edit.html.erb_spec.rb. Then let's explore writing custom matchers.

Custom Matchers

As an application grows, we find ourselves expecting similar content in many view specs. We see the same links, form fields, custom HTML components, and even convention-based markup. For these cases, we can write custom matchers to reduce duplication and produce more expressive examples.

Simple Matcher Example

Consider the case where your project utilizes a convention for using anchor tags and CSS to produce visually stunning buttons used throughout the site. Rather than rely on manually typing in the appropriate selector, we can improve on this by writing a custom matcher.

Take this example:

Download rails_views/messages/16/spec/views/buttons.html.erb_spec.rb

```
it "should have a button to create a message" do
  render "buttons.html.erb"
  response.should have_selector("a.button", :href => messages_path)
end
```

We can make a small improvement by pulling out the pattern into a custom matcher rather than duplicate the effort for having to remember the appropriate CSS selector each time we use a button. Here's our new matcher, using RSpec's simple_matcher and Webrat's have_selector matcher to do most of the work for us:

Download rails_views/messages/16/spec/spec_helpers/views/matchers.rb

```
def have_button(href)
  simple_matcher("a button to #{href}") do |response|
    response.should have_selector("a.button", :href => href)
  end
end
```

Now our examples can be slightly improved:

Download rails_views/messages/16/spec/views/buttons.html.erb_spec.rb

```
it "should have a button to create a message" do
  render "buttons.html.erb"
  response.should have_button(messages_path)
end
```

This example is a little win. It increases the clarity of each example that expects a button. If the convention changes, we've isolated that change to one place—the have_button() matcher.

Organizing Matchers

Custom matchers can be put anywhere you want as long as they are required from spec_helper.rb and included for the appropriate specs. A simple guideline to follow is to organize custom matchers by the type of spec they're going to be used in.

Consider the have_button() matcher we just looked at. A good home for this might be a ViewHelpers::Matchers module in spec/spec_helpers/views/matchers.rb. Here's what the file would look like:

Download rails_views/messages/16/spec/spec_helpers/views/matchers.rb

```
module ViewHelpers
  module Matchers
    def have_button(href)
      simple_matcher("a button to #{href}") do |response|
        response.should have_selector("a.button", :href => href)
      end
    end
  end
end
```

To get access to the view matchers in view specs, all you need to do is update spec_helper.rb to include them. You do this by adding a line that requires every ruby file in the spec/spec_helpers/ directory. This line goes before the Spec::Runner.configure block:

Download rails_views/messages/16/spec/spec_helper.rb

```
Dir[File.dirname(__FILE__) + "/spec_helpers/**/*.rb"].each do |file|
  require file
end
```

Next, inside the Spec::Runner.configure block add the following line to include the matcher module:

Download rails_views/messages/16/spec/spec_helper.rb

```
config.include ViewHelpers::Matchers, :type => :view
```

That's it. Now any custom matchers we add to ViewHelpers::Matchers will be made available to our view specs.

21.5 What We Just Learned

- User Interface changes more often than anything else in an application.
- Specifying syntactic detail makes code examples brittle. We want to specify view semantics without getting bogged down in syntactic detail.
- View specs use a custom example group provided by the rspec-rails library.

- View specs live in a directory tree parallel to the views themselves, and follow a naming convention of spec/path/to/view.html.erb_spec.rb for app/path/to/view.html.erb.
- Use Webrat's have_xpath() and have_selector() matchers for view specs.
- Use mock_model() and stub_model() to isolate view specs from the database and underlying business logic of your models.
- Partials can have their own view specs.
- Extract shared examples and custom matchers to help keep your view specs DRY.

One thing we didn't cover is how to specify helper methods that we extract from views into helper modules. That's because rspec-rails supports specifying helpers in isolation from the controllers and views that use them, in the form of Helper Specs. We'll be discussing that in detail in the (as yet) unwritten *chp.railsHelpers*.

Chapter 22

Rails Helpers

Coming soon ...

Rails' controllers are like waiters in a restaurant. A customer orders a steak dinner from a waiter. The waiter takes the request and tells the kitchen that he needs a steak dinner. When the steak dinner is ready the waiter delivers it the customer for her enjoyment.

► Craig Demyanovich

Chapter 23

Rails Controllers

The restaurant metaphor does a great job describing the role of controllers in a Rails application. Just as a waiter doesn't need to know how to prepare a steak dinner, a controller doesn't need to know the details of building a model. Keeping these details out of the controller provides a natural separation of concerns between the controller and the model, which makes the models easier to change, extend, and reuse.

This chapter will show you how to develop controllers outside-in using controller specs as the driving force.

23.1 Writing Controller Specs

A controller spec is a collection of examples of the expected behaviour for actions on a controller. Whereas views are inherently state-based, controllers are naturally interaction-based. They wait at the edges of a Rails app to mediate interaction between models and views given an incoming request. So we specify their expectations through interactions, process the action, and look at assigned instance variables and flash messages made available for the view.

In most cases, it's the interaction between the controller and its collaborators (usually models) that we're interested in when writing controller specs. And by default, controller specs don't render views.[1] Combine that fact with judicious use of mocks and stubs for interaction with the model, and now we can specify controller interactions in complete isolation from the other components. This pushes us to build skinny

1. You can tell controller specs to render views with the integrate_views() method.

controllers and helps us to discover objects with well named methods to encapsulate the real work.

A simple guideline for a controller is that it should know *what* to do, but not *how* to do it. Controllers that know too much about the *how* tend to become responsible for too many things and as a result become bloated, messy, and hard understand. This will become clear as we work through an example.

We'll use the same Rails app that we used in the last chapter.

Simple Example

In the last chapter we built up the view which contained the form to create a message. Now we're going to develop a controller and action responsible for processing that form submission. The action will also create the message. We'll follow Rails' conventions and make a MessagesController with a create action.

Start by creating a messages_controller_spec.rb file in the RAILS_ROOT/spec/controllers/ directory. That directory probably doesn't exist yet, so you may need to create it. Here is the barebones spec that we'll start with:

Download rails_controllers/messages/01/spec/controllers/messages_controller_spec.rb

```
Line 1  require File.expand_path(File.dirname(__FILE__) + '/../spec_helper')
   2
   3  describe MessagesController, "POST create" do
   4
   5    it "should build a new message"
   6
   7    it "should save the message"
   8
   9  end
```

The examples on lines 5 and 7 are considered *pending* because they have no blocks. You can read more about pending examples in Section 10.2, *Pending Examples*, on page 113.

Run the spec and you should get *uninitialized constant MessagesController*, generated by the reference on line 3 to a non-existent MessagesController. Go ahead and create that class in app/controllers/messages_controller.rb, run the spec again, and now the output should tell you that there are 2 examples, 0 failures, 2 pending.

The first pending example suggests that we want the create() action to build a new message, so let's add a block that sets that expectation.

Joe Asks...

Isn't Message.should_receive(:new) implementation?

At some level, yes it is, but it's not the same as specifying internal implementation details that only occur within the object being spec'd. We're specifying the interaction with other objects in order to isolate this example from anything that might go wrong, or does not yet exist in the other objects. That way when a controller spec fails, you know that it's because the controller is not behaving correctly and can quickly diagnose the problem.

One of the motivations for this approach in Rails controller specs is that we don't have to worry about changes to model validation rules causing failures in controller specs. Rails fixtures can also help solve that problem if you use them judiciously. Test data builders like Fixjour, Factory Girl, Object Daddy and Machinist can also help. But fixtures and Test Data Builders all use a database, which slows down the specs, even if they maintain rapid fault isolation.

Download rails_controllers/messages/02/spec/controllers/messages_controller_spec.rb

```
it "should build a new message" do
  Message.should_receive(:new).with("body" => "a quick brown fox")
  post :create, :message => { "body" => "a quick brown fox" }
end
```

Running the spec should result in a failure since we haven't added a create action on the controller. Based on our example we can make this pass by building a new message in the action:

Download rails_controllers/messages/02/app/controllers/messages_controller.rb

```
def create
  Message.new params[:message]
end
```

The spec should now be at 2 examples, 0 failures, 1 pending.

Moving right along, the second example suggests that we we want the controller to save the message. Again, add a block to express the expectation:

Download rails_controllers/messages/02/spec/controllers/messages_controller_spec.rb

```
it "should save the message" do
  message = mock_model(Message)
  Message.stub!(:new).and_return message
  message.should_receive(:save)
  post :create
end
```

The example should fail with the following message:

```
Mock 'Message_1001' expected :save with (any args) once, but received it 0 times
```

To get this to pass all we need to do is call save() on the message:

Download rails_controllers/messages/03/app/controllers/messages_controller.rb

```
def create
  message = Message.new params[:message]
  message.save
end
```

Run the spec again and you'll see the second example is now passing, but we broke the first example in the process: 2 examples, 1 failure. There is no message object in the first example, and there needs to be one for the code in the action to run.

We can get the first example to pass without impacting the second example by introducing a mock message:

Download rails_controllers/messages/03/spec/contr . . . rs/messages_controller_refactor1_spec.rb

```
Line 1  it "should build a new message" do
2     message = mock_model(Message, :save => nil)
3     Message.should_receive(:new).
4       with("body" => "a quick brown fox").
5       and_return(message)
6     post :create, :message => { "body" => "a quick brown fox" }
7   end
```

Here we create a mock message on line 2 and tell the Message class to return it in response to new() on line 5.

Run the examples and you'll see 2 examples, 0 failures.

We've made progress, but we've also introduced some duplication between the two examples. We can clean that up by extracting out the common bits to a before(:each) block:

Download rails_controllers/messages/03/spec/contr . . . rs/messages_controller_refactor2_spec.rb

```
describe MessagesController, "POST create" do

  before(:each) do
```

```
    @message = mock_model(Message, :save => nil)
    Message.stub!(:new).and_return(@message)
  end

  it "should build a new message" do
    Message.should_receive(:new).
      with("body" => "a quick brown fox").
      and_return(@message)
    post :create, :message => { "body" => "a quick brown fox" }
  end

  it "should save the message" do
    @message.should_receive(:save)
    post :create
  end

end
```

The spec should still have 2 examples, 0 failures.

Adding context specific examples

Our spec isn't done though. Controllers typically do different things depending on whether or not the work they delegate succeeds or fails. Let's start with the *happy day* case and specify what should happen when the save() succeeds. Add these pending examples after the second example:

Download rails_controllers/messages/04/spec/controllers/messages_controller_spec.rb

```
context "when the message saves successfully" do
  before(:each) do
    @message.stub!(:save).and_return true
  end

  it "should set a flash[:notice] message"

  it "should redirect to the messages index"
end
```

We use the context() method to express the *given* context for an example or a group of examples.[2] By convention, though not enforced programatically, we express the same *given* in a before() block within the context block. In this case, we'll stub the save() method, telling it to return true. This is Rails' way of indicating that the save() succeeded.

Let's fill in the code for the first example:

2. context() is an alias for describe()

Download rails_controllers/messages/04/spec/controllers/messages_controller_flash_spec.rb

```
it "should set a flash[:notice] message" do
  post :create
  flash[:notice].should == "The message was saved successfully."
end
```

That example fails because the flash[:notice] is nil. Let's update the create action:

Download rails_controllers/messages/05/app/controllers/messages_controller.rb

```
def create
  message = Message.new params[:message]
  if message.save
    flash[:notice] = "The message was saved successfully."
  end
end
```

That example should be passing now, so let's move on to the next example:

Download rails_controllers/messages/05/spec/contr . . . ers/messages_controller_redirect_spec.rb

```
it "should redirect to the messages index" do
  post :create
  response.should redirect_to(messages_path)
end
```

This example fails with expected redirect to "/messages", got no redirect. Add the redirect to get it to pass:

Download rails_controllers/messages/06/app/controllers/messages_controller.rb

```
def create
  message = Message.new params[:message]
  if message.save
    flash[:notice] = "The message was saved successfully."
    redirect_to messages_path
  end
end
```

Both happy day examples should be passing. Now we can move on to the failure case. Add the following failure context to the spec:

Download rails_controllers/messages/06/spec/contr . . . lers/messages_controller_failure_spec.rb

```
context "when the message fails to save" do
  before(:each) do
    @message.stub!(:save).and_return false
  end

  it "should assign @message"

  it "should render the new template"
```

```
end
```

Just like we did before, we express the context in a before() block, but this time we tell the save() to return false, indicating that the save() failed.

Run the spec and you should see 6 examples, 0 failures, 2 pending. Let's fill in the first pending example:

Download rails_controllers/messages/06/spec/contr . . . llers/messages_controller_assign_spec.rb

```
it "should assign @message" do
  post :create
  assigns[:message].should == @message
end
```

The assigns() method allows us to peek at instance variables assigned to the view. Normally, we wouldn't encourage peeking at the internal state of an object, but Rails' controllers reach directly into views to set their state instead of using public methods that we could capture with message expectations. So, we'll make an exception to this guideline in order to ensure that our expectations are being met.

This example fails saying it expected a Message object, but got nil. Let's update the action to make it pass:

Download rails_controllers/messages/07/app/controllers/messages_controller.rb

```
def create
  @message = Message.new params[:message]
  if @message.save
    flash[:notice] = "The message was saved successfully."
    redirect_to messages_path
  end
end
```

6 examples, 0 failures, 1 pending. Now let's fill out the last example:

Download rails_controllers/messages/07/spec/contr . . . sages_controller_render_template_spec.rb

```
it "should render the new template" do
  post :create
  response.should render_template("new")
end
```

This example fails with expected "new", got "messages/create". We can get this to pass by rendering the new action. Let's update the create action:

Download rails_controllers/messages/08/app/controllers/messages_controller.rb

```
def create
  @message = Message.new params[:message]
  if @message.save
```

```
      flash[:notice] = "The message was saved successfully."
      redirect_to messages_path
    else
►     render :action => "new"
    end
  end
```

6 examples, 0 failures (none pending)! Now we've got a fully implemented controller action. While it's fresh in our heads let's reflect on the spec and the action.

What we just did

The create() action we just implemented is pretty typical of a Rails app. The controller passes the params it receives to the model, delegating the real work. By specifying the interactions with the model instead of the result of the model's work, we are able to keep the spec and the implementation simple and readable.

This is what it means to have a controller know *what* to do without knowing the details of *how* to do it. Any complexity related to building a message will be specified and implemented in the Message model.

The spec we used to drive this action into existence can be used to illustrate some basic conventions we like to follow for controller specs:

Directory organization The directory structure for controller specs parallels the directory structure found in RAILS_ROOT/app/controllers/.

File naming Each controller spec is named after the controller it provides examples for, with _spec.rb appended to the filename. For example, sessions_controller_spec.rb contains the specs for sessions_controller.rb.

Always require spec_helper.rb Each controller spec should require the spec_helper.rb file, which sets up the environment with all the right example group classes and utility methods.

Example group names The docstring passed to the outer-most describe() block in a controller spec typically includes the type of request and the action the examples are for.

While spec-driving the create() action we focused on one example at a time. Once each example passed, we looked for and extracted any duplication to a before block, allowing each example to stay focused, clear, and DRY. And when we found examples that pertained to a given context, we used context blocks with clear descriptions to organize them.

This spec also introduced a number of methods which provide a good foundation for writing controller specs. Many of these methods come directly from ActionController::TestCase, which Rails uses for functional tests. Let's look a closer at each of the methods we used.

assigns()

We use the assigns() method to specify the instance variables we expect to be assigned in the view. It takes a single argument—a symbol that indicates the name of the instance variable.

Note that the assigns() method available in controller specs is different from the one available in view specs. In view specs assigns() is used to set instance variables for a view *before rendering the view*. In controller specs we use assigns() to set expectations about instance variables assigned for the view *after calling the controller action*.

flash()

We use the flash() method to specify messages we expect to be stored in the flash hash. It uses the same API to access flash in the spec as you would use in the controller, which makes it convenient and easy to remember when working with flash.

post()

We use the post() method to simulate a POST request. It can take three arguments. The first argument is the name of the action to call. The second argument (optional) is a hash of key/value pairs to make up the params. The third argument (also optional) is a hash of key/value pairs that make up the session hash for the controller.

```
# no params or session data
post :create

# with params
post :create, :id => 2

# with params and session data
post :create, { :id => 2 }, { :user_id => 99 }
```

The post() method comes directly from ActionController::TestCase, which offers similar methods for get, put, delete, head and even xml_http_request requests. All but the xml_http_request and its alias, xhr, have the same signature as the post() method.

The xml_http_request() and xhr() methods introduce one additional argument to the front: the type of request to make. Then the other arguments are just shifted over. For example:

```
# no params or session data
xhr :get

# with params
xhr :get, :id => 2

# with params and session data
xhr :get, { :id => 2 }, { :user_id => 99 }
```

render_template()

We use the render_template() method to specify the template we expect a controller action to render. It takes a single argument—the path to the template that we are rendering.

There are three variations to the argument that render_template will accept. The first way is to pass in the path to the template minus the RAILS_ROOT/app/views/ portion:

```
response.should render_template("messages/new")
```

The second way is a short hand form of the first. If the template being rendered is a part of the controller being spec'd you can pass in just the template name:

```
# this will expand to "messages/new" in a MessagesController spec
response.should render_template("new")
```

As of Rails 2.3 and RSpec 1.2, we can specify the full filename of the template to be rendered including the filename extension. This lets us specify that the controller should pick a template in the same way it does when the app runs. For example, we can set an expectation that the controller will find and render the messages/new.js.erb template when making a request for JavaScript:

```
# controller action
def new
  respond_to :js, :html
end

# in the spec
get :new, :format => "js"
response.should render_template("new.js.erb")
```

redirect_to()

We use the redirect_to() method to specify that the action should redirect to a pre-defined location. It has the same API as its Rails' counterpart, assert_redirected_to().

```
# relying on route helpers
response.should redirect_to(messages_path)

# relying on ActiveRecord conventions
response.should redirect_to(@message)

# being specific
response.should redirect_to(:controller => "messages", :action => "new")
```

With the basics under your belt let's look at some common controller scenarios that build on what you've just done—starting with *before filters*.

23.2 Before Filters

Before filters are methods that get executed before controller actions. They are an incredibly powerful part of the standard Rails toolbox, and help us to to remove duplication across controllers and actions, change the path of execution for an incoming request, and even stop a request dead in its tracks.

We generally don't specify before filters directly. If a code example specifies that an anonymous user can not view a given resource, it doesn't really care whether authentication is handled directly in the action or in a before filter. However, specs are interested in the behaviour of a controller and before filters can drastically impact those. Not taking them into account can lead to overly verbose and complicated specs. This will become clear as we work through an example.

Login Required Example

We're going to update the message functionality we implemented earlier in this chapter so that you must log in to create a message. Let's add a new group of examples with an anonymous user posting to the create action.

Open the messages_controller_spec.rb file and add the following example group (beginning with only one example) to the top of the POST create describe block:

Download rails_controllers/messages/09/spec/controllers/messages_controller_spec.rb

```
describe "anonymous user" do

  it "should redirect to the login page" do
    post :create
    response.should redirect_to(login_path)
  end

end
```

Running the spec should result in 7 examples, 1 failure, with the new example being the failure. The current controller implementation doesn't care whether a user is anonymous or authenticated. Let's fix this by adding a login_required() before filter:

Download rails_controllers/messages/10/app/controllers/messages_controller.rb

```
class MessagesController < ApplicationController

  before_filter :login_required

  def create
    @message = Message.new params[:message]
    if @message.save
      flash[:notice] = "The message was saved successfully."
      redirect_to messages_path
    else
      render :action => "new"
    end
  end

  protected

  def login_required
    unless current_user
      redirect_to login_path
      false
    end
  end
end
```

7 examples, 7 failures. Whoops! We broke all of the examples—undefined local variable or method `current_user'. The very last thing we changed in the code was addition of the login_required() before filter. Based on how before filters work we know these failures were expected given the addition of login_require(). To address the additional failures we'll need to make those examples aware of the login requirement. Knowing the reason behind the failures and having a solid grasp of what we'll need to do to fix them, we can confidently progress forward with the current

example while letting the others stay red—just until we gets the latest example passing.

We need a current_user() method to represent a logged in user. If we had a login process in place we could rely on it to tell us how to represent a logged in user. Since we don't, let's make the decision to allow the controller to access a logged in user via a current_user() method. Add an empty current_user() method to the controller to satisfy the login_required() before filter:

Download rails_controllers/messages/11/app/controllers/messages_controller.rb

```
class MessagesController < ApplicationController

  before_filter :login_required

  def create
    @message = Message.new params[:message]
    if @message.save
      flash[:notice] = "The message was saved successfully."
      redirect_to messages_path
    else
      render :action => "new"
    end
  end

  protected

  def login_required
    unless current_user
      redirect_to login_path
      false
    end
  end

  def current_user
  end

end
```

Run the spec again—the anonymous user example is still failing. There's no login_path() method. Update the routes to map a login_path:

Download rails_controllers/messages/11/config/routes.rb

```
ActionController::Routing::Routes.draw do |map|
  map.login '/login', :controller => 'sessions', :action => 'new'
  map.resources :messages
end
```

This change puts us at 7 examples, 6 failures, with our anonymous user

example passing! In getting this to pass we found a need for the SessionsController, which we currently don't have in the app. When driving functionality outside-in with Cucumber we wouldn't be able to get the scenario to pass without implementing the login process. For now our focus isn't on the login process. It's on creating a message. Let's keep moving forward without shifting focus.

Fixing what we broke

At this point the anonymous user example is passing and there is one more anonymous user example we need to add, but first let's fix the failing examples. The examples that are failing now represent what happens when an authenticated user POSTS to the create action. To get these examples passing again we need to stub the current_user() method on the controller to return a user. Let's add the stub to the before:

Download rails_controllers/messages/12/spec/controllers/messages_controller_spec.rb

```
before(:each) do
  controller.stub!(:current_user).and_return mock_model(User)
  @message = mock_model(Message, :save => nil)
  Message.stub!(:new).and_return @message
end
```

Running the spec now should result in the examples still failing—*uninitialized constant User*. Let's add a User model in the in app/models/user.rb:

Download rails_controllers/messages/12/app/models/user.rb

```
class User
end
```

Run the spec again—7 examples, 1 failure. We've fixed the failing examples, but now we've made the anonymous user example fail. Oops! Up until now our original examples have lived in the top-level describe. Its before block is being used for our anonymous user examples. We don't want that, so let's wrap the original examples in their own describe block. This will push them down one level of nesting, and it will communicate that these examples are for *authenticated users*. You should end up with the two describe blocks at the same level of nesting in the spec:

```
describe MessagesController, "POST create" do
  describe "anonymous user" do
    ...
  end

  describe "authenticated user" do
    ...
```

```
  end
end
```

The spec should be back to green, 7 examples, 0 failures. The path we took to get the spec back to green had us work with a red bar for a hot minute. Knowing why the examples broke gave us confidence to get the new example passing first before addressing the broken examples. If we didn't have that understanding we would have stopped, undone our changes, and then proceeded with smaller steps.

Adding another anonymous user example

Now that we've got a green bar let's add that second anonymous user example. Not only do we want an anonymous user to be redirected, we also want to make sure the create action is never run. We know that it won't be executed since we've implemented it as a before filter, but there's nothing in our spec communicating that as expected behaviour. Let's add it as an example to the anonymous user describe:

Download rails_controllers/messages/12/spec/controllers/messages_controller_spec.rb

```
describe "anonymous user" do

  it "should redirect to the login page" do
    post :create
    response.should redirect_to(login_path)
  end

  it "should not execute the #create action" do
    controller.should_not_receive(:create)
    post :create
  end

end
```

You might be wondering if this latest example is necessary. Isn't it just confirming how before filters work in Rails? No, it isn't. The example specifies the expectation that an anonymous user should not be able to execute the create action. This is a pretty important expectation, without it anonymous users could wreak havoc upon the app creating unauthorized messages! Making the behaviour explicit in the spec also has the added benefit of providing regression against future controller changes.

With this latest example you should have 8 examples, 0 failures. The create action is now only accessible to authenticated users. The spec is looking good, but let's make a minor refactoring the implementation. Authentication usually isn't restricted to one controller and even though we're

only using it in one controller it makes it easier to find and re-use if we push the current_user() and login_required() methods down to ApplicationController. Do that now:

Download rails_controllers/messages/13/app/controllers/application.rb

```
class ApplicationController < ActionController::Base
  helper :all # include all helpers, all the time

  protect_from_forgery # :secret => '71534792e644ba098610c70211b734e5'

  protected

  def login_required
    unless current_user
      redirect_to login_path
      false
    end
  end

  def current_user
  end
end
```

Run the examples again just to make sure we didn't break anything—8 examples, 0 failures.

Extracting a login helper

Currently we have only one action requiring an authenticated user. When building an app it usually doesn't take long for additional actions to also require an authenticated user. It will become tedious and redundant typing the same line into the before block of every controller spec that requires an authenticated user. Not to mention it'd be very time consuming to have to change every one of those lines should the login process in the app change how it exposes authenticated users.

Let's extract the simple stub that represents a logged in user to a spec helper. This will improve the readability of our specs and have one point of change should the login process be updated.

Create the spec/spec_helpers/controllers/ directory and add to it the login_helpers.rb file. Add a ControllerHelpers module with a login_as_user() method. Copy the line from the before block in the messages_controller_spec.rb that logs in a user to this method. It should look like this:

Download rails_controllers/messages/13/spec/spec_helpers/controllers/login_helpers.rb

```
module ControllerHelpers
```

```
  def login_as_user
    controller.stub!(:current_user).and_return mock_model(User)
  end

end
```

You'll need to update spec/spec_helper.rb to include the ControllerHelpers module so its available in any controller spec. Let's do that now:

Download rails_controllers/messages/13/spec/spec_helper.rb

```
Spec::Runner.configure do |config|
  config.include Webrat::Matchers, :type => :view
  config.include Webrat::HaveTagMatcher, :type => :view
  config.include ViewMatchers, :type => :view
  config.include ControllerHelpers, :type => :controller
end
```

In the last chapter we had updated the spec_helper.rb to include all ruby files in the spec/spec_helpers/ directory. Because of this, the login_helpers.rb file will be loaded automatically for us. Now we can update the before block in messages_controller_spec.rb to rely on the login_as_user() method:

Download rails_controllers/messages/13/spec/controllers/messages_controller_spec.rb

```
before(:each) do
  login_as_user
  @message = mock_model(Message, :save => nil)
  Message.stub!(:new).and_return @message
end
```

Running the spec should still result in 8 examples, 0 failures. That was an easy update and now we've got a well-named spec helper that can be easily re-used in other controller examples!

23.3 Spec'ing ApplicationController

Most of the time the behaviour you add to a controller is exposed directly through an action. However, there are times when you need to introduce behaviour which is applied to every controller and invoked indirectly. Perhaps you're adding an around filter which logs every incoming request processed by the app. Or maybe you're adding application wide error handling. In either case, specifying these behaviours one action at a time can be quite tedious. So let's explore a technique which works well to keep us moving forward swiftly and confidently.

We're going to add uniform error handling for AccessDenied exceptions in the app we've been working on. Create spec/controllers/application_controller_spec.rb

with the following contents:

Download rails_controllers/messages/14/spec/controllers/application_controller1_spec.rb

```
require File.dirname(__FILE__) + '/../spec_helper'

describe ApplicationController, "handling AccessDenied exceptions" do
  it "should redirect to the /401.html (access denied) page"
end
```

At this point we've got 1 example, 0 failures, 1 pending. To express what we want to happen let's add an example that simply calls an action:

Download rails_controllers/messages/14/spec/controllers/application_controller2_spec.rb

```
describe ApplicationController, "handling AccessDenied exceptions" do

  it "should redirect to the /401.html (access denied) page" do
    get :index
    response.should redirect_to('/401.html')
  end

end
```

This spec should fail with

```
No route matches {:action=>"index", :controller=>"application"}
```

In most controller specs we write examples for controllers used directly in the app. Here we are going to specify behaviour of every controller's superclass, ApplicationController, which isn't exposed to the app.

One common approach is to create a controller right in the spec. In this case we need an index action, so we'll add that to the controller, programming it raise the AccessDenied error that we're expecting in the example.

Download rails_controllers/messages/14/spec/controllers/application_controller3_spec.rb

```
describe ApplicationController, "handling AccessDenied exceptions" do
  class FooController < ApplicationController
    def index
      raise AccessDenied
    end
  end

  it "should redirect to the /401.html (access denied) page" do
    get :index
    response.should redirect_to('/401.html')
  end
end
```

Now it fails with uninitialized constant ApplicationController::AccessDenied. We can get past this by adding an AccessDenied exception. Add it in RAILS_ROOT/lib/access_denied.rb:

Download rails_controllers/messages/14/lib/access_denied.rb

```
class AccessDenied < StandardError
end
```

Now we're back to the earlier failure—No route matches {:action=>"index", :controller=>"application"}. It's trying to call index() on ApplicationController. We want to call it on FooController. We can use rspec-rails' controller_name() method to tell the examples to do just that:

Download rails_controllers/messages/14/spec/controllers/application_controller4_spec.rb

```
describe ApplicationController, "handling AccessDenied exceptions" do
  class FooController < ApplicationController
    def index
      raise AccessDenied
    end
  end
  controller_name 'foo'

  it "should redirect to the /401.html (access denied) page" do
    get :index
    response.should redirect_to('/401.html')
  end
end
```

No route matches {:action=>"index", :controller=>"foo"}. This is similar to the failure we got before, but now it is trying to hit the index action on the FooController. The failure message is correct. There is no route to the FooController as it just exists in our spec. We can update the routes for the sake of our spec to map routes for the FooController. Add the following before to the spec:

Download rails_controllers/messages/14/spec/controllers/application_controller5_spec.rb

```
describe ApplicationController, "handling AccessDenied exceptions" do
  class FooController < ApplicationController
    def index
      raise AccessDenied
    end
  end
  controller_name 'foo'

  before(:each) do
    ActionController::Routing::Routes.draw do |map|
      map.resources :foo
    end
```

```
    end

    after(:each) do
      ActionController::Routing::Routes.reload!
    end

    it "should redirect to the /401.html (access denied) page" do
      get :index
      response.should redirect_to('/401.html')
    end
  end
```

This fixed the spec—1 example, 0 failures—but we're not quite done yet. Drawing new routes has a side-effect that we don't want—it overwrites any previous routes drawn. While this doesn't affect this spec it will affect any controller specs that run after it, such as when you run multiple specs at the same time with rake spec. All you need to do is add a after(:each) block which reloads the routes:

Download rails_controllers/messages/14/spec/controllers/application_controller5_spec.rb

```
describe ApplicationController, "handling AccessDenied exceptions" do
  class FooController < ApplicationController
    def index
      raise AccessDenied
    end
  end
  controller_name 'foo'

  before(:each) do
    ActionController::Routing::Routes.draw do |map|
      map.resources :foo
    end
  end

  after(:each) do
    ActionController::Routing::Routes.reload!
  end

  it "should redirect to the /401.html (access denied) page" do
    get :index
    response.should redirect_to('/401.html')
  end
end
```

The spec should still be at 1 example, 0 failures. The techniques we applied here work great for specifying behaviour that is applied across the entire controller landscape. You will most likely not rely on these techniques that often, but when you find an appropriate situation you'll have a technique that can save time, effort, and still give you confi-

Joe Asks...

How can I spec file uploads?

From the controller's perspective an uploaded file is nothing more than another parameter in the params hash that gets passed through to a model. There's nothing interesting about the uploaded file to spec in a controller. And since uploading files involve integrating controllers, models, the database, and even the file system—we encourage you to rely on Cucumber and Webrat to provide that level of integration.

However, it is possible to utilize a controller spec to provide the necessary integration to spec file uploads. And while we don't encourage this it's a technique you should be aware of.

Rails' provides a ActionController::TestUploadedFile class which can be used to represent an uploaded file in the params hash of a controller spec. Here's a sample spec which does that:

```
describe UsersController, "POST create" do
  after do
    # if files are stored on the file system
    # be sure to clean them up
  end

  it "should be able to upload a user's avatar image" do
    image = fixture_path + "/test_avatar.png"
    file = ActionController::TestUploadedFile.new image, "image/png"
    post :create, :user => { :avatar => file }
    User.last.avatar.original_filename.should == "test_avatar.png"
  end
end
```

This spec would require that you have a test_avatar.png image in the RAILS_ROOT/spec/fixtures directory. It would take that file, upload it to the controller, and the controller would create and save a real User model.

dence that controllers are working as expected.

23.4 Sending Emails

Spec'ing a controller action that sends an email is a lot like spec'ing an action that creates a model. After all a mailer is just another collaborator that a controller interacts with. Mailers themselves are best spec'd

in isolation, which the (as yet) unwritten *chp.railsMiscStuff* will cover.

When you need to implement an action that will send email you should fall back to the same techniques we applied when implementing the create action on the MessagesController. It will lead to interaction-based examples like this:

```
describe UsersController, "POST create" do
  before(:each) do
    @user = mock_model(User)
    User.stub!(:new).and_return @user
    UserMailer.stub!(:deliver_confirmation)
  end

  it "should send a confirmation email to the user" do
    UserMailer.should_receive(:deliver_confirmation).with(@user)
    post :create
  end

  # ...
end
```

23.5 Custom Macros

As an application grows, we find ourselves having similar expectations on different controllers and actions. We apply before filters to many controller and actions, and we find ourselves repetitively typing the same expectations for several specs. When we notice these patterns in our specs, we can introduce custom macros and matchers to reduce duplication and produce more expressive examples.

should_require_login() Macro

Earlier in this chapter we introduced the application requirement that only authenticated users could create a message. Let's apply the same requirement to the new action on the MessagesController. Add the following examples to your messages_controller_spec.rb:

Download rails_controllers/messages/15/spec/controllers/messages_controller_spec.rb

```
describe MessagesController, "GET new" do

  describe "anonymous user" do
    it "should redirect to the login page" do
      get :new
      response.should redirect_to(login_path)
    end
```

Isolation and Integration Modes

Controller specs use a custom example group, which works in Isolation mode by default. This supports specifying controllers in isolation from the views that they render. By stubbing out the model layer as well, we can drive out controllers in complete isolation from views, models, and the database.

This keeps the controller specs lean, not having to manage a spiderweb of dependencies in the view or the model. It also provides quick fault isolation. You'll always know that a failing controller spec means that the controller is not behaving correctly.

The view templates do need to exist, but they can be empty, or broken, and the controller specs will pass as long as the controller is doing its job.

If you're more comfortable with the views being rendered, you can use the Integration mode, which allows the controller to render views. Just tell the example group to integrate views with the integrate_views() method:

```
describe MessagesController do
  integrate_views
  ...
```

In this mode, controller specs are like Rails functional tests—one set of examples for both controllers and views. The benefit of this approach is that you get wider coverage from each spec. Experienced Rails developers may find this an easier approach to begin with, however we encourage you to explore using the isolation mode and revel in its benefits.

```
  it "should not call the #new action" do
    controller.should_not_receive(:new)
    get :new
  end
end

describe "authenticated user" do
  before(:each) do
    login_as_user
    @message = mock_model(Message)
    Message.stub!(:new).and_return @message
  end
```

```
    it "should build a message" do
      Message.should_receive(:new)
      get :new
    end

    it "should assign @message" do
      get :new
      assigns[:message].should == @message
    end

    it "should render the new template" do
      get :new
      response.should render_template("new")
    end
  end

end
```

With these examples the spec have 13 examples, 0 failures. Take a minute to look over the entire spec. You'll notice that the anonymous user examples for both the create action and the new actions are very similar except for the action we process. Let's remove this duplication by pulling out the pattern to a should_require_login() macro.

Start by creating a macros.rb file in the spec/spec_helpers/controllers/ directory with the following contents:

Download rails_controllers/messages/15/spec/spec_helpers/controllers/macros.rb

```
module ControllerMacros

  def should_require_login
  end

end
```

Macros are normal methods which are executed at the same level as it() or before(). We can take advantage of this to wrap the creation of examples behind a simple method. This will become clear as we work this example. Let's focus on the anonymous user examples for the new action first. Copy and paste those examples, describe block etc., in our should_require_login() method:

Download rails_controllers/messages/15/spec/spec_helpers/controllers/macros2.rb

```
def should_require_login
  describe "anonymous user" do
    it "should redirect to the login page" do
      get :new
      response.should redirect_to(login_path)
    end
```

```
    it "should not call the #new action" do
      controller.should_not_receive(:new)
      get :new
    end
  end
end
```

Now add a call to the should_require_login() macro in the spec right above the anonymous user examples we just copied:

Download rails_controllers/messages/15/spec/contr . . . rs/messages_controller_refactor2_spec.rb

```
describe MessagesController, "GET new" do
  should_require_login

  describe "anonymous user" do
```

Running the spec should result in an error—undefined local variable or method 'should_require_login'. We need to configure rspec to make our macros accessible to controller specs. Update spec_helper.rb to have the config extend our ControllerMacros module for controllers:

Download rails_controllers/messages/15/spec/spec_helper.rb

```
Spec::Runner.configure do |config|
  config.include Webrat::Matchers, :type => :view
  config.include Webrat::HaveTagMatcher, :type => :view
  config.include ViewMatchers, :type => :view
  config.include ControllerHelpers, :type => :controller
  config.extend ControllerMacros, :type => :controller
end
```

Here we used config.extend rather than config.include. We did this because *include* is used to include methods that can be used within an it() example whereas *extend* is used to make methods available when defining spec. See Chapter 15, *Extending RSpec*, on page 177 chapter for more information on extending rspec.

The spec should now have 15 examples, 0 failures. The should_require_login() macro is working! Go ahead an remove the original anonymous user examples in the spec and run the spec. You should be back to 13 examples, 0 failures.

Currently the should_require_login() macro is hard coded to make a GET request for the new action. We'll need to make it more flexible so we can use it for other actions as well, such as the create action. Let's parameterize the macro (e.g. should_require_login(:get, :new)) so it will be able to work for any request method and any action. Go ahead and

change the macro to accept two parameters—the request method and the action to call:

Download rails_controllers/messages/15/spec/spec_helpers/controllers/macros3.rb

```
def should_require_login(request_method, action)
```

Next, update the macro body to rely on those parameters. Let's replace the hard-coded references to :new and calls to get with the parameters that are being passed in. Passing the request method in as a parameter provides a challenge for us though. How can we invoke it as a method call? Fortunately, Ruby solves this problem for us. We can rely on Ruby's send() method to invoke it:

Download rails_controllers/messages/15/spec/spec_helpers/controllers/macros4.rb

```
def should_require_login(request_method, action)
  describe "anonymous user" do
    it "should redirect to the login page" do
      send request_method, action
      response.should redirect_to(login_path)
    end

    it "should not call the #{action} action" do
      controller.should_not_receive(action)
      send request_method, action
    end
  end
end
```

Running the spec should fail—`should_require_login': wrong number of arguments (0 for 2). Update the spec to pass in the parameters we just added to our macro:

Download rails_controllers/messages/15/spec/contr . . . rs/messages_controller_refactor3_spec.rb

```
should_require_login :get, :new
```

The spec is now back to 13 examples, 0 failures. Now let's update the POST create example to use the macro. Add a call to the macro at the top of its describe:

Download rails_controllers/messages/15/spec/contr . . . rs/messages_controller_refactor4_spec.rb

```
should_require_login :post, :create
```

The spec is still green with 15 examples, 0 failures. Go ahead and remove the anonymous user describe block and run the spec—13 examples, 0 failures.

That's it! You now have a controller macro under your belt. And you can apply what we did here in any situation where you discover a pattern

emerging from the specs.

23.6 What We Just Learned

- Controllers coordinate the interaction between the user and the application and should know *what* to do, but not *how* to do it.
- Specifying the desired interaction helps us to discover objects with well named methods to encapsulate the real work.
- Controller specs use a custom example group provided by the rspec-rails library.
- Controller specs live in a directory tree parallel to the controllers themselves, and follow a naming convention of spec/controllers/my_controller_spec.rb for app/controllers/my_controller.rb.
- Use the redirect_to() matcher to confirm redirects.
- Use the render_template() matcher to confirm the template being rendered.
- Use the assigns() method to confirm the instance variables assigned for the view.
- Use the flash() method to confirm the flash messages stored for the view.
- Use mock_model() and stub_model() to isolate controller specs from the database and underlying business logic of your models.
- Extract spec helpers and custom macros to help keep your controller specs DRY.

Chapter 24

Rails Models

Coming soon ...

Appendix A

RubySpec

Coming soon ...

Appendix B

RSpec's Built-In Expectations

Here is a summary of all of the expectations that are supported directly by RSpec.

Equality

Expression	Passes if ...
actual.should equal(expected)	actual.equal?(expected)
actual.should eql(expected)	actual.eql?(expected)
actual.should == expected	actual == expected
actual.should === expected	actual === expected

Expression	Passes unless ...
actual.should_not equal(expected)	actual.equal?(expected)
actual.should_not eql(expected)	actual.eql?(expected)
actual.should_not == expected	actual == expected
actual.should_not === expected	actual === expected

Arbitrary Predicates

Expression	Passes if ...
actual.should be_[predicate]	actual.predicate?
actual.should be_a_[predicate]	actual.predicate?
actual.should be_an_[predicate]	actual.predicate?

Expression	Passes if ...
actual.should be_[predicate](*args)	actual.predicate?(*args)
actual.should be_a_[predicate](*args)	actual.predicate?(*args)
actual.should be_an_[predicate](*args)	actual.predicate?(*args)

Expression	Passes unless ...
actual.should_not be_[predicate]	actual.predicate?
actual.should_not be_a_[predicate]	actual.predicate?
actual.should_not be_an_[predicate]	actual.predicate?

Expression	Passes unless ...
actual.should_not be_[predicate](*args)	actual.predicate?(*args)
actual.should_not be_a_[predicate](*args)	actual.predicate?(*args)
actual.should_not be_an_[predicate](*args)	actual.predicate?(*args)

Regular Expressions

Expression	Passes if ...
actual.should match(expected)	actual.match?(expected)
actual.should =~ expected	actual =~ expected

Expression	Passes unless ...
actual.should_not match(expected)	actual.match?(expected)
actual.should_not =~ expected	actual =~ expected

Comparisons

Expression	Passes if ...
actual.should be < expected	actual < expected
actual.should be <= expected	actual <= expected
actual.should be >= expected	actual >= expected
actual.should be > expected	actual > expected

Collections

Expression	Passes if ...
actual.should include(expected)	actual.include?(expected)
actual.should have(n).items	actual.items.length == n or actual.items.size == n
actual.should have_exactly(n).items	actual.items.length == n or actual.items.size == n
actual.should have_at_least(n).items	actual.items.length >= n or actual.items.size >= n
actual.should have_at_most(n).items	actual.items.length <= n or actual.items.size <= n

Expression	Passes unless ...
actual.should_not include(expected)	actual.include?(expected)
actual.should_not have(n).items	actual.items.length == n or actual.items.size == n
actual.should_not have_exactly(n).items	actual.items.length == n or actual.items.size == n

Errors

Expression	Passes if ...
proc.should raise_error	proc raises any error
proc.should raise_error(type)	raises specified type of error
proc.should raise_error(message)	raises error with specified message
proc.should raise_error(type, message)	raises specified type of error with specified message

Expression	Passes unless ...
proc.should_not raise_error	proc raises any error
proc.should_not raise_error(type)	raises specified type of error
proc.should_not raise_error(message)	raises error with specified message
proc.should_not raise_error(type, message)	raises specified type of error with specified message

Symbols

Expression	Passes if ...
proc.should throw_symbol	proc throws any symbol
proc.should throw_symbol(type)	proc throws specified symbol

Expression	Passes unless ...
proc.should_not throw_symbol	proc throws any symbol
proc.should_not throw_symbol(type)	proc throws specified symbol

Floating Point Comparisons

Expression	Passes if ...
actual.should be_close(expected, delta)	actual < (expected + delta) or > (expected - delta)

Expression	Passes unless ...
actual.should_not be_close(expected, delta)	actual < (expected + delta) or > (expected - delta)

Duck Typing

Expression	Passes if ...
actual.should respond_to(*messages)	messages.each { |m| m.respond_to?(m) }

Expression	Passes unless ...
actual.should_not respond_to(*messages)	messages.each { |m| m.respond_to?(m) }

When All Else Fails...

Expression	Passes if ...
actual.should satisfy { |actual| block }	the block returns true

Expression	Passes unless ...
actual.should_not satisfy { |actual| block }	the block returns true

Appendix C

Bibliography

[Bec02] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley, Reading, MA, 2002.

[Coh04] Mike Cohn. *User Stories Applied: For Agile Software Development*. Boston, MA, Addison-Wesley Professional, 2004.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley Longman, Reading, MA, 1999.

[HT00] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, Reading, MA, 2000.

[Mar02] Robert C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, Englewood Cliffs, NJ, 2002.

[MRB97] Robert C. Martin, Dirk Riehle, and Frank Buschmann. *Pattern Languages of Program Design 3*. Addison-Wesley Professional, Boston, MA, 1997.

[Rai04] J. B. Rainsberger. *JUnit Recipes : Practical Methods for Programmer Testing*. Manning Publications Co., Greenwich, CT, 2004.

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

The RSpec Book's Home Page
http://pragprog.com/titles/achbd
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/achbd.

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)