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

Prepared exclusively for aidy lewis

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**

Prepared exclusively for aidy lewis

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

Prepared exclusively for aidy lewis

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

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

Copyright © 2009 The Pragmatic Programmers LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

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.

B1.0 printing, January 28, 2009

Version: 2009-1-29

Prepared exclusively for aidy lewis

Contents

I Getting Started with RSpec and Cucumber 8

1 Introduction 9
1.1 Test Driven Development: Where it All Started 9
1.2 Behaviour Driven Development: The Next Step 10
1.3 RSpec 11
1.4 Cucumber 12
1.5 The BDD Cycle 13

2 Describing Application Behaviour with Cucumber 16
2.1 Selecting Stories for the First Iteration 17
2.2 Deriving Features from Stories 18
2.3 Automating Acceptance Criteria 20
2.4 Steps and Step Definitions 22
2.5 What We Just Did 29

3 Working from the Outside-In with RSpec 31
3.1 Red: Start With a Failing Code Example 34
3.2 Green: Get the Example To Pass 35
3.3 Refactor to Remove Duplication 43
3.4 What We Just Did 46

4 Outlining Scenarios 48

5 Random Expectations 49

II Behaviour Driven Development 50

6 Writing Software that Matters 51

Prepared exclusively for aidy lewis

7 Mock Objects **52**

III RSpec **53**

8 Code Examples **54**
8.1 Describe It! 55
8.2 Pending Examples 59
8.3 Before and After 62
8.4 Helper Methods 65
8.5 Shared Examples 68
8.6 Nested Example Groups 70

9 Expectations **74**
9.1 should and should_not 76
9.2 Built-In Matchers 77
9.3 Arbitrary Predicate Matchers 86
9.4 Explicit Predicate Matchers 87
9.5 Have Whatever You Like 88
9.6 Operator Expressions 92
9.7 Generated Descriptions 93
9.8 Subject-ivity 94

10 Mocking in RSpec **97**

11 RSpec and Test::Unit **98**
11.1 Running Test::Unit tests with the RSpec runner 99
11.2 Refactoring Test::Unit Tests to RSpec Code Examples . 103
11.3 What We Just Did 108

12 Tools And Integration **109**
12.1 The spec Command 109
12.2 TextMate 116
12.3 Autotest 117
12.4 Heckle 118
12.5 Rake 119
12.6 RCov 120

13 Extending RSpec **122**

Prepared exclusively for aidy lewis

14 Cucumber 123

IV Behaviour Driven Rails 124

15 BDD in Rails 125
- 15.1 Traditional Rails Development 126
- 15.2 Outside-In Rails Development 127
- 15.3 Setting up a Rails project 130
- 15.4 What We Just Learned 132

16 Cucumber with Rails 133
- 16.1 Working with Cucumber in Rails 133
- 16.2 Step Definition Styles 136
- 16.3 Direct Model Access 137

17 Webrat 147

18 Rails Views 148

19 Rails Helpers 149

20 Rails Controllers 150

21 Rails Models 151

V RSpec in the Wild 152

22 RSpec and the RubySpec Project 153

A RSpec's Built-In Expectations 154

B Bibliography 158

Index 159

Prepared exclusively for aidy lewis
this copy is (B1.0 printing, January 28, 2009)

Part I

Getting Started with RSpec and Cucumber

Prepared exclusively for aidy lewis

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 perpsective, 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]

Prepared exclusively for aidy lewis

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 becomes delivery of high quality code to testers, and those testers are responsible for testing practices

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 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 Steve'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.empty?.should be_true
    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.empty?.should be_true should be self-explanatory. Just read it out loud: movie_list.empty? should be true.

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:

```
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 def-*

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 libary, 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 HellesÃÿy 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 fiancÃl'e, 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.

initions 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

3. Although Cucumber is focused on high level behaviour and RSpec on more granular aspects of behaviour, each can be used for either purpose.

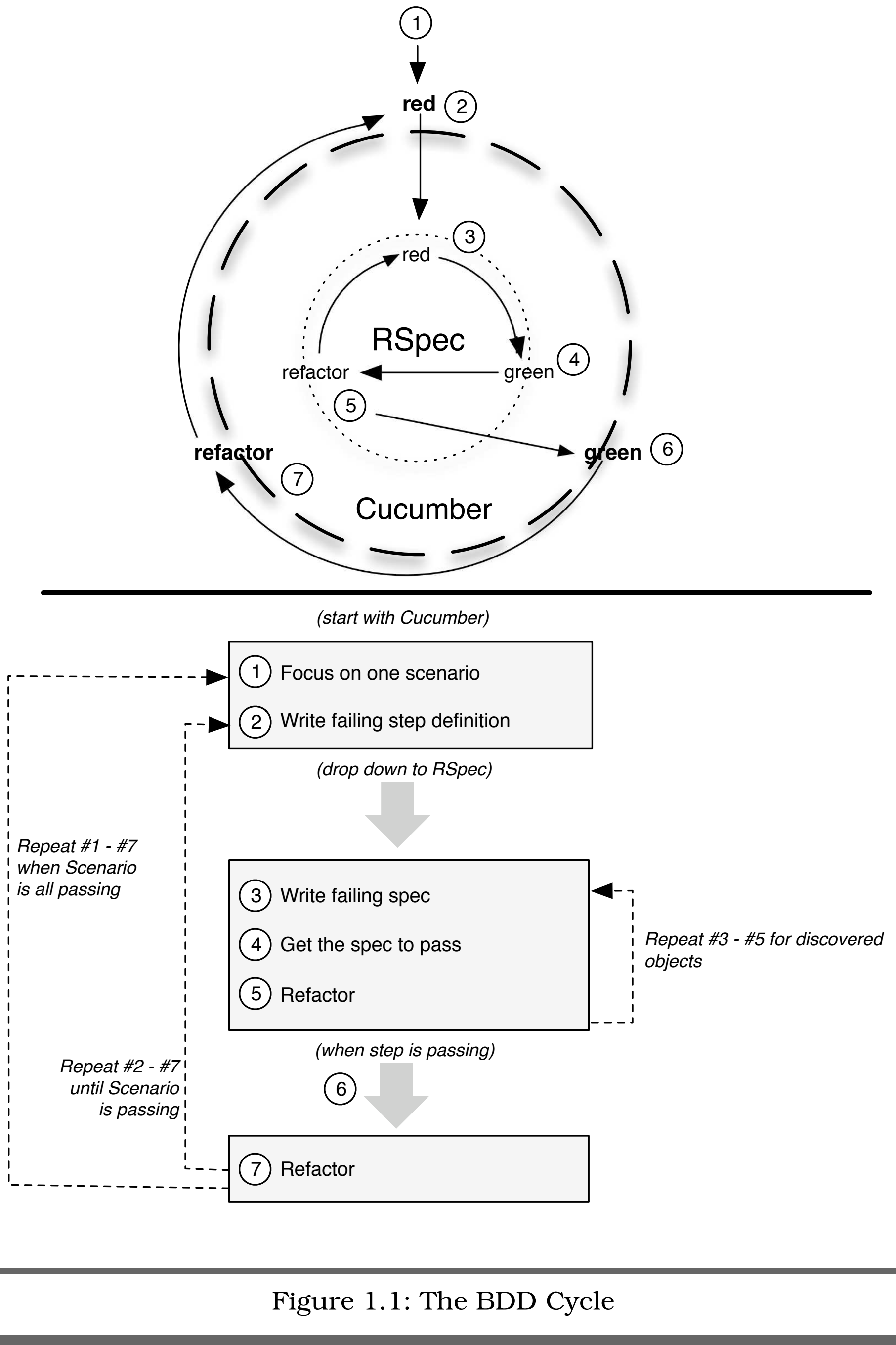


Figure 1.1: The BDD Cycle

like Cucumber, we'll actually have two concentric red/green/refactor cycles, as depicted in Figure 1.1, on the preceding 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 to learn more of the detail and philosophy behind each feature.

So now its 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.

Prepared exclusively for aidy lewis

Chapter 2

Describing Application Behaviour 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.

Prepared exclusively for aidy lewis

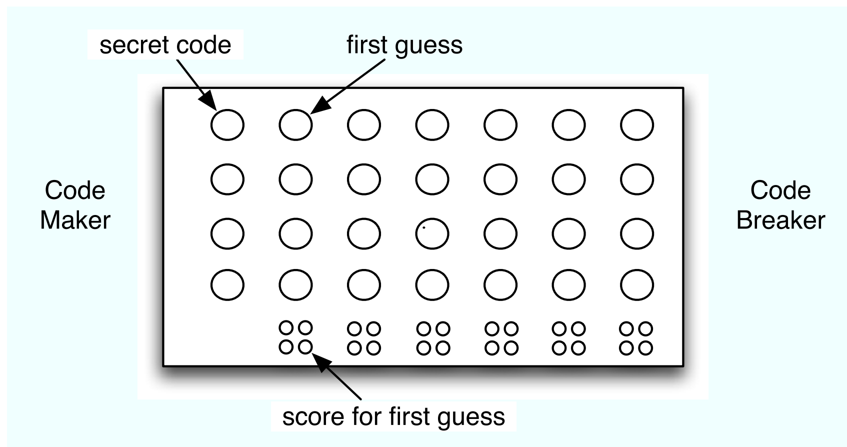


Figure 2.1: Mastermind

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 features, spec, and lib 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.

By 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

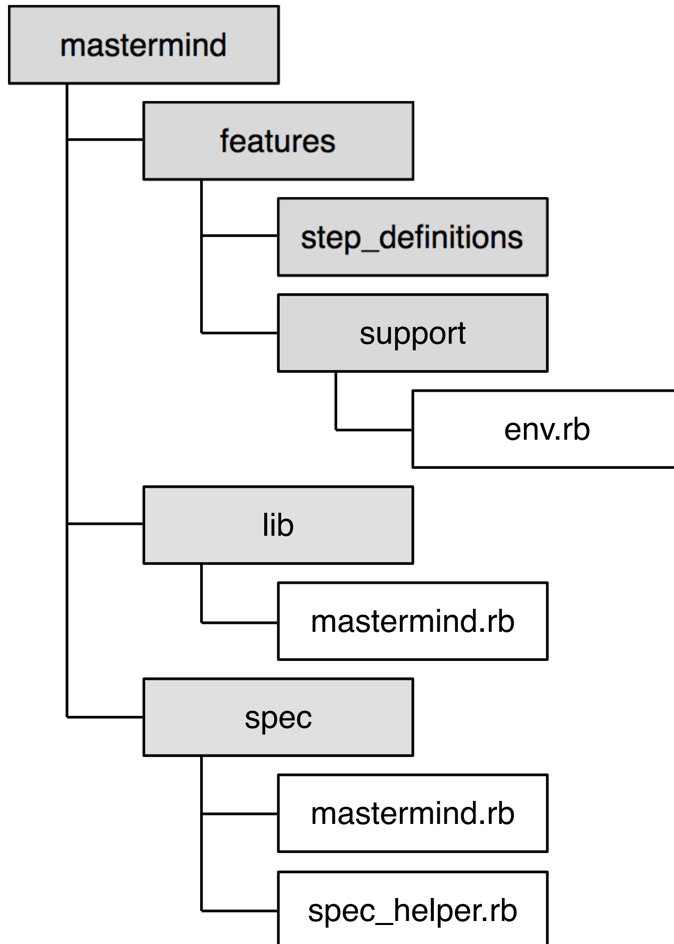


Figure 2.2: Project Structure

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 mastermind/01/01/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 mastermind/01/01/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
```

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.

```
  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 steps pending (4 with no step definition)

You can use these snippets to implement pending steps:

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

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

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

Then /^the game should say "Enter guess:"$/ do
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 enter 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
1 step passed
3 steps pending (3 with no step definition)

You can use these snippets to implement pending steps:

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

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

Then /^the game should say "Enter guess:"$/ do
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 mastermind/01/02/features/step_definitions/mastermind.rb

```
When /^I start a new game$/ do
  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 mastermind/01/02/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 mastermind/01/03/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 mastermind/01/04/features/step_definitions/mastermind.rb

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

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 you 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.

Prepared exclusively for aidy lewis

```
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 sets an expectation that the value of the @message variable should equal (using Ruby's ==() method) the message from the step in the scenario. This is RSpec's way of setting expectations about equality.[2]

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 mastermind/01/05/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 mastermind/01/05/bin/mastermind

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

game = Mastermind::Game.new
```

2. We'll be talking about all the different expectations you can set with RSpec later in Chapter 9, *Expectations*, on page 74.

Prepared exclusively for aidy lewis

```
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 mastermind/01/06/bin/mastermind

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

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

Much better! Go ahead and copy that code into bin/mastermind.bat if you're on Windows. Otherwise, use bin/mastermind (and don't forget to chmod 755 bin/mastermind so you'll be able to execute it later).

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 mastermind/01/06/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)
      ./01/06/features/step_definitions/mastermind.rb:7:in \
                                `When /^I start a new game$/'
      01/06/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 step passed
1 step failed
2 steps skipped
```

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.[3]

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

3. Cucumber can do this because it is built on top of *Treetop*, a library for building grammars in Ruby.

```
Feature: code-breaker starts game \
                        # 0/1/06/features/codebreaker_starts_game.feature
  As a code-breaker
  I want to start a game
  So that I can break the code
  Scenario: start game \
                        # 0/1/06/features/codebreaker_starts_game.feature:8
    Given I am not yet playing \
                        # 0/1/06/features/step_definitions/mastermind.rb:1
    When I start a new game \
                        # 0/1/06/features/step_definitions/mastermind.rb:5
      uninitialized constant Mastermind (NameError)
      ./01/06/features/step_definitions/mastermind.rb:7:in \
                        `When /^I start a new game$/'
      01/06/features/codebreaker_starts_game.feature:10:in \
                        `When I start a new game'
    Then the game should say "Welcome to Mastermind!" \
                        # 0/1/06/features/step_definitions/mastermind.rb:11
    And the game should say "Enter guess:" \
                        # 0/1/06/features/step_definitions/mastermind.rb:11


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

Now we see file and line numbers for the feature and 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 to the first step in the concentric cycles described at the beginning of this chapter: 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

Working from the Outside-In 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 mastermind/01/06/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.

Prepared exclusively for aidy lewis

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
```

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 mastermind/01/07/lib/mastermind/game.rb

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

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

Download mastermind/01/07/lib/mastermind.rb

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

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

Download mastermind/01/07/spec/spec_helper.rb

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

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

Download mastermind/01/07/spec/mastermind/game_spec.rb

```ruby
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
```

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

```
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 mastermind/01/07/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 \
                              # 0/1/07/features/codebreaker_starts_game.feature
  As a code-breaker
  I want to start a game
  So that I can break the code
  Scenario: start game \
                              # 0/1/07/features/codebreaker_starts_game.feature:8
    Given I am not yet playing \
                              # 0/1/07/features/step_definitions/mastermind.rb:1
    When I start a new game \
                              # 0/1/07/features/step_definitions/mastermind.rb:5
      wrong number of arguments (1 for 0) (ArgumentError)
      ./01/07/features/step_definitions/mastermind.rb:7:in \
                              `initialize'
      ./01/07/features/step_definitions/mastermind.rb:7:in \
                              `new'
      ./01/07/features/step_definitions/mastermind.rb:7:in \
                              `When /^I start a new game$/'
      01/07/features/codebreaker_starts_game.feature:10:in \
                              `When I start a new game'
    Then the game should say "Welcome to Mastermind!" \
                              # 0/1/07/features/step_definitions/mastermind.rb:11
    And the game should say "Enter guess:" \
                              # 0/1/07/features/step_definitions/mastermind.rb:11


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

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 a 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 mastermind/01/08/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 the (as yet) unwritten *chp.mockFramework*, 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 mastermind/01/09/spec/mastermind/game_spec.rb

```ruby
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
```

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, and run the file again:

Download mastermind/01/10/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
```

```
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 mastermind/01/11/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 mastermind/01/12/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 mastermind/01/13/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:

```
ruby 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 –colour 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 \
                            # 0/1/13/features/codebreaker_starts_game.feature
  As a code-breaker
  I want to start a game
  So that I can break the code
  Scenario: start game      # 0/1/13/features/codebreaker_starts_game.feature:8
    Given I am not yet playing \
                            # 0/1/13/features/step_definitions/mastermind.rb:1
    When I start a new game # 0/1/13/features/step_definitions/mastermind.rb:5
    Then the game should say "Welcome to Mastermind!" \
                            # 0/1/13/features/step_definitions/mastermind.rb:11
      undefined method `include' for #<Object:0x520814> (NoMethodError)
      ./01/13/features/step_definitions/mastermind.rb:12:in \
                            `Then /^the game should say "(.*)"$/'
      01/13/features/codebreaker_starts_game.feature:11:in \
```

```
                                `Then the game should say "Welcome to Mastermind!"'
    And the game should say "Enter guess:" \
                                # 0/1/13/features/step_definitions/mastermind.rb:11


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

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

Download mastermind/01/14/features/support/env.rb

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

Run the feature again, and:

```
Feature: code-breaker starts game \
                    # 0/1/14/features/codebreaker_starts_game.feature
  As a code-breaker
  I want to start a game
  So that I can break the code
  Scenario: start game    # 0/1/14/features/codebreaker_starts_game.feature:8
    Given I am not yet playing \
                    # 0/1/14/features/step_definitions/mastermind.rb:1
    When I start a new game \
                    # 0/1/14/features/step_definitions/mastermind.rb:5
    Then the game should say "Welcome to Mastermind!" \
                    # 0/1/14/features/step_definitions/mastermind.rb:11
    And the game should say "Enter guess:" \
                    # 0/1/14/features/step_definitions/mastermind.rb:11
      expected ["Welcome to Mastermind!"] to include \
                    "Enter guess:" (Spec::Expectations::ExpectationNotMetError)
      ./01/14/features/step_definitions/mastermind.rb:12:in
                    `And /^the game should say "(.*)"$/'
      01/14/features/codebreaker_starts_game.feature:12:in \
                    `And the game should say "Enter guess:"'


1 scenario
3 steps passed
1 step failed
```

Progress! Now one of the two *Thens* are passing, so it looks like we're about half way done with this feature. Actually we're quite a bit more than half way 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 mastermind/01/14/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 mastermind/01/16/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 mastermind/01/17/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 [GHJV95], and is supported by RSpec's mock framework with a the as_null_object() method:

Download mastermind/01/18/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 behavior 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 behavior 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* 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 behavior.

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

Download mastermind/01/19/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 mastermind/01/20/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
```


Prepared exclusively for aidy lewis
this copy is (B1.0 printing, January 28, 2009)

```
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 mastermind/01/21/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 \
                           # 0/1/21/features/codebreaker_starts_game.feature
  As a code-breaker
  I want to start a game
  So that I can break the code
  Scenario: start game \
                           # 0/1/21/features/codebreaker_starts_game.feature:8
    Given I am not yet playing \
                           # 0/1/21/features/step_definitions/mastermind.rb:1
    When I start a new game \
                           # 0/1/21/features/step_definitions/mastermind.rb:5
    Then the game should say "Welcome to Mastermind!" \
                           # 0/1/21/features/step_definitions/mastermind.rb:11
    And the game should say "Enter guess:" \
                           # 0/1/21/features/step_definitions/mastermind.rb:11
```

```
1 scenario
4 steps passed
```

And viola! 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 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, and expose you to some really cool features in both Cucumber and RSpec. So take a few minutes break, drink up that brain juice, and meet me at the top of the next chapter.

Prepared exclusively for aidy lewis

Chapter 4

Outlining Scenarios

Coming soon ...

Prepared exclusively for aidy lewis

Chapter 5

Random Expectations

Coming soon ...

Prepared exclusively for aidy lewis

Part II

Behaviour Driven Development

Prepared exclusively for aidy lewis

Chapter 6

Writing Software that Matters

Coming soon …

Prepared exclusively for aidy lewis

Chapter 7

Mock Objects

Coming soon ...

Prepared exclusively for aidy lewis

Part III

RSpec

Prepared exclusively for aidy lewis

Chapter 8

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:

- The *subject code* is the code whose behaviour we are specifying with RSpec.
- An *expectation* is an expression of how the subject code is expected to behave. You'll read about state based expectations in Chapter 9, *Expectations*, on page 74, and interaction expectations in the (as yet) unwritten *chp.mockFramework*.
- A *code example* is executable documentation 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.
- An *example group* is a group of code examples.
- A *spec*, or *spec file*, is 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

Prepared exclusively for aidy lewis

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.

code before and after each example, and even share examples across groups.

8.1 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() 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.

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.

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, 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
```

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.

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.[1] 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:

```
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

1. As you'll see later in the (as yet) unwritten *chp.extendingRspec*, you can coerce the describe() method to return your own custom ExampleGroup subclass.

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.

8.2 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!

8.3 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.

Prepared exclusively for aidy lewis
this copy is (B1.0 printing, January 28, 2009)

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 the other. 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

before(: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.

8.4 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 consisent 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 the (as yet) unwritten *sec.globalConfiguration* for more information):

```
Spec::Runner.configure do |config|
  config.include(UserExampleHelperMethods)
end
```

So now that we can share helper methods across example groups, how about sharing *examples*?

8.5 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 grouip 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 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 the (as yet) unwritten *chp.extendingRspec*.

8.6 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, etc declared in the 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 before

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 9

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*.

Prepared exclusively for aidy lewis

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 to help 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.

9.1 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 it's name tells you: 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 9.1, 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

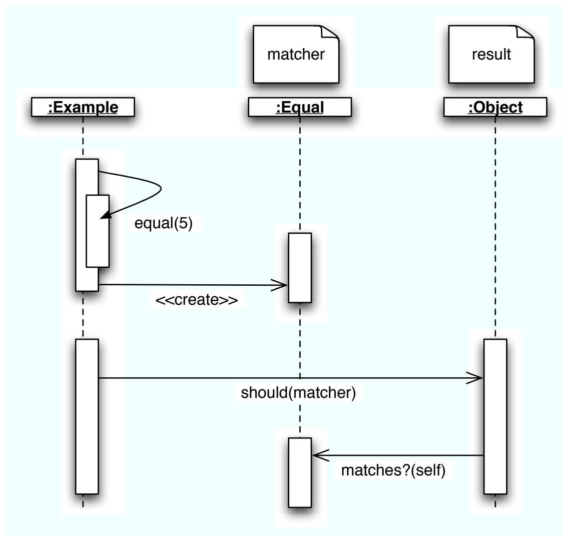


Figure 9.1: Should/Matcher Interaction Diagram

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 9.6, *Operator Expressions*, on page 92. Right now, let's take a closer look at RSpec's built in matchers.

9.2 Built-In Matchers

RSpec ships with several built-in matchers with obvious names that you can use in your examples to access them. 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 let's 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]
{'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 9.5, *Have Whatever You Like*, on page 88. 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 principal 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 this 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 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 specifics 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_to_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_to_hard, true)
}

# or ...

lambda {
  team.working_too_hard.should throw_symbol(:working_to_hard, false)
}
```

9.3 Arbitrary 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").

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, and so RSpec includes a means of assigning your own Explicit Predicate Matchers.

9.4 Explicit Predicate Matchers

There are going to be times when it doesn't make sense to prefix the predicate with "be_". If our SunsetMatch application matches elderly singles for potential sunset-years romance, we're going to have some users who are interested in finding candidates who can drive at night.

```
candidate.should be_can_drive_at_night
```

Eeeewwwww. What we really want to say is something more like:

```
candidate.should be_able_to_drive_at_night
```

Or, for short:

```
candidate.should drive_at_night
```

RSpec helps us say exactly what we want to say in this case with a simple means of configuring Custom Predicate Matchers.

```
describe "sunset match examples" do
  predicate_matchers[:drive_at_night] = :can_drive_at_night?
  it "should be able to drive at night" do
    candidate.should drive_at_night
  end
end
```

We can also configure these so they are available throughout your examples using Spec::Runner.configure:

```
Spec::Runner.configure do |config|
  config.predicate_matchers[:drive_at_night] = :can_drive_at_night?
end
```

See the (as yet) unwritten *chp.extendingRspec* for more info on how and where to use Spec::Runner.configure.

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 is not in the object itself, but in an object that it owns?

9.5 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 let's 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 should 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.

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 9.2, on the following page 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.

Prepared exclusively for aidy lewis
this copy is (B1.0 printing, January 28, 2009)

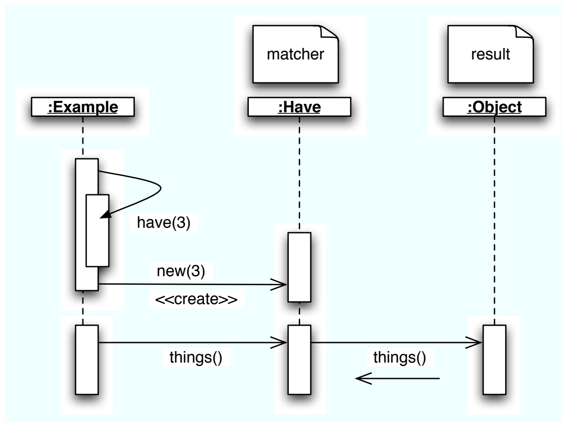


Figure 9.2: Have Matcher Sequence

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.

9.6 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.

9.7 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
```


Prepared exclusively for aidy lewis
this copy is (B1.0 printing, January 28, 2009)

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.

9.8 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 even 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 in 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
- Arbitrary predicate matchers
- Explicit predicate matchers
- Operator expressions
- Generated descriptions
- Using the implicit subject()
- Declaring an explicit 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 the (as yet) unwritten *chp.extendingRspec*, along with a number of other techniques for extending RSpec and tuning its DSL towards your specific projects.

In the mean time, 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 10

Mocking in RSpec

Coming soon ...

Prepared exclusively for aidy lewis

Chapter 11

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 straight forward, 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`

Prepared exclusively for aidy lewis

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.

11.1 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 tranforming 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 -rtest/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's edge, 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 -rtest/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.

11.2 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 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') # <label id="code.lone_describe"
     -  
     -    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() 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 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 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 the setup() and teardown() with before() and after():

Download testunit/test/person_spec_with_before_and_after.rb

```ruby
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 converstion, 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.

11.3 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 12

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.

12.1 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'
```

Prepared exclusively for aidy lewis

```
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 in them 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.

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.

You probably don't want to look at the HTML in a console, so you should 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 colour \
                      --format nested \
                      --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 12.1, 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.

```
RSpec Results                         30 examples, 20 failures, 1 pending
                                           Finished in 2.081589 seconds
Running specs with --diff

should print diff of different strings

expected: "RSpec is a\nbehaviour driven development\nframework for Ruby\n",
     got: "RSpec is a\nbehavior driven development\nframework for Ruby\n" (using ==)
Diff:
@@ -1,4 +1,4 @@
 RSpec is a
-behavior driven development
+behaviour driven development
 framework for Ruby

./failing_examples/diffing_spec.rb:13:

11 framework for Ruby
12 EOF
13     usa.should == uk
14   end

should print diff of different objects' pretty representation
```

Figure 12.1: HTML Report

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.

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, dots for passing examples are printed in green, F's for failing examples are red, *'s for pending examples are yellow. Also, 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:

```
--colour
--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
```

12.2 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 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.

12.3 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 Application Behaviour with Cucumber*, on page 16. 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.

Prepared exclusively for aidy lewis

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, it's code examples in spec/models/account_spec.rb will be run automatically.

12.4 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. Line 9 tells us that at least one code example failed after modifying the code in each of those four ways.

You can also 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
```

As of version 1.4.1, released back in 2006, 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.

12.5 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|
  t.spec_opts = ["--color", "--format", "specdoc"]
end
```

Check the rdoc for Spec::Rake::SpecTask to see the full list of configuration options.

12.6 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

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.

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 13

Extending RSpec

Coming soon ...

Prepared exclusively for aidy lewis

Chapter 14

Cucumber

Coming soon ...

Prepared exclusively for aidy lewis

Part IV

Behaviour Driven Rails

Prepared exclusively for aidy lewis

Chapter 15

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

Prepared exclusively for aidy lewis

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.

15.1 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 due. At this juncture, you might hear your conscience telling you to "do the simplest thing."

The Illusion of "simple" with Inside-Out

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 assumptions of what think you'll 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.

15.2 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 15.1, on the next page is the same cycle depicted in Figure 1.1, on page 14, 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 129).
- 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,

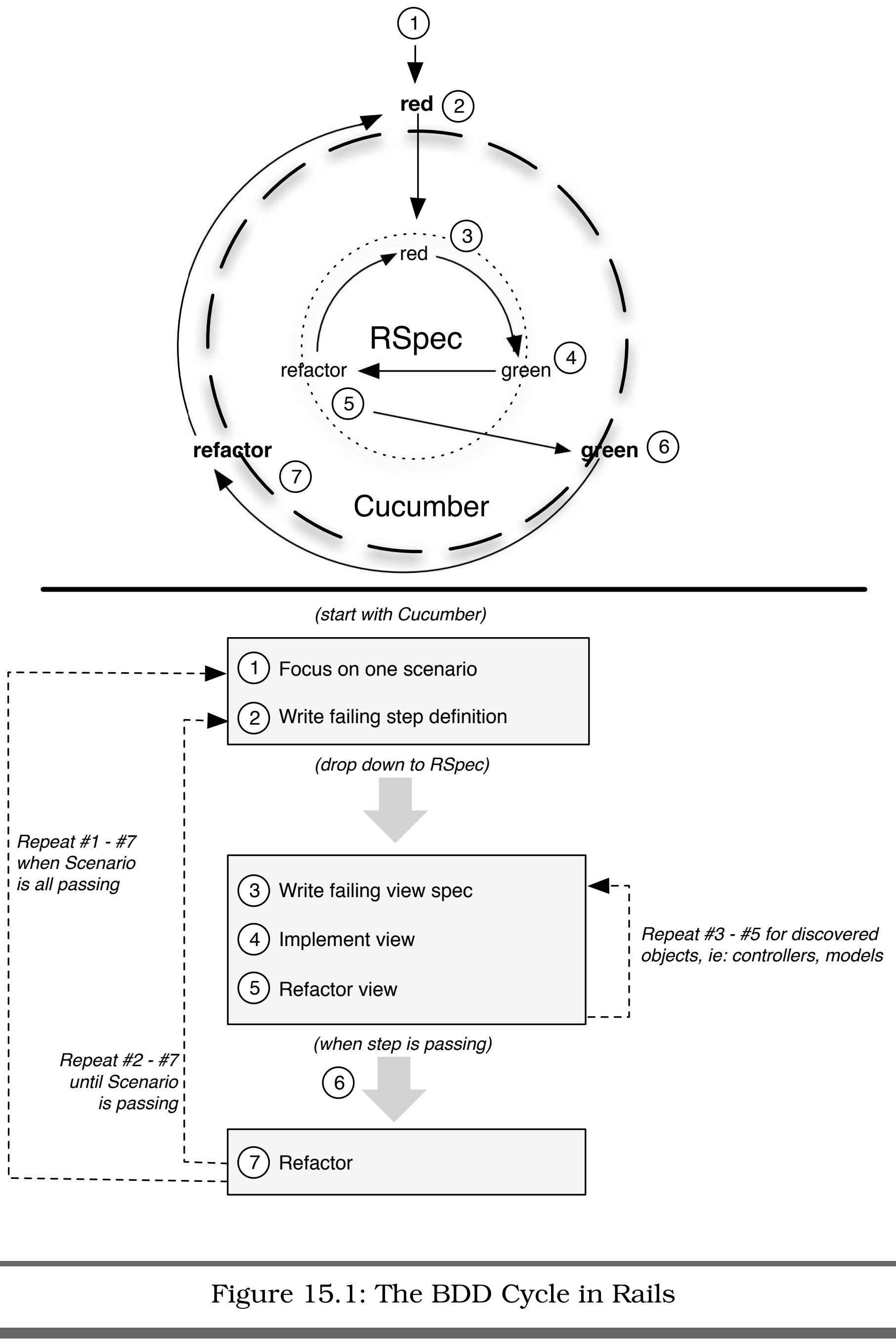


Figure 15.1: The BDD Cycle in Rails

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 the 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 onto the next unimplemented step and continue working outside-in. When a scenario is done move onto the next scenario or find the nearest customer and have them validate that its working as it should—then move onto 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 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.

15.3 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—it's included 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

Rails has built-in gem management which works well for gems that the application always requires. However, it doesn't work very well for gems that are only loaded in certain environments. No one needs or wants RSpec, Cucumber, rspec-rails, or Webrat to be loaded in development and production environments.

We do not recommend this installation method. However, it's here for completeness. If you decide to try this know that you may run into issues. If you do it may be best to revert to one of the previously mentioned installation methods.

In order to take advantage of Rails' built-in gem management, you'll need to setup your test environment to load the gems. Add the following lines to config/environments/test.rb:

```
config.gem 'rspec-rails', :lib => 'spec/rails'
config.gem 'cucumber'
config.gem 'webrat'
```

After saving this file you would ideally be able to perform the following commands:

```
# see what gems are required in the test environment
RAILS_ENV=test rake gems

# install required gems to your system
```

```
RAILS_ENV=test rake gems:install

# unpack required gems from your system to the app's vendor/gems
RAILS_ENV=test rake gems:unpack

# unpack required dependencies for your app's gems
RAILS_ENV=test rake gems:unpack:dependencies
```

Unfortunately, Rails gem management has been plagued with issues. When I executed these rake tasks, only gems:install and gems:unpack worked. As mentioned previously, we do not encourage or recommend this for Cucumber, RSpec, rspec-rails, or Webrat.

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
```

Now your project is ready for outside-in development.

15.4 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.

Prepared exclusively for aidy lewis

this copy is (B1.0 printing, January 28, 2009)

Chapter 16

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.

16.1 Working with Cucumber in Rails

Cucumber scenarios serve as a high level description of a Rails application's behavior 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 IntegrationTests, they'll give you tremendous confidence to refactor your code and evolve the application code in response to changing requirements. In addition, they'll document the system's behavior and your progress in implementing it by connecting those requirements directly to Ruby code.

As we saw in Chapter 15, *BDD in Rails*, on page 125, installing Cucumber into a Rails app is simple. The last step of the installation process is to run the generator included in Cucumber.

Prepared exclusively for aidy lewis

```
$ ./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 the (as yet) unwritten *chp.webrat*.
- 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 16.1, 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.

Prepared exclusively for aidy lewis
this copy is (B1.0 printing, January 28, 2009)

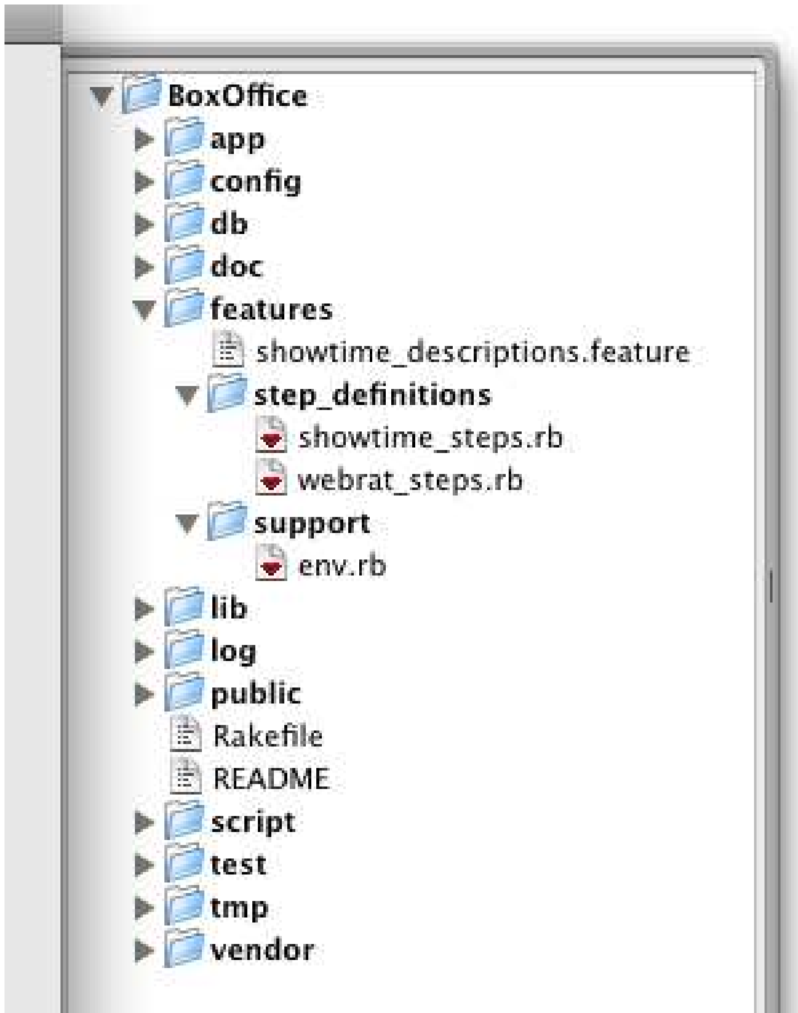


Figure 16.1: Rails Project Tree with Cucumber Features

16.2 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 behavior 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 it's behavior:

- *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 it's support for piggy-backing on Selenium. This style is fully integrated but can be slow to run and cumbersome to maintain.

Prepared exclusively for aidy lewis
this copy is (B1.0 printing, January 28, 2009)

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 16.2, 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 valued 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 the (as yet) unwritten *chp.webrat*.

16.3 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

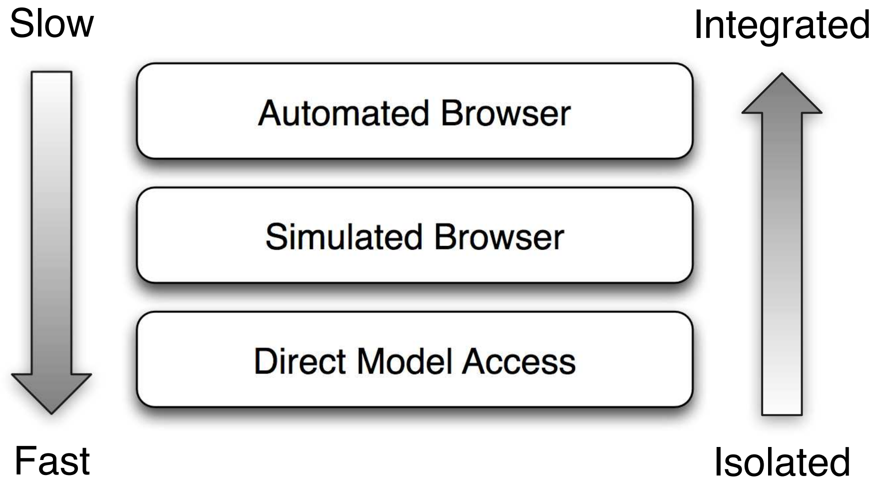


Figure 16.2: Comparing step definition styles

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 behaviors 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:

Download cucumber_rails/01/features/showtime_descriptions.feature

```
Feature: Showtime Descriptions
```

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 behavior. 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 behavior 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.

```
  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
```


Prepared exclusively for aidy lewis
this copy is (B1.0 printing, January 28, 2009)

```
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 a 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]

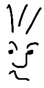
Just as instance variables can be created in before(:each) blocks and referenced in individual examples, the @movie instance variable is created in the Given() and available to all subsequent steps.

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]

Download cucumber_rails/01/out/02.failing

```
./script/cucumber features/showtime_descriptions.feature:7
```

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 was introduced in cucumber-0.1.9.



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 defintions, 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.

```
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:

```
./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 behavior.

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 17

Webrat

Coming soon ...

Prepared exclusively for aidy lewis

Chapter 18

Rails Views

Coming soon ...

Prepared exclusively for aidy lewis

Chapter 19

Rails Helpers

Coming soon ...

Prepared exclusively for aidy lewis

Chapter 20

Rails Controllers

Coming soon ...

Prepared exclusively for aidy lewis

Chapter 21

Rails Models

Coming soon ...

Prepared exclusively for aidy lewis

Part V

RSpec in the Wild

Prepared exclusively for aidy lewis

Chapter 22

RSpec and the RubySpec Project

Coming soon ...

Prepared exclusively for aidy lewis

Appendix A

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

Prepared exclusively for aidy lewis

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

Prepared exclusively for aidy lewis
this copy is (B1.0 printing, January 28, 2009)

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

Prepared exclusively for aidy lewis
this copy is (B1.0 printing, January 28, 2009)

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

Prepared exclusively for aidy lewis

Appendix B

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.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995.

[Mar02] Robert C. Martin. *Agile Software Development, Principles, Patterns, and Practices.* Prentice Hall, Englewood Cliffs, NJ, 2002.

[Rai04] J. B. Rainsberger. *JUnit Recipes : Practical Methods for Programmer Testing.* Manning Publications Co., Greenwich, CT, 2004.

Prepared exclusively for aidy lewis

Index

Prepared exclusively for aidy lewis

First page of blurb

Prepared exclusively for aidy lewis

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 in the news.

Buy the Book

If you liked this PDF, 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

Phone Orders: 1-800-699-PROG (+1 919 847 3884)
Online Orders: www.pragprog.com/catalog
Customer Service: orders@pragprog.com
Non-English Versions: translations@pragprog.com
Pragmatic Teaching: academic@pragprog.com
Author Proposals: proposals@pragprog.com

Prepared exclusively for aidy lewis