

The Cucumber Book

Behaviour-Driven
Development for
Testers and
Developers



Matt Wynne
and Aslak Hellesøy

edited by Jacquelyn Carter

What Readers Are Saying About *The Cucumber Book*

Few tools have managed to bridge the developer-customer divide as well as Cucumber has. Cucumber is not a tool for testing applications. Cucumber is a philosophy for communicating requirements. This book brings that philosophy to life.

► **Robert C. Martin (Uncle Bob)**

I devoured the Cucumber book on a train ride from Grenoble to Brussels a few days after watching Matt's presentation "BDD As It's Meant to Be Done." These two resources helped me understand in just a few hours how to avoid dozens of common mistakes writing scenarios in the Cucumber style. It's as though I received an injection of perhaps two years of experience writing scenarios poorly so that I didn't have to go through it all myself. What a gift. I recommend this book to everyone working with Cucumber.

► **J. B. Rainsberger**
Author, *JUnit Recipes*

Teams can use Cucumber to get a better understanding of what software to build for their customers. In this book, Aslak and Matt do a brilliant job explaining how you get started with Cucumber with plenty of easy-to-follow examples.

- **Rachel Davies**
Author, *Agile Coaching*

To those of you wondering how to use Cucumber effectively, *The Cucumber Book* is the answer. Not content to write just a testing book, Aslak and Matt have packed it with practical insights on many aspects of software development. Studying this book will make you a better software developer.

- **Pat Maddox, B.D.D.M.F.**
RSpec Core team

This is a much-needed book, providing not only an expanded description of how to use Cucumber but an opinionated one to suggest how to use it for the best effect. Reading this book is like having Aslak and Matt sitting next to you, patiently helping you through your first project with Cucumber. Not only will you learn effective use of Cucumber, but you'll also be introduced to several other Ruby tools that can be used with Cucumber.

- **George Dinwiddie**
Software development coach at iDIA Computing, LLC

Matt and Aslak show you how Cucumber can save you from stale documentation, unclear requirements, and absentee tests. By the end of the book, your team's programmers, testers, and product owners will be talking excitedly about the next great product you're going to build together.

► **Ian Dees**

Author, *Scripted GUI Testing with Ruby*

This book had me at “Cucumber is designed to help build bridges between the technical and nontechnical members of a software team.” Wynne and Hellesøy understand the whole-team approach to specification by example, with diverse team members collaborating to deliver what the customer really wants. They use examples to teach us how to automate regression checks with Cucumber, use it to build a safety net to allow refactoring, and free testers to contribute their most valuable skills to the team.

► **Lisa Crispin**

Author, *Agile Testing: A Practical Guide for Testers* (with Janet Gregory)

This book is a tale of how to do effective acceptance testing, with Cucumber as the filling in the sandwich. The authors don't just scratch the surface; they get right under the skin and show us how versatile Cucumber can be.

► **Robert Chatley**

Principal, Devlogical

Lots of great tips for Cucumber newbies and experts alike—Matt and Aslak have done a great job of explaining everything from getting started to how to get the most out of Cucumber. You'll want to read this book cover to cover and keep it close as a reference!

► **Gojko Adzic**

Author, *Specification by Example* and *Bridging the Communication Gap*

The Cucumber Book is a must-read for anyone thinking about using Cucumber; it is scattered with treasures for even the most experienced Cucumber users.

► **Antony Marcano**

RiverGlide

The Cucumber Book

Behaviour-Driven Development
for Testers and Developers

Matt Wynne
Aslak Hellesøy

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



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, PragProg 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://pragprog.com>.

The team that produced this book includes:

Jackie Carter (editor)
Potomac Indexing, LLC (indexer)
Kim Wimpsett (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Copyright © 2012 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-13: 978-1-934356-80-7
Printed on acid-free paper.
Book version: P1.0—January 2012

Contents

Foreword	xiii
Acknowledgments	xv
Preface	xvii

Part I — Cucumber Fundamentals

1.	Why Cucumber	3
1.1	Automated Acceptance Tests	4
1.2	Behaviour-Driven Development	4
1.3	Living Documentation	6
1.4	How Cucumber Works	7
1.5	What We Just Learned	8
2.	First Taste	11
2.1	Understanding Our Goal	11
2.2	Creating a Feature	12
2.3	Creating Step Definitions	14
2.4	Implementing Our First Step Definition	16
2.5	Running Our Program	17
2.6	Changing Formatters	18
2.7	Adding an Assertion	19
2.8	Making It Pass	20
2.9	What We Just Learned	23
3.	Gherkin Basics	25
3.1	What's Gherkin For?	25
3.2	Format and Syntax	28
3.3	Feature	29
3.4	Scenario	30
3.5	Comments	33

3.6	Spoken languages	34
3.7	What We Just Learned	35
4.	Step Definitions: From the Outside	39
4.1	Steps and Step Definitions	40
4.2	Capturing Arguments	45
4.3	Multiple Captures	49
4.4	Flexibility	50
4.5	Returning Results	52
4.6	What We Just Learned	58
5.	Expressive Scenarios	61
5.1	Background	61
5.2	Data Tables	64
5.3	Scenario Outline	70
5.4	Nesting Steps	75
5.5	Doc Strings	78
5.6	Staying Organized with Tags and Subfolders	79
5.7	What We Just Learned	82
6.	When Cucumbers Go Bad	85
6.1	Feeling the Pain	86
6.2	Working Together	89
6.3	Caring for Your Tests	97
6.4	Stop the Line and Defect Prevention	105
6.5	What We Just Learned	106

Part II — A Worked Example

7.	Step Definitions: On the Inside	111
7.1	Sketching Out the Domain Model	112
7.2	Removing Duplication with Transforms	117
7.3	Adding Custom Helper Methods to the World	120
7.4	Organizing the Code	128
7.5	What We Just Learned	131
8.	Support Code	133
8.1	Fixing the Bug	134
8.2	Bootstrapping the User Interface	141
8.3	Making the Switch	142
8.4	Using Hooks	147

8.5	Building the User Interface	150
8.6	What We Just Learned	154
9.	Dealing with Message Queues and Asynchronous Components	157
9.1	Our New Asynchronous Architecture	157
9.2	How to Synchronize	158
9.3	Implementing the New Architecture	161
9.4	Fixing the Flickering Scenario	165
9.5	What We Just Learned	171
10.	Databases	173
10.1	Introducing ActiveRecord	174
10.2	Refactoring to Use a Database	175
10.3	Reading and Writing to the Database	178
10.4	Cleaning the Database with Transactions	181
10.5	Cleaning the Database with Truncation	185
10.6	What We Just Learned	186

Part III — Cucumber Applied

11.	The Cucumber Command-Line Interface	191
11.1	Cucumber's Command-Line Options	191
11.2	Running a Subset of Scenarios	192
11.3	Changing Cucumber's Output	194
11.4	Specifying the Location of Step Definitions	196
11.5	Managing Your Work in Progress (WIP)	197
11.6	Using Profiles	198
11.7	Running Cucumber from Rake	198
11.8	Running Cucumber in Continuous Integration	199
11.9	What We Just Learned	200
12.	Testing a REST Web Service	201
12.1	In-Process Testing of Rack-Based REST APIs	202
12.2	Out-of-Process Testing of Any REST API	213
12.3	What We Just Learned	220
13.	Adding Tests to a Legacy Application	221
13.1	Characterization Tests	222
13.2	Squashing Bugs	224
13.3	Adding New Behavior	225

13.4	Code Coverage	227
13.5	What We Just Learned	227
14.	Bootstrapping Rails	229
14.1	Running the Generators	230
14.2	Creating a User	232
14.3	Posting a Message	236
14.4	Associating a Message with a User	239
14.5	Creating a Controller by Hand	240
14.6	Implementing the View	242
14.7	What We Just Learned	243
14.8	Try this	244
15.	Using Capybara to Test Ajax Web Applications	245
15.1	Implementing a Simple Search Without Ajax	247
15.2	Searching with Ajax	257
15.3	The Capybara API	268
15.4	Taking Screenshots	271
15.5	What We Just Learned	272
16.	Testing Command-Line Applications with Aruba	275
16.1	Simple Interfaces	275
16.2	Our First Aruba Feature	276
16.3	Working with Files and Executables	280
16.4	Interacting with User Input	288
16.5	Using Aruba's Ruby DSL	291
16.6	What We Just Learned	292
A1.	Using Cucumber with Other Platforms	295
A2.	Installing Cucumber	299
A2.1	Installing Ruby	299
A2.2	HTTP Proxy Settings	300
A2.3	Installing Bundler	301
A2.4	Installing Cucumber (and RSpec)	301
A2.5	Installing Other Gems	301
A2.6	Choosing a Text Editor	302
A3.	Ruby Gem Versions	303
A4.	Bibliography	305
	Index	307

Foreword

Behaviour-driven development has come a long way since I first started talking about it in 2003. At that time, I was simply trying to find better ways to explain the revelatory practice of TDD, usually to nervous, suspicious, or at the very least skeptical programmers. Why would you write tests ahead of any code? That didn't make sense. And why were we writing tests anyway—don't we have testers for that?

Very few things represent a genuine paradigm shift. Mostly the term is used by marketers to convince you to change your brand of toothpaste. According to the Free Online Dictionary, a paradigm¹ is “A set of assumptions, concepts, values, and practices that constitutes a way of viewing reality for the community that shares them.” That's right, a paradigm shift involves messing with someone's sense of *reality!* No wonder they get uncomfortable.

TDD is one of those rare genuine cases, so it's no surprise that many people are deeply skeptical when you start trying to introduce it. And it's also not surprising that it took us several attempts at articulating it, in different ways, from different angles, and with different audiences, before we found something that worked. At first we started deep in the code, because that's where the programmers were. Over time we were able to take the action closer to the business stakeholders and describe the multilayered approach that is modern BDD (and is also, ironically, classic TDD, which Kent Beck described from the very outset as working on multiple levels of abstraction).

Aslak Hellesøy has been part of the effort to describe that shift almost from the very beginning. As well as an early adopter of BDD—and a passionate advocate of TDD—he rewrote my sad efforts at building a scenario runner for RSpec into the tool we now call Cucumber. He has invested enormous time and effort into both the tool and its community, so it came as no surprise to learn he and Matt were writing a book on BDD in Cucumber. I love that they

1. <http://www.thefreedictionary.com/paradigm>

are targeting both developers and testers. If the tool isn't bringing these two worlds closer together, then it's the wrong tool.

I was delighted to learn that Aslak's partner in crime would be Matt Wynne. Another passionate and experienced TDDer—and BDDer—Matt has been involved with Cucumber since day one. He is a fun and engaging speaker, is a great teacher, and has a wealth of knowledge and wit that comes across in his writing. In fact, I'm proposing that Matt Wynne be inaugurated as a unit of excellence, somewhere between a Win and an Epic Win. (*Oh, cool, did you see that? That was better than a win; that was a Matt Wynne!*)

I hope you enjoy this book as much as I did. My review notes from an earlier draft seem to contain the phrase “Oh, this is lovely” more times than I remembered. It feels like you are being led into a strange, yet somehow oddly familiar, world with two knowledgeable and accommodating guides. Oodles of examples, descriptions, and sidebars (Joe—who you'll meet early on—quickly became my friend by asking the things I found myself asking.) help you on your way, and the authors manage to keep the plot moving quickly enough to keep you engaged, which is always a challenge in a technical book.

I can't say where BDD will be in another eight years, but with folks like Matt and Aslak sharing their innovation and clarity, now is a very exciting time to be involved in agile software development.

—Dan North, *Lean technology specialist at DRW Trading and originator of BDD*

Acknowledgments

The first people we want to thank are the hundreds of you who contribute to the Cucumber community. Whether you're sharing ideas, experiences, and opinions on the mailing list, helping people in the IRC channel, or contributing new features and bug fixes to the codebase, it all helps. Without your contribution, there would be no Cucumber and therefore no book.

Writing this book has taken much more effort than either of us had anticipated. Throughout it all, our editor Jackie Carter has patiently been there at our side, cajoling us when we needed it, chiding us when we deserved it, and giving us thoughtful feedback at every opportunity. Jackie has made a massive contribution to the quality of what you're reading, and her name fully deserves its place on the cover.

Thanks to our reviewers:

Lasse Koskela	George Dinwiddie
Rachel Davies	Ian Dees
Arti Mathanda	Robert Chatley
Kevin Rutherford	Luis Lavena
Gojko Adzic	Crain Riecke
Sean Miller	Mike Sassak

Your suggestions and encouragement were greatly appreciated.

Thanks to all the beta readers who left us feedback, helping us iron out the little mistakes we would never have seen ourselves.

Thanks to Dan North for his enthusiastic and generous foreword. Matt blames Dan for introducing him to the idea of BDD, and it was his experiments with JBehave and RBehave that caused Aslak to create Cucumber in the first place. Dan has a lot to answer for.

From Matt: I want to thank the team at Songkick, especially Sabrina Leandro, Niko Felger, Dan Lucraft, Phil Cowans, and Matt Johnson. Many of the lessons

in this book I learned with you. Greatest thanks go to my wonderful wife, Anna, for believing in this project and giving me the support I needed to actually get it done. Imagine all the things we'll be able to do now that it's finished!

From Aslak: Dad, thanks for having the foresight to buy me a Commodore 64 in 1981. Patricia, my dear wife—thank you for the countless hours of patience and encouragement. And for coming up with the silly but catchy name Cucumber!

Preface

Cucumber is a friendly tool. It wants to be part of your team, and it doesn't mind being the nitpicky nerd who can remember every single little detail about what your system can and can't do. Every team needs someone like that.

Even better than that, Cucumber will volunteer to do all the boring repetitive checks that you need to run to make sure that the system is working as expected. This frees your testers up to do interesting, creative work instead, and it gives programmers the courage to perform major surgery to the code when it's required. Business stakeholders warm to Cucumber's open attitude, sharing everything that the development team is doing in terminology that they can actually understand.

Cucumber is a young tool, but it's already become clear that some people misunderstand it. Those of us who were drawn to Cucumber from the beginning instinctively realized that it's more than a testing tool; it's a collaboration tool. By writing this book, we hope to show you not just how to use Cucumber but how to use it well.

Who This Book Is For

Cucumber is designed to help build bridges between the technical and non-technical members of a software team, and we've tried to consider both of those readers. The majority of the book is written to the technical reader, someone who is interested in test automation and already has at least some programming skill. However, several of the chapters—especially in the first part of the book where we explain how to write the specifications themselves—are written with the nontechnical reader very much in mind. Specifically, those chapters are as follows:

- [Chapter 1, Why Cucumber?, on page 3](#)
- [Chapter 3, Gherkin Basics, on page 25](#)
- [Chapter 5, Expressive Scenarios, on page 61](#)

- [Chapter 6, When Cucumbers Go Bad, on page 85](#)
- [Chapter 13, Adding Tests to a Legacy Application, on page 221](#)

As the book develops, we'll look at more complex testing situations, and the level of technical know-how required to read the chapters will increase. We've tried to make this buildup as gradual as possible so that if you're only just beginning your journey into test automation, you should be able to follow along and learn as we go.

You Don't Need to Know Ruby, But It Helps

Ruby¹ is an open source programming language that can be installed and run on all major operating systems. The original—and still most popular—version of Cucumber is written in Ruby, and this book is about that version.

That doesn't mean the system you're testing has to be written in Ruby. One of Ruby's many strengths is how good it is at talking to other languages and platforms, and we'll show you how to use Ruby tools to test web-based systems that could be written in any language.

To follow along with the coding examples in the technical chapters, it will help if you know a little Ruby. Ruby is an easy language to learn, and the Ruby examples we'll use are deliberately simple. To get the best out of this book, we suggest that Ruby novices accompany it with a copy of *Everyday Scripting with Ruby* [Mar07] or *Programming Ruby: The Pragmatic Programmer's Guide* [TFH08].

It's OK if You're Not Test-Driven

We've had our greatest success with Cucumber as part of an *outside-in* approach, starting with a failing Cucumber test and using that to drive our development work on the application code. As developers, this way of working helps us stay honest and avoid the temptation to build in functionality that nobody asked us for, just in case it might be needed one day in the future.

Cucumber is a tool that facilitates this way of working, but it doesn't force it on you. Some teams use Cucumber to automate tests for the work that developers have already done. This can often be a first step toward adopting an outside-in approach, as Cucumber's readable tests start to attract the attention of the team's nontechnical stakeholders, drawing them into the process. Even if you're using Cucumber to write tests against existing code, you'll still get a great deal of benefit from Cucumber over alternatives like QTP and

1. <http://www.ruby-lang.org>

Selenium IDE, and we think you'll still get a lot out of this book. We're not here to preach to you about process, but we will share our insights about what has worked for us and why.

Why You Should Listen to Us

We've both been building software for a living for almost twenty years, and using automated tests for nearly ten of those. Aslak created Cucumber in 2008, and Matt has been one of its most active users from day one.

We've used Cucumber to test all kinds of systems: from Ruby on Rails web applications, through Flash games, to enterprise Java web services. We've also trained hundreds of developers in how to use Cucumber, teaching the material in this book at events and companies around the world.

The Cucumber community is full of lively debate, and we've spent many hours of our spare time having our ideas challenged and honed in discussions with other users. We hope we've distilled as much of that knowledge and experience as possible in this book.

How This Book Is Organized

The book is in three parts. In Part I, we'll take you through the core concepts you need to know in order to make use of Cucumber. Novice readers will learn everything they need to know to get up and running, and readers who are already experienced with Cucumber should pick up plenty of useful detail too.

Part II works through a practical example of developing a new application using Cucumber. You'll pair with us as we build a simple application from scratch, giving you a chance to experience how we like to build software using Cucumber and to consolidate what you've learned in Part I. We'll also teach you some advanced features of Cucumber that are easier to learn in the context of an example.

Cucumber provides the framework for specifying and executing tests, but there are a wide variety of systems that you might want to test. In Part III, we've provided you with a broad selection of guides to using Cucumber in some common situations, such as testing REST APIs, Ajax web applications, and command-line applications.

What Is Not in This Book

Although it is possible to test Flash and mobile applications using Cucumber, the details are sadly beyond the scope of this edition. Similarly, the flavors

of Cucumber that run natively on the JVM, JavaScript and C#, allowing you to write your Cucumber code in the same language as your production code, will not be covered. Cucumber's wire protocol (a protocol for driving remote systems over a TCP socket) is also out of scope.

We have provided a list of pointers to further information on this subject in [Appendix 1, Using Cucumber with Other Platforms, on page 295](#). For further information about using Ruby to automate and test different kinds of system, we recommend getting a copy of *Scripted GUI Testing with Ruby* [Dee08] by Ian Dees to compliment our book.

Running the Code Examples

This book is full of practical examples, and we encourage you to follow along with them to get the most out of the book. You'll learn the most if you type them in by hand as you read along, but if you'd prefer, you can always download the code examples from http://pragprog.com/titles/hwcuc/source_code.

To run the examples, you'll need to install the Ruby language itself as well as some additional libraries, which in Ruby are called *gems*. You can find the full instructions in [Appendix 2, Installing Cucumber, on page 299](#).

Windows Users

Most of the code examples work just the same on Windows and *nix operating systems. On the rare occasions that they differ, you'll find the Windows version in a sidebar nearby, with a note in the body of the text pointing you to the sidebar.

You'll soon notice that we've used the \$ symbol for the command prompt. This is familiar to Linux or Mac users but might feel a little unfamiliar to Windows users. So, when you're looking at something like this:

\$ cucumber

try to imagine you're seeing this instead:

C:\> cucumber

Other than that, everything should work just the same for everyone.

Getting Help

If you're stuck on one of the exercises in this book, there is a discussion forum where you can ask for help at <http://forums.pragprog.com/forums/166>.

If you have a general question about Cucumber, the Cucumber community will welcome you to their mailing list at <https://groups.google.com/forum/#!forum/cukes>. Cucumber is an open source tool, which means that everyone contributing to the group is volunteering their time, so please make sure you've researched your question as thoroughly as you can before you ask for help on the mailing list. People will be much more likely to help you if they can see you're trying to help yourself.

Part I

Cucumber Fundamentals

CHAPTER 1

Why Cucumber?

Software starts as an idea.

Let's assume it's a good idea—an idea that could make the world a better place, or at least make someone some money. The challenge of the software developer is to take the idea and make it real, into something that actually delivers that benefit.

The original idea is perfect, beautiful. If the person who has the idea happens to be a talented software developer, then we might be in luck: the idea could be turned into working software without ever needing to be explained to anyone else. More often, though, the person with the original idea doesn't have the necessary programming skill to make it real. Now the idea has to travel from that person's mind into other people's. It needs to be *communicated*.

Most software projects involve teams of several people working collaboratively together, so high-quality communication is critical to their success. As you probably know, good communication isn't just about eloquently describing your ideas to others; you also need to solicit feedback to ensure you've been understood correctly. This is why agile software teams have learned to work in small *increments*, using the software that's built incrementally as the feedback that says to the stakeholders "Is this what you mean?"

Even this is not enough. If the developers spend a two-week iteration implementing a misunderstanding, not only have they wasted two weeks of effort, but they've corrupted the integrity of the codebase with concepts and functionality that do not reflect the original idea. Other developers may have already innocently started to build more code on top of those bad ideas, making it unlikely they'll ever completely disappear from the codebase.

We need a kind of filter to protect our codebase from these misunderstood ideas.

1.1 Automated Acceptance Tests

The idea of *automated acceptance tests* originates in eXtreme Programming¹ (XP), specifically in the practice of Test-Driven Development² (TDD).

Instead of a business stakeholder passing requirements to the development team without much opportunity for feedback, the developer and stakeholder collaborate to write automated tests that express the outcome that the stakeholder wants. We call them acceptance tests because they express what the software needs to do in order for the stakeholder to find it *acceptable*. The test fails at the time of writing, because no code has been written yet, but it captures what the stakeholder cares about and gives everyone a clear signal as to what it will take to be *done*.

These tests are different from *unit tests*, which are aimed at developers and help them to drive out and check their software designs. It's sometimes said that unit tests ensure you *build the thing right*, while acceptance tests ensure you *build the right thing*.

Automated acceptance testing has been an established practice among good XP teams for years, but many less experienced agile teams seem to see TDD as being a programmer activity only. As Lisa Crispin and Janet Gregory point out in *Agile Testing: A Practical Guide for Testers and Agile Teams* [CG08], without the business-facing automated acceptance tests, it's hard for the programmers to know which unit tests they need to write. Automated acceptance tests help your team to focus, ensuring the work you do each iteration is the most valuable thing you could possibly be doing. You'll still make mistakes—but you'll make a lot less of them—meaning you can go home on time and enjoy the rest of your life.

1.2 Behaviour-Driven Development

Behaviour-Driven Development³ (BDD) builds upon Test-Driven Development (TDD) by formalizing the good habits of the best TDD practitioners. The best TDD practitioners work from the *outside-in*, starting with a failing customer acceptance test that describes the behavior of the system from the customer's point of view. As BDD practitioners, we take care to write the acceptance tests as *examples* that anyone on the team can read. We make use of the process of writing those examples to get feedback from the business stakeholders about whether we're setting out to build the right thing before we get started.

-
1. *Extreme Programming Explained: Embrace Change* [Bec00]
 2. *Test Driven Development: By Example* [Bec02]
 3. <http://behaviour-driven.org/>

As we do so, we make a deliberate effort to develop a shared, *ubiquitous language* for talking about the system.

Ubiquitous language

As Eric Evans describes in his book *Domain Driven Design* [Eva03], many software projects suffer from low-quality communication between the domain experts and programmers on the team:

“A project faces serious problems when its language is fractured. Domain experts use their jargon while technical team members have their own language tuned for discussing the domain in terms of design... Across this linguistic divide, the domain experts vaguely describe what they want. Developers, struggling to understand a domain new to them, vaguely understand.”

With a conscious effort by the team, a ubiquitous language can emerge that is used and understood by everyone involved in the project. When the team uses this language consistently in their conversations, documentation, and code, the friction of translating between everyone’s different little dialects is gone, and the chances of misunderstandings are greatly reduced.

Cucumber helps facilitate the discovery and use of a ubiquitous language within the team, by giving the two sides of the linguistic divide a place where they can meet. Cucumber tests interact directly with the developers’ code, but they’re written in a medium and language that business stakeholders can understand. By working together to write these tests—*specifying collaboratively*—not only do the team members decide what behavior they need to implement next, but they learn how to describe that behavior in a common language that everyone understands.

When we write these tests before development starts, we can explore and eradicate many misunderstandings long before they ooze their way into the codebase.

Examples

What makes Cucumber stand out from the crowd of other testing tools is that it has been designed specifically to ensure the acceptance tests can easily be read—and written—by anyone on the team. This reveals the true value of acceptance tests: as a communication and collaboration tool. The easy readability of Cucumber tests draws business stakeholders into the process, helping you really explore and understand their requirements.

Here’s an example of a Cucumber acceptance test:

Feature: Sign up

Sign up should be quick and friendly.

Scenario: Successful sign up

New users should get a confirmation email and be greeted personally by the site once signed in.

Given I have chosen to sign up

When I sign up with valid details

Then I should receive a confirmation email

And I should see a personalized greeting message

Scenario: Duplicate email

Where someone tries to create an account for an email address that already exists.

Given I have chosen to sign up

But I enter an email address that has already registered

Then I should be told that the email is already registered

And I should be offered the option to recover my password

Notice how the test is specified as *examples* of the way we want the system to behave in particular scenarios. Using examples like this has an unexpectedly powerful effect in enabling people to visualize the system before it has been built. Anyone on the team can read a test like this and tell you whether it reflects their understanding of what the system should do, and it may well spark their imagination into thinking of other scenarios that you'll need to consider too. Gojko Adzic's book *Specification by Example* [Adz11] contains many case studies of teams who have discovered this and used it to their advantage.

Acceptance tests written in this style become more than just tests; they are *executable specifications*.

1.3 Living Documentation

Cucumber tests share the benefit of traditional specification documents in that they can be written and read by business stakeholders, but they have a distinct advantage in that you can give them to a computer at any time to tell you how accurate they are. In practice, this means that your documentation, rather than being something that's written once and then gradually goes out of date, becomes a living thing that reflects the true state of the project.

Source of Truth

For many teams, they become the definitive *source of truth* as to what the system does. Having a single place to go for this information saves a lot of time that is often wasted trying to keep requirements documents, tests, and code all in sync. It also helps to build trust within the team, because different parts of the team no longer have their own personal versions of the truth.

1.4 How Cucumber Works

Before we dive into the meat of the book, let's give you some context with a high-level overview of a typical Cucumber test suite.

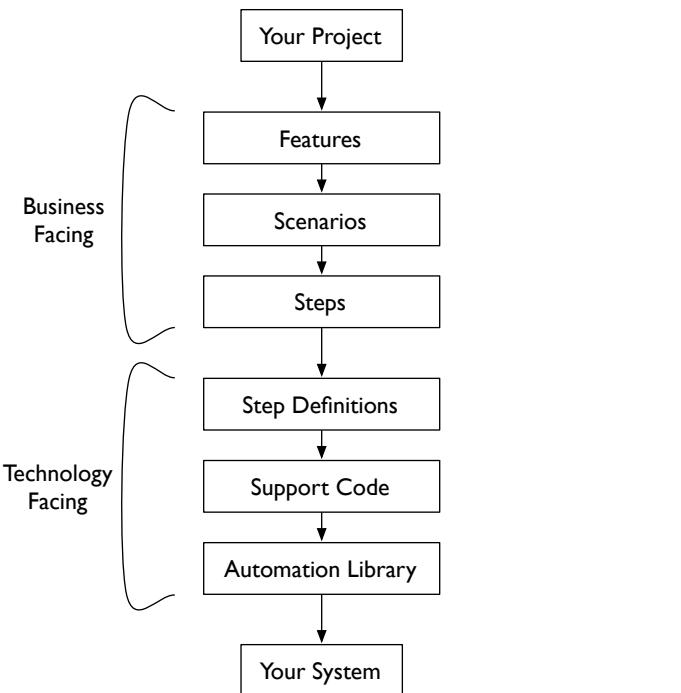
Cucumber is a command-line tool. When you run it, it reads in your specifications from plain-language text files called *features*, examines them for *scenarios* to test, and runs the scenarios against your system. Each scenario is a list of *steps* for Cucumber to work through. So that Cucumber can understand these feature files, they must follow some basic syntax rules. The name for this set of rules is *Gherkin*.

Along with the features, you give Cucumber a set of *step definitions*, which map the business-readable language of each step into Ruby code to carry out whatever action is being described by the step. In a mature test suite, the step definition itself will probably just be one or two lines of Ruby that delegate to a library of *support code*, specific to the domain of your application, that knows how to carry out common tasks on the system. Normally that will involve using an *automation library*, like the browser automation library Capybara, to interact with the system itself.

This hierarchy, from features down to automation library, is illustrated by [Figure 1, Cucumber testing stack, on page 8](#).

If the Ruby code in the step definition executes without error, Cucumber proceeds to the next step in the scenario. If it gets to the end of the scenario without any of the steps raising an error, it marks the scenario as having passed. If any of the steps in the scenario fail, however, Cucumber marks the scenario as having failed and moves on to the next one. As the scenarios run, Cucumber prints out the results showing you exactly what is working and what isn't.

That's it in a nutshell. There are many other advantages to Cucumber that make it an excellent choice: you can write your specifications in more than forty different spoken languages, you can use tags to organize and group your scenarios, and you can easily integrate with a host of high-quality Ruby

**Figure 1—Cucumber testing stack**

automation libraries to drive almost any kind of application. All that and more will become clear as you read the rest of the book.

1.5 What We Just Learned

Let's review what we've covered so far.

Software teams work best when the developers and business stakeholders are communicating clearly with one another. A great way to do that is to collaboratively specify the work that's about to be done using automated acceptance tests.

When the acceptance tests are written as examples, they stimulate people's imaginations and help them see other scenarios they hadn't previously considered.

When the team write their acceptance tests collaboratively, they can develop their own ubiquitous language for talking about their problem domain. This helps them avoid misunderstandings.

Cucumber was designed specifically to help business stakeholders get involved in writing acceptance tests.

Each test case in Cucumber is called a *scenario*, and scenarios are grouped into features. Each scenario contains several steps.

The business-facing parts of a Cucumber test suite, stored in feature files, must be written according to syntax rules—known as Gherkin—so that Cucumber can read them.

Under the hood, step definitions translate from the business-facing language of steps into Ruby code.

To illustrate these concepts, in the next chapter we're going to dive right in and build a very simple application, using Cucumber to drive the development.

Try This

Cucumber has its own ubiquitous language. Can you list the terms about Cucumber's domain you've learned in this chapter and describe what you think each of them means?

First Taste

We figured you'd be eager to start playing with your shiny new toy right away, so we're going to work through a simple example that will give you a feel for what it's like to work with Cucumber. You might not quite understand everything that happens here just yet, but try not to let that worry you. We'll come back and fill in the details over the next few chapters.

We're going to build a simple command-line application from the outside-in, driving our development with Cucumber. Watch how we proceed in very small baby steps, going back to run Cucumber after we make each change. This patient rhythm is an important characteristic of working effectively with Cucumber, but it's much easier to demonstrate than to explain.

Assuming you want to follow along with the action in this chapter (it'll be a lot more fun if you do), you'll need to have Cucumber installed. If you haven't already done so, see [Appendix 2, Installing Cucumber, on page 299](#) for installation instructions.

Right...shall we get started then?

2.1 Understanding Our Goal

Our goal is to write a program that can perform calculations. Some people might call it a calculator.

We have an incredible vision of what this calculator will one day become: a cloud-based service that runs on mobile phones, desktops, and browsers, uniting the world with the opportunity of ubiquitous mathematical operations. We're pragmatic businesspeople, though, so the first release of our program has to be as simple as possible. The first release will be a command-line program, implemented as a Ruby script. It will take input containing the calculation to be done and display the result at the command prompt.

So, for example, if the input file looks like this:

```
2+2
```

then the output would be 4.

Similarly, if the input file looks like this:

```
100/2
```

then the output would be 50.

You get the idea.

2.2 Creating a Feature

Cucumber tests are grouped into *features*. We use this name because we want them to describe the features that a user will be able to enjoy when using our program. The first thing we need to do is make a directory where we'll store our new program along with the features we'll be writing for it.

```
$ mkdir calculator  
$ cd calculator
```

We're going to let Cucumber guide us through the development of our calculator program, so let's start right away by running cucumber in this empty folder:

```
$ cucumber
```

You don't have a 'features' directory. Please create one to get started.
See <http://cukes.info/> for more information.

Since we didn't specify any command-line arguments, Cucumber has assumed that we're using the conventional features folder to store our tests. That would be fine, except that we don't have one yet. Let's follow the convention and create a features directory:

```
$ mkdir features
```

Now run Cucumber again.

```
$ cucumber
```

```
0 scenarios  
0 steps  
0m0.000s
```

Each Cucumber test is called a *scenario*, and each scenario contains *steps* that tell Cucumber what to do. This output means that Cucumber is happily

scanning the features directory now, but it didn't find any scenarios to run. Let's create one.

Our user research has shown us 67 percent of all mathematical operations are additions, so that's the operation we want to support first. In your favorite editor, create a plain-text file called `features/adding.feature`:

[Download first_taste/01/features/adding.feature](#)

Feature: Adding

```
Scenario: Add two numbers
  Given the input "2+2"
  When the calculator is run
  Then the output should be "4"
```

This `.feature` file contains the first scenario for our calculator program. We've translated one of the examples we were given in the previous section into a Cucumber scenario that we can ask the computer to run over and over again. You can probably see that Cucumber expects a little bit of structure in this file. The keywords Feature, Scenario, Given, When, and Then are the structure, and everything else is documentation. Although some of the keywords are highlighted here in the book—and they may be in your editor too—it's just a plain-text file. The structure is called *Gherkin*.

When you save this file and run cucumber, you should see a great deal more output than the last time:

`$ cucumber`

Feature: Adding

```
Scenario: Add two numbers
  Given the input "2+2"
  When the calculator is run
  Then the output should be "4"
```

```
1 scenario (1 undefined)
3 steps (3 undefined)
0m0.003s
```

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

```
Given /^the input "([^"]*)"/ do |arg1|
  pending # express the regexp above with the code you wish you had
end
```

```
When /^the calculator is run$/ do
  pending # express the regexp above with the code you wish you had
end
```

```
Then /^the output should be "([^\"]*)"$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end
```

If you want snippets in a different programming language, just make sure a file with the appropriate file extension exists where cucumber looks for step definitions.

Wow, that's a lot of output all of a sudden! Let's take a look at what's going on here. First, we can see that Cucumber has found our feature and is trying to run it. We can tell this because Cucumber has repeated the content of the feature back to us. You might also have noticed that the summary output 0 scenarios has changed to 1 scenario (undefined). This means that Cucumber has read the scenario in our feature but doesn't know how to run it yet.

Second, Cucumber has printed out three code snippets. These are sample code for *step definitions*, written in Ruby, which tell Cucumber how to translate the plain English steps in the feature into actions against our application. Our next step will be to put these snippets into a Ruby file where we can start to flesh them out.

Before we explore beneath the layer of business-facing Gherkin features into step definitions, it's worth taking a quick look at the map in case anyone is feeling lost. [Figure 2, The main layers of a Cucumber test suite, on page 15](#) reminds us how things fit together. We start with features, which contain our scenarios and steps. The steps of our scenarios call step definitions that provide the link between the Gherkin features and the application being built.

Now we'll implement some step definitions so that our scenario is no longer undefined.

2.3 Creating Step Definitions

Without thinking too much about what they mean, let's just copy and paste the snippets from Cucumber's last output into a Ruby file. Just like the features, Cucumber is expecting the step definitions to be found in a conventional place:

```
$ mkdir features/step_definitions
```

Now create a Ruby file in `features/step_definitions`, called `calculator_steps.rb`. Cucumber won't mind what you call it as long as it's a Ruby file, but this is a good name to use. Open it in your text editor and paste in those snippets:

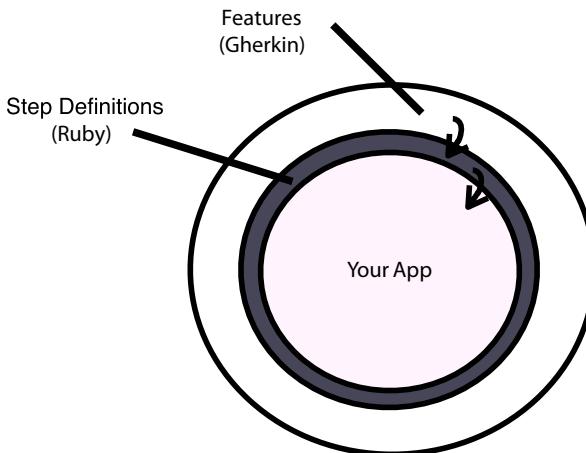


Figure 2—The main layers of a Cucumber test suite

[Download first_taste/02/features/step_definitions/calculator_steps.rb](#)

```
Given /^the input "([^"]*)"$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end
```

```
When /^the calculator is run$/ do
  pending # express the regexp above with the code you wish you had
end
```

```
Then /^the output should be "([^"]*)"$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end
```

Running cucumber will tell us what to do next:

Feature: Adding

```
Scenario: Add two numbers
  Given the input "2+2"
    TODO (Cucumber::Pending)
    ./features/step_definitions/calculator_steps.rb:2
    features/adding.feature:4
  When the calculator is run
  Then the output should be "4"
```

```
1 scenario (1 pending)
3 steps (2 skipped, 1 pending)
0m0.002s
```

The scenario has graduated from undefined to pending. This is good news, because it means Cucumber is now running the first step, but as it did so, it hit the call to pending inside our copied-and-pasted step definition code, which tells Cucumber that this scenario is still a work in progress. We need to replace that pending marker with a real implementation.

Notice that Cucumber reports the two other steps as skipped. As soon as it encounters a failed or pending step, Cucumber will stop running the scenario and skip the remaining steps.

Let's implement the first step definition.

2.4 Implementing Our First Step Definition

We've decided this first release of our calculator is going to take its input from the user as a command-line argument, so our job in the step definition for Given the input "2+2" is just to remember the input so that we know what to pass on the command line when we run the calculator in the next step. In the features/step_definitions folder, edit the calculator_steps.rb file so that the first step definition looks like this:

```
Download first_taste/03/features/step_definitions/calculator_steps.rb
Given /^the input "([^"]*)"$/ do |input|
  @input = input
end
```

All we've done here is store the input from the feature in a Ruby instance variable. That instance variable will be around for as long as this particular scenario is running, so we can use it again in the next step when we actually run the calculator.

Great, that was easy. Now, where were we again? Let's ask cucumber:

Feature: Adding

```
Scenario: Add two numbers
  Given the input "2+2"
  When the calculator is run
    TODO (Cucumber::Pending)
    ./features/step_definitions/calculator_steps.rb:9
      features/adding.feature:5
    Then the output should be "4"
```

```
1 scenario (1 pending)
3 steps (1 skipped, 1 pending, 1 passed)
0m0.002s
```

Yay! Our first step passed! The scenario is still marked as pending, of course, because we still have the other two steps to implement, but we're starting to get somewhere.

2.5 Running Our Program

To implement the next step, edit `features/step_definitions/calculator_steps.rb` so that the second step definition looks like this:

```
Download first_taste/04/features/step_definitions/calculator_steps.rb
When /^the calculator is run$/ do
  @output = `ruby calc.rb #{@input}`
  raise('Command failed!') unless $? .success?
end
```

This code attempts to run our calculator program `calc.rb`, passing it the input we stored in the first step and storing any output in another instance variable. Then it checks a specially—but rather cryptically—named Ruby variable `$?` to check whether the command succeeded and raises an error if it didn't. Remember that Cucumber fails a step if the step definition raises an error; this is the simplest way to do it.

This time when we run Cucumber, we should see that it has actually tried to run our calculator:

Feature: Adding

```
Scenario: Add two numbers
  Given the input "2+2"
ruby: No such file or directory -- calc.rb (LoadError)
  When the calculator is run
    Command failed! (RuntimeError)
      ./features/step_definitions/calculator_steps.rb:10
        features/adding.feature:5
    Then the output should be "4"
```

```
Failing Scenarios:
cucumber features/adding.feature:3
```

```
1 scenario (1 failed)
3 steps (1 failed, 1 skipped, 1 passed)
0m0.027s
```

Our step is failing, because we don't have a `calc.rb` program to run yet. You should see that Cucumber has highlighted the output from our raised error in red just beneath the step, helping you spot the problem.

You might well think it's a bit odd that we've written and run code that tries to run our `calc.rb` program, knowing perfectly well that the file doesn't even

exist yet. We do this deliberately, because we want to make sure we have a fully functioning test in place before we drop down to working on the solution. Having the discipline to do this means we can trust our tests, because we've seen them fail, and this gives us confidence that when the tests pass, we're really done. This gentle rhythm is a big part of what we call *outside-in development*, and while it might seem strange at first, we hope to show you throughout the book that it has some great benefits.

Another benefit of working from the outside-in is that we've had a chance to think about the command-line interface to our calculator from the point of view of a user, without having made any effort to implement it yet. At this stage, if we realize there's something we don't like about the interface, it's very easy for us to change it.

2.6 Changing Formatters

We think it's starting to get distracting looking at the whole content of the feature in Cucumber's output each time we run. Let's switch to the progress *formatter* to get a more focused output. Run this command:

```
$ cucumber --format progress
```

You should see the following output:

```
.ruby: No such file or directory -- calc.rb (LoadError)
F-

(:) failed steps (:)

Command failed! (RuntimeError)
./features/step_definitions/calculator_steps.rb:10
features/adding.feature:5

Failing Scenarios:
cucumber features/adding.feature:3 # Scenario: Add two numbers

1 scenario (1 failed)
3 steps (1 failed, 1 skipped, 1 passed)
0m0.023s
```

Instead of printing the whole feature, the progress formatter has printed three characters in the output, one for each step. The first `.` character means the step passed. The `F` character means the second step, as we know, has failed. The final `-` character means that the last step has been skipped. Cucumber has several different formatters that you can use to produce different types of output as your features run; you'll learn more about them through the course of the book.

Formatters

Cucumber formatters allow you to visualize the output from your test run in different ways. There are formatters that produce HTML reports, formatters that produce JUnit XML for continuous integration servers like Jenkins, and many more.

Use cucumber --help to see the different formatters that are available and try some out for yourself. We'll explain more about formatters in [Chapter 11, The Cucumber Command-Line Interface, on page 191](#).

That was an interesting little diversion, but let's get back to work. We have a failing test to fix!

2.7 Adding an Assertion

So, following Cucumber's lead, we need to create a Ruby file for our program. Let's just create an empty Ruby file for the time being so that we can stay on the outside and get the test finished before we go on to the solution. Linux/Mac users can type this to create an empty file:

```
$ touch calc.rb
```

If you're using Windows, there is no touch command, so either just create an empty text file named calc.rb in your editor or use this little trick:

```
C:\> echo.>calc.rb
```

When we run cucumber again, we should see that the second step is passing and we're on to the final step:

```
$ cucumber --format progress
..P
(::) pending steps (:)

features/adding.feature:6:in 'Then the output should be "4"'
  1 scenario (1 pending)
  3 steps (1 pending, 2 passed)
  0m0.460s
```

To get the last step definition working, change the last step definition in features/step_definitions/calculator_steps.rb to look like this:

```
Download first_taste/07/features/step_definitions/calculator_steps.rb
Then /^the output should be "(.*)"$/ do |expected_output|
  @output.should == expected_output
end
```

We're using an RSpec¹ assertion to check that the expected output specified in the feature matches the output from our program that we stored in @output in the previous step definition. If it doesn't, RSpec will raise an error just like we did using raise in the last step definition.

Now when we run cucumber, we have ourselves a genuine failing test:

```
$ cucumber --format progress

..F

(:) failed steps (:)

expected: "4"
      got: "" (using ==) (RSpec::Expectations::ExpectationNotMetError)
./features/step_definitions/calculator_steps.rb:16
features/adding.feature:6

Failing Scenarios:
cucumber features/adding.feature:3 # Scenario: Add two numbers

1 scenario (1 failed)
3 steps (1 failed, 2 passed)
0m0.587s
```

Great! Now our test is failing for exactly the right reason: it's running our program, examining the output, and telling us just what the output *should* look like. This is a natural point to pause for a break. We've done the hard work for this release: when we come back to this code, Cucumber will be telling us exactly what we need to do to our program to make it work. If only all our requirements came ready-rolled with a failing test like this, building software would be easy!

Try This

Can you write an implementation of calc.rb that makes the scenario pass? Remember at this stage, we have only a single scenario to satisfy, so you might be able to get away with quite a simple solution.

We'll show you our solution in the next section.

2.8 Making It Pass

So, now that we have our solid failing Cucumber scenario in place, it's time to let that scenario drive out a solution.

1. *The RSpec Book* [CADH09]



Joe asks:

I Feel Weird: You're Making Tests Pass but Nothing Works!

We've implemented a step that calls the calculator program and passes, even though the "program" is just an empty file. What's going on here?

Remember that a step isn't a test in itself. The test is the whole scenario, and that isn't going to pass until all of its steps do. By the time we've implemented all of the step definitions, there's only going to be one way to make the whole scenario pass, and that's to build a calculator that can perform additions!

When we work outside-in like this, we often use temporary *stubs* like the empty calculator program as placeholders for details we need to fill in later. We know that we can't get away with leaving that as an empty file forever, because eventually Cucumber is going to tell us to come back and make it do something useful in order to get the whole scenario to pass.

This principle, *deliberately doing the minimum useful work the tests will let us get away with*, might seem lazy, but in fact it's a discipline. It ensures that we make our tests thorough: if the test doesn't drive us to write the right thing, then we need a better test.

There is a very simple solution that will make the test pass, but it's not going to get us very far. Let's try it anyway, for fun:

[Download first_taste/08/calc.rb](#)

```
print "4"
```

Try it. You should see the scenario pass at last:

...

```
1 scenario (1 passed)
3 steps (3 passed)
0m0.479s
```

Hooray! So, what's wrong with this solution? After all, we already said that we want to do the minimum work that the tests will let us get away with, right?

Actually, that's not quite what we said. We said we want to do the minimum *useful* work that the tests will let us get away with. What we've done here might have made the test pass, but it isn't very useful. Apart from the fact that it certainly isn't going to function as a calculator yet, let's look at what we've missed out on testing with our smarty-pants one-liner solution:

- We haven't tried to read the input from the command line.
- We haven't tried to add anything up.

In *Crystal Clear: A Human-Powered Methodology for Small Teams* [Coc04], Alistair Cockburn advocates building a *walking skeleton* as early as possible in a project to try to flush out any potential problems with your technology choices. Obviously, our calculator is trivially simple, but it's still worth considering this principle: why don't we build something more useful that passes this scenario *and* helps us learn more about our planned implementation?

If you're unconvinced by that argument, try looking at it as a problem of duplication. We have a hard-coded value of 4 in two places: once in our scenario and once in our calculator program. In a more complex system, this kind of duplication might go unnoticed, and we'd have a brittle scenario.

Let's force ourselves to fix this, using what Kent Beck calls *triangulation* (*Test Driven Development: By Example* [Bec02]). We'll add another scenario to our feature, using a new keyword called a Scenario Outline:

[Download first_taste/09/features/adding.feature](#)

Feature: Adding

Scenario Outline: Add two numbers
Given the input "<input>"
When the calculator is run
Then the output should be "<output>"

Examples:

input	output
2+2	4
98+1	99

We've turned our scenario into a Scenario Outline, which lets us specify multiple scenarios using a table. We could have copied and pasted the whole scenario and just changed the values, but we think this is a more readable way of expressing the examples, and we want to give you a taste of what's possible with Gherkin's syntax. Let's see what the output looks like now:

\$ **cucumber**

Feature: Adding

Scenario Outline: Add two numbers
Given the input "<input>"
When the calculator is run
Then the output should be "<output>"

Examples:

```

| input | output |
| 2+2 | 4      |
| 98+1 | 99    |
expected: "99"
     got: "4" (using ==) (RSpec::Expectations::ExpectationNotMetError)
./features/step_definitions/calculator_steps.rb:15
features/adding.feature:6

Failing Scenarios:
cucumber features/adding.feature:3

2 scenarios (1 failed, 1 passed)
6 steps (1 failed, 5 passed)
0m1.035s

```

We can see from the summary 2 scenarios (1 failed, 1 passed) that Cucumber has run two scenarios. Each row in the Examples table is expanded into a scenario when Cucumber runs the scenario outline. The first example—where the result is 4—still passes, but the second example is failing.

Now it definitely makes sense to reimplement our program with a more realistic solution:

[Download first_taste/10/calc.rb](#)

```
print eval(ARGV[0])
```

First we read the command-line argument ARGV[0] and send it to Ruby's eval method. That should be able to work out what to do with the calculation input. Finally, we print the result of eval to the console.

Try that. Do both scenarios pass? Great! You've just built your first program with Cucumber.

2.9 What We Just Learned

We've taken a quick skim over a lot of different things in this chapter, all of which will be covered again in more detail later. Let's recap and highlight some of the most important points.

Directory Structure

Cucumber likes you to use a conventional directory structure for storing your features and step definitions:

```
features/
  adding.feature
  ...
  step_definitions/
    calculator_steps.rb
  ...

```

You can override this structure, if you really want to, by passing arguments to Cucumber, but this is the conventional and simplest way to store your files.

Baby Steps

As we progressed through the example, did you notice how often we ran cucumber?

One of the things we love about working outside-in with Cucumber is how it helps us to stay focused. We can let Cucumber guide us through the work to be done, leaving us free to concentrate on creating an elegant solution. By running Cucumber every time we make a change, any mistakes we make are found and resolved quickly, and we get plenty of feedback and encouragement about our progress.

Gherkin

Cucumber tests are expressed using a syntax called Gherkin. Gherkin files are plain text and have a .feature extension. We'll talk more about Gherkin in [Chapter 3, Gherkin Basics, on page 25](#).

Step Definitions

Step definitions are the Ruby glue that binds your Cucumber tests to the application you're testing. When the whole thing plays together, it looks a bit like [Figure 2, The main layers of a Cucumber test suite, on page 15](#).

You'll learn more about step definitions in [Chapter 4, Step Definitions: From the Outside, on page 39](#).

After that whistle-stop tour of Cucumber's features, we're going to slow down and get into a bit more depth. We'll work our way in through the layers over the next few chapters, starting with a look at Gherkin, the language we use to write Cucumber features.

Try This

See whether you can add a second feature, division.feature, using the other example from the start of this chapter. Do you need to change the solution to make it pass?

Gherkin Basics

Now that you've gained some confidence with how Cucumber works, it's worth stepping back for a few moments and doing a little studying. We're going to look at Gherkin, the language we use for writing Cucumber features.

By the end of this chapter you'll understand how to write specifications for your software that can be both read by your stakeholders and tested by Cucumber. You'll learn what each of the Gherkin keywords does and how they all fit together to make readable, executable Cucumber specifications. You won't know how to run the tests yet, but you'll be ready to write them.

3.1 What's Gherkin For?

When we build software for people (let's call them *stakeholders*), it's notoriously difficult to figure out exactly what they want us to build. In his famous essay, *No Silver Bullet* [Bro95], Fred Brooks says:

"The hardest single part of building a software system is deciding precisely what to build."

We've all worked on projects where, because of a misunderstanding, code that we'd worked hard on for several days or more had to be thrown away. Better communication between developers and stakeholders is essential to help avoid this kind of wasted time. One technique that really helps facilitate this communication is the use of *concrete examples* to illustrate what we want the software to do.

Concrete Examples

By using real-world examples to describe the desired behavior of the system we want to build, we stay grounded in language and terminology that makes sense to our stakeholders: *we're speaking their language*. When we talk in terms of these examples, they can really imagine themselves using the system,

and that means they can start to give us useful feedback and ideas before a line of code has been written.

To illustrate this, let's imagine you're building a credit card payment system. One of the requirements is to make sure users can't enter bad data. Here's one way of expressing that:

Customers should be prevented from entering invalid credit card details.

This is an example of what agile teams often call *acceptance criteria* or *conditions of satisfaction*.¹ We use the word *acceptance* because they tell us what the system must be able to do in order for our stakeholders to find it acceptable.

The previous requirements statement is useful, but it leaves far too much room for ambiguity and misunderstanding. It lacks *precision*. What exactly makes a set of details invalid? How exactly should the user be prevented from entering them? We've seen too many projects get dragged into the tar pit² by these kind of worthy but vague statements. Let's try illustrating this requirement with a concrete example:

If a customer enters a credit card number that isn't exactly 16 digits long, when they try to submit the form, it should be redisplayed with an error message advising them of the correct number of digits.

Can you see how much more specific this second statement is? As a developer implementing this feature, we know almost everything we need to be able to sit down and start working on the code. As a stakeholder, we have a much clearer idea of what the developer is going to build.

In fact, a stakeholder reading this might point out that there are certain types of cards that are valid with less than 16 digits and give us another example. This is the real power of examples: they stimulate our imagination, enabling us to explore and discover edge cases we might otherwise have found much later.

By giving an example to illustrate our requirement, we've turned an acceptance criterion into an *acceptance test*. Now we have something unambiguous that we can use to test the behavior of the system, either manually or using an automated test script.

1. *Agile Estimating and Planning* [Coh05]

2. The tar pit metaphor comes from the seminal book by Fred Brooks, *The Mythical Man Month: Essays on Software Engineering* [Bro95].

Try This

Think about a feature you're working on right now or have worked on recently. Can you write down three concrete examples of the behavior needed for that feature to be acceptable?

Executable Specifications

Another advantage of using concrete examples is that they're much easier to validate against the running system than vague requirement statements. In fact, if we're neat and tidy about how we express them, we can get the computer to check them for us. We call this *automated acceptance testing*.³

The challenge with writing good automated acceptance tests is that, for them to be really effective, they need to be readable by not only the computer but also by our stakeholders. It's this human readability that allows us to get feedback about what we're building while we're building it. This is where Gherkin comes in.

Gherkin gives us a lightweight structure for documenting examples of the behavior our stakeholders want, in a way that it can be easily understood both by the stakeholders and by Cucumber. Although we can call Gherkin a programming language,⁴ its primary design goal is human readability, meaning you can write automated tests that read like documentation. Here's an example:

[Download gherkin_basics/sample.feature](#)

Feature: Feedback when entering invalid credit card details

In user testing we've seen a lot of people who made mistakes entering their credit card. We need to be as helpful as possible here to avoid losing users at this crucial stage of the transaction.

Background:

Given I have chosen some items to buy
And I am about to enter my credit card details

Scenario: Credit card number too short

When I enter a card number that's only 15 digits long
And all the other details are correct
And I submit the form
Then the form should be redisplayed
And I should see a message advising me of the correct number of digits

-
3. *Extreme Programming Explained: Embrace Change* [Bec00]
 4. A note for the pedantic reader: the Gherkin language does have a grammar enforced by a parser, but the language is not Turing Complete.

Scenario: Expiry date invalid

When I enter a card expiry date that's in the past

And all the other details are correct

And I submit the form

Then the form should be redisplayed

And I should see a message telling me the expiry date must be wrong

One interesting feature of Gherkin's syntax is that it is not tied down to one particular spoken language. Each of Gherkin's keywords has been translated into more than forty different spoken languages, and it is perfectly valid to use any of them to write your Gherkin features. No matter if your users speak Norwegian or Spanish, med Gherkin kan du beskrive funksjonalitet i et språk de vil forstå.⁵ Tocino grueso!⁶ More on that later.

3.2 Format and Syntax

Gherkin files use the .feature file extension. They're saved as plain text, meaning they can be read and edited with simple tools. In this respect, Gherkin is very similar to file formats like Markdown, Textile, and YAML.

Keywords

A Gherkin file is given its structure and meaning using a set of special keywords. There's an equivalent set of these keywords in each of the supported spoken languages, but for now let's take a look at the English ones:

- Feature
- Background
- Scenario
- Given
- When
- Then
- And
- But
- *
- Scenario Outline
- Examples

We'll spend the rest of this chapter exploring how to use the most common of these keywords, which will be enough to get you started writing your own

5. Gherkin lets you to write your features in a language they will understand.

6. Chunky Bacon!

Cucumber features. We'll come back to look at the remaining keywords later in [Chapter 5, Expressive Scenarios, on page 61](#).

Dry Run

All of the examples in this chapter are valid Gherkin and can be parsed by Cucumber. If you want to play around with them as we go through the chapter, just create a test.feature working file. Then run it with the following:

```
$ cucumber test.feature --dry-run
```

The --dry-run switch tells Cucumber to parse the file without executing it. It will tell you if your Gherkin isn't valid.

3.3 Feature

Each Gherkin file begins with the Feature keyword. This keyword doesn't really affect the behavior of your Cucumber tests at all; it just gives you a convenient place to put some summary documentation about the group of tests that follow.

Here's an example:

```
Feature: This is the feature title
  This is the description of the feature, which can
  span multiple lines.
  You can even include empty lines, like this one:
```

In fact, everything until the next Gherkin keyword is included in the description.

The text immediately following on the same line as the Feature keyword is the *name* of the feature, and the remaining lines are its *description*. You can include any text you like in the description except a line beginning with one of the words Scenario, Background, or Scenario Outline. The description can span multiple lines. It's a great place to wax lyrical with details about who will use the feature, and why, or to put links to supporting documentation such as wireframes or user research surveys.

It's conventional to name the feature file by converting the feature's name to lowercase characters and replacing the spaces with underscores. So, for example, the feature named User logs in would be stored in user_logs_in.feature.

In valid Gherkin, a Feature must be followed by one of the following:

- Scenario
- Background
- Scenario Outline

Although Background and Scenario Outline are handy keywords to know once you've written a few scenarios, we don't need to worry about them just yet. They're covered later in [Chapter 5, Expressive Scenarios, on page 61](#). Right now all we need is the Scenario.

3.4 Scenario

To actually express the behavior we want, each feature contains several scenarios. Each scenario is a single concrete example of how the system should behave in a particular situation. If you add together the behavior defined by all of the scenarios, that's the expected behavior of the feature itself.

When Cucumber runs a scenario, if the system behaves as described in the scenario, then the scenario will pass; if not, it will fail. Each time you add a new scenario to your Cucumber test suite and make it pass, you've added some new functionality to the system, and that's time for a high-five.

Each feature typically has somewhere between five and twenty scenarios, each describing different examples of how that feature should behave in different circumstances. We use scenarios to explore edge cases and different paths through a feature.

Scenarios all follow the same pattern:

1. Get the system into a particular state.
2. Poke it (or tickle it, or ...).
3. Examine the new state.

So, we start with a *context*, go on to describe an *action*, and then finally check that the *outcome* was what we expected. Each scenario tells a little story describing something that the system should be able to do.

Given, When, Then

In Gherkin, we use the keywords Given, When, and Then to identify those three different parts of the scenario:

```
Scenario: Successful withdrawal from an account in credit
  Given I have $100 in my account # the context
  When I request $20           # the event(s)
  Then $20 should be dispensed # the outcome(s)
```

So, we use Given to set up the context where the scenario happens, When to interact with the system somehow, and Then to check that the outcome of that interaction was what we expected.

A Template for Describing a Feature

Although feature descriptions are often helpful documentation, they're not mandatory. If you're struggling to work out what to say, the following template can be a great place to start:

```
In order to <meet some goal>
As a <type of stakeholder>
I want <a feature>
```

By starting with the goal or value that the feature provides, you're making it explicit to everyone who ever works on this feature why they're giving up their precious time. You're also offering people an opportunity to think about other ways that the goal could be met. Maybe you don't actually need to build this feature at all, or you could deliver the same value in a much simpler way.

This template is known as a *Feature Injection* template, and we are grateful to Chris Matts and Liz Keogh for sharing it with us.

And, But

Each of the lines in a scenario is known as a *step*. We can add more steps to each Given, When, or Then section of the scenario using the keywords And and But:

```
Scenario: Attempt withdrawal using stolen card
Given I have $100 in my account
But my card is invalid
When I request $50
Then my card should not be returned
And I should be told to contact the bank
```

Cucumber doesn't actually care which of these keywords you use; the choice is simply there to help you create the most readable scenario. If you don't want to use And or But, you could write the previous scenario like this, and it would still work exactly the same way:

```
Scenario: Attempt withdrawal using stolen card
Given I have $100 in my account
Given my card is invalid
When I request $50
Then my card should not be returned
Then I should be told to contact the bank
```

But that doesn't read as nicely, does it?

Replacing Given/When/Then with Bullets

Some people find Given, When, Then, And, and But a little verbose. There is an additional keyword you can use to start a step: * (an asterisk). We could have written the previous scenario like this:

```
Scenario: Attempt withdrawal using stolen card
  * I have $100 in my account
  * my card is invalid
  * I request $50
  * my card should not be returned
  * I should be told to contact the bank
```

To Cucumber, this is exactly the same scenario. Do you find this version easier to read? Maybe. Did some of the meaning get lost? Maybe. It's up to you and your team how you want to word things. The only thing that matters is that everybody understands what's communicated.

Stateless

When writing scenarios, here's a really important concept you need to grasp:

Each scenario must make sense and be able to be executed independently of any other scenario.

That means you can't put some money into the account in one scenario and then expect that money to be there in the next scenario. Cucumber won't stop you from doing this, but it's extremely bad practice: you'll end up with scenarios that fail unexpectedly and are harder to understand.

This might seem a little dogmatic, but trust us, it really helps keep your scenarios simple to work with. It avoids building up brittle dependencies between scenarios and also gives you the flexibility to run just the scenarios you need to when you're working on a particular part of the system, without having to worry about getting the right test data set up. We explain these problems in depth in [Chapter 6, When Cucumbers Go Bad, on page 85](#).

When writing a scenario, always assume that it will run against the system in a default, blank state. Tell the story from the beginning, using Given steps to set up all the state you need for that particular scenario.

Name and Description

Just like a Feature, a Scenario keyword can be followed by a name and description. Normally you'll probably just use the name, but it's valid Gherkin to follow the name with a multiline description—everything up until the first Given, When, or Then will be slurped up into the description of the scenario.

Stale scenario names can cause confusion. When modifying existing scenarios (or copying and pasting them), take care to check that the name still makes sense. Since the scenario name is just documentation, Cucumber won't fail the scenario even if its name no longer has anything to do with what's actually going on in the steps. This can be really confusing for anyone reading the scenario later.

Try This

- Now that you understand how to write Gherkin scenarios, try converting some of the concrete examples you wrote down earlier for your own project into Gherkin.
- Show them to someone who knows nothing about your project, and ask them what they think your application does.
- Practice describing what you're doing with Given/When/Then while you're doing everyday things such as starting your car, cooking breakfast, or switching channels on the TV. You'll be surprised how well it fits.

3.5 Comments

As well as the description fields that follow Feature and Scenario keywords, Cucumber allows you to precede these keywords with comments.

Like in Ruby, comments start with a # character. Unlike in Ruby, comments have to be the first and only thing on a line (well, apart from whitespace).

Here's an example:

```
Download gherkin_basics/comments_example.feature
# This feature covers the account transaction and hardware-driver modules
Feature: Withdraw Cash
  In order to buy beer
  As an account holder
  I want to withdraw cash from the ATM

  # Can't figure out how to integrate with magic wand interface
  Scenario: Withdraw too much from an account in credit
    Given I have $50 in my account
    # When I wave my magic wand
    And I withdraw $100
    Then I should receive $100
```

You can also put comments within a scenario. The most common use for this is to comment out a step, as we've shown in the previous example.

As in any programming language, comments can quickly go stale and become meaningless or downright confusing. When this happens, the comment



Matt says:

Take Care with Your Naming Scenarios

Even though they can't make your tests pass or fail, scenario names are surprisingly important to get right. Here are some reasons why it's a good idea to pay attention to them:

- When your tests break, it's the failing scenario's name that will give you the headline news on what's broken. A concise, expressive name here can save everyone a lot of time.
- Once you have a few scenarios in a feature file, you don't want to have to read the detail of the steps unless you really need to do so. If you're a programmer, think of it a bit like method naming. If you name the method well, you won't need to read the code inside it to work out what it does.
- As your system evolves, your stakeholders will quite often ask you to change the expected behavior in an existing scenario. A well-composed scenario name will still make sense even if you add an extra `Then` step or two.

A good tip is to avoid putting anything about the outcome (the `Then` part) of the scenario into the name and concentrate on summarizing the context and event (Given and When) of the scenario.

causes more harm than good. We advise you to use them as sparingly as you can and put the important stuff into scenarios where it can be tested.

3.6 Spoken languages

Remember earlier we said that you can write your Gherkin features in the spoken language of your stakeholders? Here's how.

Putting a `# language: no` language comment on the first line of a feature file tells Cucumber which spoken language that feature is written in. If you omit this header, Cucumber will default to English, as you've already seen.

Here's an example of a feature written in Norwegian:

```
# language: no
Egenskap: Summering
For å unngå at firmaet går konkurs
Må regnskapsførerere bruke en regnemaskin for å legge sammen tall

Scenario: to tall
Gitt at jeg har tastet inn 5
Og at jeg har tastet inn 7
Når jeg summerer
Så skal resultatet være 12
```



Joe asks:

Why Do We Have Both Comments and Descriptions? Which One Should I Use and When?

Here's how we think about this: the description, which you can put after each keyword, is part of the structured Gherkin document and is the right place to put documentation for your stakeholders.

Comments, on the other hand, can be used more to leave notes for testers and programmers who are working with the features. Think of a comment as something more temporary, a bit like a sticky note.

Don't forget that programmers and testers need documentation too. If there are technical details that need to be documented with the feature, you should feel free to put them into the description too, provided that the business-facing members of your team are comfortable with it being there.

If you're wondering whether Cucumber knows how to speak your language, you can ask it for the list of all valid languages with this command:

```
$ cucumber --i18n help
```

When you're working with a particular language, you can discover the keywords by passing the language code (as listed by the previous command) to the `--i18n` switch. Here's Japanese, for example:

```
$ cucumber --i18n ja
```

One of the greatest benefits of working with a tool like Cucumber is the conversations you have with your stakeholders as you write the scenarios. These conversations can help you find gaps and misunderstandings that otherwise might have emerged only after you'd spent days or even weeks working on the code. So, even if you never run the tests, just writing them can help make you go faster.

3.7 What We Just Learned

Let's review what we've talked about in this chapter:

- We saw how the core Gherkin keywords Feature, Scenario, Given, When, and Then can be used to describe the behavior your stakeholders want as concrete examples.
- There is a fundamental pattern to each Gherkin scenario, with a context (Given), an event (When), and an outcome (Then).



Joe asks:

So, Can I Mix Spoken languages in My Features?

A project can have a mix of features that use different spoken languages, yes. However, remember that the setting is for a *feature*, so all the scenarios in a feature have to use the same spoken language.

- Each scenario must be capable of being run on its own and should not depend on data set up by other scenarios. This means every scenario has to include enough Given steps to set up all the data it needs.
- You can add descriptions and comments to your .feature files to turn them into useful documentation of your system.
- Using the #language: header, you can write your features in different spoken languages.

At this point you have all the knowledge you need to get started writing your own Gherkin features. Even though there are some keywords we haven't covered yet, there's a huge amount of value in what you already know. Just pretend you have a machine that can turn your Cucumber scenarios into perfect working code, and play the game of working with your team to create the best descriptions you can of what you want the software to do.

In the next chapter, we'll start to explore step definitions, the layer beneath the Gherkin features where you interact with your application, and bring your scenarios to life.

Try This

Here are some exercises for you to try.

Practice Given/When/Then

Let's build a robot! Here's a scenario:

Scenario: Tickle a happy robot
Given I am in a good mood
When you tickle me
Then I will giggle

What happens if you change the context in which this scenario happens? Write another scenario where the first step puts the robot into a bad mood. Leave the action the same. What will be the outcome when you tickle a grumpy robot?

Now try changing the action:

Scenario: Attack a happy robot
Given I am in a good mood
When you kick me in the shins
Then I will ...

What will the outcome be now?

Your First Scenario

Write the first formal Gherkin scenario for your project. Pick a feature you're working on right now, and try to describe the way the system should behave when you're done. Notice when you have questions about the precise language you should use, or the precise behavior, and write down those questions. Try to get some time with the right people on your team to answer those questions. Show them the scenario: does it make sense to them? How would they have worded it?

Step Definitions: From the Outside

Now that you know how to use Gherkin to describe *what* you want your tests to do, the next task is to tell them *how* to do it. Whether you choose to drive your acceptance tests from Cucumber scenarios or simple Test::Unit scripts, there's no escaping the fact that you're going to need to write some code eventually. It's about that time.

Step definitions sit right on the boundary between the business's domain and the programmer's domain. They're written in Ruby,¹ and their responsibility is to translate each plain-language step in your Gherkin scenarios into concrete actions in Ruby code. As an example, take this step from the ATM scenario in the previous chapter:

Given I have \$100 in my Account

The step definition for this step needs to make the following things happen:

- Create an account for the protagonist in the scenario (if there isn't one already).
- Set the balance of that account to be \$100.

How exactly those two goals are achieved depends a great deal on your specific application. Automated acceptance tests generally try to simulate user interactions with the system, and the step definitions are where you'll tell Cucumber how you want it to poke around with your system. That might involve clicking buttons on a user interface or reaching beneath the covers and directly inserting data into a database, writing to files, or even calling a web service. We think of step definitions themselves as distinct from the

1. In fact, Cucumber supports step definitions written in many other programming languages too, but we're concentrating on Ruby in this book.

automation code that does the actual poking so that the layers separate out like [Figure 3, Step definitions are translators, on page 41](#).

There are two sides to a step definition. On the outside, it translates from plain language into Ruby, and on the inside it tells your system what to do using Ruby automation code. Ruby has an incredibly rich set of libraries for automating a whole variety of systems, from JavaScript-heavy web applications to REST web services. We're not going to show you how to use all of those libraries in this chapter; that will come later in the book. Here we're going to concentrate on the main responsibility of this layer of your Cucumber tests, which is to interpret a plain-language Gherkin step and decide what to do.

We're going to start by explaining some of the mechanics of how step definitions match up to plain-language steps and then work through an example of how to write a single step definition that can handle many different steps. We'll finish off by explaining how Cucumber executes step definitions and deals with their results. When we're done, you should understand enough to start writing and running your own step definitions.

4.1 Steps and Step Definitions

Let's start by clarifying the distinction between a *step* and a *step definition*.

Each Gherkin scenario is made up of a series of steps, written in plain language. On its own, a step is just documentation; it needs a step definition to bring it to life. A step definition is a piece of Ruby code that says to Cucumber, "If you see a step that looks like this..., then here's what I want you to do...."

When Cucumber tries to execute each step, it looks for a matching step definition to execute. So, how does Cucumber match a step definition to a step?

Matching a Step

Gherkin steps are expressed in plain text. Cucumber scans the text of each step for patterns that it recognizes, which you define using a *regular expression*. If you haven't used regular expressions before, then just think of them like a slightly more sophisticated version of the wildcards you'd use to search for a file. Although they can look intimidating at first, there are actually only a small number of patterns you need to get a great deal of mileage out of them. All of those patterns will be covered in this chapter; if you're already well familiar with regular expressions, you might want to skim over the next few sections, up until [Section 4.5, Returning Results, on page 52](#).

Let's take the ATM example from the previous chapter:

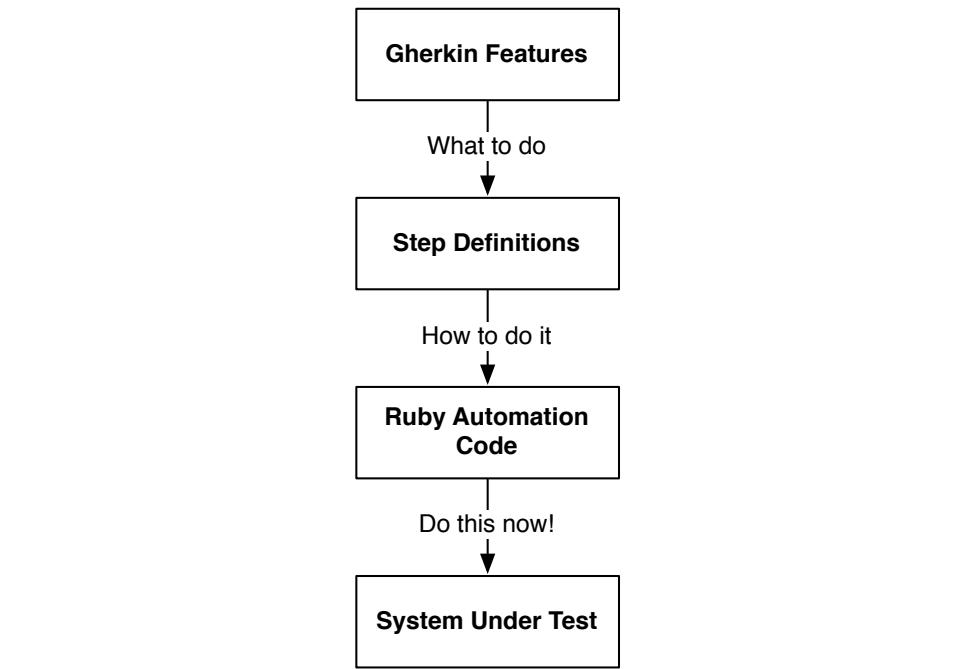


Figure 3—Step definitions are translators.

[Download step_definitions/intro/features/cash_withdrawal.feature](#)

Feature: Cash withdrawal

Scenario: Successful withdrawal from an account in credit

Given I have \$100 in my account

When I request \$20

Then \$20 should be dispensed

As Cucumber executes this feature, it will come to the first step of the scenario, Given I have \$100 in my Account and say to itself, *Now, do I have any step definitions that match the phrase I have \$100 in my Account?*

A simple regular expression that will match this step would look like this:

/I have \\$100 in my Account/

In Ruby, regular expressions are denoted by those forward slashes. Notice that we've had to escape the dollar sign with a backslash. That's because the dollar sign can have a special meaning in a regular expression, but in this case we want to interpret it literally. We'll come back to these special characters a bit later in the chapter.



Aslak says: Avoiding Regular Expressions

Cucumber allows you to define step definitions using strings instead of regular expressions. This might seem simpler at first, but it has other problems, which I'll illustrate with an example. Here is a step definition that uses a plain string:

```
Given "I have 100 in my Account" do
end
```

We couldn't write \$100 here, because the \$ has a special meaning when you define step definitions with strings. Any \$ signs—including any letter or number following it—will be interpreted as an argument, which you will learn more about in [Section 4.2, Capturing Arguments, on page 45](#). This step definition uses an argument:

```
Given "I have $amount in my Account" do |amount|
end
```

This step definition would match both of the following Gherkin steps:

```
Given I have 100 in my Account
Given I have $100 in my Account
```

In the first case, the step definition's amount argument would have the value "100". In the second case, it would have the value "\$100". If our step definition expects a string with only digits, this can be problematic. We have no way to enforce a consistent way to write Gherkin steps, and the step definitions have to anticipate many kinds of input.

This is why using strings instead of regular expressions is not as advantageous as you might think. They give far less control over what gets matched and what arguments a step definition can receive.

If Cucumber sees a step definition with this regular expression, it will execute it when it comes to the first step of our scenario. So, how do we create a step definition?

Creating a Step Definition

Step definitions live in regular Ruby files in the features/step_definitions folder. To create a step definition, you can use the special Cucumber method Given, like this:

```
Given /I have \$100 in my Account/ do
  # TODO: code that puts $100 into User's Account goes here
end
```

You'll typically put several step definitions like this together in the same Ruby file. Cucumber will load all the files in the features/step_definitions directory, so it's really up to you how exactly you want to organize them. We suggest

keeping a separate file per domain entity so that step definitions that work with similar parts of the system are kept together.

Let's examine the step definition in detail. This is a Ruby file, and we're calling the special Cucumber method `Given`, which tells Cucumber that we want to register a step definition. We pass the `Given` method two things: a regular expression to match one or more steps (the bit between the forward slashes) and a block of Ruby code (the bit between the `do...end`) to execute if it does. Cucumber stores the code block for later to execute if it comes across a matching step.

You can also use the methods `When` or `Then` to create a step definition in just the same way.

Given, When, Then Are the Same

It doesn't actually matter which of the three methods you use to register a step definition, because Cucumber ignores the keyword when matching a step. Under the hood, all of the keyword methods are aliases for `Cucumber::RbDsl#register_rb_step_definition`. The keywords are really just there for extra documentation to help you express the intent of each step or step definition.

This means that, whether it was created with the method `Given`, `When`, or `Then`, a step definition will match any Gherkin step as long as the regular expression matches the main text of the step. [Figure 4, Cucumber ignores keywords when matching a step, on page 44](#) highlights what Cucumber sees when it scans a scenario for matching step definitions.

This flexibility can be really handy, as we'll show you later, but there is one gotcha to watch out for. Let's take a look at an example.

Imagine you have already implemented your ATM withdrawal scenario, including writing a step definition for `Given I have $100 in my Account`. So, you have a step definition that matches the text `I have $100 in my Account` and creates an account with `$100` in it. A few weeks later that scenario is a dim and distant memory, and you get a new requirement to give all new accounts a `$1` gift. You sit down with your domain expert and write the following scenario:

```
Scenario: New accounts get a $1 gift
Given I have a brand new Account
And I deposit $99
Then I have $100 in my Account
```

That looks reasonable, doesn't it? We set up the new account, deposit some money, and then check that the new balance is what we'd expect it to be. But

Feature: Cash Withdrawal

Scenario: Attempt withdrawal using stolen card

Given I have \$100 in my account

But my card is invalid

When I request \$50

Then my card should not be returned

And I should be told to contact the bank

Figure 4—Cucumber ignores keywords when matching a step.

can you see what's going to happen if we run this new scenario together with our original ATM withdrawal scenario?

Let's look at our original step definition again:

```
Given /I have \$100 in my Account/ do
  # TODO: code that puts $100 into User's Account goes here
end
```

Now that we've learned that Cucumber ignores the Given/When/Then keyword when matching a step, we can see that this original step definition is also going to match the last step of our new scenario, Then I have \$100 in my Account. Surprise! We expected that step to check the balance of the account, but instead it's going to put \$100 *into* the account!

You obviously need to be careful in this situation, because we could easily have had a scenario that was giving us a *false positive*: passing when it should have been failing. It might not seem like it, but Cucumber's flexibility has actually helped us here, by exposing some quite subtle ambiguity in the language used in each of the steps. The best way we've found to avoid this kind of problem is to pay careful attention to the precise wording in your steps. You could change both steps to be less ambiguous:

```
Given I have deposited $100 in my Account
Then the balance of my Account should be $100
```

By rewording the steps like this, you've made them better at communicating exactly what they will do when executed. Learning to spot and remove this kind of ambiguity is something that takes practice. Paying attention to the distinction in wording between two steps like this can also give you hints about concepts that may not be expressed in your code but need to be. It might seem pedantic, but we've found that teams who pay this much careful attention to detail write much better software, faster.

Speaking in Tongues

If you're using a different spoken language than English in your features, you can still use the same language when registering step definitions. Cucumber creates an alias of Cucumber::RbDsl#register_rb_step_definition for every spoken language, so a team working in Spain, for example, could use the following:

```
Dado /tengo \$100 en mi Cuenta/ do
  # TODO: code that puts $100 into User's Account goes here
end
```

Remember, you can discover the keywords for your spoken language using the command cucumber -i18n <language>.

4.2 Capturing Arguments

You'll notice that in the step we've been using as an example, we've talked about the sum of \$100 the whole time. What if we had another scenario where we needed to deposit a different amount of money into the account? Would we need another step definition, like this?

```
Given /I have deposited \$100 in my Account/ do
  # TODO: code goes here
end
```

```
Given /I have deposited \$250 in my Account/ do
  # TODO: code goes here
end
```

Happily, we don't. This is where the flexibility of regular expressions comes into play. We can use two of regular expressions' most useful features here to capture any dollar amount as an *argument* to the step definition. Those features are *capture groups* and *wildcards*.

Capture Groups

When you surround part of a regular expression with parentheses, it becomes a capture group. Capture groups are used to highlight particular parts of a pattern that you want to lift out of the matching text and use. In a Cucumber step definition, the text matched within each capture group is passed to the code block as an argument:

```
Given /I have deposited $(100) in my Account/ do |amount|
  # TODO: code goes here
end
```

Here the block argument amount will receive the string value 100 when this step definition matches. The previous example is a bit silly, because this

regular expression is still only ever going to match steps that talk about the amount of \$100. We need to use a wildcard inside the capture group to open it up to other values.

Alternation

There are a few different ways to specify a wildcard in a regular expression. One of the simplest is *alternation*, where we express different options separated by a pipe character |, like this:

```
Given /I have deposited \$(100|250) in my Account/ do |amount|
  # TODO: code goes here
end
```

This step definition will now match a step with either of the two values 100 or 250 in it, and the number will be captured and passed to the block as an argument. Alternation can be useful if there are a fixed set of values that you want to accept in your step definition, but normally you'll want something a little looser.

The Dot

The dot is a *metacharacter*, meaning it has magical powers in a regular expression. Literally, a dot means *match any single character*. So, we can try this instead:

```
Given /I have deposited \$(...) in my Account/ do |amount|
  # TODO: code goes here
end
```

That will now match a step with any three-figure dollar sum and send the matched amount into the block. This is definitely a step in the right direction, but there are a couple of problems with what we've done. For one, remember that the dot matches *any character*, so we could end up capturing letters in here instead of numbers. More importantly, what if we wanted a step that deposits just \$10 in the account, or \$1,000? This step definition won't match those steps because it's always looking for three characters. We can fix this by using a *modifier*.

The Star Modifier

In regular expressions, a *repetition modifier* takes a character (or metacharacter) and tells us how many times over it can appear. The most flexible modifier is the star:



Joe asks:

What If I Actually Want to Match a Dot?

Any of the metacharacters like the dot can be escaped by preceding them with a backslash. So, if you wanted to specifically match, say 3.14, you could use /3\.\14/.

You might have noticed that there's a backslash in front of the dollar amount in the step definition we're using. That's because \$ itself is a metacharacter (it's an anchor, which we'll explain later), so we need to escape to make it match a normal dollar sign.

```
Given /I have deposited \$(..) in my Account/ do |amount|
  # TODO: code goes here
end
```

The star modifier means *any number of times*. So, with .* we're capturing *any character, any number of times*. Now we're getting somewhere. This will definitely allow us to capture all those different amounts. But there's still a problem.

The star modifier is a bit of a blunt instrument. Because we're using it with the dot that matches *any character*, it will gobble up any text at all up until the phrase in my Account. This is why, in regex terminology, the star modifier is known as a *greedy* operator. For example, it would happily match this step:

```
Given I have deposited $1 and a cucumber in my Account
```

Ridiculously, the amount argument captured by our regular expression in this case would be 1 and a cucumber. We need to be a bit more specific about the characters we want to match so that we just capture numbers. Instead of a dot, we can use something else.

Character Classes

Character classes allow you to tell the regular expression engine to match one of a range of characters. You just place all of the characters you would accept inside square brackets:

```
Given /I have deposited \$([0123456789]*) in my Account/ do |amount|
  # TODO: code goes here
end
```

For a continuous range of characters like we have, you can use a hyphen:

```
Given /I have deposited \$([0-9]*) in my Account/ do |amount|
  # TODO: code goes here
end
```

Now we've restricted the character we'll accept to be numeric. We're still modifying the character with the star to accept any number of them, but we're now being specific that we'll accept only a continuous string of numbers.

Shorthand Character Classes

For common patterns of characters like [0-9], there are a few *shorthand character classes* that you can use instead. You may find this just makes your regular expressions more cryptic, but there are only a few to learn. For a digit, you can use \d as a shorthand for [0-9]:

```
Given /I have deposited \$\d*/ in my Account/ do |amount|
  # TODO: code goes here
end
```

See [Useful Shorthand Character Classes, on page 49](#) for more examples.

There's just one last problem with this. Can you see what it is?

The Plus Modifier

The star is one example of a repetition modifier, but there are others. A subtle problem with the star is that *any number of times* can mean zero.

So, this step would match:

```
Given I have deposited $ in my Account
```

That's no good. To fix this, we can use the + modifier, which means *at least once*:

```
Given /I have deposited \$\d+/ in my Account/ do |amount|
  # TODO: code goes here
end
```

There we go. We took a rambling route to get to the answer, but on the way we've visited almost every one of the features of regular expressions that are useful to us when building Cucumber step definition. There are only a couple more to cover.

Try This

Imagine you're building a system for airport departure lounge screens. You need to be able to capture examples of flight codes from the Cucumber scenarios. Can you write a single step definition that can capture the flight codes from all of these steps?

Useful Shorthand Character Classes

Here are the most useful shorthand character classes:

- \d stands for *digit*, or [0-9].
- \w stands for *word character*, specifically [A-Za-z0-9_]. Notice that underscores and digits are included but not hyphens.
- \s stands for *whitespace character*, specifically [\t\r\n]. That means a space, a tab, or a line break.
- \b stands for *word boundary*, which is a lot like \s but actually means the opposite of \w. Anything that is not a word character is a word boundary.

You can also negate shorthand character classes by capitalizing them, so for example, \D means *any character except a digit*.

```
Given the flight EZY4567 is leaving today
```

...

```
Given the flight C038 is leaving today
```

...

```
Given a flight BA01618 is leaving today
```

Start by writing a step definition that works for the first step, and then make it more and more generic so that it works with the other steps too.

4.3 Multiple Captures

You don't have to stop at capturing just a single argument. Cucumber will pass an argument to your block for every capture group in your regular expression, so you can grab as many details as you like from a step.

Here's an example. Let's imagine our bank wants to start offering its customers savings accounts as well as their regular checking account. Customers can use the ATM to transfer money between their accounts. Here's one of the scenarios for this new feature:

Scenario: Transfer funds from savings into checking account

Given I have deposited \$10 in my Checking Account

And I have deposited \$500 in my Savings Account

When I transfer \$500 from my Savings Account into my Checking Account

Then the balance of the Checking Account should be \$510

And the balance of the Savings Account should be \$0

Let's try to write a step definition that can handle the first two steps. As well as the amount deposited, we need to capture the type of account so that we know where to put it.

We can use a modified version of the regular expression we used previously to capture the type of account:

```
Given /I have deposited \$(\d+) in my (\w+) Account/ do |amount, account_type|
  # TODO: code goes here
end
```

We use the shorthand character class `\w`, modified with the plus to mean *any word character, at least once*, effectively capturing a single word. That word is then passed to the block in the second argument, which we've named `account_type`.

Try This

Write a step definition for the next step in the scenario, When I transfer \$500 from my Savings Account into my Checking Account. The step definition should capture three arguments:

- The amount of money being transferred
- The type of account being debited in the transfer
- The type of account that receives the credit in the transfer

Test it by writing simple puts statements in your step definition to print the value captured in each argument to the console.

4.4 Flexibility

The readability of Cucumber features helps teams learn to use a ubiquitous language when talking about the system they're building. This is a really important benefit, because that consistent use of terminology helps reduce misunderstandings and allow communication to flow more smoothly between everyone on the team.

So, we want to encourage our feature authors to be consistent about the nouns and verbs they use in the Cucumber features, because it helps make features that can be readily understood by anyone on the team. Equally, we also want feature authors to be able to express themselves as naturally as possible, which means they may often use slightly different phrasing to mean exactly the same thing. This is fine; in fact, it's to be encouraged. Cucumber features are all about communicating with business users in their language, and it's important that we don't force them to sound like robots.

To keep the features readable and natural, it's useful to develop the skill to make your step definitions flexible enough to match the different ways something might be expressed by a feature author. This isn't as hard as you might think.

The Question Mark Modifier

When matching business-facing Gherkin text, you'll often want to indicate that you don't care about the odd character in your match, such as when a word could be singular or plural:

```
Given I have 1 cucumber in my basket
Given I have 256 cucumbers in my basket
```

Like the star and the plus, the question mark modifies the character that precedes it, specifying how many times it can be repeated. The question mark modifier means *zero or one times*; in other words, it makes the preceding character optional. In step definitions, it's particularly useful for plurals:

```
Given /I have (\d+) cucumbers? in my basket/ do |number|
  # TODO: code goes here
end
```

By putting a question mark after the s in cucumbers, we're saying that we don't care whether the word is singular or plural. So, this step definition will match both of the previous steps.

You'll find that the question mark comes in handy to knock off some rough edges from your step definitions and make them nice and flexible for your feature authors. Another useful technique is to use a *noncapturing group*.

Noncapturing Groups

Remember back in [Alternation, on page 46](#) we showed you how you can list a set of possible values for part of a regular expression, separated by pipe. We can use this same technique to add flexibility to our step definitions, allowing feature authors to use slightly different ways to say the same thing. There's one little change we'll need to make, but we'll get to that in a minute.

Take this extremely common step for a web application:

```
When I visit the homepage
```

Suppose someone comes along and writes another step that looks like this:

```
When I go to the homepage
```

Both of these steps have identical meaning to the reader, but unfortunately a step definition for the first one won't match the second one without some modification. It would be helpful to have a step definition to recognize both phrases, because it really doesn't matter whether you say visit or go to—they both mean the same thing. We can use an *alternate* to relax the step definition to accept this slightly different phrasing:

```
When /I(?:visit|go to) the homepage/ do
  # TODO: code goes here
end
```

Notice that we've had to prefix the list of alternates with another bit of regular expression magic. The ?: at the start of the group marks it as *noncapturing*, meaning Cucumber won't pass it as an argument to our block.

Anchors

You might have noticed that the step definition snippets that Cucumber prints for undefined steps start with a ^ and end with a \$. Perhaps you've become so used to seeing them that you've stopped noticing them altogether. These two metacharacters are called *anchors*, because they're used to tie down each end of the regular expression to the beginning and end of the string that they match on.

You don't have to use them, and we deliberately left them out of the example up to this point because we wanted to wait until we'd explained what they do. If you omit one or both of them, you'll find you end up with a much more flexible step definition—perhaps too flexible. As a silly example, suppose we add the ^ anchor to the beginning but omit the \$ at the end of our bank account step definition:

```
Given /^I have deposited \$(\d+) in my Account/ do |amount|
  # TODO: code goes here
end
```

This will allow a particularly creative feature author to write something like this:

```
Given I have deposited $100 in my Account from a check my Grandma gave to me
```

Generally, it's best to keep your regular expressions as tight as you can so that there's less chance of two step definitions clashing with each other. That's why the snippets that Cucumber generates for undefined steps always include the anchors. Still, leaving off the anchors is a trick worth knowing about that can sometimes come in handy.

4.5 Returning Results

Cucumber is a testing tool, and it's in the Ruby code of a step definition where our tests find out whether a step has succeeded in whatever it set out to do. So, how does a step definition tell Cucumber whether it passed or failed?

Like most other testing tools, Cucumber uses exceptions to communicate the failure of a test. As it executes a scenario, one step at a time, Cucumber

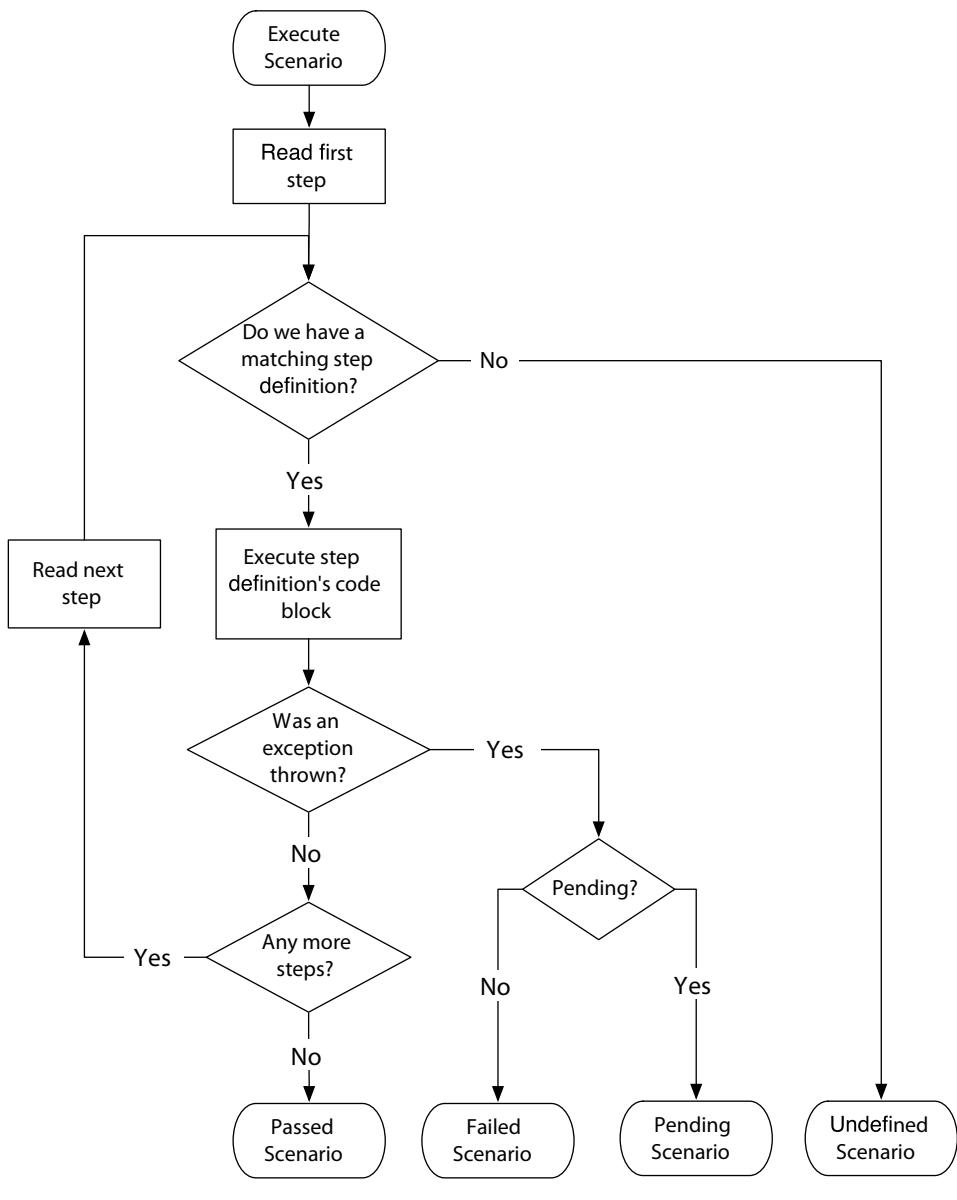


Figure 5—How Cucumber executes a scenario

assumes that a step has passed unless its step definition raises an exception. If a step passes, it moves on to the next step. [Figure 5, How Cucumber executes a scenario, on page 53](#) shows how this plays out.

In Cucumber, results are a little more sophisticated than a simple pass or fail. A scenario that's been executed can end up in any of the following states:

- Undefined
- Pending
- Failed
- Passed

These states are designed to help indicate the progress that you make as you develop your tests. Let's run through an example of automating the ATM withdrawal scenario to illustrate what we mean.

Undefined Steps

When Cucumber can't find a step definition that matches a step, it marks the step as undefined (yellow) and stops the scenario. The rest of the steps in the scenario will be either skipped or marked as undefined too if they don't have a matching step definition themselves.

To show you how this works, we can run our ATM withdrawal scenario. Create a file called `features/cash_withdrawal.feature`, and put the following into it:

[Download step_definitions/00/features/cash_withdrawal.feature](#)

```
Feature: Cash Withdrawal
  Scenario: Successful withdrawal from an account in credit
    Given I have deposited $100 in my account
    When I request $20
    Then $20 should be dispensed
```

We haven't written any step definitions yet, so when we run this feature, we should see the steps all come up as undefined:

Feature: Cash Withdrawal

```
Scenario: Successful withdrawal from an account in credit
  Given I have deposited $100 in my account
  When I request $20
  Then $20 should be dispensed
```

```
1 scenario (1 undefined)
3 steps (3 undefined)
0m0.001s
```

If you have color output in your terminal, you should see each step is yellow, indicating that it's neither failing (red) nor passing (green) but somewhere in between. Notice also that Cucumber has printed out a snippet for each missing step definition. We can use these as a starting point for implementing our own step definitions.

Assertions and Exceptions

Even if you're used to using a testing library like `Test::Unit` or `RSpec`, you might not have realized that the assertions in those libraries work by raising exceptions.

You can prove this to yourself by writing a little Ruby program that runs a failing assertion:

```
Download step_definitions/assertions_sidebar/assertion.rb
require 'test/unit/assertions'
include Test::Unit::Assertions

begin
  assert false, "Hello World"
rescue Exception => exception
  puts "Exception was raised: #{exception.class}"
end
```

When you run it, you should find that this program raises an exception of type `MiniTest::Assertion` (or a `Test::Unit::AssertionFailedError` if you're using Ruby 1.8)

Pending Steps

When Cucumber discovers a step definition that's halfway through being implemented, it marks the step as pending (yellow). Again, the scenario will be stopped, and the rest of the steps will be skipped or marked as undefined.

How does Cucumber know whether a step definition has been implemented?

Of course, you have to tell it. A few special Cucumber methods are available from within your step definitions' code blocks (which we'll talk about more in [Chapter 7, Step Definitions: On the Inside, on page 111](#)); one of them is `pending`.

When you call `pending` from within a step definition, it raises a `Cucumber::Pending` error. This tells Cucumber's runtime that the step has failed but in a particular way: the step definition is still being worked on. You'll probably have noticed that the snippets Cucumber generates for undefined steps have a call to `pending` in them; now you understand why.

Let's get back to our ATM withdrawal scenario to show you what we mean.

Create a file called `features/step_definitions/steps.rb` and paste in this step definition:

```
Download step_definitions/01/features/step_definitions/steps.rb
Given /^I have deposited \$(\d+) in my account$/ do |amount|
  pending("Need to design the Account interface")
end
```

Now when we run cucumber, we'll see that it has only tried to execute the first step and has marked it as pending. All the rest are still undefined:

Feature: Cash Withdrawal

```
Scenario: Successful withdrawal from an account in credit
  Given I have deposited $100 in my account
    Need to design the Account interface (Cucumber::Pending)
    ./features/step_definitions/steps.rb:2
      features/cash_withdrawal.feature:3
  When I request $20
  Then $20 should be dispensed
```

```
1 scenario (1 pending)
3 steps (2 undefined, 1 pending)
0m0.002s
```

The pending status is bit like those *under construction* signs you used to see all over the Internet in the 1990s. You can use it as a temporary signpost to your teammates that you're in the middle of working on something.

When we're developing a new scenario from the outside-in like this, we'll tend to work right across the same layer before diving into the next one. Right now we're concentrating on adding step definitions, so let's do that for each of the other steps and add a pending call in each one:

[Download step_definitions/02/features/step_definitions/steps.rb](#)

```
Given /^I have deposited \$\(\d+\) in my account$/ do |amount|
  pending("Need to design the Account interface")
end
When /^I request \$\(\d+\)$/ do |amount|
  pending("How do we simulate cash being requested?")
end
Then /^\$\(\d+\) should be dispensed$/ do |amount|
  pending("How do we validate that cash was dispensed?")
end
```

As we do this, we can check to see whether there are any existing step definitions we could use and, if not, just create a new one with a call to pending. The pending scenarios become our to-do list for the work we'll do when we drop down to the next layer and start implementing the step definitions.

Failing Steps

If the block of code executed by a step definition raises an exception, Cucumber will mark that step as failed (red) and stop the scenario. The rest of the steps in the scenario will be skipped.

In practice, there are two reasons why a step definition will fail:

Strict Mode

If you use the `--strict` command-line option, cucumber will return an exit code of 1 (to indicate an error) if there are any undefined or pending steps.

This can be useful in a continuous integration build to spot any half-finished features that have been accidentally checked in or when you've refactored your step definitions and some of your steps are no longer matching.

- The scenario couldn't finish because you have a bug in your step definition code, or in the system under test, which has caused it to throw an error. You'll get used to seeing these failures all the time during development if you use Cucumber to drive your development from the outside-in. Each failure message tells you what you need to do next.
- The step definition has used an *assertion* to check something about the state of the system, and the check didn't pass. You'll typically get these errors right at the end of your outside-in cycle or long after the feature has been implemented if someone accidentally introduces a bug.

An assertion is a check in your tests that describes some condition that you expect to be satisfied. Failures because of assertions tend to happen in Then steps, whose job is to check things about the state of the system. These are the alarm bells you're fitting to the system that will go off if it starts to behave in unexpected ways. Either way, Cucumber will show you the exception message and backtrace in its output, so it's up to you to investigate.

We're almost done with this chapter, but we want to show you an example of a failing step first. We're at the point where we need to start designing the interface between our tests and our system. Let's start with something simple and imagine an Account class that we can use to create a bank account for the actor in our scenario. We can modify the step definition like this:

[Download step_definitions/03/features/step_definitions/steps.rb](#)

```
Given /^I have deposited $(\d+) in my account$/ |amount|
  Account.new(amount.to_i)
end
```

What's going to happen when we run this? We don't actually have an Account class, so it's going to blow up:

Feature: Cash Withdrawal

```
Scenario: Successful withdrawal from an account in credit
  Given I have deposited $100 in my account
    uninitialized constant Account (NameError)
```

```
./features/step_definitions/steps.rb:3
  features/cash_withdrawal.feature:3
When I request $20
  Then $20 should be dispensed

Failing Scenarios:
cucumber features/cash_withdrawal.feature:2

1 scenario (1 failed)
3 steps (1 failed, 2 skipped)
0m0.002s
```

What's this failure telling us? It's telling us we need to create an Account class, of course! We're not going to start building our banking system right now, but we'll pick this example up again in [Chapter 7, Step Definitions: On the Inside, on page 111](#). As you've seen, Cucumber has caught the exception and displayed it to us just beneath the step. In the summary at the bottom we can see that the step failed, and the rest were skipped.

4.6 What We Just Learned

It's useful to think of a step definition as being a special kind of method. Unlike a regular Ruby method, whose name has to match exactly, a step definition can be invoked by any step that matches its regular expression. Because regular expressions can contain wildcards, this means you have the flexibility to make the Gherkin steps nice and readable, while keeping your Ruby step definition code clean and free of duplication.

- Step definitions provide a mapping from the Gherkin scenarios' plain-language descriptions of user actions into Ruby code, which simulates those actions.
- Step definitions are registered with Cucumber by calling Given, When, Then, or one of the aliases for your spoken language.
- Step definitions use regular expressions to declare the steps that they can handle. Because regular expressions can contain wildcards, one step definition can handle several different steps.
- A step definition communicates its result to Cucumber by raising, or not raising, an exception.

Now that you've seen how Gherkin and step definitions fit together, you're ready to start using Cucumber to test your own applications. To get your step definitions talking to the application, you'll need to learn how to use one of Ruby's automation libraries, many of which are covered in the recipes part of this book. If you're testing a web application, for example, see [Chapter 15](#),

[*Using Capybara to Test Ajax Web Applications*](#), on page 245, and if your web application runs on Ruby on Rails, see [*Chapter 14, Bootstrapping Rails*](#), on page 229. If you want to test a command-line application, try [*Chapter 16, Testing Command-Line Applications with Aruba*](#), on page 275. Web services are covered in [*Chapter 12, Testing a REST Web Service*](#), on page 201.

Over here in the fundamentals part, we're going to start putting some more flesh on the bones of your Cucumber knowledge. There's much more to Gherkin than the basic keywords we taught you in the previous chapter, and that's what we'll explore next.

Try This

At the end of the previous chapter, we suggested that you write some scenarios for your own project. Now try running them with Cucumber and use the snippets to create your first step definitions. Think about which domain entity each step is working with, and use that to decide which file to put the step definition into. Look for places you can use what you've learned about regular expressions to make the step definitions more flexible.



Matt says: Choosing Your Assertion Library

When making assertions in Cucumber step definitions, there are a few different libraries you could choose. The main candidates are RSpec,^a MiniTest,^b and Wrong.^c We'll give you a quick overview of each one's strengths, and you can choose your favorite. Here's a quick example of each of them in use:

```
# MiniTest
>> require 'test/unit'
>> include Test::Unit::Assertions
>> assert_equal 'green', 'cucumber'
MiniTest::Assertion: <"green"> expected but was
<"cucumber">.

# RSpec
>> require 'rspec/expectations'
>> "green".should == "cucumber"
RSpec::Expectations::ExpectationNotMetError: expected: "cucumber"
      got: "green" (using ==)

# Wrong
>> require 'wrong'
>> include Wrong
>> assert { 'green' == 'cucumber' }
Wrong::Assert::AssertionFailedError: Expected ("green" == "cucumber"), but
Strings differ at position 0:
  first: "green"
  second: "cucumber"
```

MiniTest's assertions are built into Ruby, which makes them a handy tool to reach for. They use a style of assertion that you'll be familiar with if you're used to making assertions in any xUnit testing framework.

RSpec's assertions are automatically loaded by Cucumber. Their style emphasizes readability.

Wrong is a new kid on the block. It uses Ruby magic tricks to read the code you've passed to the assertion to produce really helpful error messages.

In the end, the choice is a personal preference as to which syntax you prefer; all of the libraries provide roughly the same set of functionality.

-
- a. *The RSpec Book* [CADH09]
 - b. <http://rubydoc.info/gems/minitest/2.6.1/MiniTest/Assertions>
 - c. <http://rubydoc.info/gems/wrong/0.6.0/frames>

Expressive Scenarios

In [Chapter 3, Gherkin Basics, on page 25](#), we deliberately didn't give you the whole story and showed you just a core set of keywords instead. They're the fundamental building blocks you need to get started working with Cucumber, and we wanted to get you started as quickly as possible. Now it's time to refine your skills.

When you're writing Cucumber features, make readability your main goal. Otherwise, a reader can easily feel more like they're reading computer program than a specification document, which is something we want you to try to avoid at all costs. After all, if your features aren't easy for nonprogrammers to read, you might as well just be writing your tests in plain old Ruby code.

The real key to expressive scenarios is having a healthy vocabulary of domain language to use to express your requirements. That said, using only the basic set of Gherkin keywords can often make your features repetitive, making them cluttered and awkward to read. By the end of this chapter you'll know everything there is to know about Gherkin's syntax, giving you all the tools you need to write clear, readable Cucumber acceptance tests. We'll also show you how to use tags and folders to stay organized as you write more features for your project.

First we want to concentrate on helping you remove that repetitive clutter. We're going to show you how to use *scenario outlines* and *data tables* to help make your Gherkin scenarios more readable, but we'll start with a new keyword called `Background`.

5.1 Background

A *background* section in a feature file allows you to specify a set of steps that are common to every scenario in the file. Instead of having to repeat those

steps over and over for each scenario, you move them up into a Background element. There are a couple of advantages to doing this:

- If you ever need to change those steps, you have to change them in only one place.
- The importance of those steps fades into the background so that when you're reading each individual scenario, you can focus on what is unique and important about that scenario.

To show you what we mean, let's take an existing scenario that uses only the basic Gherkin Scenario element and improve its readability by *refactoring* it to use a Background. Here's our feature before the refactoring starts:

Feature: Change PIN

Customers being issued new cards are supplied with a Personal Identification Number (PIN) that is randomly generated by the system.

In order to be able to change it to something they can easily remember, customers with new bank cards need to be able to change their PIN using the ATM.

Scenario: Change PIN successfully

Given I have been issued a new card
And I insert the card, entering the correct PIN
When I choose "*Change PIN*" from the menu
And I change the PIN to 9876
Then the system should remember my PIN is now 9876

Scenario: Try to change PIN to the same as before

Given I have been issued a new card
And I insert the card, entering the correct PIN
When I choose "*Change PIN*" from the menu
And I try to change the PIN to the original PIN number
Then I should see a warning message
And the system should not have changed my PIN

You can see that there are two scenarios here, but without reading them carefully, it's quite hard to see what exactly is going on in each one. The first three steps in each scenario, while necessary to clarify the context of the scenario, are completely repeated in both scenarios. That repetition is distracting, making it harder to see the *essence* of what each scenario is testing.

Let's factor out the three repeated steps into a Background, like this:

Feature: Change PIN

As soon as the bank issues new cards to customers, they are supplied with a Personal Identification Number (PIN) that is randomly generated by the system.

In order to be able to change it to something they can easily remember, customers with new bank cards need to be able to change their PIN using the ATM.

Background:

Given I have been issued a new card
And I insert the card, entering the correct PIN
And I choose "*Change PIN*" from the menu

Scenario: Change PIN successfully

When I change the PIN to 9876
Then the system should remember my PIN is now 9876

Scenario: Try to change PIN to the same as before

When I try to change the PIN to the original PIN number
Then I should see a warning message
And the system should not have changed my PIN

Our refactoring hasn't changed the behavior of the tests at all: at runtime, the steps in the background are executed at the beginning of each scenario, just as they were before. What we have done is made each individual scenario much easier to read.

You can have a single Background element per feature file, and it must appear before any of the Scenario or Scenario Outline elements. Just like all the other Gherkin elements, you can give it a name, and you have space to put a multiline description before the first step. For example:

Feature: Change PIN

In order to be able to change it to something they can easily remember, customers with new bank cards need to be able to change their PIN using the ATM.

Background: Insert a newly issued card and sign in

Whenever the bank issues new cards to customers, they are supplied with a Personal Identification Number (PIN) that is randomly generated by the system.

Given I have been issued a new card
And I insert the card, entering the correct PIN

...

Using a `Background` element isn't always necessary, but it's often useful to improve the readability of your features by removing repetitive steps from individual scenarios. Here are some tips for using it well:

- Don't use `Background` to set up complicated state unless that state is something the reader actually needs to know. For example, we didn't mention the actual digits of the system-generated PIN in the previous example, because that detail wasn't relevant to any of the scenarios.
- Keep your `Background` section short. After all, you're expecting the user to actually remember this stuff when reading your scenarios. If the background is more than four lines long, can you find a way to express that action in just one or two steps?
- Make your `Background` section vivid. Use colorful names and try to tell a story, because your reader can keep track of stories much better than they can keep track of dull names like User A, User B, Site 1, and so on. If it's worth mentioning at all, make it really stand out.
- Keep your scenarios short, and don't have too many. If the `Background` is more than three or four steps long, think about using higher-level steps or splitting the feature file in two. You can use a background as a good indicator of when a feature is getting too long: if the new scenarios you want to add don't fit with the existing background, consider splitting the feature.
- Avoid putting technical details such as clearing queues, starting backend services, or opening browsers in a background. Most of these things will be assumed by the reader, and there are ways to push those actions down into your support code that we'll explain later in the book, such as in [*Tagged Hooks*, on page 147](#).

Backgrounds are useful for taking `Given` (and sometimes `When`) steps that are repeated in each scenario and moving them to a single place. This helps keep your scenarios clear and concise.

5.2 Data Tables

Sometimes steps in a scenario need to describe data that doesn't easily fit on a single line of `Given`, `When`, or `Then`. Gherkin allows us to place these details in a table right underneath a step. *Data tables* give you a way to extend a Gherkin step beyond a single line to include a larger piece of data.

For example, consider these steps:

Refactoring to Background

Refactoring^a is the process of changing code to improve its readability or design without changing its behavior. This technique applies to Gherkin features just as well as it does to the rest of your codebase. As your understanding of your domain grows through the course of the project, you'll want to reflect that learning by updating your features.

Often you don't see a background immediately. You might start out by writing one or two scenarios, and it's only as you write the third that you notice some common steps. When you spot a feature where the same or similar steps are repeated in several scenarios, see whether you can refactor to extract those steps into a background. It can take a little bit of courage to do this, because there's a risk you might make a mistake and break something, but this is a pretty safe refactoring. Once you're done, you should end up with the feature doing exactly the same thing as it did before you started but easier to read.

a. *Refactoring: Improving the Design of Existing Code* [FBBO99]

Given a User "*Michael Jackson*" born on August 29, 1958

And a User "*Elvis*" born on January 8, 1935

And a User "*John Lennon*" born on October 9, 1940

...

Boring! We wouldn't tolerate this kind of repetitive stuff in a traditional specification document, and we don't have to tolerate it in a Cucumber specification either. We can collect those steps together into a single step that uses a table to express the data:

Given these Users:

name	date of birth
Michael Jackson	August 29, 1958
Elvis	January 8, 1935
John Lennon	October 9, 1940

That's much clearer. The table starts on the line immediately following the step, and its cells are separated using the pipe character: |. You can line up the pipes using whitespace to make the table look tidy, although Cucumber doesn't mind whether you do; it will strip out the values in each cell, ignoring the surrounding whitespace.

In the previous table, we've used a heading for each column in the table, but that's only because it made sense for that particular step. You have the freedom to specify data in different ways, such as putting the headings down the side:

Then I should see a vehicle that matches the following description:

Wheels	2	
Max Speed	60 mph	
Accessories	lights, shopping basket	

Or just to specify a list:

Then my shopping list should contain:

Onions	
Potatoes	
Sausages	
Apples	
Relish	

To explain how to work with these different shaped tables, we need to take a short dive down into the step definition layer. If you're not interested in writing Ruby step definition code, feel free to skip this bit.

Working with Data Tables in Step Definitions

We'll illustrate how to use a data table in your step definitions with a quick game of tic-tac-toe. Let's imagine we're building a tic-tac-toe game and we've started working on the basic feature for making moves on the board. We start with a scenario like this:

Download expressive_scenarios/01/features/tic_tac_toe.feature

Feature:

Scenario:

Given a board like this:

	1	2	3	
1				
2				
3				

When player x plays in row 2, column 1

Then the board should look like this:

	1	2	3	
1				
2	x			
3				

We'll show you how to grab the table from the first step, manipulate it in the second, and finally compare the expected board from the scenario with the actual one.

Run cucumber to generate the step definition snippets, and paste them into features/step_definitions/board_steps.rb:

[Download expressive_scenarios/01/features/step_definitions/board_steps.rb](#)

```
Given /^a board like this:$/ do |table|
  # table is a Cucumber::Ast::Table
  pending # express the regexp above with the code you wish you had
end

When /^player x plays in row (\d+), column (\d+)/ do |arg1, arg2|
  pending # express the regexp above with the code you wish you had
end

Then /^the board should look like this:$/ do |table|
  # table is a Cucumber::Ast::Table
  pending # express the regexp above with the code you wish you had
end
```

Notice that the snippets for the two step definitions where we're going to receive a table are a little different. There's a comment telling you that the argument you're being passed is a Cucumber::Ast::Table object. That's a really rich object with lots of methods for interacting with its data. We'll show you some of the most useful ones now.

Let's start fleshing out those step definitions.

Turning the Table into an Array

Under the hood, the table is just a two-dimensional array. Often we'll want to work with it in that raw form, so we can call the raw method on it to do just that. Let's get the raw data from the table and store it in an instance variable @board, which we can manipulate in our second step when we want to make a move.

As an experiment, add an implementation for the second step definition that just prints the raw board out so we can see what it looks like:

[Download expressive_scenarios/02/features/step_definitions/board_steps.rb](#)

```
Given /^a board like this:$/ do |table|
  @board = table.raw
end

When /^player x plays in row (\d+), column (\d+)/ do |row, col|
  puts @board
  pending # express the regexp above with the code you wish you had
end

Then /^the board should look like this:$/ do |table|
  # table is a Cucumber::Ast::Table
  pending # express the regexp above with the code you wish you had
end
```

When you run cucumber, you should see the two-dimensional array is printed out.

```
$ cucumber -f progress

[[ "", "1", "2", "3"], ["1", "", "", ""], ["2", "", "", ""], ["3", "", "", ""]]

P-


(:) pending steps (:)

features/tic_tac_toe.feature:8:in 'When player x plays in row 2, column 1'

1 scenario (1 pending)
3 steps (1 skipped, 1 pending, 1 passed)
0m0.008s
```

Notice that the raw table includes the column and row headings.

Comparing Tables with Diff

So that we can start out with a failing test, we'll skip doing any manipulation of the board in this step for now. So, remove the body of the second step definition and make the following implementation in the last step definition:

```
Download expressive_scenarios/03/features/step_definitions/board_steps.rb
Given /^a board like this:$/
  |table|
  @board = table.raw
end
When /^player x plays in row (\d+), column (\d+)/$/
  |row, col|
end
Then /^the board should look like this:$/
  |expected_table|
  expected_table.diff!(@board)
end
```

We've used the `diff!` method on the table that describes how things should look, passing it the actual board as we see it in our application. When you run cucumber again, you should see that the step has failed because the tables were not identical:

```
$ cucumber
```

Feature:

```
Scenario:
  Given a board like this:
    |   | 1 | 2 | 3 |
    | 1 |   |   |   |
    | 2 |   |   |   |
    | 3 |   |   |   |
  When player x plays in row 2, column 1
```

```
Then the board should look like this:
```

	1	2	3
1			
2	x		
2			
3			

```
Tables were not identical (Cucumber::Ast::Table::Different)
./features/step_definitions/board_steps.rb:9
features/tic_tac_toe.feature:9
```

Failing Scenarios:

```
cucumber features/tic_tac_toe.feature:2
```

```
1 scenario (1 failed)
3 steps (1 failed, 2 passed)
0m0.012s
```

If you have color in your console output, you'll see the cells that were the same are green. Expected cells that weren't found are yellow, and the actual cells are printed next to them in gray. It's possible to tweak the comparison behavior by passing a hash of options to `diff!`, which you can read about in the documentation for `Cucumber::Ast::Table`.

Let's fix the When step to make the scenario pass. Add this implementation to the second step definition:

```
Download expressive_scenarios/04/features/step_definitions/board_steps.rb
When /^player x plays in row (\d+), column (\d+)/ do |row, col|
  row, col = row.to_i, col.to_i
  @board[row][col] = 'x'
end
```

Run the scenario, and you should see it pass.

This is just a taste of what you can do with data tables in Cucumber. We encourage you to read the documentation² for `Cucumber::Ast::Table` and play around with it yourself.

Data tables are a great feature of Gherkin. They're really versatile, and they help you express data concisely, as you'd want to in a normal specification document. With backgrounds and data tables, you can do a lot to reduce the noise and clutter in your scenarios. Even when you use these tools, you'll still sometimes see a pattern where one scenario looks a lot like the one that came before it and the one after it. This is where a *scenario outline* can help.

2. <http://cukes.info/cucumber/api/ruby/latest/Cucumber/Ast/Table.html>

5.3 Scenario Outline

Sometimes you have several scenarios that follow exactly the same pattern of steps, just with different input values or expected outcomes. For example, suppose we're testing each of the fixed amount withdrawal buttons on the ATM:

Feature: Withdraw Fixed Amount

The "*Withdraw Cash*" menu contains several fixed amounts to speed up transactions for users.

Scenario: Withdraw fixed amount of \$50

Given I have \$500 in my account
When I choose to withdraw the fixed amount of \$50
Then I should receive \$50 cash
And the balance of my account should be \$450

Scenario: Withdraw fixed amount of \$100

Given I have \$500 in my account
When I choose to withdraw the fixed amount of \$100
Then I should receive \$100 cash
And the balance of my account should be \$400

Scenario: Withdraw fixed amount of \$200

Given I have \$500 in my account
When I choose to withdraw the fixed amount of \$200
Then I should receive \$200 cash
And the balance of my account should be \$300

Once again, all the repetition in this feature makes it boring to read. It's hard to see the essence of each scenario, which is the amount of money involved in each transaction. We can use a scenario outline to specify the steps once and then play multiple sets of values through them. Here's that scenario again, refactored to use a scenario outline:

Feature: Withdraw Fixed Amount

The "*Withdraw Cash*" menu contains several fixed amounts to speed up transactions for users.

Scenario Outline: Withdraw fixed amount

Given I have <Balance> in my account
When I choose to withdraw the fixed amount of <Withdrawal>
Then I should receive <Received> cash
And the balance of my account should be <Remaining>

Examples:

Balance	Withdrawal	Received	Remaining
---------	------------	----------	-----------

\$500	\$50	\$50	\$450	
\$500	\$100	\$100	\$400	
\$500	\$200	\$200	\$300	

We indicate *placeholders* within the scenario outline using angle brackets (`<..>`) where we want real values to be substituted. The scenario outline itself is useless without a table of Examples, which lists rows of values to be substituted for each placeholder.

You can have any number of Scenario Outline elements in a feature and any number of Examples tables under each scenario outline. Behind the scenes, Cucumber converts each row in the Examples table into a scenario before executing it. You can prove this to yourself by using the `--expand` option, which will print each example in a scenario outline as though it were a scenario:

```
$ cucumber --expand
```

One of the advantages of using a scenario outline is that you can clearly see gaps in your examples. In our example, we haven't tested any edge cases, such as when you try to withdraw more money than you have available. This becomes much more obvious when you can see all the values lined up together in a table.

Remember that although the syntax for writing them in Gherkin is the same, these tables are totally different from the data tables we described earlier in this chapter. Data tables just describe a lump of data to attach to a single step of a single scenario. In a scenario outline, each row of an Examples table represents a whole scenario to be executed by Cucumber. In fact, you might want to use the keyword Scenarios (note the extra `s`) in place of Examples if you find that more readable.

Bigger Placeholders

It's easy to imagine that you can use scenario outline placeholders only where there's a piece of data in the step. In fact, when Cucumber compiles a scenario outline's table of examples down into scenarios ready to execute, it doesn't care where the placeholders are. So, you can substitute as much or as little as you like from any step's text.

Let's illustrate this by testing the edge case where we try to withdraw more money than we have. What should we do in this case? Give the user as much as we can, given their remaining balance, or just show them an error message? We ask our stakeholders for clarification, and they're happy for us to show them an error message. Here's how we write the scenario first:

Scenario: Try to withdraw too much

Given I have \$100 in my account

When I choose to withdraw the fixed amount of \$200

Then I should see an error message

And the balance of my account should be \$100

There's still a good deal of duplication here with the flow of the scenarios above it, but because the Then step is so different, we can't put this one into the scenario outline. Or can we?

Let's change the scenario outline, replacing the <Received> placeholder with a more abstract <Outcome>:

Scenario Outline: Withdraw fixed amount

Given I have <Balance> in my account

When I choose to withdraw the fixed amount of <Withdrawal>

Then I should <Outcome>

And the balance of my account should be <Remaining>

Examples:

Balance	Withdrawal	Remaining	Outcome
\$500	\$50	\$450	receive \$50 cash
\$500	\$100	\$400	receive \$100 cash
\$500	\$200	\$300	receive \$200 cash

Now we can simply add our failure case to the bottom of that table:

Scenario Outline: Withdraw fixed amount

Given I have <Balance> in my account

When I choose to withdraw the fixed amount of <Withdrawal>

Then I should <Outcome>

And the balance of my account should be <Remaining>

Examples:

Balance	Withdrawal	Remaining	Outcome
\$500	\$50	\$450	receive \$50 cash
\$500	\$100	\$400	receive \$100 cash
\$500	\$200	\$300	receive \$200 cash
\$100	\$200	\$100	see an error message

We can use a placeholder to replace any of the text we like in a step. Notice that it doesn't matter what order the placeholders appear in the table: what counts is that the column header matches the text in the placeholder in the scenario outline.

Although this is a useful technique, be careful that your programmer's instinct to reduce duplication at all costs doesn't take over here.⁵ If you move too

5. See this point explained in detail by David Chelimsky: <http://confreaks.net/videos/434>.



Joe asks:

How Many Examples Should I Use?

Once you have a scenario outline with a few examples, it's very easy to think of more examples, and even easier to add them. Before you know it, you have a huge, very comprehensive table of examples—and a problem.

Why?

On a system of any serious complexity, you can quite quickly start to experience what mathematicians call *combinatorial explosion*, where the number of different combinations of inputs and expected outputs becomes unmanageable. In trying to cover every possible eventuality, you end up with rows and rows of example data for Cucumber to execute. Remember that each of those little rows represents a whole scenario that might take several seconds to execute, and that can quickly start to add up. When your tests take longer to run, you slow down your feedback loop, making the whole team less productive as a result.

A really long table is also very hard to read. It's better to aim to make your examples *illustrative* or *representative* than *exhaustive*. Try to stick to what Gojko Adzic calls the *key examples*.^a If you study the code you're testing, you'll often find that some rows of your examples table cover the same logic as another row in the table. You might also find that the test cases in your table are already covered by unit tests of the underlying code. If they're not, consider whether they should be.

Remember that readability is what's most important. If your stakeholders feel comforted by exhaustive tests, perhaps because your software operates in a safety-critical environment, then by all means put them in. Just remember that you'll never be able to prove there are no bugs. As logicians say, absence of proof is not proof of absence.

If you're struggling to pick a representative set from an exhaustive list of examples, take a look at Joseph Wilk's pairwise^b Ruby gem.

-
- a. *Specification by Example* [Adz11]
 - b. <https://github.com/josephwilk/pairwise/wiki>

much of the text of a step into the examples table, it can be very hard to read the flow of the scenario. Remember your goal is readability, so don't take this too far, and always test your features by getting other people to regularly read them and give you feedback.

Multiple Tables of Examples

Cucumber will happily handle any number of Examples elements beneath a Scenario Outline, meaning you can group different kinds of examples together, if you want. For example:

Scenario Outline: Withdraw fixed amount

Given I have <Balance> in my account
When I choose to withdraw the fixed amount of <Withdrawal>
Then I should <Outcome>
And the balance of my account should be <Remaining>

Examples: Successful withdrawal

Balance	Withdrawal	Outcome	Remaining
\$500	\$50	receive \$50 cash	\$450
\$500	\$100	receive \$100 cash	\$400

Examples: Attempt to withdraw too much

Balance	Withdrawal	Outcome	Remaining
\$100	\$200	see an error message	\$100
\$0	\$50	see an error message	\$0

As usual, you have the option of a name and description for each Examples table. When you have a large set of examples, splitting it into multiple tables can make it easier for a reader to understand.

Explain Yourself

You'll normally be using a scenario outline and examples table to help specify the implementation of a business rule. Remember to include a plain-language description of the underlying rule that the examples are supposed to illustrate! It's amazing how often people forget to do this.

For example, look at this feature:

Feature: Account Creation

Scenario Outline: Password validation

Given I try to create an account with password "<Password>"
Then I should see that the password is <Valid or Invalid>

Examples:

Password	Valid or Invalid
abc	invalid
ab1	invalid
abc1	valid
abcd	invalid
abcd1	valid

If you have to implement the code to make this feature pass, can you tell what the underlying rule is?

Not very easily. So, let's modify the feature to make it more self-explanatory, like this:

Feature: Account Creation**Scenario Outline:** Password validation

Given I try to create an account with password "<Password>"
Then I should see that the password is <Valid or Invalid>

Examples: Too Short

Passwords are invalid if less than 4 characters

Password	Valid or Invalid
abc	invalid
ab1	invalid

Examples: Letters and Numbers

Passwords need both letters and numbers to be valid

Password	Valid or Invalid
abc1	valid
abcd	invalid
abcd1	valid

By separating the examples into two sets and giving each one a name and description, we've explained the rule *and* given examples of the rule at the same time.

5.4 Nesting Steps

Finding the right level of detail, or *abstraction*, to use in your scenarios is a skill that takes some time to master. What many people don't realize is that different levels of detail are appropriate for different scenarios in the same system—sometimes in the same feature—depending on what it is they're describing.

As an example, here's a scenario for the user of our ATM authenticating with their PIN:

Scenario: Successful login with PIN

Given I have pushed my card in the slot
When I enter my PIN
And I press "OK"
Then I should see the main menu

It's entirely appropriate for us to go into this much detail about the authentication process in this scenario, because that's where our focus is. Now consider our cash withdrawal scenario from earlier, which has a different focus but still needs to be authenticated. Does it make sense to express the PIN authentication steps of this scenario at the same level of detail? Let's try it:

```
Scenario: Withdraw fixed amount of $50
  Given I have $500 in my account
  And I have pushed my card into the slot
  And I enter my PIN
  And I press "OK"
  When I choose to withdraw the fixed amount of $50
  Then I should receive $50 cash
  And the balance of my account should be $450
```

That's awful! There's so much noise about authentication that we hardly notice the important part: the part about withdrawing cash. That detail was useful in the PIN scenario where it was relevant, but now it's just distracting. We'll talk more about the dangers of over-detailed or *imperative* scenarios in [Chapter 6, When Cucumbers Go Bad, on page 85](#); right now we want to show you a simple refactoring to push these details down into the step definitions so our scenario is easy to read.

Nest Steps Refactoring

Let's take the three authentication steps and summarize what they do with a single high-level step:

```
Given I have authenticated with the correct PIN
```

Cut the three lines from the scenario into your clipboard, and replace them with that single step. Now run cucumber to generate the step definition snippet for your new high-level step. It should look like this:

```
Given /^I have authenticated with the correct PIN$/ do
  pending # express the regexp above with the code you wish you had
end
```

Now, use Cucumber's built-in steps method to call out to the original three steps in the body of your high-level step's definition:

```
Given /^I have authenticated with the correct PIN$/ do
  steps %{
    And I have pushed my card into the slot
    And I enter my PIN
    And I press "OK"
  }
end
```

The %{ ... } construct is just a way to tell Ruby that you have a string going across multiple lines. When Cucumber runs this high-level step, it will delegate to each of the lower-level steps. So, the behavior of the scenario is exactly the same, but the language in the business-facing feature is at a higher level.

Arguments and Nested Steps

If the low-level steps you are nesting within a new high-level step take arguments, you'll want to be able to capture those arguments in the high-level step and pass them to the low-level steps. Since the `steps` method just takes a string, you can use Ruby's string interpolation to pass in the arguments:

```
Given /^an activated customer (\w+) exists$/ do |name|
  steps %{
    Given I create a customer with login #{name}
    And I register the customer with login #{name}
    And I activate the customer with login #{name}
  }
end
```

The only time this won't work is if the argument you're passing is a data table. In that case you can use another method, `step`, to call a specific step, passing the actual typed `DataTable` object as an argument:

```
Given /^a (\w+) widget with the following details:$/
  step "I create a #{color} widget with the following details:", details_table
  steps %{
    And I register the #{color} widget
    And I activate the #{color} widget
  }
end
```

Nested steps work best when they're simple: a high-level step delegating to a few lower-level ones. As soon as you start adding arguments into the mix, you're already starting to make things more complicated than they probably need to be. Remember that underneath it all there are Ruby methods being called. Often it makes sense to use the Ruby methods directly rather than obscuring them with layers of nested steps.

The Dangers of Nested Steps

Nested steps are a useful tool to reach for when you need to quickly refactor an overly detailed scenario to make it more readable. However, you need to watch out: they can lure you down a path to complexity and frustration. Here's a story from Andrew Premdas, an experienced Cucumber user who once was a big fan of nested steps:

Don't Use Nested Steps

by: Andrew Premdas

Nested steps are a temptation. They beguile you with the idea that you can achieve something valuable with great ease. But this is a mirage, because anything you do with nested steps can be done a simpler way by using Ruby code instead.

When implementing step definitions, what is important is to make it crystal clear exactly what you are doing so you can be sure your scenario does what it says, and not something different. Nested steps allow you to implement step definitions without really knowing what you are doing or how you are doing it. They put unnecessary layers of obscurity between the scenario and the code that actually does the work.

Nesting steps prevents you from becoming a better programmer. The programming involved in implementing step definitions is generally quite simple. Every time you use a nested step you lose an opportunity to interact directly with the underlying code that talks to your system and understand how it works. Instead of becoming empowered and learning with each new scenario, you become more and more dependent on nested steps. As your use of nested steps increases, this dependency increases, as does the complexity of your step implementations.

The important thing to remember is this: *you can apply the extract method refactoring to the body of every step definition*. This choice is always preferable to nesting because it's always easier to find, read, and work with a Ruby method call than another step definition. Why bother nesting when you can extract methods instead?

On our project that used nested steps, the more we nested, the harder it became to implement scenarios—they just kept breaking for really strange reasons. Getting rid of the nested steps made things much more transparent. Our colleagues noticed the improvement immediately and soon were telling us off for nesting steps.

5.5 Doc Strings

Doc strings just allow you to specify a larger piece of text than you could fit on a single line. For example, if you need to describe the precise content of an email message, you could do it like this:

```
Scenario: Ban Unscrupulous Users
When I behave unscrupulously
Then I should receive an email containing:
    """
        Dear Sir,
        Your account privileges have been revoked due to your unscrupulous behavior.
        Sincerely,
        The Management
    """
And my account should be locked
```

Just like a data table, the entire string between the `"""` triple quotes is attached to the step above it. The indentation of the opening `"""` is not important, although common practice is to indent two spaces in from the enclosing step, as we've shown. The indentation inside the triple quotes, however, is significant: imagine the left margin running down from the start of the first `"""`. If you want to include indentation within your string, you need to indent it within this margin.

Doc strings open up all kinds of possibilities for specifying data in your steps. We've seen teams use these arguments to specify snippets of JSON or XML data when writing features for an API, for example. There's an example of how to do this in [Chapter 12, Testing a REST Web Service, on page 201](#). You do need to be cautious when including this much detail in a scenario. It's easy to create a lot of clutter with a large piece of data, making it hard to read the scenario as a whole. You can also easily create brittle scenarios, where the slightest change to the system causes the scenario to fail because it's behaving slightly differently than the way it was described in the doc string.

OK, we've covered the more advanced features of Gherkin you can use to express your business requirements. The last thing we want to talk about in this chapter is keeping things organized.

5.6 Staying Organized with Tags and Subfolders

It's easy to be organized when you have only a couple of features, but as your test suite starts to grow, you'll want to keep things tidy so that the documentation is easy to read and navigate. One simple way to do this is to start using subfolders to categorize your features. This gives you only one axis for organization, though, so you can also use tags to attach a label to any scenario, allowing you to have as many different ways of slicing your features as you like.

Subfolders

This is the easiest way to organize your features. You may find yourself torn as to how to choose a category, though: do you organize by user type, with an `features/admins` folder, a `features/logged_in_users` folder, and a `features/visitors` folder, for example? Or do you organize them by domain entity or something else?

Of course, this is a decision for you and your team to take, but we can offer a little bit of advice. We've had most success using subfolders to represent different high-level tasks that a user might try to do. So, if we were building an intranet reporting system, we might organize it like this:

```
features/
  reading_reports/
  report_building/
  user_administration/
```

Don't get too hung up about getting your folder structure right the first time. Make a decision to try a structure, reorganize all the existing feature files, and then stick to it for a while as you add new features. Put a note in the

calendar to take some time out in a couple of weeks and reflect on whether the new structure is working.

If you think about your features as a book that describes what your system does, then the subfolders are like the chapters in that book. So, as you tell the story of your system, what do you want the reader to see when they scan the table of contents?

Running Features in a Subfolder

If you have put some of your features into subfolders, you can still run them using the command `cucumber`, which will automatically find and run all the `.feature` files anywhere within the `features` folder. However, if you want to run just one feature from a subfolder, there's a little trick to getting it to work. Here's what you might expect to be able to do:

```
$ cucumber features/reading_reports/widgets_report.feature
# all your steps come up as undefined :(
```

If you try to run a single feature from a subfolder, you'll find that all your steps come up as undefined, even though the step definitions are there in `features/step_definitions`. What's going on? Unfortunately, Cucumber has become a bit disoriented and no longer knows where to find your step definitions. You need to tell it explicitly to *require* the step definition code from the root `features` folder:

```
$ cucumber features/reading_reports/widgets_report.feature -r features
```

You can use a *profile* (see [Chapter 11, The Cucumber Command-Line Interface, on page 191](#)) to tell Cucumber to do this automatically.

Tags

If subfolders are the chapters in your book of features, then tags are the sticky notes you've put on pages you want to be able to find easily. You tag a scenario by putting a word prefixed with the `@` character on the line before the `Scenario` keyword, like this:

```
@widgets
Scenario: Generate report
  Given I am logged in
  And there is a report "Best selling widgets"
  ...
```

In fact, you can attach multiple tags to the same scenario, separated with spaces:



Aslak says: Features Are Not User Stories

Long ago, Cucumber started life as a tool called the RSpec Story Runner. In those days, the plain-language tests used a `.story` extension. When I created Cucumber, I made a deliberate decision to name the files features rather than stories. Why did I do that?

User stories are a great tool for planning. Each story contains a little bit of functionality that you can prioritize, build, test, and release. Once a story has been released, we don't want it to leave a trace in the code. We use refactoring to clean up the design so that the code absorbs the new behavior specified by the user story, leaving it looking as though that behavior had always been there.

We want the same thing to happen with our Cucumber features. The features should describe how the system behaves today, but they don't need to document the history of how it was built; that's what a version control system is for!

We've seen teams whose features directory looks like this:

```
features/
  story_38971_generate_new_report.feature
  story_38986_run_report.feature
  story_39004_log_in.feature
  ...
```

We strongly encourage you *not* to do this. You'll end up with fragmented features that just don't work as documentation for your system. One user story might map to one feature, but another user story might cause you to go and add or modify scenarios in several existing features—if the story changes the way users have to authenticate, for example. It's unlikely that there will always be a one-to-one mapping from each user story to each feature, so don't try to force it. If you need to keep a story identifier for a scenario, use a tag instead.

```
@slow @widgets @nightly
Scenario: Generate overnight report
  Given I am logged in
  And there is a report "Total widget sales history"
  ...
```

If you want to tag all the scenarios in a feature at once, just tag the Feature element at the top, and all the scenarios will inherit the tag. You can still tag individual scenarios as well.

```
@nightly @slow
Feature: Nightly Reports

@widgets
Scenario: Generate overnight widgets report
```

...

`@doofers`

Scenario: Generate overnight doofers report

...

In the previous example, the scenario called Generate overnight widgets report will have three tags: @nightly, @slow, and @widgets, whereas the scenario called Generate overnight doofers report will have the tags @nightly, @slow, and @doofers. You can also tag Scenario Outline elements and even the individual Examples tables beneath them.

There are three main reasons for tagging scenarios:

- *Documentation:* You simply want to use a tag to attach a label to certain scenarios, for example to label them with an ID from a project management tool.
- *Filtering:* Cucumber allows you to use tags as a filter to pick out specific scenarios to run or report on. You can even have Cucumber fail your test run if a certain tag appears too many times.
- *Hooks:* Run a block of Ruby code whenever a scenario with a particular tag is about to start or has just finished.

We'll cover hooks later in [Tagged Hooks, on page 147](#), and we'll explain how to filter based on tags in [Filtering with Tag Expressions, on page 193](#). In case you can't wait until then, here's a quick example of how to run Cucumber, selecting just scenarios with a certain tag:

```
$ cucumber --tags @javascript
```

That will select and run only the scenarios tagged with @javascript.

5.7 What We Just Learned

Congratulations, you've graduated from Gherkin school! Let's look back over what we've learned in this chapter.

- Readability should be your number-one goal when writing Gherkin features. Always try to sit together with a stakeholder when you write your scenarios, or at the very least pass them over for feedback once you've written them. Keep fine-tuning the language in your scenarios to make them more readable.
- Use a Background to factor out repeated steps from a feature and to help tell a story.

- Repetitive scenarios can be collapsed into a Scenario Outline.
- Steps can be extended with multiline strings or data tables.
- You can organize features into subfolders, like chapters in a book.
- Tags allow you to mark up scenarios and features so you select particular sets to run or report on.

Have you noticed that we're going outside-in as we work through this part of the book? We started with an overview, then looked at the most useful bits of Gherkin, and then did the same for step definitions. Now that you've learned about the more advanced features of Gherkin, you're almost ready to dive deep into step definition Ruby code. In the next part of the book, that's exactly what we'll do, with a worked example that will give you a chance to practice everything you've learned, and more. First, though, we're going to take a step back and examine some of the common problems you and your team might encounter as you start to use Cucumber and what to do about them.

Try This

Review the scenarios that you've written for your system, and see whether you can find an opportunity to use a Background, a Scenario Outline, or a data table. Refactor the feature to make use of the new keyword and compare the new version of the feature with the old one. Which one do you think is more readable? Show it to someone else on your team; what do they think?

CHAPTER 6

When Cucumbers Go Bad

When your team first starts to use Cucumber, it won't be long before you start to notice that you seem to be creating code with fewer bugs than you did before. You might find yourself bravely refactoring in code that previously you would have been too scared to touch. You'll continue adding feature after feature, inspired by the delight you felt on seeing your first passing scenario.

After a while, however, things can start to turn sour. Suddenly it dawns on you that the tests take a really long time to run. Or perhaps you've started to notice a couple of scenarios that seem to fail at random, usually just when you're up against a tight deadline. Perhaps the nontechnical stakeholders have lost interest in the process, and only developers are reading the features anymore. People might even start to ask this:

Is Cucumber holding us back?

The good news is, there are solutions to these problems. In our coaching and consulting work, we've seen all kind of problems experienced by all kinds of teams as they learn to use Cucumber. In this chapter, we'll describe the most common problems we've seen. We'll help you understand their root causes, and we'll make suggestions for tackling them or, ideally, avoiding them in the first place. There won't be much code in this chapter, but you'll find lots of useful advice.

We'll start where it hurts, by describing four different symptoms your team might be experiencing. Then we'll dig down into the underlying causes of these, before finally looking at solutions. By the end of the chapter, you should feel much more confident about how to help your team stay successful with Cucumber in the long run.

6.1 Feeling the Pain

We've identified four main types of pain that your team might start to feel if their Cucumber goes bad. Take a look and see whether you recognize any of these:

Symptom	Problem
Some of our tests fail randomly.	<i>Flickering scenarios</i>
We keep breaking tests unintentionally.	<i>Brittle features</i>
Our features take too long to run.	<i>Slow features</i>
Our stakeholders don't read our features.	<i>Bored stakeholders</i>

Let's take a closer look at each of these symptoms.

Flickering Scenarios

When a scenario that was passing yesterday is failing today, with the same source code running in the same environment, you have what we call a *flickering scenario*. Here's our definition of a flickering scenario:

Flickering scenario

A flickering scenario fails occasionally and at random. The same scenario, run on the same codebase in the same environment, will mostly pass but sometimes fail. These apparently uncontrollable failures cause the team to lose confidence in their tests, in their code, and in themselves.

The worst thing about a flickering scenario is that as soon as you try to reproduce it so that you can fix it, it refuses to fail. Fixing a flickering scenario is one of the hardest tasks you can take on, yet it's also one of the most important. For a suite of automated tests to be useful, the team must have absolute trust in it. When even just a single test is compromising that trust, it has a corrosive effect on how everyone feels about the whole test suite.

To fix a flickering scenario, you have to study the code and try to understand why it might be happening. This is a scientific process of making a hypothesis about the cause of the failure, creating an experiment to prove or disprove that hypothesis, and then running the experiment to see whether you were right. You might need to go around this loop several times before you crack the problem, and it might take several days to run an experiment, if the flickering scenario fails only intermittently. If you run out of ideas, consider simply deleting the test altogether rather than have it come back and fail on you at a time of its own choosing.

Flickering scenarios are normally caused by one of the following underlying problems:

- [Shared Environments, on page 100](#)
- [Leaky Scenarios, on page 97](#)
- [Race Conditions and Sleepy Steps, on page 99](#)

Brittle Features

When you feel like you can hardly move in the test suite without making an apparently unrelated test fail for no good reason, you have what we call *brittle features*. Here's our definition:

Brittle feature

A brittle feature is easy to break. When features are brittle, a necessary change in one part of the test suite or main codebase causes apparently unrelated scenarios to break.

When you encounter a brittle scenario, it's usually when you're in the middle of doing something else. You're interrupted by the unexpected failure and waste time dashing over to fix the unexpected broken test. On a bad day, this can happen several times before you emerge from the rabbit warren. Brittle features are self-fulfilling: when developers perceive their tests to be brittle, they tend to be less courageous about refactoring and cleaning up test code and instead try to get in and out as quickly as they can, leaving the tests and production codebase in an increasingly hard-to-maintain state.

Brittle features are normally caused by one of the following underlying problems:

- [Fixture Data, on page 101](#)
- [Duplication, on page 93](#)
- [Leaky Scenarios, on page 97](#)
- [Tester Apartheid, on page 101](#)

Slow Features

Each time you add a new scenario to your test suite, you're adding a few seconds to the total test run time. For a successful application whose users continue to demand new features, that test run time is only going to get longer. A long test run creeps up on you: first five minutes seems like an eternity to wait, then fifteen minutes seems bad, but you get used to going to grab a coffee while it runs. Pretty soon, you come back from your coffee and it still hasn't finished, and fifteen minutes becomes twenty-five minutes. Before you know it, your features are taking more than an hour or even longer.

Once a new scenario is passing, the main reason to keep running it is for feedback: you want that scenario to warn you if you somehow accidentally break the functionality that it checks for. The value of that feedback diminishes as your test run starts taking longer and longer. When the build is slow, developers don't run all the tests before committing code and will rely on the continuous integration server to give them feedback instead. If a few developers do this at the same time, the chances of all of their changes integrating successfully are slim, and a broken build becomes the norm.

Long test runs also mean people are scared to refactor and do other general maintenance on the Cucumber tests themselves. Refactoring the code in a step definition that's used in 340 scenarios is scary, because you'll need to run all 340 scenarios to tell you for certain whether your change broke anything.

A slow feature run is normally caused by a combination of the following:

- [*Race Conditions and Sleepy Steps*, on page 99](#)
- [*Lots of Scenarios*, on page 103](#)
- [*Big Ball of Mud*, on page 104](#)

Bored Stakeholders

"Our stakeholders don't read our features." This is a common complaint from teams that have tried Cucumber but failed to make it stick as anything other than a tool for automating their test scripts. Yet many other teams attest to the transformative effect Cucumber has had, helping development teams to collaborate much more effectively with their business stakeholders. What could be the difference between these two experiences?

The answer lies partly in starting with the right kind of collaborative relationship with those business stakeholders. If they think they're too busy to help you understand exactly what they want, then you have a deeper team problem that Cucumber can't help you solve. On the other hand, many teams who start out with keen and interested stakeholders waste the opportunity Cucumber gives them to build that collaborative relationship. When features are written by testers or developers working alone, they inevitably use technical terms that make the stakeholders feel marginalized when they read them. This becomes a vicious circle: as stakeholders lose interest, they spend less time helping write the features in a language that makes sense to them. Before you know it, the features have become nothing more than a testing tool.

This painful symptom is normally caused by a combination of the following underlying problems:

Running Cucumber Tests in Parallel

If you're stuck with a slow set of features, a pragmatic option can be to run them in parallel. The simplest approach to this is to partition your features using tags or folders and then run each of those partitioned sets at the same time. Many continuous integration tools like Jenkins^a allow you to delegate builds to slave machines so you can ensure that each partitioned set of features gets its own dedicated environment.

Another approach is to use a tool like Testbot^b or Hydra^c to distribute your features automatically across several slave machines.

Whatever approach you choose, you'll definitely need a *One-Click System Setup*, on page 102.

- a. <http://jenkins-ci.org/>
- b. <http://rubygems.org/gems/testbot>
- c. <http://rubygems.org/gems/hydra>

- *Incidental Details*, on page 90
- *Imperative Steps*, on page 91
- *Duplication*, on page 93
- *Ubiquitous What?*, on page 94
- *Siloed Features*, on page 95

Once you've spotted any of these symptoms in your team, you need to know what to do. It's time to look at the underlying problems that are at work and what you can do about them.

6.2 Working Together

Cucumber features are what Gojko Adzic⁴ calls *living documentation*. That term neatly sums up the two main benefits of using Cucumber:

- *Living*: It tests the system automatically so you can work on it safely.
- *Documentation*: It facilitates good communication about the current or planned behavior of the system.

When your team is struggling with Cucumber, the problems you're having will hit you in one of these two places. Either they'll result in Cucumber scenarios that provide poor feedback for the developers or they'll mean Cucumber fails to help your team communicate. We'll start by looking at what might be holding you back from making the features work as a communication tool.

4. *Specification by Example* [Adz11]

Incidental Details

Consider the following scenario for an online email client:

```
Scenario: Check inbox
Given a User "Dave" with password "password"
And a User "Sue" with password "secret"
And an email to "Dave" from "Sue"
When I sign in as "Dave" with password "password"
Then I should see 1 email from "Sue" in my inbox
```

There is a lot of detail in this scenario: we have the username and password of the main actor, Dave, and we also have the username and password of another user, Sue. The usernames are quite useful, because they help tell the story of the scenario, but the passwords are just noise: the passwords of the users have nothing to do with what's being tested and in fact are making it harder to read. For example, Sue has a different password than Dave. As you read the scenario, wondering whether this is relevant, you're distracted from the main point of the scenario: to check that Dave can see Sue's email.

We call details like the passwords *incidental details*,⁵ which are details that are mentioned in the scenario but that actually have no relevance to the purpose of the scenario. This kind of irrelevant detail makes the scenario harder to read, which in turn can cause your stakeholders to lose interest in reading them. Let's rewrite the scenario without the passwords:

```
Scenario: Check inbox
Given a User "Dave"
And a User "Sue"
And an email to "Dave" from "Sue"
When I sign in as "Dave"
Then I should see 1 email from "Sue" in my inbox
```

This is definitely an improvement, making it easier to read and understand the essence of the scenario. Let's try stripping away some more of the noise:

```
Scenario: Check inbox
Given I have received an email from "Sue"
When I sign in
Then I should see 1 email from "Sue" in my inbox
```

Now we have a simple three-step scenario that's clear and concise. It's also more maintainable: if our product owner wants us to change the authentica-

5. This term comes from the excellent paper "Writing Maintainable Acceptance Tests" by Dale Emery: http://dhemery.com/pdf/writing_maintainable_automated_acceptance_tests.pdf.

tion mechanism, we can just rewrite the underlying step definition code without having to touch the features.

Avoiding Incidental Details

If you're a programmer, you're probably practiced at filtering out irrelevant details as you read code each day. Bear that in mind when you write scenarios, because you may not even notice these incidental details slipping in.

Try to avoid being guided by existing step definitions when you write your scenarios and just write down exactly what you want to happen, in plain English. In fact, try to avoid programmers or testers writing scenarios on their own. Instead, get nontechnical stakeholders or analysts to write the first draft of each scenario from a purely business-focused perspective or ideally in a pair with a programmer to help them share their mental model. With a well-engineered support layer, you can confidently and quickly write new step definitions to match the way the scenario has been expressed.

Imperative Steps

In computer programming, there are two contrasting styles for expressing the instructions you give to a computer to make it do something for you. These styles are called *imperative programming* and *declarative programming*.

Imperative programming means using a sequence of commands for the computer to perform in a particular order. Ruby is an example of an imperative language: you write a program as a series of statements that Ruby runs one at a time, in order. A declarative program tells the computer *what* it should do without prescribing precisely *how* to do it. CSS is an example of a declarative language: you tell the computer what you want the various elements on a web page to look like, and you leave it to take care of the rest.

Gherkin is, of course, an imperative language. Cucumber executes each step in a scenario, one at a time, in the sequence you've written them in. However, that doesn't mean those steps need to be read like the instructions for assembling a piece of flat-pack furniture. Let's take a look at a typical example written in an imperative style:

```
Scenario: Redirect user to originally requested page after logging in
  Given a User "dave" exists with password "secret"
  And I am not logged in
  When I navigate to the home page
  Then I am redirected to the login form
  When I fill in "Username" with "dave"
  And I fill in "Password" with "secret"
  And I press "Login"
```

Then I should be on the home page

What's good about this scenario? Well, it uses very generic step definitions, like `/^I fill in "(.)" with "(.)"$/`, which means you can write lots of scenarios like this without having to create much step definition code. You could also probably just about make the argument that it acts as a guide to what the user interface will look like, since it names the fields and buttons that will be used in the login form.

However, when a team uses such an imperative style for their step definitions, it won't be long before they're experiencing the pain of brittle tests *and* bored stakeholders. Scenarios written in this style are not only noisy, long, and boring to read, but they're easy to break: if the user-experience people decided to change the wording on the submit button from Login to Log in, the scenario will fail, for no good reason at all.

Worst of all, scenarios that use generic step definitions like this are failing to create their own domain language. The language of this scenario, using words and phrases like *fill in* and *press*, is expressed in the domain of user interface widgets, a generic and relatively low-level domain.

Use a Declarative Style Instead

Let's raise the level of abstraction in this scenario and rewrite it using a more declarative style:

```
Scenario: Redirect user to originally requested page after logging in
Given I am an unauthenticated User
When I attempt to view some restricted content
Then I am shown a login form
When I authenticate with valid credentials
Then I should be shown the restricted content
```

The beauty of this style is that it is not coupled to any specific implementation of the user interface. This same scenario could apply to a thick-client or mobile application. The words it uses aren't technical and are instead written in a language (*unauthenticated*, *restricted*, *credentials*) that any stakeholder interested in security should be able to clearly understand. It's by expressing every scenario at this level of abstraction that you discover your team's ubiquitous language.

It's true that using declarative style will mean you have to write more step definitions, but you can keep the code in those step definitions short and easy to maintain by pushing the actual work off into helper methods in your support code. We'll show you how to do this in [Chapter 8, Support Code, on page 133](#).

The Spectrum from Imperative to Declarative

In Gherkin features, there isn't a clear line between imperative and declarative styles. Instead, there's a spectrum, and the right place on that spectrum for each step in each scenario depends on lots of things: the area of the system you're describing, the kind of application you're building, the domain expertise of the programmers, and the level of trust that the nontechnical stakeholders have in the programmers. If your stakeholders want to see a lot of detail in the features, it may indicate that you need to work on that trust, but it may just mean you're working on the kind of system that needs to be specified in lots of detail.

You can take declarative style too far, removing so much of the detail from a scenario that it loses its ability to tell a story:

```
Scenario: The whole system
  Given the system exists
  When I use it
  Then it should work, perfectly
```

This scenario is ridiculous, of course, but it illustrates what can happen when you raise the level of abstraction so high that your scenario doesn't tell the reader anything interesting at all. A team that used this scenario would need an incredibly high level of trust in their programmers. We're encouraging you to push your team toward the more abstract, declarative end of the spectrum, but as always, the most important thing is to work with your nontechnical stakeholders to get the level right for them.

Duplication

All good computer programmers understand how destructive duplication is to the maintainability of their code. Yet we often see duplication rife in teams' Cucumber features. Duplication obviously makes your scenarios brittle, but it also makes them boring.

Gherkin has the Background and Scenario Outline keywords you can use to reduce duplication, as we showed you in [Chapter 5, Expressive Scenarios, on page 61](#), but stay vigilant for where the duplication is a sign that your steps are written at too low a level of abstraction. If you have steps that are too imperative, any amount of moving them into Backgrounds or Scenario Outlines won't help you. Work with nontechnical members of your team to get their feedback about the kind of duplication they can accept and the kinds that make their eyes glaze over.

Let Your Examples Flow

by: Dan North

The DRY principle⁶ (Don't Repeat Yourself) says that the definition of any concept should appear once and only once in your code. This is an admirable aim in that if you have to change the behavior of the system, you want to be able to change it in one place and be confident that your change will apply consistently across the codebase. If there are multiple definitions of that behavior scattered around the code, the chances are that not only will you not catch them all but someone before you didn't catch them all and the multiple definitions are already inconsistent, and who wants that?

However, when you are using examples to drive your code, there is another principle in play that I believe trumps the DRY principle: *the examples should tell a good story*. They are the documentation narrative that will guide future programmers (including you when you come back to change this code in three months time and you've forgotten what it does). In this case, clarity of intent is found in the quality of the narrative, not necessarily in minimizing duplication.

Some years ago I had my first experience of pair programming with Martin Fowler. That is, I had done quite a bit of pair programming, just not with Martin before. We were looking at some Ruby code I had written test-first, and Martin asked to see the tests "to find out what the code does." Then he did a rather odd thing. He started *moving the tests around*. I had a few helper classes and utility methods in the source file, neatly at the end out of the way. He moved them up and dropped them inline just ahead of the first test that used them.

Madness! I thought—now the supporting code is all over the place! It really offended my sense of tidiness. But then I saw a pattern beginning to emerge: the test code was starting to read like a story. He would introduce these little methods and classes just before their one walk-on line in the narrative. It was quite an eye-opener for me. The test code flowed and unfolded the story of the class under test.

The a-ha! moment for me was when I imagined reading a book where the plot and characters had been DRYed out. Everything would be in footnotes or appendixes. All the character descriptions, plot elements, subtexts, and so on, would be carefully extracted into fully cross-referenced paragraphs. That is great if you are reading an encyclopedia but not so appropriate if you want to get into the flow and find out what happens. You would be forever flicking back and forth in the book, and you would very quickly forget where you even were in the story. In the words of the old joke, dictionaries have lousy plots, but at least they explain all the words as they go.

Some people refer to this as the DAMP principle: Descriptive and Meaningful Phrases. When you're writing examples, readability is paramount, and DAMP trumps DRY.

Ubiquitous What?

The ubiquitous language your team uses will be driven by the domain you're working in. If you're building a system for live-music fans, your ubiquitous language will include words like *concert*, *performance*, *artist*, and *venue*. If

6. *The Pragmatic Programmer: From Journeyman to Master* [HTOO]

you're building a catalog of TV shows, you'll have words like *broadcaster*, *genre*, *duration*, and *transmission date* in your ubiquitous language.

The point is for *everyone* on the team to use the same words, everywhere. It's not OK to have a database table called `tbl_Performer` if the rows in that table represent things that most of the team refers to as *artists*. Wherever you see schism like this, stop, decide which is the right word to use, make the appropriate correction, and then stick with it.

We talk about *developing* a ubiquitous language because it's an ongoing process. That development takes work. It takes effort to really listen to one another and agree on the words you'll use, and it takes discipline to stick to those commitments.

The rewards are great. Teams that use a ubiquitous language make fewer mistakes and *enjoy* their work more because they can communicate effectively about the work. When a team doesn't appreciate the value of a ubiquitous language, they'll be careless with the wording of their scenarios, missing a valuable opportunity to build strong bridges between the technical and business-focused sides of the team. When you try to correct people or clarify terminology, you might end up feeling like you're just being picky.

Take time to explain the concept of a ubiquitous language to your team and what its benefits are. Once everyone understands why it's important, you'll find they're much more willing to help make the effort to discuss and decide on the right words to use.

Used correctly, Cucumber helps a team to develop their ubiquitous language. When programmers and businesspeople work together to write scenarios, you'll find all kinds of arguments breaking out about how precisely to word things. Great! Each of those disagreements has exposed a potential misunderstanding between the two groups, in other words, a bug magnet. For a new team, these sessions can be hard at first, but as the language develops, they get easier and easier. [Three Amigos, on page 96](#) is a good way to structure these meetings.

Siloed Features

Cucumber can feel like a very technical tool. It's run from the command line, and the feature files are designed to be checked into source control along with the code that they test. Yet it's supposed to help the business stakeholders on your team feel more in control of the development process. When testers and developers tuck their features away in source control, the rest of the team

Three Amigos

The best Gherkin scenarios are created when the three amigos come together, bringing three very different perspectives:

- The first amigo is a tester, who thinks about how to break things. The tester will typically come up with lots of scenarios, sometimes covering obscure edge cases and sometimes covering very important ones that no one else had thought of.
- The second amigo is a programmer, who thinks about how to make things. The programmer will typically add steps to scenarios, as he asks clarifying questions about what exactly should happen.
- The third amigo is the product owner, who cares about scope. When the tester thinks of scenarios that hit edge cases the product owner doesn't care about, she can tell the team to drop that scenario out of scope altogether, or the group can decide to test it with unit tests only. When the programmer explains that implementing a particular scenario will be complicated, the product owner has the authority to help decide on alternatives or to drop it altogether.

Many teams practicing BDD have found the three amigos make a great partnership for thrashing out Gherkin scenarios that the whole team believes in. You don't have to stop at three amigos, though: invite your team's user experience specialists or operations staff, for example, if the feature being discussed will affect them.

can feel as though their documentation had been locked away in a cupboard to which they don't have the keys.

Your features act as a design tool for specifying new features, but they also act as a great reference document for what the system already does today. For a system of any significant size, no one person will remember exactly what it will do in every situation, so when you get a bug report from a user or are considering adding new functionality to some part of the system, you want this reference right at your side.

Cucumber itself has only limited support for sharing features in a way that's accessible for nontechnical audiences, but there are plenty of plug-ins and tools springing up around it that do. For example, if you use GitHub for source control, the pages for your project will have syntax-highlighted features that people can even comment on.

Relish⁷ is a service that was created by members of the Cucumber and RSpec teams to provide an easy way to publish Cucumber features as documentation.

7. <http://relishapp.com>

The RSpec project now uses its Relish documentation as its home page, and your team can use it too.

You can achieve at least half of the benefit of Cucumber just by having the discipline to sit down with your business stakeholders and write scenarios collaboratively. The conversations sparked by that process will uncover so many potential bugs or schedule overruns that you'll already have made a huge win, even if you choose to never automate your features.

Assuming you do want to automate them, however, read on to find out how to do it well.

6.3 Caring for Your Tests

The benefit of automating your features is that you'll be able to trust them as living documentation in the long run, because you'll be checking each scenario against the production code to make sure it's still valid. For the programmers working on the code, there's another benefit too: those tests act as a safety net when they're working on the system, alerting them to any mistakes they make that break existing behavior.

So, your features work as a feedback mechanism to the whole team about the behavior of the system and to the programmers about whether they've broken anything. For these feedback loops to be useful, the tests need to be fast and they need to be reliable. Let's start by looking at problems that affect the reliability of your tests.

Leaky Scenarios

Cucumber scenarios are basically state-transition tests: you put the system into a Given state A, you perform action X (When), and Then you check that it has moved into expected state B. So, each scenario needs the system to be in a certain state before it begins, and yet each scenario also leaves the system in a new, dirty state when it's finished.

When the state of the system is not reset between tests, we say that they allow state to *leak* between them. This is a major cause of brittle tests.

When one scenario depends upon state left behind by another earlier scenario in order for it to pass, you've created a dependency between the two scenarios. When you have a chain of several scenarios depending on each other like this, it's only a matter of time before you end up with a train wreck.

If that first scenario, the one that happens to leave the system in just the right state for the next one to pick it up, is ever changed, the next scenario will suddenly start to fail. Even if you don't change that earlier scenario, what

Test Data Builders

If you're using Ruby, you're probably familiar with the FactoryGirl^a gem. FactoryGirl is an excellent implementation of the Test Data Builder^b pattern. For the uninitiated, here's a quick summary of its benefits.

Suppose you're testing a payroll system and you need to create a PayCheck record as part of a scenario. The way your domain model is structured, a PayCheck needs an Employee, and the Employee in turn needs an Address. Each of them also has a few mandatory fields. Instead of having to create all these objects individually in your step definition code or having a big fat set of fixture data, you can simply say this:

```
Given /^I have been paid$/ do
  Factory :pay_check
end
```

Once you've configured FactoryGirl with the structure of your data model (see the FactoryGirl documentation for details on how this is done), all you need to do is ask her for a PayCheck, and she'll create not only the PayCheck record but all the dependent objects as well, setting the mandatory fields with reasonable default values. If you care about a field having a specific value, you can tell her to override the default:

```
Given /^I have been paid 50 dollars$/ do
  Factory :pay_check, :amount => 50
end
```

When it's this easy to create data, you no longer need to carry around the baggage of big fixture data sets. There's a small amount of up-front investment in creating these builders, of course, but it quickly pays off in reliable, readable scenarios and step definition code. If your team isn't using Ruby, they could still point ActiveRecord and FactoryGirl at their database with minimal effort. Otherwise, look for similar tools for your language.

-
- a. http://rubygems.org/gems/factory_girl
 - b. <http://www.natpryce.com/articles/000714.html>

happens if you want to run only the second scenario on its own? Without the state leaked out by the earlier scenario, it will fail.

The opposite of this, *independent scenarios*, ensures they put the system into a clean state and then add their own data on top of it. This makes them able to stand on their own, rather than being coupled to the data left behind by other tests or shared fixture data. Investing in building up a good reliable library of [Test Data Builders, on page 98](#) makes this much easier to achieve.

We can't stress enough how fundamental independent scenarios are to successful automated testing. Apart from the added reliability of scenarios that independently set up their own data, they're also clearer to read and debug. When you can see precisely what data is used by a scenario just by reading



Matt says:

Fixture Is an Overloaded Term

The word *fixture* has at least three meanings in the domain of automated testing, which can cause confusion. We've used the term *fixture data* in this chapter to mean data that's used to set up the context for a scenario or test case. This is the common meaning of the term as used in various xUnit testing tools^a and by the Ruby on Rails framework.

There is a long tradition (coming from the hardware world, where test fixtures originated) of calling the link between the test system and the system under test a fixture. This is the “glue code” role that we've referred to in this book as *automation code*. The FIT testing framework^b uses this meaning of the term.

Some unit testing tools (such as NUnit) have further confused the issue by referring to the test case class itself as a fixture. So much for a ubiquitous language!

-
- a. [xUnit Test Patterns \[Mes07\]](#)
 - b. <http://fit.c2.com/>

it, rather than having to root around in a fixture data script or, worse, in the database itself, you'll be able to understand and diagnose a failure much more easily.

Race Conditions and Sleepy Steps

If you write end-to-end integration tests for a reasonably complex system, you'll eventually encounter this problem. Race conditions occur when two or more parts of the system are running in parallel, but success depends on a particular one of them finishing first. In the case of a Cucumber test, your When step might cause the system to start some work that it runs in the background, such as generating a PDF or updating a search index. If this background task happens to finish before Cucumber runs your Then step, the scenario will pass. If Cucumber wins the race and the Then step executes before the background task is finished, the scenario will fail.

When it's a close race, you'll have a flickering scenario, where the scenario will pass and fail intermittently. If there's a clear winner, a race condition can go unnoticed for a long time, until a new change to the system evens up the stakes, and the scenario starts to fail at random.

A crude solution to this problem is to introduce a fixed-length pause or *sleep* into the scenario to give the system time to finish processing the background task. Although this is definitely a useful technique in the very short term to

diagnose a race condition, you should resist the temptation to leave a sleep in your tests once you understand the cause of the problem. Introducing sleepy steps won't solve the flickering problem but just make it less likely to happen. In the meantime, you've added a few extra seconds to your total test run time, swapping one set of problems for another.

If we really had to choose, we'd choose slow, reliable tests over faster unreliable ones, but there's no need to make this compromise. When testers and programmers pair up to automate scenarios, they can craft tests that are built with knowledge of how the system works. This means they can make use of cues from the system to let the tests know when it's safe to proceed, so instead of using crude fixed-length sleeps, the tests can proceed as quickly as possible. For an example of working with asynchronous code and further detail, see [Chapter 9, Dealing with Message Queues and Asynchronous Components, on page 157.](#)

Shared Environments

This is a problem that we've often found in teams that are transitioning from a manual acceptance testing regime to using automated acceptance tests. Traditionally, the manual testers on the team would have a special environment, often called *system test*, where a recent build of the system would be deployed. They'd run their manual tests in that environment, reporting bugs back to the development team. If more than one team member needed to run tests in that same environment, they'd communicate between each other to make sure they didn't tread on one another's toes.

If it's even slightly awkward to install the system in a new environment, the likelihood is that when the team starts to automate their tests, they'll follow the path of least resistance and point their test scripts at this existing system test environment. Now the environment is shared between not only the human members of the team but the test scripts too. Suppose a developer gets a bug report and wants to reproduce it for himself. He logs in to the system test environment and clicks a few buttons, but he doesn't realize the automated tests are running at the same time. As part of the steps to reproduce the bug, the developer unwittingly deletes a database record that the automated test relied on, and the automated test fails. This kind of situation is a classic cause of flickering scenarios.

The shared use of a single environment can also contribute to unreliable tests by causing heavy and inconsistent load on in-demand resources like databases. When the shared database is under excessive load, normally reliable tests will time out and fail.

To deal with this problem, it needs to be so easy to spin up the system in a new environment that you can do it for fun. You need a [One-Click System Setup, on page 102](#).

Tester Apartheid

Testers are too often unfairly regarded as second-class citizens on a software team. As we'll explain in [Chapter 8, Support Code, on page 133](#), developing a healthy suite of Cucumber features requires not only testing skill but programming skill too. When testers are left alone to build their own Cucumber tests, they may lack the software engineering skill to keep their step definition and support code well organized. Before you know it, the tests are a brittle muddle that people are scared to change.

Combat this problem by encouraging programmers and testers to work together when writing step definition and support code. The programmers can show the tester how to keep the code organized and factor out reusable components or libraries that other teams can use. This is precisely how libraries like Capybara (see [Chapter 15, Using Capybara to Test Ajax Web Applications, on page 245](#)) were created, when a programmer extracted reusable code from their team's step definitions. By pairing with testers like this, programmers also develop a better understanding of what it takes to make their code *testable*.

When Cucumber is being used to good effect on a team, a tester should be able to delegate the work of running basic checks to Cucumber. This frees them up to do the much more interesting, creative work of *exploratory testing*, as explained in *Agile Testing: A Practical Guide for Testers and Agile Teams* [CG08].

Fixture Data

When manually testing a system, it's useful to populate it with realistic data so that you can wander around the system just like you would in the live application. When your team transitions from manual to automated tests, you can often be tempted to just port over a subset of production data so that the automated tests have a functioning system to work with straightaway.

The alternative, having each test set up its own data, might seem like it's just too hard. In a legacy system—especially one where the design has evolved organically—where creating the single object actually needed for the test you're working on means you need to create a huge tree of other dependent objects, you'll feel like the easiest option is just to create this tree once in the fixture data and then share it with all the other tests.

One-Click System Setup

To avoid flickering scenarios that result from using shared environments, the team needs a setup script that will create a new instance of the system from scratch, at the click of a button.

If the system has a database, the database generated by the script should contain the latest schema, as well as any stored procedures, views, functions, and so on. It should contain just the very minimum baseline data necessary for the system to be able to function, such as configuration data. Any more should be left for the independent scenarios to create for themselves.

If there are message queues, or memcache daemons, the setup script should start them too, with the minimal configuration that you'd expect to be there on the running system.

Even if your team is not using Ruby for their production code, they could experiment with using Ruby's ActiveRecord^a gem for managing database schemas and migration scripts. ActiveRecord turns this kind of chore into a breeze.

a. <http://rubygems.org/gems/activerecord>

This approach has a couple of significant problems. A set of fixture data, even if it starts out relatively lean, will only tend to grow in size over time. As more and more scenarios come to rely on the data, each needing their own specific little tweaks, the size and complexity of the fixture data grows and grows. You'll start to feel the pain of brittle features when you make a change to the data for one scenario, but that change causes other scenarios to fail. Because you have so many different scenarios depending on the fixture data, you'll tend to plaster on more data because it's safer than changing the existing data. When a scenario does fail, it's hard to know which of the masses of data in the system could be relevant to the failure, making diagnosis much more difficult.

When you have a large set of fixture data, it can be slow to set it up between each test. This puts pressure on you to write leaky scenarios that don't reset the state of the system between each scenario, because it's quicker not to do so. We've already explained where this can lead.

We consider fixture data to be an antipattern. We much prefer using [Test Data Builders](#), on page 98 like FactoryGirl¹³ where the relevant data is created within the test itself, rather than being buried away in a big tangled set of fixture data.

13. https://github.com/thoughtbot/factory_girl

Nightly Build

When you have [Slow Features, on page 87](#), caused by [Lots of Scenarios, on page 103](#), it's worth considering splitting your build in two. Use tags to annotate the scenarios that should be run on every check-in, and demote the rest to a nightly build.

Use of this pattern depends on your team's appetite for risk, as well as their tendency to make mistakes. The scenarios that should be demoted to a nightly build are the ones that rarely, if ever, fail. They're the scenarios for functionality that hasn't changed for months, and they cover stable code that isn't being worked on. They're the scenarios that if you had to, you'd be prepared to delete altogether.

There's a maintenance overhead in keeping the tags for the check-in build on the right scenarios. Over time, some of those scenarios will stabilize and should be demoted to the nightly build, to be replaced by newer scenarios.

Although a nightly build can be a good way to get you out of a hole, usually the right long-term solution is to break up your [Big Ball of Mud, on page 104](#).

Lots of Scenarios

It might seem like stating the obvious, but having a lot of scenarios is by far the easiest way to give yourself a slow overall feature run. We're not trying to suggest you give up on BDD and go back to cowboy coding, but we do suggest you treat a slow feature run as a red flag. Having lots of tests has other disadvantages than just waiting a long time for feedback. It's hard to keep a large set of features organized, making them awkward for readers to navigate around. Maintenance is also harder on the underlying step definitions and support code.

We find that teams that have a single humongous build also tend to have an architecture that could best be described as a big ball of mud. Because all of the behavior in the system is implemented in one place, all the tests have to live in one place, too, and have to all be run together as one big lump. This is a classic ailment of long-lived Ruby on Rails applications, which tend to grow organically without obvious interfaces between their subsystems.

We'll talk more about what to do with big balls of mud in the next section. It's really important to face up to this problem and tackle it once you realize it's happening, but it isn't a problem you'll be able to solve overnight.

In the meantime, you can keep your features organized using subfolders and tags (see [Chapter 5, Expressive Scenarios, on page 61](#)). Tagging is especially helpful, because you can use tags to partition your tests. You can choose to run partitioned sets of tests in parallel or even demote some of them to run in a [Nightly Build, on page 103](#).

It's also worth thinking about whether some of the behavior you've specified in Cucumber scenarios could be pushed down and expressed in fast unit tests instead. Teams that enthusiastically embrace Cucumber sometimes forget to write unit tests as well and rely too much on slow integration tests for feedback. Try to think of your Cucumber scenarios as broad brush strokes that communicate the general behavior of the code to the business, but still try to get as good a coverage as you can from fast unit tests. Help make this happen by having testers and programmers work in pairs when implementing Cucumber scenarios. This pair can make good decisions about whether a piece of behavior necessarily needs to be implemented in a slow end-to-end Cucumber scenario and drive out the behavior using a fast unit test instead.

Big Ball of Mud

The Big Ball of Mud¹⁴ is an ironic name given to the type of software design you see when nobody has really made much effort to actually do any software design. In other words, it's a big, tangled mess.

We've explained where a big ball of mud will manifest itself as problems in your Cucumber tests: slow features, fixture data, and shared environments are all examples of the trouble it can cause. Look out for these signals and be brave about making changes to your system's design to make it easier to test.

We suggest Alistair Cockburn's *ports and adapters* architecture¹⁵ as a way of designing your system to be testable. Michael Feathers' *Working Effectively with Legacy Code* [Fea04] gives many practical examples of breaking up large systems that weren't designed to be tested.

Hold regular sessions with your team to talk about your architecture: what you like, what you don't like, and where you'd like to take it. It's easy to get carried away with ambitious ideas that evaporate into thin air soon after you get back to your desks, so make sure you try to leave these sessions with realistic, practical steps that will move you in the right direction.

That covers the most common problems you're likely to hit as you adopt Cucumber into your team. However, it's one thing to understand these problems, but it's quite another to get the time to do anything about them. Next we'll talk about an important technique for making that time.

14. <http://www.laputan.org/mud/>

15. <http://alistair.cockburn.us/Hexagonal+architecture>

6.4 Stop the Line and Defect Prevention

Of all the activities that your team does, which do you think is the most important? Writing code for new features? Fixing bugs found in testing? Fixing bugs found in production? Speeding up your features?

Sadly, test maintenance does not come near the top of most software teams' list of priorities. If the elevators are broken in your office building, you can be sure that someone will be on the phone to the facilities team straightaway. When your tests are slow or brittle, the problem is invisible to everyone but the programmers and testers who rely on them. If you do test maintenance at all, you generally do it when things have gotten so bad that you can't stand it any longer, or you simply can't get a release out because the tests are so broken. There just always seems to be something more important to do.

Team members who think in this way about their tests have got it all wrong. The automated tests are the heartbeat of the team that relies on them, and they need meticulous care and attention to keep them healthy.

Stop the Line at Toyota

In Toyota's manufacturing plants, every shop-floor worker has the authority and the responsibility to stop an entire production line whenever a problem arises. The problem is then given immediate and focused attention by experienced staff, and the line is restarted only once the problem has been resolved. Once the line has restarted, a team is tasked with performing a *root-cause analysis* on the problem to understand why it happened so that the source of the problem can be resolved.

When Taichi Ohno¹⁶ first introduced this idea, his managers thought he was crazy. At the time, it was taken for granted within the manufacturing industry that the most important thing you could do was keep your assembly lines running, day and night if necessary.

When Ohno first told his managers to implement this new system, some of them listened and some of them didn't. At first, the managers who had implemented the policy saw their productivity drop. Stopping to deal with each problem immediately was slowing them down, and when they compared their output numbers with the managers who had ignored their boss, it looked like the boss had got it wrong.

Gradually, however, those managers who allowed their lines to stop and deal with every problem started to see their lines stopping less frequently. Because each problem was being dealt with using defect prevention, those lines had been investing in continuously improving the quality of the machines and processes that ran the line. That investment started to pay off, and their lines started to get faster and faster. Soon their output was much greater than that of the lines controlled by the managers who had ignored their apparently crazy boss. Their production lines were still clunking along at the same old rate, suffering the same old problems.

16. *Toyota Production System - Beyond Large Scale Production* [Ohn88]

Defect Prevention

Toyota's counterintuitive but hugely successful policy of stopping the line works because it's part of a wider process, known as *defect prevention*, that focuses on continuously improving the manufacturing system. Without this wider process, stop the line itself would have very little effect. There are four steps to this process:

1. Detect the abnormality.
2. Stop what you're doing.
3. Fix or correct the immediate problem.
4. Investigate the root cause and install a countermeasure.

This fourth step is crucial because it seizes the opportunity offered by the problem at hand to understand something more fundamental about your process. It also means that fixing things becomes a habit, rather than something you put off to do someday later when you're not in such a hurry.

For example, suppose the build has broken with a failing test. It turns out that the guy who pushed the commit with the failing test didn't run all the tests before he pushed. Why not? Well, it turns out that he thinks the tests take too long to run, so he ran just what he thought were the ones covering the change he made and then crossed his fingers and pushed his commit anyway. So, the underlying cause is that the features are slow. Now that we understand the root cause, we can work to fix it.

Some teams keep a log of build failures, recording the root cause each time. When they have sufficient evidence that a particular root cause is worth tackling, they can put some concentrated effort into tackling it properly.

Imagine your team as a production line, cranking out valuable features for your users. If you spot a problem that's slowing the production line down, *stop the line* and fix the problem for good. Implementing stop the line means you've decided to make a fast, high-quality, reliable test run your whole team's top priority, second only to dealing with production issues that are affecting your customers. When there's a problem with the tests—whether that's an urgent problem like a failing test or a nagging annoyance like a flickering scenario—put your best people on it, and fix it forever.

6.5 What We Just Learned

Cucumber features are a valuable asset to your company. We've seen teams that have ripped out and rewritten big parts of their systems, safe in the knowledge that they had a set of accurate, executable specifications to ensure the new solution worked just as well as the original. To those teams, the

features were more valuable than the production code itself. If you're going to invest in writing Cucumber features, you need to protect that investment by caring for them so that they're as useful as possible to the whole team. Don't settle for features that are slow, that fail intermittently, or that are read by only half the team: iron out problems as they happen, and use each problem as a reason to make the tests even better than they were before.

Cucumber might just seem like a testing tool, but at its heart it's really a *collaboration* tool. If you make a genuine effort to write features that work as documentation for the nontechnical stakeholders on your team, you'll find you are forced to talk with them about details that you might never have otherwise made the time to talk about. Those conversations reveal insights about their understanding of the problem, insights that will help you build a much better solution than you would have otherwise. This is Cucumber's big secret: the tests and documentation are just a happy side effect; the real value lies in the knowledge you discover during those conversations.

Try This

Here are some exercises for you to try for yourself.

Defect Prevention on Your Team

Think of three things that are slowing down your team's production line. What is the root cause of each of them? What could you do to change them for the better?

Incidental Details Practice

Here is a scenario, written in a hideously imperative style, riddled with incidental details:

```
Scenario: Create an invoice
Given I am a signed in user with role: admin
And a client "Test Client" exists with name: "Test Client"
And a project "Test Project" exists with:
| name | "Test Project" |
| client | client "Test Client" |
And an issue "Test Issue" exists with:
| project | project "Test Project" |
| name | "Test Issue" |
And a work_unit "Test Work Unit" exists with:
| issue | issue "Test Issue" |
| completed_on | "2011-08-24" |
| hours | "7.5" |
And I am on the admin invoices page
Then I should see "Test Client"
And I follow "Test Client"
```

```
And I fill in "invoice_id" with "abc"
And I press "Submit"
Then I am on the admin invoices page
And I should not see "Test Client"
```

Start by just trying to work out what's going on: what do you think this system does? What is the purpose of the scenario? What behavior is it trying to test? Notice how the incidental details are like overgrown weeds, getting in the way of you figuring out what the test is actually trying to do.

Now that you understand the *essence* of the scenario, rewrite it in your own words. You should need much fewer steps, but you might want to consider using more than one scenario.

Now that you've rewritten the scenario in a more declarative style, can you spot the crucial Then step that was missing from the original scenario?

Part II

A Worked Example

Step Definitions: On the Inside

It's time to pull together everything that you've learned in the first part of the book and use it in practice. There are a few advanced concepts left about Cucumber that we want to explain to you, and they'll be much easier to demonstrate with an example. A lot of what we'll do in this part of the book will blur the line between testing and development. If you're more of a tester than a developer, don't let that worry you: the Ruby code we'll build is just about as simple as it gets. By following along, you'll get a good sense of how we like to work, as well as pick up some new knowledge about working with Cucumber.

At the end of [Section 4.5, Returning Results, on page 52](#), we'd just started work on a greenfield project to build the software for an ATM. We had a single scenario for the most important behavior of the system: letting someone walk up to the machine and withdraw cash.

[Download step_definitions_inside/01/features/cash_withdrawal.feature](#)

Feature: Cash Withdrawal

Scenario: Successful withdrawal from an account in credit

Given I have deposited \$100 in my account

When I request \$20

Then \$20 should be dispensed

Now we're going to pick up this scenario and work outside-in, designing the system as we go, just as we would on a real project. In this chapter, we'll get the scenario to pass by driving out a simple domain model for our ATM. Then, in the next chapter, we'll get a nasty surprise when we discover that there's a missing step in our scenario. Finally, we'll demonstrate the benefits of well-engineered test code by introducing a user interface around the domain model.

By the end of this chapter, you'll have learned about Cucumber's World and how you can use it to contain state that's shared between step definitions. We'll write some custom helper methods that will introduce a layer of decoupling between our step definitions and the application we're building. We'll show you how to use *transforms* to reduce duplication in your step definitions and make their regular expressions more readable. Finally, we'll show you how we like to organize the files in our projects so that they're easy to work with and maintain.

7.1 Sketching Out the Domain Model

The heart of any object-oriented program is the domain model. When we start to build a new system, we like to work directly with the domain model. This allows us to iterate and learn quickly about the problem we're working on without getting distracted by user interface gizmos. Once we have a domain model that really reflects our understanding of the system, it's easy to wrap it in a pretty skin.

We're going to let Cucumber drive our work, building the domain model classes directly in the step definitions. As usual, we start by running cucumber on our scenario to remind us what to do next:

`Feature: Cash Withdrawal`

```
Scenario: Successful withdrawal from an account in credit
  Given I have deposited $100 in my account
    uninitialized constant Account (NameError)
    ./features/step_definitions/steps.rb:2
    features/cash_withdrawal.feature:3
  When I request $20
  Then $20 should be dispensed
```

`Failing Scenarios:`
`cucumber features/cash_withdrawal.feature:2`

```
1 scenario (1 failed)
3 steps (1 failed, 2 skipped)
0m0.002s
```

When we last worked on this scenario, we'd just reached the point where we had written the regular expressions for each of our step definitions and started to implement the first one. Here's how our steps file looks:

[Download step_definitions_inside/01/features/step_definitions/steps.rb](#)

```
Given /^I have deposited \$(\d+) in my account$/ do |amount|
  Account.new(amount.to_i)
end
```

```
When /^I request \$(\d+)/ do |arg1|
  pending # express the regexp above with the code you wish you had
end
```

```
Then /^$ (\d+) should be dispensed$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end
```

In that first step definition, we've sketched out a call to an imaginary class called Account. Ruby has given us an error that tells us that the next thing we need to do is define the Account class. Let's go ahead and do that:

[Download step_definitions_inside/02/features/step_definitions/steps.rb](#)

```
class Account
  def initialize(amount)
  end
end
```

```
Given /^I have deposited \$(\d+) in my account$/ do |amount|
  Account.new(amount.to_i)
end
```

Notice that we're defining the class right here in our steps file. Don't worry—it's not going to stay here forever, but it's most convenient for us to create it right here where we're working. Once we have a clear idea of how we're going to work with the class, then we can refactor and move it to a more permanent home. We're also converting the amount captured from the Gherkin step as a string into a number before we pass it into the domain model.

Let's run cucumber again and see what it thinks of that:

Feature: Cash Withdrawal

```
Scenario: Successful withdrawal from an account in credit
  Given I have deposited $100 in my account
  When I request $20
    TODO (Cucumber::Pending)
    ./features/step_definitions/steps.rb:13
      features/cash_withdrawal.feature:4
    Then $20 should be dispensed
```

```
1 scenario (1 pending)
3 steps (1 skipped, 1 pending, 1 passed)
0m0.002s
```

Well, that was easy! Perhaps...too easy? Let's review the code in our step definition and see what we think. There are a couple of things we're not happy about:

- There's some inconsistent language creeping in; the step talks about *depositing* funds into the account, but the code passes the funds to the Account class's constructor.
- The step is lying to us! It says Given I have deposited \$100 in my account, and it's passed. Yet we know from our implementation that nothing has been deposited anywhere.
- Having to convert the amount to an integer is messy. If we have a variable called amount, we should expect it to already be a number of some kind, not the string captured by the regular expression.

We'll work through each of these points before we move onto the next step in the scenario.

Getting the Words Right

We want to clarify the wording before we do anything else, so let's think about how we could make the code in the step definition read more like the text in the step. We could go back and reword the step to say something like Given an Account with a balance of \$100. In reality, though, the only way that an account would have a balance is if someone deposited funds into it. So, let's change the way we talk to the domain model inside our step definition to reflect that:

```
Download step_definitions_inside/03/features/step_definitions/steps.rb
class Account
  def deposit(amount)
    end
end

Given /^I have deposited $(\d+) in my account$/ do |amount|
  my_account = Account.new
  my_account.deposit(amount.to_i)
end
```

That seems better.

There's something else in the wording that bothers us. In the step, we talk about *my account*, which implies the existence of a protagonist in the scenario who has a relationship to the account, perhaps a Customer. This is a sign that we're probably missing a domain concept. However, until we get to a scenario where we have to deal with more than one customer, we'd prefer to keep things simple and focus on designing the fewest classes we need to get this scenario running. So, we'll park this concern for now.

Telling the Truth

Now that we're happier with the interface to our Account class, we can resolve the next issue from our code review. After we've deposited the funds in the account, we can check its balance with an assertion:

```
Download step_definitions_inside/04/features/step_definitions/steps.rb
Given /^I have deposited \$\d+ in my account$/ do |amount|
  my_account = Account.new
  my_account.deposit(amount.to_i)
  my_account.balance.should eq(amount.to_i),
    "Expected the balance to be #{amount} but it was #{my_account.balance}"
end
```

We've used an RSpec assertion here, but if you prefer another assertion library, feel free to use that. It might seem odd to put an assertion in a Given step, but it communicates to future readers of this code what state we expect the system to be in once the step has run. We'll need to add a balance method to the Account so that we can run this code:

```
Download step_definitions_inside/04/features/step_definitions/steps.rb
class Account
  def deposit(amount)
  end

  def balance
  end
end
```

Notice that we're just sketching out the interface to the class, rather than adding any implementation to it. This way of working is fundamental to outside-in development. We try not to think about *how* the Account is going to work yet but concentrate on *what* it should be able to do.

Now when we run the test, we get a nice helpful failure message:

Feature: Cash Withdrawal

```
Scenario: Successful withdrawal from an account in credit
  Given I have deposited $100 in my account
    Expected the balance to be 100 but it was
      (RSpec::Expectations::ExpectationNotMetError)
      ./features/step_definitions/steps.rb:15
      features/cash_withdrawal.feature:3

  When I request $20
  Then $20 should be dispensed
```

Failing Scenarios:

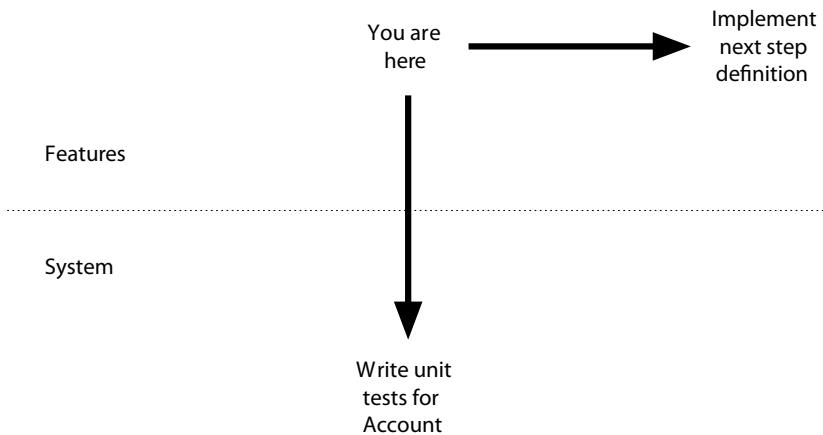
cucumber features/cash_withdrawal.feature:2

```
1 scenario (1 failed)
3 steps (1 failed, 2 skipped)
0m0.002s
```

Now our step definition is much more robust, because we know it will sound an alarm bell if it isn't able to deposit the funds into the account as we've asked it to do. Adding assertions to Given and When steps like this means that if there's ever a regression later in the project, it's much easier to diagnose because the scenario will fail right where the problem occurs. This technique is most useful when you're sketching things out; eventually, we'll probably move this check further down the testing stack into a *unit test* for the Account and take it out of the step definition.

Doing the Simplest Thing

We're at a decision point here. We've effectively finished implementing our first step definition, but we can't move on to the next one until we've made some changes to the implementation of the Account class so that the step passes.



It's tempting to pause here, move the Account class into a separate file, and start driving out the behavior we want using unit tests. We're going to try to resist that temptation for now and stay on the outside of the Account class. If we can get a full tour through the scenario from this perspective, we'll be more confident in the design of the class's interface once we do step inside and start implementing it.

So, we'll make a very simple implementation in the Account class that's obviously incomplete but is just right enough to make this first step pass. Think of this like putting up scaffolding on a construction site: we're going to take it down eventually, but it will help things to stand up in the meantime.

Change Account to look like this, and the first step should pass:

```
Download step_definitions_inside/05/features/step_definitions/steps.rb
class Account
  def deposit(amount)
    @balance = amount
  end

  def balance
    @balance
  end
end
```

Good. We still have one issue left on our list, which is the duplication of the calls to `to_i`. Now that our step is passing, we can do that refactoring with confidence.

7.2 Removing Duplication with Transforms

Another issue we have with this step definition is that we have to convert the string captured by the regular expression into an integer. In fact, now that we've added an assertion, we've had to do it twice. As our test suite grows, we can imagine these calls to `to_i` littering our step definitions. Even these four characters count as duplication, so let's stamp them out.

To do this, we're going to learn about a new Cucumber method, called `Transform`.

Transforms work on captured arguments. Each transform is responsible for converting a certain captured string and turning it into something more meaningful. For example, we can use this transform to take a matched argument that contains a number and turn it into a Ruby `Fixnum` integer:

```
Download step_definitions_inside/06/features/step_definitions/steps.rb
Transform /^d+$/ do |number|
  number.to_i
end
```

We define the transform by giving Cucumber a regular expression that describes the argument we're interested in transforming. Notice that we've used the `^` and `$` to anchor the transform's regular expression to the ends of the captured string. This is really important, because we want our transform to match only captures that are numbers, not just captures that contain a number somewhere in them.

When Cucumber matches a step definition, it checks for any transforms that match each argument. When an argument matches a transform, Cucumber passes the captured string to the transform's block, and the result of running

the block is what's then yielded to the step definition. This is shown in [Figure 6, Transforms: how do they work?, on page 119](#).

With the transform in place, we can now remove the duplicated calls to `to_i` in our step definition:

[Download step_definitions_inside/06/features/step_definitions/steps.rb](#)

```
Given /^I have deposited \$(\d+) in my account$/ do |amount|
  my_account = Account.new
  my_account.deposit(amount)
  my_account.balance.should eq(amount),
    "Expected the balance to be #{amount} but it was #{my_account.balance}"
end
```

Great! That code looks much cleaner and easier to read. Introducing the transform has brought in a new kind of duplication, though. The regular expression that we use to capture the number is now duplicated: we have `\d+` both in the step definition and in the transform's definition. This could be a problem if we wanted, for example, to start using cents as well as dollars in our features; we'd have to change the regular expression in the step definition and in the transform. Fortunately, Cucumber allows us to define the regular expression once, in the transform, and then reuse it in the step definition, like this:

[Download step_definitions_inside/07/features/step_definitions/steps.rb](#)

```
CAPTURE_A_NUMBER = Transform /^(\d+)/ do |number|
  number.to_i
end

Given /^I have deposited \$(#{CAPTURE_A_NUMBER}) in my account$/ do |amount|
  my_account = Account.new
  my_account.deposit(amount)
  my_account.balance.should eq(amount),
    "Expected the balance to be #{amount} but it was #{my_account.balance}"
end
```

We store the result of calling `Transform` in the constant `CAPTURE_A_NUMBER` and then use that constant as we build the regular expression in the step definition. As well as making it easier to change and reuse this capturing regular expression in the future, this refactoring makes it more obvious to someone reading the step definition that this argument will be transformed.

We can tidy this up a little further by moving the dollar sign into the transform's capture. This makes the code more cohesive, because we're bringing together the whole regular expression statement for capturing the amount of funds deposited. It also gives us the option to capture other currencies in the future.

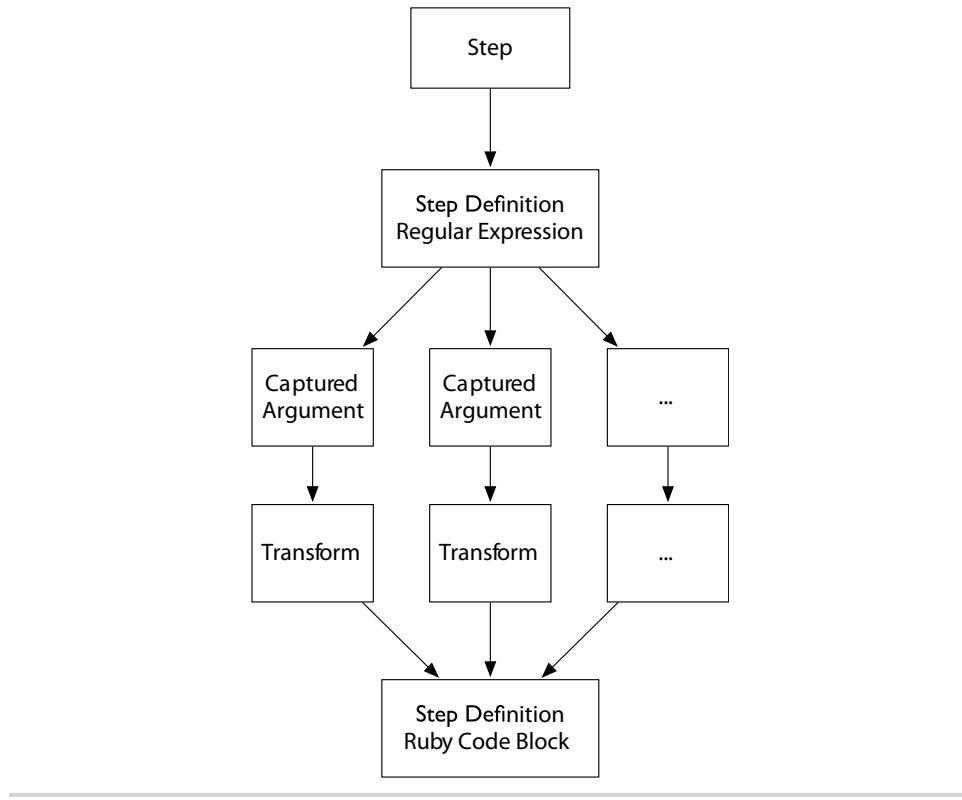


Figure 6—Transforms: how do they work?

[Download step_definitions_inside/08/features/step_definitions/steps.rb](#)

```
CAPTURE_CASH_AMOUNT = Transform /^\$((\d+))/ do |digits|
  digits.to_i
end
```

```
Given /I have deposited (#{$CAPTURE_CASH_AMOUNT}) in my account$/ do |amount|
  my_account = Account.new
  my_account.deposit(amount)
  my_account.balance.should eq(amount),
  "Expected the balance to be #{amount} but it was #{my_account.balance}"
end
```

Notice that we've used a capture group inside the transform to separate the numbers from the currency symbol. This is how we tell Cucumber that we're interested in transforming only that part of the capture we've been passed, so that's all that will be passed into our block. If we wanted to capture the currency symbol as well, we could put another capture group around it, and it would be yielded to the transform's block as another argument:

```
Transform /^(£|$|€)(\d+)/ do | currency_symbol, digits |
  Currency::Money.new(digits, currency_symbol)
end
```

Let's take another look at our to-do list. Using the transform has cleared up the final point from the initial code review. As we went along, we collected a new to-do list item: that we need to implement the Account properly, with unit tests. Let's leave that one on the list for now and move on to the next step of the scenario.

7.3 Adding Custom Helper Methods to the World

We've implemented the first step of our scenario to set up an account with a sufficient balance that the withdrawal should work. After all that talking about transforms, it's hard to remember exactly what we need to do next, but we can always rely on cucumber to remind us where we are:

Feature: Cash Withdrawal

```
Scenario: Successful withdrawal from an account in credit
  Given I have deposited $100 in my account
  When I request $20
    TODO (Cucumber::Pending)
    ./features/step_definitions/steps.rb:28
    features/cash_withdrawal.feature:4
  Then $20 should be dispensed
```

```
1 scenario (1 pending)
3 steps (1 skipped, 1 pending, 1 passed)
0m0.002s
```

The first step is passing as we'd expect, and the second one is failing with a pending message. So, our next task is to implement the step to simulate a customer withdrawing cash from the ATM. Here's the empty step definition:

```
Download step_definitions_inside/09/features/step_definitions/steps.rb
When /^I request \$(\d+)/ do |arg1|
  pending # express the regexp above with the code you wish you had
end
```

First we can reuse the transform we created earlier to translate the cash amount, captured as a string by Cucumber, into a number:

```
When /^I request (\#{CAPTURE_CASH_AMOUNT})$/ do |amount|
  pending # express the regexp above with the code you wish you had
end
```

We need somewhere to withdraw the cash from, which means we need to bring a new actor into this scenario. This new class is going to handle our

request to withdraw cash from our account. If we walked into a bank in real life, that role would be played by a *teller*. Thinking outside-in again, let's sketch out the code we'd like:

```
When /^I request (#{$CAPTURE_CASH_AMOUNT})$/ do |amount|
  teller = Teller.new
  teller.withdraw_from(my_account, amount)
end
```

That looks pretty good. The teller will need to know which account to take the cash from and how much to take. There's a little inconsistency creeping into the language again, though: the step definition talks about *requesting* the cash, but in the code we're *withdrawing* it. *Withdraw* is the term we use most often, so let's change the text in the scenario to match.

[Download step_definitions_inside/10/features/cash_withdrawal.feature](#)

Feature: Cash Withdrawal

Scenario: Successful withdrawal from an account in credit

```
Given I have deposited $100 in my account
When I withdraw $20
Then $20 should be dispensed
```

[Download step_definitions_inside/10/features/step_definitions/steps.rb](#)

```
When /^I withdraw (#{$CAPTURE_CASH_AMOUNT})$/ do |amount|
  teller = Teller.new
  teller.withdraw_from(my_account, amount)
end
```

That's better. Now the language in the scenario more closely reflects what's going on in the code beneath it. Run cucumber again, and you should be prompted to create a Teller class. Let's create one:

[Download step_definitions_inside/10/features/step_definitions/steps.rb](#)

```
class Teller
  def withdraw_from(account, amount)
  end
end
```

Again, we've just sketched out the interface for now, without adding any implementation. With that in place, try running cucumber again:

Feature: Cash Withdrawal

Scenario: Successful withdrawal from an account in credit

```
Given I have deposited $100 in my account
When I withdraw $20
  undefined local variable or method 'my_account' for
  #<Object:0x63756b65> (NameError)
  ./features/step_definitions/steps.rb:32
features/cash_withdrawal.feature:4
```

```
Then $20 should be dispensed
```

Failing Scenarios:

```
cucumber features/cash_withdrawal.feature:2
```

```
1 scenario (1 failed)
3 steps (1 failed, 1 skipped, 1 passed)
0m0.003s
```

A-ha. We defined `my_account` in the first step definition, but when we tried to use it in the second step definition, Ruby can't see it. How can we make `my_account` available to both step definitions? The answer lies in understanding something fundamental about how Cucumber executes step definitions.

Storing State in the World

Just before it executes each scenario, Cucumber creates a new object. We call this object the `World`. The step definitions for the scenario execute in the context of the `World`, effectively as though they were methods of that object. Just like methods on a regular Ruby class, we can use *instance variables* to pass state between step definitions.

Here's how the code looks with `my_account` stored as an instance variable in the step definition:

```
Given /^I have deposited (\#{CAPTURE_CASH_AMOUNT}) in my account$/ do |amount|
  @my_account = Account.new
  @my_account.deposit(amount)
  @my_account.balance.should eq(amount), "Expected the balance to be #{amount}"
end

When /^I withdraw (\#{CAPTURE_CASH_AMOUNT})$/ do |amount|
  teller = Teller.new
  teller.withdraw_from(@my_account, amount)
end
```

Try it. It works!

This solution is OK, but we don't really like leaving instance variables in step definitions like this. The problem with instance variables is that, if you don't set them, they just return `nil`. We hate `nil`s, because they creep around your system, causing weird bugs that are hard to track down. For example, if someone were to later come along and use the second step definition in another scenario where they hadn't already set `@my_account`, they'd end up passing a `nil` into `Teller#withdraw_from`.

We're going to show you a quick refactoring to get rid of the instance variable from the step definition, by pushing the code that creates the Account into a helper method.

Creating Custom Helper Methods

In a regular class we'd avoid nils by putting the instance variables behind an accessor method, like this:

```
def my_account
  @my_account ||= Account.new
end
```

Using `||=` ensures that the new Account will be created only once and then stored in an instance variable. You can do the same in Cucumber. To add custom methods to the World, you define them in a module and then tell Cucumber you want them to be mixed into your World. Here's what we need to do:

```
Download step_definitions_inside/11/features/step_definitions/steps.rb
module KnowsMyAccount
  def my_account
    @my_account ||= Account.new
  end
end

World(KnowsMyAccount)
```

We defined `my_account` on a module called `KnowsMyAccount` and then told Cucumber to mix that module into our World by calling Cucumber's special `World` method. As usual, we've done all this inline in the step definition file for now, but we're going to tidy it away soon.

This means we can get rid of the code that initializes the Account from the first step definition, and we can get rid of the instance variable, so the step definitions go back to using `my_account` again:

```
Download step_definitions_inside/11/features/step_definitions/steps.rb
Given /I have deposited (#{$CAPTURE_CASH_AMOUNT}) in my account$/ do |amount|
  my_account.deposit(amount)
  my_account.balance.should eq(amount),
    "Expected the balance to be #{amount} but it was #{my_account.balance}"
end

When /I withdraw (#{$CAPTURE_CASH_AMOUNT})$/ do |amount|
  teller = Teller.new
  teller.withdraw_from(my_account, amount)
end
```

This time `my_account` won't use a local variable in the step definition, but it will call our helper method instead. With this change in place, run cucumber, and everything should be passing.

Customizing the World

Our second step is passing now, so let's take a break from the example to learn a bit more about the World.

The main way you'll use the World is by extending it with modules of code that support your step definitions by performing common actions against your system:

```
module MyCustomHelperModule
  def my_helper_method
    ...
  end
end

World(MyCustomHelperModule)
```

In this case, you'll be extending a `World` object that Cucumber has created for you. By default, Cucumber creates each `World` by simply calling `Object.new`, but you can override this and use a different class if you need to, by passing a block to the `World` method:

```
class MySpecialWorld
  def some_helper_method
    # ...
  end
end

World { MySpecialWorld.new }
```

Whatever class you use for your worlds, Cucumber always mixes in the module `Cucumber::RbSupport::RbWorld`, which contains various helper methods you can use to talk to Cucumber from your step definitions. The method `pending` that we've used a few times already in the book is an example of this. There are explanations of many of these methods scattered throughout the book, but if you want a complete reference of them, the best thing to do is look up the API documentation¹ for `Cucumber::RbSupport::RbWorld`.

You can discover all the modules that are mixed into the world by calling `puts self` from within a step definition. A list of modules will be printed to the output. And because Cucumber and RSpec are friends, you'll find that Cucumber

1. <http://cukes.info/cucumber/api/ruby/latest/Cucumber/RbSupport/RbWorld.html>

automatically mixes in RSpec's RSpec::Matchers if it can see that you're using the RSpec gem.

Remember that a new world is created for each scenario, so it is destroyed at the end of the scenario. This helps isolate scenarios from one another, because any instance variables set during one scenario will be destroyed along with the world in which they were created when the scenario ends.

Designing Our Way to the Finish Line

Back in the real world, we were trying to get our final step to pass. When we run cucumber, we can see that the first two steps are passing, and the final one is pending. Almost there! Let's take a look at that last step definition:

[Download step_definitions_inside/11/features/step_definitions/steps.rb](#)

```
Then /^$($\d+) should be dispensed$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end
```

The big question here is: where will the cash be dispensed? Which part of the system can we examine for evidence of whether it doled out the readies? It seems as if we're missing a domain concept. In the physical ATM, the cash will end up poking out of a slot on the ATM, something like this:

[Download step_definitions_inside/12/features/step_definitions/steps.rb](#)

```
Then /^(#{$CAPTURE_CASH_AMOUNT}) should be dispensed$/ do |amount|
  cash_slot.contents.should == amount
end
```

That looks good. When we hook our code up to the real hardware, we're going to need some way of talking to it, and this object will work fine as a *test double* in the meantime. Let's start running cucumber to drive this out. First it tells us that we need to define the `cash_slot` method, of course. Let's add that method to our helper module and rename the module to reflect its new role.

[Download step_definitions_inside/12/features/step_definitions/steps.rb](#)

```
module KnowsTheDomain
  def my_account
    @my_account ||= Account.new
  end

  def cash_slot
    @cash_slot ||= CashSlot.new
  end
end
```

`World(KnowsTheDomain)`

Asking the Operator

One great example of the methods built into Cucumber::RbSupport::RbWorld is the ask method. You can call ask from a step definition, and Cucumber will pause the test execution to ask the operator who's running the tests a question. This is useful in a situation where you still want to use Cucumber to document a step that's hard or impossible to automate—just ask a human instead!

We run cucumber again, and this time it wants us to define the CashSlot class, so let's go ahead and do as we're told. Again, we just build a sketch of the interface we want to use, with minimal implementation:

[Download step_definitions_inside/12/features/step_definitions/steps.rb](#)

```
class CashSlot
  def contents
    raise("I'm empty!")
  end
end
```

Now when you run cucumber, you'll see we've moved closer to our goal: all the classes and methods are wired up, and our final step is failing just because there's no cash coming out of the CashSlot:

Feature: Cash Withdrawal

```
Scenario: Successful withdrawal from an account in credit
  Given I have deposited $100 in my account
  When I withdraw $20
  Then $20 should be dispensed
    I'm empty! (RuntimeError)
    ./features/step_definitions/steps.rb:19
    ./features/step_definitions/steps.rb:55
    features/cash_withdrawal.feature:5
```

Failing Scenarios:

cucumber features/cash_withdrawal.feature:2

```
1 scenario (1 failed)
3 steps (1 failed, 2 passed)
0m0.004s
```

To get this last step to pass, someone needs to tell the CashSlot to dispense the cash when the customer makes a withdrawal. It's the Teller who's in charge of the transaction, but at the moment he doesn't know anything about the CashSlot. We'll use *dependency injection* to pass the CashSlot in to Teller's constructor. Now we can imagine a new CashSlot method that the Teller can use to tell it to dispense the cash:

```
Download step_definitions_inside/13/features/step_definitions/steps.rb
class Teller
  def initialize(cash_slot)
    @cash_slot = cash_slot
  end

  def withdraw_from(account, amount)
    @cash_slot.dispense(amount)
  end
end
```

This seems like the simplest implementation of `Teller` that we need to get the scenario to pass. It's odd that when we designed this method from the outside we thought we'd need the `account` parameter, but now we don't seem to need it. Let's stay focused, though: we'll make a note on our to-do list to look into this later and carry on getting this step to pass.

There are two changes we need to make now. We need to add the new `dispense` method to `CashSlot`, and we have to change the second step definition to create the `Teller` correctly:

```
When /^I withdraw (#\{CAPTURE_CASH_AMOUNT}\)$/
  do |amount|
    teller = Teller.new(cash_slot)
    teller.withdraw_from(my_account, amount)
  end
```

This call to `Teller.new` seems out of place in a step definition now. All our other classes are created inside helper methods in the `World`, so let's do the same here:

```
Download step_definitions_inside/13/features/step_definitions/steps.rb
module KnowsTheDomain
  def my_account
    @my_account ||= Account.new
  end

  def cash_slot
    @cash_slot ||= CashSlot.new
  end

  def teller
    @teller ||= Teller.new(cash_slot)
  end
end

World(KnowsTheDomain)
```

This means our step definition becomes a lot less cluttered:

[Download step_definitions_inside/13/features/step_definitions/steps.rb](#)

```
When /^I withdraw (\#{CAPTURE_CASH_AMOUNT})$/ do |amount|
  teller.withdraw_from(my_account, amount)
end
```

The step definition code all reads very nicely now. Pushing some of the details down into our `World` module means the step definition code is at a *higher level of abstraction*. This makes it less of a mental leap when you come into the step definitions from a business-facing scenario, because the code doesn't contain too much detail.

This scenario isn't going to pass until we do some work on our `CashSlot`, though. Run cucumber, and you'll see that it's missing the `dispense` method. A simple implementation of `CashSlot` should get this working:

[Download step_definitions_inside/13/features/step_definitions/steps.rb](#)

```
class CashSlot
  def contents
    @contents or raise("I'm empty!")
  end

  def dispense(amount)
    @contents = amount
  end
end
```

Run cucumber one last time, and you should see the scenario pass:

Feature: Cash Withdrawal

```
Scenario: Successful withdrawal from an account in credit
  Given I have deposited $100 in my account
  When I withdraw $20
  Then $20 should be dispensed
```

```
1 scenario (1 passed)
3 steps (3 passed)
0m0.002s
```

Excellent! Go and grab a drink—then we can sit down, review the code, and do some refactoring.

7.4 Organizing the Code

Before we finish this session, let's be kind to our future selves and tidy up a little. As we worked, we just created everything we needed inline in our `features/step_definitions/step.rb` file. We'll move most of that stuff out of there, and put it into a more conventional place. Here's a list of what we'd like to fix:

- The application's domain model classes should go into a `lib` directory in the root of the project.
- The `KnowsTheDomain` module can move into its own file.
- The transform can also move into its own file.
- The steps file can be split to organize the step definitions better. This is arguably unnecessary for a project with only three step definitions, but we'll do it anyway to illustrate how we'd do this on a bigger project.

Separating the Application Code

It's conventional in Ruby projects to store the system's code in a `lib` directory in the root of the project. We need a name for our application too, since another convention is that the entry point to the `lib` folder is a single file named after your application. Our company is trying to stand out from the crowd with the brand name `NiceBank`, so create a file `lib/nice_bank.rb` and move the three classes, `Account`, `Teller`, and `CashSlot`, into there.

Now replace the code you just took out of `steps.rb` with a line to load the files from the `lib` directory:

```
Download step_definitions_inside/14/features/step_definitions/steps.rb
require File.join(File.dirname(__FILE__), '..', '..', 'lib', 'nice_bank')
```

Save the files and run cucumber.

This works, but it isn't perfect. Loading the application code is something we want to do right at the start of the test run, before Cucumber even starts looking at the step definitions. Fortunately, Cucumber gives us another special folder where we can do just that.

Booting the Cucumber Environment

When Cucumber first starts a test run, before it loads the step definitions, it loads the files in a directory called `features/support`. We haven't needed to use this directory so far in the book, because our examples have been so simple. The support directory is for code that supports the step definitions so that the step definitions themselves stay simple and easy to read.

Just like `features/step_definitions`, Cucumber will load all the Ruby files it finds in `features/support`. This is convenient, because it means you don't need to pepper `require` statements everywhere, but it does mean that the precise order in which files are loaded is hard to control. In practice, this means you can't have dependencies between files in the support directory, because you can't expect

that another file will have already been loaded. There is one exception to that, a special file called `env.rb`

The file `features/support/env.rb` is always the very first file to be loaded when Cucumber starts a test run. You use it to prepare the environment for the rest of your support and step definition code to operate. Loading our application is pretty fundamental to the test run, so move the require statement into your very first `features/support/env.rb` file:

[Download step_definitions_inside/15/features/support/env.rb](#)

```
require File.join(File.dirname(__FILE__), '..', '..', 'lib', 'nice_bank')
```

Run cucumber, and everything should still pass.

Transforms and World Module

We should also tidy away the transform and the world module. Create a file called `features/support/transforms.rb` and move the transform out of `steps.rb` and into there. It's useful to keep the transforms together like this because when we make changes to them, we can read all the other transforms to make sure everything is consistent. Run cucumber, and everything should still pass.

Now let's take the `KnowsTheDomain` module and the call to `World` and move them into a new file, called `features/support/world_extensions.rb`. As we add more methods to our `World`, we would split them into multiple modules, each in their own file; however, this is fine for now.

Organizing Our Step Definitions

We've managed to get this far with a single file of step definitions called `steps.rb`, and for a project of this size, we might well stick with a single file for a while longer. As the number of step definitions starts to grow, we'll want to split up the files so that the code is more cohesive. We've found that our favorite way to organize step definition files is to organize them with one file per domain entity. So, in our example, we'd have three files:

```
features/step_definitions/account_steps.rb
features/step_definitions/teller_steps.rb
features/step_definitions/cash_slot_steps.rb
```

For the time being, each of those files will contain only a single step definition, but now we have room to grow.

Dry Run and `env.rb`

As we start to move files around, it's useful to test that everything still matches using `cucumber --dry-run`. A dry run aims to parse your features and

step definitions but not actually run any of them. This is much faster than a real test run if you just want to print off your features or check for undefined steps with the usage formatter.

The main difference between a normal run and a dry run of cucumber is that a dry run will not boot your environment, so `env.rb` will not be loaded. This means you need to be a little bit careful how you lay out your files:

- Make sure that the other files in support can be loaded with or without `env.rb` having been loaded.
- Put all the slow stuff in `env.rb` so that a dry run can start quickly.

Get used to using dry-run, especially with the usage formatter, to help refactor your scenarios and step definitions with confidence.

7.5 What We Just Learned

It might seem like the system we've been building here is just a toy. There aren't any concrete components that a user could interact with yet: our `CashSlot` is just a plain old Ruby class, and there aren't any buttons for the user to push. What we do have, though, are the beginnings of a domain model and a greater understanding of the problem. Outside-in doesn't always mean starting with the user interface; it means starting with the outside of whatever you want to discover.

We know it isn't always possible to work like this. You'll often be adding tests to a legacy system or working on a project where the user interface is already well-defined by the time you're asked to develop the code. Even in these situations, trying to model your domain in Ruby classes will help your own understanding and also make the test code more maintainable in the long term.

Here are some of the more concrete subjects we've covered in this chapter:

- Transforms help with maintainability by removing annoying duplicate code to process captured arguments from steps. They also give names to the useful parts of your regular expressions.
- Ruby code that supports the step definitions can go into `features/support`, which is loaded before the step definitions.
- The file `features/support/env.rb` is loaded first of all, and that's where you need to boot up your application. This file isn't loaded during a dry run.
- It's good practice to organize step definition files with one file per domain entity.

- Step definitions execute in the context of an object called the World. You can pass state between steps using instance variables and mix in helper methods defined in Ruby modules.

By adding our own World module, we've made the step definition code easier to read, and we've started to decouple the step definitions from the system. The benefit of this decoupling will come later as the system itself evolves. In fact, in the next chapter, we'll show you how we can introduce a web user interface for withdrawing the cash without having to change a line in the step definitions.

Try This

There are lots of places you could take this example now that we have it up and running. Here's one idea for you to try:

The Big Rewrite

Now that we've discovered the domain model together, why not see whether you can do it again, for practice? Delete everything except the feature, the transform, and the step definitions. Then delete the body of each step definition, and change it back to pending. Close the book, run cucumber, and off you go!

Try to forget about what we did, and enjoy the process of discovering a domain model for yourself.

Bug Hunt

There's one remaining item on our to-do list, reminding us that we need to investigate why we originally designed the `Teller#withdraw_from` method to take two parameters, but we're using only one of them.

See whether you can figure out what this inconsistency means, and think about what changes you'd like to make in order to resolve it. Play with the code and try some solutions. We'll work on this issue at the beginning of the next chapter, so you won't have long to wait if you're not sure of the answer.

Edge Cases

We have a single scenario here for the *happy path* through the process of withdrawing cash. Can you think of some simple variations on the scenario that would cause a different outcome? For example, what would happen if your account had a lower balance?

Write your new scenarios out in the same `cash_withdrawal.feature` file, and if you're up for the challenge, have a go at automating them.

Support Code

In the previous chapter, we started working through an example of how to use Cucumber to build a real application, outside-in. The system we're building is an Automated Teller Machine (ATM) for a bank, and we used Cucumber to help us design a simple domain model that satisfied this scenario:

```
Download support_code/01/features/cash_withdrawal.feature
```

Feature: Cash Withdrawal

Scenario: Successful withdrawal from an account in credit

Given I have deposited \$100 in my account

When I withdraw \$20

Then \$20 should be dispensed

The code we've written makes the scenario pass, but the system isn't really of any use yet: there's no external interface for a user to interact with, just a handful of Ruby classes. Now we're going to fix that, by wrapping the domain model with a user interface where the user can request the amount of cash they want to withdraw.

In this chapter, we're going to focus mostly on the code in the features/support folder. This is the lowest level of your test code, where it connects or *couples* to your actual application. If this coupling is well-engineered, your tests will be a pleasure to modify as your project grows. If the coupling is too tight, your tests will be brittle and break any time anything moves. That's why we created a separation layer between the step definitions and the system using a module of helper methods mixed into the World. This separation layer provides just the decoupling we'll need as we start to introduce a user interface.

Before we get started with the user interface, we have one item left on our to-do list we need to check off first.

8.1 Fixing the Bug

Sometimes the signals you'll get from the code that something is wrong are pretty faint. We always try to keep a to-do list handy and write down everything that concerns us as we go. That way, we can stay focused on the task at hand but know that we'll get a chance to check over and clean up any inconsistencies before we move onto the next task.

As we were finishing off the last scenario, we noticed that `Teller#withdraw_from` takes two arguments, but only one of them was used in the method to make the scenario pass. This inconsistency means that something is wrong, but we need to investigate further to know what we need to do about it. Let's take a look at the code that uses the method:

```
Download support_code/01/features/step_definitions/teller_steps.rb
When /^I withdraw (\#{CAPTURE_CASH_AMOUNT})$/ do |amount|
  teller.withdraw_from(my_account, amount)
end
```

It seems pretty obvious what we intended to happen here. The Teller should take the specified amount of money out of the account and hand it to the CashSlot. So, how did we manage to make the scenario pass without having to do anything to the account when we implemented that method? Let's take another look at the scenario:

```
Download support_code/01/features/cash_withdrawal.feature
Feature: Cash Withdrawal
Scenario: Successful withdrawal from an account in credit
  Given I have deposited $100 in my account
  When I withdraw $20
  Then $20 should be dispensed
```

A-ha! There's nothing here that actually checks that the balance of the account has been reduced to \$80. It's a good thing we caught this now—our code would have been handing out cash to customers for free!

Let's tack on this extra outcome as another Then step:

```
Download support_code/02/features/cash_withdrawal.feature
Feature: Cash Withdrawal
Scenario: Successful withdrawal from an account in credit
  Given I have deposited $100 in my account
  When I withdraw $20
  Then $20 should be dispensed
  ➤ And the balance of my account should be $80
```

Run cucumber and paste the snippet for the new step definition into the file `features/step_definitions/account_steps.rb`:



Matt says:

That's Not a Bug; It's Just a Missing Scenario

One of the wonderful things I discovered when I first used Cucumber to build a complete application from the outside-in was when we started manual exploratory testing and discovered some bugs. Almost without exception, every one of those bugs could be traced back to a gap in the Cucumber scenarios we'd written for that part of the system. Because we'd written the scenarios collaboratively, with businesspeople and developers working together to get them right, it wasn't anybody's fault that there was a bug. It was just an edge case we hadn't originally anticipated.

In my experience, bugs are a big source of friction and unhappiness in software teams. Businesspeople blame them on careless developers, and developers blame them on inadequate requirements from the businesspeople. Actually they're just a natural consequence of software development being a really complex endeavor. Using Cucumber really helped the team see that closing these gaps is everyone's job. The blaming just didn't happen, and we were a much happier team as a result.

[Download support_code/02/features/step_definitions/account_steps.rb](#)

```
Given /'I have deposited (\#\{CAPTURE_CASH_AMOUNT\}) in my account$/ do |amount|
  my_account.deposit(amount)
  my_account.balance.should eq(amount),
    "Expected the balance to be #\{amount\} but it was #\{my_account.balance\}"
end
```

```
Then /^the balance of my account should be \$\(\d+\)$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end
```

We can tidy up the regular expression for the step definition using the transform we created in the previous chapter and duplicate the assertion from the previous step definition:

[Download support_code/03/features/step_definitions/account_steps.rb](#)

```
Given /'I have deposited (\#\{CAPTURE_CASH_AMOUNT\}) in my account$/ do |amount|
  my_account.deposit(amount)
  my_account.balance.should eq(amount),
    "Expected the balance to be #\{amount\} but it was #\{my_account.balance\}"
end
```

```
Then /^the balance of my account should be (\#\{CAPTURE_CASH_AMOUNT\})$/ do |amount|
>   my_account.balance.should eq(amount),
>   "Expected the balance to be #\{amount\} but it was #\{my_account.balance\}"
end
```

We'll make a note on our to-do list to tidy up that duplication and carry on trying to fix this bug. Run cucumber now to see whether we have it trapped:

Feature: Cash Withdrawal

```
Scenario: Successful withdrawal from an account in credit
  Given I have deposited $100 in my account
  When I withdraw $20
  Then $20 should be dispensed
  And the balance of my account should be $80
    Expected the balance to be 80 but it was 100
      (RSpec::Expectations::ExpectationNotMetError)
      ./features/step_definitions/account_steps.rb:9
      features/cash_withdrawal.feature:6
```

Failing Scenarios:

```
cucumber features/cash_withdrawal.feature:2
```

```
1 scenario (1 failed)
4 steps (1 failed, 3 passed)
0m0.003s
```

Great—the bug has been caught by our scenario. Just as we suspected, the account's balance remains untouched by the withdrawal. But not for long! Crack open lib/nice_bank.rb, and take a look at our Teller class:

[Download support_code/03/lib/nice_bank.rb](#)

```
class Teller
  def initialize(cash_slot)
    @cash_slot = cash_slot
  end
  def withdraw_from(account, amount)
    @cash_slot.dispense(amount)
  end
end
```

Change the withdraw_from method to tell the account what to do:

[Download support_code/04/lib/nice_bank.rb](#)

```
class Teller
  def initialize(cash_slot)
    @cash_slot = cash_slot
  end
  def withdraw_from(account, amount)
    >   account.debit(amount)
    >   @cash_slot.dispense(amount)
  end
end
```

Now when we run cucumber, we're told we need to create this new method on the Account. On a real project, we'd drop down a gear at this point and start writing unit tests for the Account class to drive out that method. We don't mind sketching out interfaces to classes with a few basic method stubs with only

Cucumber to back us up, but as soon as we start adding interesting behavior to a class, we like to make sure that behavior has been specified in a unit test. *The RSpec Book* [CADH09] and *Growing Object-Oriented Software, Guided by Tests* [FP09] both do a great job of explaining this balance with plenty of examples, but since this book is focused on Cucumber, we're going to gloss over this step and carry on.

Find the Account class in lib/nice_bank.rb and change it to look like this:

```
Download support\_code/05/lib/nice\_bank.rb
class Account
  def deposit(amount)
    @balance = amount
  end
  def balance
    @balance
  end
  def debit(amount)
    @balance -= amount
  end
end
```

We've implemented a new method called debit that decrements the @balance by the given amount. Let's see whether we've managed to fix the bug:

Feature: Cash Withdrawal

```
Scenario: Successful withdrawal from an account in credit
  Given I have deposited $100 in my account
  When I withdraw $20
  Then $20 should be dispensed
  And the balance of my account should be $80
```

```
1 scenario (1 passed)
4 steps (4 passed)
0m0.002s
```

Hooray, we did it! Now it's time for some refactoring before we move on.

Reviewing and Refactoring

In *Extreme Programming Explained* [Bec00], Kent Beck gives four criteria for a simple design. In order, with the most important first, they are as follows:

1. Passes all the tests
2. Reveals all the intention
3. Contains no duplication
4. Uses the fewest number of classes or methods

Let's review our current code against those four criteria. It's been driven from a single test made from a single Cucumber scenario that is passing, so that's rule 1 taken care of. How about rule 2? Revealing intention is essentially about how things are named, which matters to us a great deal. Let's take a look at the methods on our Account class:

[Download support_code/05/lib/nice_bank.rb](#)

```
class Account
  def deposit(amount)
    @balance = amount
  end
  def balance
    @balance
  end
  def debit(amount)
    @balance -= amount
  end
end
```

Reflecting on it, it's confusing that we've used the two method names deposit and debit. Really, we'd either like to see deposit and withdraw or credit and debit, but not a combination of the two. Which pair is more appropriate for our Account class?

We spend a bit of time chatting with one of our domain experts, and it becomes clear that credit and debit are the right names for the methods on Account. In fact, deposit is something you'd more likely ask a Teller to do for you. These conversations happen all the time as you start to establish a *ubiquitous language*, and you'll find they become easier and easier as your knowledge of the domain, and the ubiquitous language you all use to discuss it, grows.

We're going to rename the Account#deposit method to Account#credit. Since our test is nice and fast, we can just rename the method and see what breaks:

Feature: Cash Withdrawal

```
Scenario: Successful withdrawal from an account in credit
  Given I have deposited $100 in my account
    undefined method 'deposit' for #<Account:0x63756b65> (NoMethodError)
    ./features/step_definitions/account_steps.rb:2
    features/cash_withdrawal.feature:3
  When I withdraw $20
  Then $20 should be dispensed
  And the balance of my account should be $80
```

Failing Scenarios:
cucumber features/cash_withdrawal.feature:2

```
1 scenario (1 failed)
4 steps (1 failed, 3 skipped)
0m0.003s
```

In *Refactoring: Improving the Design of Existing Code* [FBBO99], this technique (make the change, see what breaks) is called *leaning on the compiler*. Since we don't have a compiler in Ruby programs, we have to settle for *leaning on the tests*. From the backtrace, it looks like we need to change line 2 of features/step_definitions/account_steps.rb. Go ahead and change that, and run cucumber again.

Our scenario is passing, but we're not quite done yet! The language in the step is now inconsistent with the code inside the step definition we've just changed:

[Download support_code/07/features/step_definitions/account_steps.rb](#)

```
Given /^I have deposited (\#{CAPTURE_CASH_AMOUNT}) in my account$/ do |amount|
  my_account.credit(amount)
  my_account.balance.should eq(amount),
    "Expected the balance to be #{amount} but it was #{my_account.balance}"
end
```

In the step, we're still using the term *deposited*, but now we're *crediting* the account in the Ruby code underneath. How much does this matter? Perhaps not much, but we'd prefer our test code to be as transparent as possible. After another discussion with our domain expert, we decide to reword the step to read Given my account has been credited with \$100. We'll change the feature and the underlying step definition:

[Download support_code/08/features/cash_withdrawal.feature](#)

Feature: Cash Withdrawal

Scenario: Successful withdrawal from an account in credit

```
Given my account has been credited with $100
When I withdraw $20
Then $20 should be dispensed
And the balance of my account should be $80
```

[Download support_code/08/features/step_definitions/account_steps.rb](#)

```
Given /my account has been credited with (\#{CAPTURE_CASH_AMOUNT})$/ do |amount|
  my_account.credit(amount)
  my_account.balance.should eq(amount),
    "Expected the balance to be #{amount} but it was #{my_account.balance}"
end
```

Looking at this step definition now, it doesn't make sense anymore to check that the balance is the same as the amount credited. If this step definition were used in another scenario that involved other credit or debit transactions before this one, the balance might not be the same as the amount credited,

Test Automation Is Software Development

You'll have noticed the painstaking care with which we've moved back and forth between the features, the step definitions, and the system itself, keeping everything as clean and consistent as we can. You might be wondering whether we honestly work this way in our everyday practice. We certainly try to do so. We've alluded to this earlier in the book, but we feel like it's time to make the point clearly: if you're writing automated tests, you're developing software. If you value those tests enough to have written them in the first place, you'll want to be able to come back and change them in the future. That means that all the same good habits we normally use to write maintainable software apply to the test code we write too.

This important point is often missed, especially in a company where testing has traditionally been done manually. People who specialize in testing sometimes aren't the best people to be automating tests, if they lack the necessary experience in software design. Without support from people who do understand how to write maintainable code, the team can end up with messy test code that's hard to change. It's around this point that people start to realize their automated tests are actually making it harder to change the software, not easier, and consider giving up on Cucumber altogether. [Figure 7. What skills do you need to write Cucumber tests?, on page 142](#) shows how software development skill becomes increasingly important as you move down the stack from Gherkin features to support code.

Testing and software design are complementary skills, and a strong team needs a mix of both specialties. Different people will sit at different points on this spectrum: some people will be great testers but not at all interested in automation or programming, and some will just want to write code without thinking about how to break it. That's fine, but the team needs to recognize that there isn't a clear division of responsibility when you're working with automated tests. Everyone needs to work together to create and maintain high-quality tests.

and that would be quite all right. So, we don't want this assertion here any more.

That gives us an easy way to cross off the last item on our to-do list: removing the duplication of the balance assertion in the two-step definitions for the account. We can simply delete the first one, because it's no longer relevant:

```
Download support_code/09/features/step_definitions/account_steps.rb
Given /^my account has been credited with (#{$CAPTURE_CASH_AMOUNT})$/ do |amount|
  my_account.credit(amount)
end
Then /^the balance of my account should be (#{$CAPTURE_CASH_AMOUNT})$/ do |amount|
  my_account.balance.should eq(amount),
    "Expected the balance to be #{$amount} but it was #{$my_account.balance}"
end
```

That completes our refactoring. The code is now as clear and communicative and free of duplication as we can imagine at this stage. We're ready to add some new functionality.

8.2 Bootstrapping the User Interface

Now that we have a model we're happy with, we're going to wrap it in a simple user interface that allows the user to specify how much cash they want to withdraw. Even though it might not be a typical choice for a real ATM, we're going to use a web form. We'll use the Sinatra¹ web framework to serve our user interface, and we'll use the Capybara² gem to automate it. We won't go into much detail about how Capybara works here, but you'll learn much more in [Chapter 15, Using Capybara to Test Ajax Web Applications, on page 245](#).

We'll start by adding the necessary gems to our project and getting the web server up and running.

Installing the Gems

Our app is growing up. We need to install another couple of Ruby gems, so let's create a Gemfile to manage the project's dependencies using Bundler.³

```
Download support_code/11/Gemfile
source :rubygems

gem 'sinatra', '1.3.1'

group :development do
  gem 'rspec', '2.7.0'
  gem 'cucumber', '1.1.3'
  gem 'capybara', '1.1.2'
end
```

Run the command `bundle` in the root of the project to install these gems.

Once the gems are installed, we can kick the tires on Sinatra by adding a few lines to the end of `lib/nice_bank.rb`:

```
Download support_code/11/lib/nice_bank.rb
require 'sinatra'
get '/' do
  'Welcome to our nice bank.'
end
```

-
1. <http://sinatrarb.com>
 2. <http://rubygems.org/gems/capybara>
 3. For instructions on installing Bundler, see [Appendix 2, Installing Cucumber, on page 299](#).

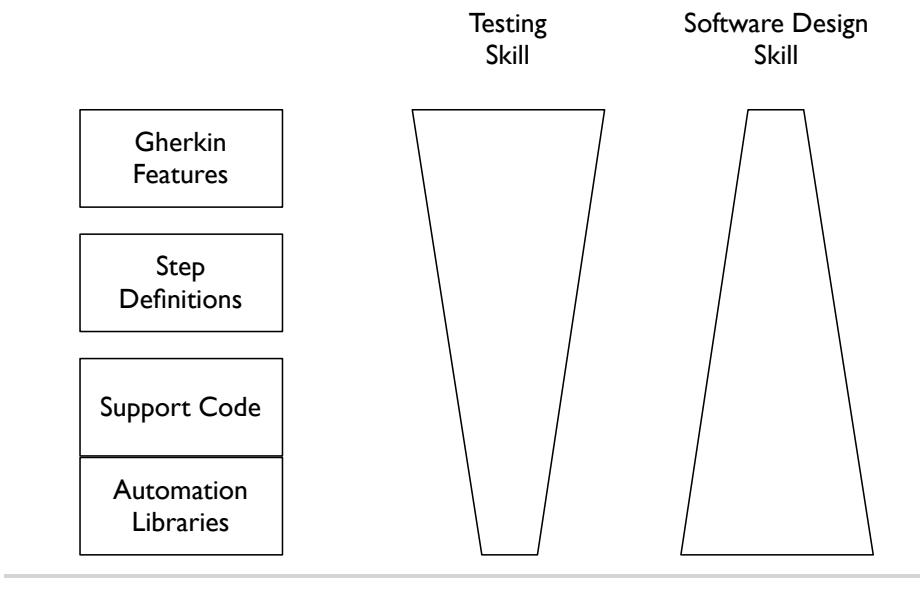


Figure 7—What skills do you need to write Cucumber tests?

If you run this file with the command `bundle exec ruby lib/nice_bank.rb`, you should see Sinatra start up a web server. You can now try hitting it from a browser at <http://localhost:4567/>. That's the beginning of our user interface.

In our tests, we need to load the Capybara library and tell it where to find our web application. Change `features/support/env.rb` to look like this:

```
Download support_code/11/features/support/env.rb
require File.join(File.dirname(__FILE__), '..', '..', 'lib', 'nice_bank')

require 'capybara/cucumber'
Capybara.app = Sinatra::Application
Sinatra::Application.set :environment, :test
```

We haven't actually changed the way our tests communicate with the application, so if you run cucumber again, the scenario should still pass. Good? Time to start building our web application.

8.3 Making the Switch

Our goal is to introduce a user interface for requesting the cash withdrawal. We want Cucumber to cover us as we make these changes, so we need to change how our test code interacts with the application. Up until now, all our step definitions were talking directly to the domain model. We're going to change that so that some of them hit the new user interface instead (see

[Figure 8, Introducing the user interface, on page 144](#)). Which steps need to change?

Let's take a look at our scenario again:

[Download support_code/11/features/cash_withdrawal.feature](#)

Feature: Cash Withdrawal

Scenario: Successful withdrawal from an account in credit

- Given** my account has been credited with \$100
- When** I withdraw \$20
- Then** \$20 should be dispensed
- And** the balance of my account should be \$80

We haven't specified anything about *how* we do the cash withdrawal, so there's nothing we need to change about the scenario at all. Great! Let's jump down into the step definition for withdrawing cash and see what needs to change there:

[Download support_code/11/features/step_definitions/teller_steps.rb](#)

```
When /^I withdraw (\#{CAPTURE_CASH_AMOUNT})$/ do |amount|
  teller.withdraw_from(my_account, amount)
end
```

All we're doing here is calling `withdraw_from` on something. Right now that something is an instance of the `Teller` class in our domain model. But what if we made `teller` return something else, like an object that represents the user interface? That's the beauty of object-oriented programming—as long as the object understands the `withdraw_from` method, this step definition is going to be happy. Let's leave this step definition alone and drop down into our `World` module:

[Download support_code/11/features/support/world_extensions.rb](#)

```
module KnowsTheDomain
  def my_account
    @my_account ||= Account.new
  end

  def cash_slot
    @cash_slot ||= CashSlot.new
  end

  def teller
    @teller ||= Teller.new(cash_slot)
  end
end
```

`World(KnowsTheDomain)`

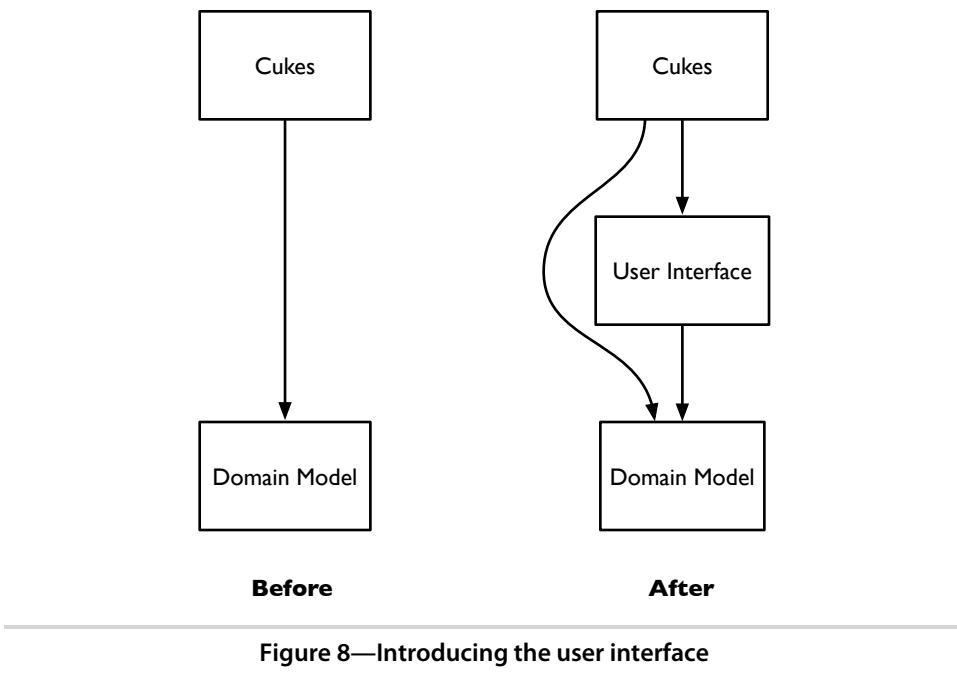


Figure 8—Introducing the user interface

What we need to do is reimplement the teller method to return a class that knows how to automate the user interface. For now, just build an empty class with the right method on it, and change teller to return an instance of the new class:

```
Download support_code/12/features/support/world_extensions.rb
module KnowsTheUserInterface
  class UserInterface
    def withdraw_from(account, amount)
    end
  end

  def my_account
    @my_account ||= Account.new
  end

  def cash_slot
    @cash_slot ||= CashSlot.new
  end

  def teller
    @teller ||= UserInterface.new
  end
end
World(KnowsTheUserInterface)
```

Now we've disconnected the action-invoking `When` step in our scenario from the domain model and connected it to this new `UserInterface` class. We've also renamed the module to reflect its new responsibility. If you run cucumber now, you should see the scenario fail:

```
Feature: Cash Withdrawal
```

```
Scenario: Successful withdrawal from an account in credit
  Given my account has been credited with $100
  When I withdraw $20
  Then $20 should be dispensed
    I'm empty! (RuntimeError)
    ./lib/nice_bank.rb:30
    ./features/step_definitions/cash_slot_steps.rb:2
    features/cash_withdrawal.feature:5
  And the balance of my account should be $80
```

Failing Scenarios:

```
cucumber features/cash_withdrawal.feature:2
```

```
1 scenario (1 failed)
4 steps (1 failed, 1 skipped, 2 passed)
0m0.003s
```

The scenario has failed because our new support module hasn't been wired up to the system yet, so nothing has been found in the cash slot. Now we have a goal: get that scenario to pass again, but this time, through the user interface.

Designing the User Interface

So, what should this shiny new user interface look like? We gathered around the whiteboard with our user experience team and sketched out a wireframe for the first iteration of the cash withdrawal form. The plan is for it to look roughly like [Figure 9, Wireframe for cash withdrawal user interface, on page 146.](#)

Let's flesh out our `UserInterface` class to talk to that form:

[Download support_code/13/features/support/world_extensions.rb](#)

```
class UserInterface
  include Capybara::DSL

  def withdraw_from(account, amount)
    visit '/'
    fill_in 'Amount', :with => amount
    click_button 'Withdraw'
  end
end
```



Figure 9—Wireframe for cash withdrawal user interface

We've included Capybara's DSL module in the `UserInterface` class to give us these new methods that let us carry out actions on the web user interface. First we visit the home page, then we fill_in the field labeled `Amount`, and finally we click the `Withdraw` button. This is the design we just looked at on the wireframe, formalized in code.

When we run cucumber against this, it's going to fail of course, because we haven't built the form yet. Let's run it anyway just to check we're on track:

`Feature: Cash Withdrawal`

```
Scenario: Successful withdrawal from an account in credit
  Given my account has been credited with $100
  When I withdraw $20
    cannot fill in, no text field, text area or password field with id, name,
    or label 'Amount' found (Capybara::ElementNotFound)
    (eval):2
    ./features/support/world_extensions.rb:9
    ./features/step_definitions/teller_steps.rb:2
    features/cash_withdrawal.feature:4
  Then $20 should be dispensed
  And the balance of my account should be $80
```

`Failing Scenarios:`
`cucumber features/cash_withdrawal.feature:2`

```
1 scenario (1 failed)
4 steps (1 failed, 2 skipped, 1 passed)
0m0.207s
```

OK, so Capybara is telling us that there isn't an `Amount` field to fill in. Wouldn't it be nice to be able to see what Cucumber is seeing? We could start Sinatra and run a manual test, but it would be more useful to see exactly what's in the browser when the test fails. To do this, we need to take some time out to learn about a new feature of Cucumber.

8.4 Using Hooks

Cucumber supports hooks, which are blocks of code that run before or after each scenario. You can define them anywhere in your support or step definition layers, using the methods `Before` and `After`.

To test them, take any Cucumber project and try adding a file `features/support/hooks.rb` that looks like this:

```
Before do
  puts "Go!"
end

After do
  puts "Stop!"
end
```

If you run `cucumber`, you'll see the two messages printed in the output at the beginning and end of the scenario. Cucumber's `Before` and `After` hooks are a lot like the setup and teardown methods in the xUnit family of testing tools.

The most common use of `Before` and `After` hooks is to clear up any residual state left in external systems like databases so that each scenario starts with the system in a known state. We'll explain this in detail in [Chapter 10, Databases, on page 173](#).

Hooks are global by default, meaning they run for every scenario in your features. If you want them to run for just certain scenarios, you need to tag those scenarios and then use a tagged hook.

Tagged Hooks

Both `Before` and `After` accept a tag expression, which you can use to selectively add the hook to only certain scenarios. For example, suppose you're writing features for the administrative area of a website. Each of the administrator features starts with the following background:

Feature: Delete Widgets

Background:

`Given I am logged in as an administrator`

...

An alternative is to tag the feature and then use a `Before` hook to run this same code to log in as an administrator.

```
@admin
Feature: Delete Widgets

...
Before('@admin') do
  user = create_user(:admin => true)
  login_as user
end
```

Now, to run a scenario logged in as an administrator, you just have to tag the scenario with @admin, and this code will automatically run before the steps of the scenario. In [Filtering with Tag Expressions, on page 193](#), we will explain more complex tag expressions using logical operations, but a simple tag will do for now.

Tagged hooks can be useful for ensuring technical things like external services are started, without making too much fuss about them in the text of the scenario itself.

Examining the Scenario

If we want it to, our hook can accept a single argument that represents the scenario. For example, we can ask a scenario for its name:

```
Before do |scenario|
  puts "Starting scenario #{scenario.name}"
end
```

We can also ask the scenario whether it failed:

```
After do |scenario|
  puts "Oh dear" if scenario.failed?
end
```

For more details on the Scenario object, look at the documentation for Cucumber::Ast::Scenario.⁴

Around Hooks

Around hooks give you more power than Before and After hooks, allowing you to actually control how many times the scenario is run. An Around hook is passed two parameters: an object that represents the scenario and a block of code that, when called, will run the scenario:

4. <http://cukes.info/cucumber/api/ruby/latest/Cucumber/Ast/Scenario.html>



Aslak says:

Step Definitions Are Global

When you define a step definition, it is defined globally. There is no way to reduce the scope of a step definition to certain scenarios like you can do with tagged hooks.

People occasionally ask for a way to scope step definitions in a similar way to tagged hooks, such as making *When I turn it off* invoke one step definition for some scenarios and another one for others.

This is a feature that would be easy to add to Cucumber, and one day I actually implemented it to get some feedback from the nice people on the Cucumber mailing list. My question was, *Can anyone think of how people might misuse this?* Richard Lawrence, an old-timer on the list, answered:

Feature-coupled steps is the extreme. The more subtle issue is that the beneficial pressure to grow a ubiquitous language goes away when it becomes too easy to say, "Oh, that's just another context, I'll use the same words to mean something different here." Thinking about some of the conversations I've had coaching teams to learn ubiquitous language, I would expect this to happen a lot.

The term “feature-coupled steps” is a term I came up with in the early days of Cucumber when I was documenting good and bad Cucumber practices in the wiki. I consider feature-coupled steps to be a code smell since they quickly cause a lot of duplication and do nothing to promote a ubiquitous language.

When Dan North—the originator of BDD—wrote his first BDD framework, step definitions were coupled to features. He told me the ability to have global step definitions shared across features was one of the *improvements* Cucumber brought on.

In retrospect, it’s rather amusing to observe the clones and spin-offs of Cucumber re-introducing mechanisms that we deliberately got rid of.

In Cucumber, the same sentence can mean only one thing.

```
Around do |scenario, block|
  puts "About to run #{scenario.name}"
  block.call
  puts "Finished running #{scenario.name}"
end
```

With the extra power of Around hooks comes increased responsibility: if you forget to call the block, your scenario won’t run at all.

Around hooks also accept tags, so you can restrict them to run for only some scenarios, just like Before and After hooks. One great trick we’ve seen them used for is to run the same scenario twice under different conditions, like this:

```
Around('@run_with_and_without_javascript') do |scenario, block|
  Capybara.current_driver = Capybara.javascript_driver
  block.call
  Capybara.use_default_driver
  block.call
end
```

Here we've used an Around hook that runs only on scenarios that are tagged with @run_with_and_without_javascript. The hook runs the scenario twice, first with Capybara set to use a driver that supports JavaScript and then again with the default driver. You'll learn more about Capybara itself in [Chapter 15, Using Capybara to Test Ajax Web Applications, on page 245](#), but this should give you an idea of what's possible using Around hooks.

Hooks That Run at Other Times

If you want to run code before all of your features start, the way to do that is simply to put it into a Ruby file inside features/support. Any file in there will be run by Cucumber before your first scenario runs.

If you want to run code after all of your features have finished, you can use Ruby's built-in `at_exit` hook, which will be run just before the Cucumber process exits. You can write one of these hooks from anywhere, including within a step definition, but you'll typically use them to tear down some external system that you've started from your support code.

There are two other built-in hooks in Cucumber, `AfterStep` and `AfterConfiguration`, which we won't go into here. For more information, refer to the documentation for `Cucumber::RbSupport::RbDsl`.⁵

Armed with this new knowledge about hooks, let's get back to work on our ATM.

8.5 Building the User Interface

We can add a debugging hook to show us what's going on in our failing scenario. Create a file `features/support/debugging.rb`:

```
Download support_code/14/features/support/debugging.rb
After do |scenario|
  save_and_open_page if scenario.failed?
end
```

The method `save_and_open_page` is provided by Capybara, but it relies on the `launchy` gem that isn't installed automatically. We'll add it to our Gemfile:

5. <http://cukes.info/cucumber/api/ruby/latest/Cucumber/RbSupport/RbDsl.html>

[Download support_code/14/Gemfile](#)
 source :rubygems

```
gem 'sinatra', '1.3.1'

group :development do
  gem 'rspec', '2.7.0'
  gem 'cucumber', '1.1.3'
  gem 'capybara', '1.1.2'
  gem 'launchy', '2.0.5'
end
```

Now run bundle to install the new gem.

What you should see if you run cucumber now is the web page that Cucumber can see has been saved to disk and opened up in your browser. This can be a very helpful debugging tool. What it's shown us is the simple greeting message we added earlier when we first added Sinatra. We need to change the Sinatra app to return a simple HTML form.

[Download support_code/15/lib/nice_bank.rb](#)

```
require 'sinatra'

get '/' do
  %{
<html>
  <body>
    <form action="/withdraw" method="post">
      <label for="amount">Amount</label>
      <input type="text" id="amount" name="amount">
      <button type="submit">Withdraw</button>
    </form>
  </body>
</html>
}
end

post '/withdraw' do
  fail "I don't know how to withdraw yet, sorry"
end
```

We've hard-coded the HTML for the form right in the Sinatra web request handler so that you can easily see what's going on. The form contains a single field with a label and a submit button that posts back to the server. We've also added a new post request handler so that the form has somewhere to post back to, although it's just a stub that will raise an error if it's called.

Try running cucumber now. If everything has gone according to plan, you'll see the error I don't know how to withdraw yet, sorry displayed in Cucumber's output, and

`save_and_open_page` should be showing the form, since that's the last thing the browser was looking at. Our next step is to implement the right code in the post request handler to actually withdraw the cash.

Dispensing the Cash

We've already built a domain model, so we have a significant head start. The class that knows how to carry out the withdrawal is `Teller`, so we'll need to create an instance of that class when we receive the posted form data on the web server. To create an instance of the `Teller`, we need a `CashSlot` to pass to the constructor. The slightly tricky thing about this is that we need to make sure that the step definitions are also looking at the same instance of `CashSlot`; otherwise, they won't be able to see the cash we've dispensed.

We can use Sinatra's settings to store a single instance of `CashSlot` where it can be accessed by both the web application and the tests. First we'll create the setting and use it in the web application:

```
Download support_code/16/lib/nice_bank.rb
set :cash_slot, CashSlot.new
post '/withdraw' do
  teller = Teller.new(settings.cash_slot)
  fail "I don't know how to withdraw yet, sorry"
end
```

Ah, now if we want to call `withdraw_from` on the teller, we'll also need an `Account`. As our project progresses, we'll probably end up determining the `Account` by reading the user's card, but at the moment our domain model doesn't go that far. For the time being, what we need is a way for the step definitions to be able to say to the web application *look, just trust me and use this account*.

Let's create another setting on our web application so it can share the `account` with the tests. This time we won't give it a default value, however, because the ATM shouldn't have a default account—it should be determined from the user's interactions with the ATM when they push their card into the slot and enter their PIN. In case we forget to do this in future scenarios, we'll cause it to throw an error if it hasn't been set:

```
Download support_code/17/lib/nice_bank.rb
set :cash_slot, CashSlot.new
set :account do
  fail 'account has not been set'
end
post '/withdraw' do
  teller = Teller.new(settings.cash_slot)
  teller.withdraw_from(settings.account, params[:amount].to_i)
end
```

Now we can try to call `Teller#withdraw_from`. Update your code to look like the previous code, and run cucumber. You should see the error message from the account setting:

Feature: Cash Withdrawal

```
Scenario: Successful withdrawal from an account in credit
  Given my account has been credited with $100
  When I withdraw $20
    account has not been set (RuntimeError)
    ./lib/nice_bank.rb:55
    ./lib/nice_bank.rb:60
    (eval):2
    ./features/support/world_extensions.rb:10
    ./features/step_definitions/teller_steps.rb:2
    features/cash_withdrawal.feature:4
  Then $20 should be dispensed
  And the balance of my account should be $80
```

Failing Scenarios:

cucumber features/cash_withdrawal.feature:2

```
1 scenario (1 failed)
4 steps (1 failed, 2 skipped, 1 passed)
0m0.269s
```

We need to update our support code so that it makes use of these two touchpoints for sharing the CashSlot and Account. For the CashSlot, we'll change our support module to read the web application's setting instead of creating its own instance:

```
Download support_code/18/features/support/world_extensions.rb
def cash_slot
  Sinatra::Application.cash_slot
end

def teller
  @teller ||= UserInterface.new
end

World(KnowsTheUserInterface)
```

Then the last thing to do is set the Account on the web application as we fill out the withdrawal form:

```
Download support_code/18/features/support/world_extensions.rb
class UserInterface
  include Capybara::DSL
```

```

def withdraw_from(account, amount)
>   Sinatra::Application.account = account
  visit '/'
  fill_in 'Amount', :with => amount
  click_button 'Withdraw'
end
end

```

This is a temporary shortcut around the process of actually authenticating the user with their physical card and PIN, because we don't want to worry about that part of the process yet.

Run cucumber now, and the scenario should be passing:

Feature: Cash Withdrawal

```

Scenario: Successful withdrawal from an account in credit
  Given my account has been credited with $100
  When I withdraw $20
  Then $20 should be dispensed
  And the balance of my account should be $80

```

```

1 scenario (1 passed)
4 steps (4 passed)
0m0.176s

```

Phew! We've managed to convert our tests to run against a web interface, and all we needed to do was change the way our `World` module's methods were implemented. If we wanted, we could set up our tests to switch between the two modules, giving us fast tests that go directly to the domain model and slower, more thorough tests that go right through the user interface.

8.6 What We Just Learned

Working outside-in with Cucumber blurs the lines between testing and development. Always be ready to learn something new about the problem domain, whether you're deciding on the wording in a Cucumber scenario or choosing the parameters for a method. By taking care to craft a clean interface between your tests and the system underneath, you'll end up with tests that can easily evolve with the system's changing requirements.

In this chapter, we showed you how to use Cucumber to trap a bug. In [Chapter 13, Adding Tests to a Legacy Application, on page 221](#), we'll talk much more about this.

We covered the basics of creating Sinatra applications and testing them using Capybara. There's a whole chapter specifically on using Capybara, [Chapter](#)

[15. Using Capybara to Test Ajax Web Applications, on page 245.](#) Sinatra is a useful tool for testers to know about. Sometimes, when the system you're testing depends on an external web service that you can't control, you'll want to create a *test double* of that external service. Sinatra is a quick and easy way of creating such a test double.

We also showed you how to use Cucumber's hooks to invoke Ruby code before and after each scenario or to run them before specific scenarios using tags.

Next we'll introduce a new problem by changing the architecture to make the system asynchronous.

Try This

Here are some exercises for you to try for yourself.

Redesign

Our user experience aficionados have had a rethink. Early reports from user testing are that most users don't want to have to type in a precise amount of money; they want to click a button with a fixed amount. Without changing the Cucumber scenario, can you change the support code and then the user interface itself so that the user has to click only a single button to get their \$20?

Can you add some more scenarios (and some more buttons) for other fixed amounts?

Preventing Overdraws

We need to help our customers not to overdraw. Add a new scenario that looks like this:

Feature: Prevent users from going overdrawn

```
Scenario: User tries to withdraw more than their balance
  Given my account has been credited with $100
  When I withdraw $200
  Then nothing should be dispensed
  And I should be told that I have insufficient funds in my account
```

Can you implement it?

Balancing Act

Our users would like to be able to check their balance from the ATM. Decide how you'd like the user interface to be—perhaps you'll show a menu from which they can choose to make a withdrawal or check their balance—and then implement this scenario:

Feature: Display balance

Scenario: User checks the balance of an account in credit

Given my account has been credited with \$100

When I check my balance

Then I should see that my balance is \$100

Dealing with Message Queues and Asynchronous Components

Our worked example has been pretty simplistic so far. We're testing a single web server process that doesn't even have a database, and we've used only a single scenario to test it. The real systems you have to work on when you put this book down are probably much more complicated than that, with several services interacting to deliver the behavior you've described in your Cucumber features.

In this chapter, we're going to make our system more enterprisey, splitting the architecture into a frontend and backend so that we can show you how to test these kinds of systems. On the way, we'll get the chance to explain some fundamental concepts about testing asynchronous systems.

9.1 Our New Asynchronous Architecture

In a real banking system, the ATM isn't the only thing making debits on your account. You might use your debit card in a restaurant or supermarket, or you might walk into a bank and withdraw cash over the counter. You might have written a check that gets cashed a few days later. You'll also get credits into your account, like when you deposit checks at the bank. The bank treats each of these events as a *transaction*, which it processes some time after the actual event has happened. Note that this kind of transaction is completely different from a *database transaction*, which we'll talk about in the next chapter.

We're going to change our architecture so that when the customer makes a withdrawal from the ATM, it posts a message about the debit transaction into a message queue. We'll move the responsibility for processing this queue into

a separate backend service. As the backend service works through the queue of transactions, it will store the updated account balance in a database so that the ATM (and our tests) can access it.

[Figure 10, Enterprise messaging architecture, on page 159](#) shows how the new architecture looks. The ATM posts messages about transactions into the Transaction Queue. The Transaction Processor reads messages off that queue, reads the existing balance from the Balance Store, and then stores the updated balance back in the Balance Store. The ATM reads the account balance from the Balance Store.

Let's look at the implications this new architecture will have on how we test the system.

9.2 How to Synchronize

In our current implementation, the debit and credit transactions are processed synchronously by our simplistic Account class:

[Download support_code/18/lib/nice_bank.rb](#)

```
class Account
  def credit(amount)
    @balance = amount
  end

  def balance
    @balance
  end

  def debit(amount)
    @balance -= amount
  end
end
```

This implementation, where the balance is updated *during* the method call to credit or debit, means that we can be certain the balance will have been updated by the time Cucumber checks the account balance in the final Then step of our scenario.

[Download support_code/18/features/cash_withdrawal.feature](#)

Feature: Cash Withdrawal

Scenario: Successful withdrawal from an account in credit
Given my account has been credited with \$100
When I withdraw \$20
Then \$20 should be dispensed
And the balance of my account should be \$80

When we change to our new architecture; however, the transactions will be processed by a separate backend service. Because the tests and backend

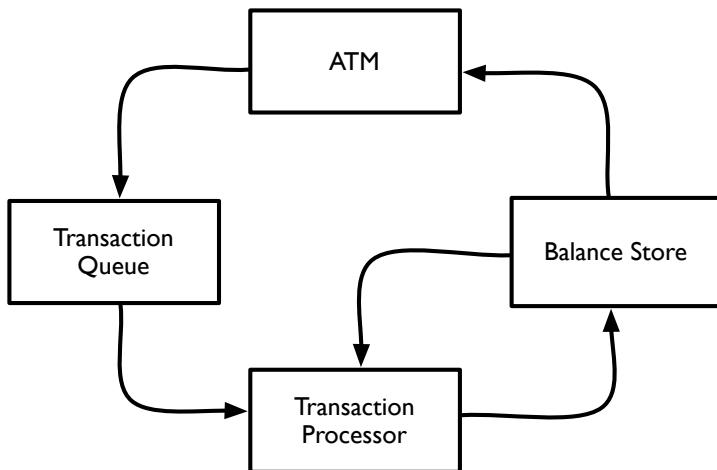


Figure 10—Enterprise messaging architecture

service are running in separate processes, it's quite possible that Cucumber will run the `Then` step before the transaction processor has finished its work. If that happened, the test would fail even though the system is actually working as expected: if only Cucumber had waited a few more moments, it would have seen the right balance. This is what we call a flickering scenario, as described in [Chapter 6, When Cucumbers Go Bad, on page 85](#). How can we tell when it's safe to check the account balance?

Adding asynchronous components into a system introduces a degree of randomness, but for our tests to be reliable, we need to ensure that the behavior is completely deterministic. To do that, we need to understand how we can *synchronize* our tests with the system, so that we make our checks only when the system is ready. In *Growing Object-Oriented Software, Guided by Tests* [FP09], Steve Freeman and Nat Pryce identify two ways to synchronize your tests with an asynchronous system: sampling and listening.

Synchronizing by Listening

Listening for events is the fastest and most reliable way to synchronize your tests with an asynchronous system. For this technique to work, the system under test has to be designed to fire events when certain things happen. The tests subscribe to these events and can use them to create *synchronization points* in the scenario.

For example, if the Transaction Processor were emitting `BalanceUpdated` events into a publish-subscribe message channel, we could wait at the top of our `Then`

Synchronizing with Delayed Job

In Ruby applications, Delayed Job^a is a popular gem for running background tasks asynchronously. We recommend working with Delayed Job by having a method in your support layer like this:

```
module DelayedJobSupport
  def process_all_jobs
    Delayed::Worker.new.work_off(Delayed::Job.count)
    if ENV['FAIL_FAST']
      raise Delayed::Job.first.last_error if Delayed::Job.count > 0
    end
  end
end

World(DelayedJobSupport)
```

The first line of the method tells Delayed Job to process all of the jobs in its queue. This is normally done by a background task in production, so by calling this code from your step definitions, you'll make sure they will wait here until the jobs are all processed. We're not actually using listening or sampling here; we're bringing what is normally an asynchronous component into our process, making it synchronous.

You'll have to call this method yourself from your step definitions at the points when you want to synchronize.

The last part of the method raises an error if any of the jobs have failed. We don't always want to do this—we might be running a scenario where the expected behavior is for the job to fail—but it can be useful to have this available for debugging. Using the environment variable like this allows us to switch to this debug mode easily:

```
$ cucumber features/background_jobs.feature FAIL_FAST=1
```

If you wanted to stay closer to how the code behaves in production and keep Delayed Job running asynchronously, you'll still need to synchronize. In this case, you could use sampling to poll Delayed Job's queue until it was empty.

a. http://rubygems.org/gems/delayed_job

step until we heard that event. Once we'd received that event, we'd know it was safe to proceed and check the balance. We would use a timeout to ensure the tests didn't wait forever if something was wrong with the system.

Using events like this involves a sophisticated coordination of your testing and development efforts, but it results in fast tests because they don't waste any time waiting for the system: as soon as they're notified of the right event, they spring back into action and carry on.

Synchronization by Sampling

When it isn't possible to listen to events from the system, the next best option is to repeatedly poll the system for the state change you're expecting. If it doesn't appear within a certain timeout, you give up and fail the test.

Sampling can result in tests that are a little bit slower than listening, because of the polling interval. The more often you poll the system for changes, the quicker the tests can react and carry on when the system is working as expected. If you poll too frequently, however, you could put excessive load on the system.

Sampling is usually the pragmatic choice when you don't have the option to receive events from the system under test. In the rest of this chapter, we'll show you how to use sampling to make our tests work reliably with the new architecture.

9.3 Implementing the New Architecture

So that we can show you what it's like to have a flickering scenario, we'll start by changing the architecture underneath the same test code. We'll demonstrate the flickering scenario and, in a final flourish, fix it by changing the tests to use sampling to synchronize with the system.

Driving Out the Interfaces

The Account class is where the ATM is going to interface with our new backend services. As shown in [Figure 10, Enterprise messaging architecture, on page 159](#), the two touch points are the TransactionQueue and the BalanceStore. Let's design the interfaces to those objects by changing our Account class to use two imaginary objects that can talk to these services:

```
Download message_queues/01/lib/nice_bank.rb
require_relative 'transaction_queue'
require_relative 'balance_store'
class Account
  def initialize
    @queue = TransactionQueue.new
    @balance_store = BalanceStore.new
  end

  def balance
    @balance_store.balance
  end

  def credit(amount)
    @queue.write("+#{amount}")
  end
end
```

```

def debit(amount)
  @queue.write("-#{amount}")
end
end

```

We've used Ruby 1.9's `require_relative` at the top of the file, so we're assuming that the two new classes will be defined in files that sit in the same directory as this one. Getting the balance is a simple matter of delegating to the `BalanceStore`. In a more realistic system, we'd need to tell the `BalanceStore` which account we wanted the balance for, but in our simple example we're dealing only with a single account, so we don't need to worry about that.

For debits and credits, we're serializing the transaction as a string, using a `+` or `-` to indicate whether the amount is a credit or a debit, and then writing it to the queue.

Building the TransactionQueue

Let's build our transaction queue. We want to keep the technology really simple for this example, so we're going to use the file system as our message store, with each message stored as a file in a `messages` directory. As a message is read from the queue, we'll delete the file. Here's the code:

[Download message_queues/01/lib/transaction_queue.rb](#)
`require 'fileutils'`

```

class TransactionQueue
  def self.clear
    FileUtils.rm_rf('messages')
    FileUtils.mkdir_p('messages')
  end

  def initialize
    @next_id = 1
  end

  def write(transaction)
    File.open("messages/#{@next_id}", 'w') { |f| f.puts(transaction) }
    @next_id += 1
  end

  def read
    next_message_file = Dir['messages/*'].first
    return unless next_message_file
    yield File.read(next_message_file)
    FileUtils.rm_rf(next_message_file)
  end
end

```

This is fairly simple Ruby code, but it's the most complicated we've had in the book so far, so let's run through how it works. First we have a class method, `TransactionQueue.clear`, which we'll use to ensure the queue is cleaned up between scenarios. When we initialize the `TransactionQueue`, we create an instance variable `@next_id`, which will be used to give each new message a unique filename. When we're asked to write a message, we create a new file in the `messages` directory, write the contents of the message into the file, and then increment `@next_id` ready for naming the next message's file.

When we're asked to read a message, we try to find a file in the `messages` directory. If the directory is empty, we just return from the method. If we do find a message, we read it, yield the contents to the caller, and then delete the message from the queue. If you're new to Ruby, you might not know how `yield` works, but try not to worry—it will become clear when you see how this code is used. As always, for more information about programming in Ruby, we recommend *Everyday Scripting with Ruby* [Mar07] or the classic *Programming Ruby: The Pragmatic Programmer's Guide* [TFH08].

We've put the `TransactionQueue` in its own file and required it from the main `lib/nice_bank.rb` program. That's because we need to use this library class both from the ATM web server and from our `TransactionProcessor`. We're going to do the same thing with the `BalanceStore`, which is what we'll build next.

Building the BalanceStore

The `BalanceStore` is a database where the latest account balance is stored. Again, we want to keep the technology simple for this example, so we'll use a very simple kind of database: a text file on disk. Here's the code:

```
Download message_queues/01/lib/balance_store.rb
require 'fileutils'

class BalanceStore
  def balance
    File.read('balance').to_i
  end

  def balance=(new_balance)
    File.open('balance', 'w') { |f| f.puts(new_balance) }
  end
end
```

We have two methods, one that reads the balance and another that sets it. They both work with a file in the root of our project called `balance`. When it's asked to set the balance, the `BalanceStore` opens the `balance` file and writes the

new balance into it. When asked for the balance, the BalanceStore reads the file and converts the contents to a number. Simple!

Now that we're persisting state to disk in our TransactionQueue and BalanceStore, we need to be careful that we don't leak any state out of our scenario. Even though we have only a single scenario in our features at the moment, we need to clean up each time it runs so that balances and messages don't leak from one test run into the next.

Adding Hooks to Reset State

We have two places where state will need to be cleaned up before our scenario runs. We need to set the user's account balance to zero, and we need to remove any messages that have been left in the transaction queue. Add a file features/support/hooks.rb with the following code in it:

[Download message_queues/01/features/support/hooks.rb](#)

```
Before do
  BalanceStore.new.balance = 0
  TransactionQueue.clear
end
```

We've created a new instance of the BalanceStore directly so that we can tell it to set the balance to zero. Then we use the TransactionQueue.clear method we created earlier to empty any messages out of the transaction queue.

Let's put the last piece in the puzzle by writing our TransactionProcessor.

Building the TransactionProcessor

The TransactionProcessor is a Ruby program that will run in the background, deep in the bowels of our bank's server room. Here's the code:

[Download message_queues/01/lib/transaction_processor.rb](#)

```
require_relative 'transaction_queue'
require_relative 'balance_store'

transaction_queue = TransactionQueue.new
balance_store = BalanceStore.new
puts "transaction processor ready"
loop do
  transaction_queue.read do |message|
    sleep 1
    transaction_amount = message.to_i
    new_balance = balance_store.balance + transaction_amount
    balance_store.balance = new_balance
  end
end
```

The program starts by creating an instance of the TransactionQueue and BalanceStore classes. It prints a message to the console to say it's started up and then enters a loop. The loop tries to read a message off the transaction queue. If it finds one, it pauses for a second, calculates the new balance, and then stores it on the BalanceStore. We've introduced the pause to demonstrate the effects of working with an asynchronous component in our system: this delay should mean the test will fail consistently because the backend will take so long to update the balance that Cucumber will have already finished the scenario.

That should complete the implementation of our new architecture. Now it's time to test it.

9.4 Fixing the Flickering Scenario

We're almost ready to run our scenario again, but before we do, there's one thing we need to change in our tests. Right now, when we run cucumber, we'll rely on Capybara to control the web server part of our architecture, but we need a way to start up the backend transaction processor too. To do that, we'll use a neat little gem called Service Manager.²

Installing and Configuring Service Manager

First we need to add service_manager to the Gemfile:

```
Download message_queues/01/Gemfile
source :rubygems

gem 'sinatra', '1.3.1'
gem 'service_manager', '0.6.2'

group :development do
  gem 'rspec', '2.7.0'
  gem 'cucumber', '1.1.3'
  gem 'capybara', '1.1.2'
  gem 'launchy', '2.0.5'
end
```

Now run bundle install to install the new gem.

Service Manager will automatically start and stop multiple backend processes when you ask it to, which takes away a lot of the pain of working with backend services in your tests. We need to give Service Manager a bit of configuration

2. http://rubygems.org/gems/service_manager. Sadly, at the time of writing, Service Manager does not support Windows. See [Building a Poor Man's Service Manager for Windows, on page 166](#) for a workaround.

Building a Poor Man's Service Manager for Windows

At the time of writing, Service Manager does not work on Windows. If you're using Windows, you'll need to ignore those parts of this chapter, but you'll still need a way to start up the backend service before the tests run and stop it again at the end. We're going to show you another way to control processes that's a little bit more low-level than Service Manager but should work on Windows.

The first thing you'll need is to add another gem, childprocess, to your Gemfile:

```
gem 'childprocess'
```

In your features/support directory, add a script to start and stop the backend service:

```
Download message_queues/03-win32/features/support/services.rb
require 'childprocess'

process = ChildProcess.build('ruby', 'lib/transaction_processor.rb')
process.start
at_exit { process.stop }
```

This illustrates the basics of using ChildProcess to control an external process. We tell ChildProcess the program we want to run and tell it to start. We use Ruby's at_exit hook to tell ChildProcess to stop the transaction processor when the Cucumber tests are all done and Cucumber is exiting.

There is one slight flaw in this technique. Unlike Service Manager, we don't wait for the transaction processor to confirm that it's started by looking for the printed ready statement in the output. This is because Windows can sometimes buffer process output, so we could be waiting forever for anything to appear! So, it's possible that we could go ahead and start running our tests before the transaction processor was ready to work. This is unlikely because the transaction processor is pretty lightweight, but you might need to consider this in your own applications and work around it by sending a signal from the backend process, like touching a file, for example.

so it knows what to do. You'll need to create a new config directory and add this file:

```
Download message_queues/01/config/services.rb
require "service_manager"

ServiceManager.define_service "transaction_processor" do |s|
  s.start_cmd = "ruby lib/transaction_processor.rb"
  s.loaded_cue = /ready/
  s.cwd = Dir.pwd
  s.pid_file = 'transaction_processor.pid'
end
```

This configuration tells Service Manager what command to run in order to start our backend service. It also gives it a string to watch out for (in this case ready) so that it knows when the service has started up. We told it the working

directory to use, and we gave it the name of a process identifier (PID) file where Service Manager can keep track of the identity of the running service. When Cucumber has finished, Service Manager will use the PID to shut down the backend service for us automatically.

Now we just need to tell Service Manager to start up when our tests run. Add a file features/support/services.rb to the project:

```
Download message_queues/01/features/support/services.rb
require "service_manager"

ServiceManager.start
```

Now, when Cucumber runs, that file will load and tell Service Manager to get started. It will check its configuration file and start up our backend transaction processor.

Finally, we're ready to run our scenario again.

Investigating the Flickering

Our tests are all hooked up to our new architecture, so we're ready to run them and do some investigation into our flickering tests. Run Cucumber and give it a try:

```
Starting transaction_processor in ~/message_queues/01 with
  'ruby lib/transaction_processor.rb'
transaction processor ready
Server transaction_processor (94557) is up.
Feature: Cash Withdrawal

  Scenario: Successful withdrawal from an account in credit
    Given my account has been credited with $100
    When I withdraw $20
    Then $20 should be dispensed
    And the balance of my account should be $80

    expected: 80
    got: 0

    (compared using ==)
    (RSpec::Expectations::ExpectationNotMetError)
    ./features/step_definitions/account_steps.rb:6
    features/cash_withdrawal.feature:6

  Failing Scenarios:
  cucumber features/cash_withdrawal.feature:2

  1 scenario (1 failed)
  4 steps (1 failed, 3 passed)
```

```
0m0.602s
Shutting down transaction_processor (94557)
Server transaction_processor (94557) is shut down
```

You'll notice the extra output from Service Manager at the top and bottom of the output, telling us that it's started up the backend service for us and stopped it again. The test has failed, though, and the cause is because of timing: Cucumber looked in the balance store and found that the balance was zero when it expected it to be 80. That's because we introduced the one-second pause into the transaction processor, meaning it loses its race with Cucumber every time. It didn't get chance to update the balance before Cucumber checked it and failed the scenario.

Let's swap things around and give the handicap to Cucumber instead of the transaction processor. Remove the sleep 1 line from the transaction processor and put it into Cucumber's Then step instead. Your files should now look like this:

```
Download message_queues/02/lib/transaction_processor.rb
require_relative 'transaction_queue'
require_relative 'balance_store'

transaction_queue = TransactionQueue.new
balance_store = BalanceStore.new

puts "transaction processor ready"
loop do
  transaction_queue.read do |message|
    transaction_amount = message.to_i
    new_balance = balance_store.balance + transaction_amount
    balance_store.balance = new_balance
  end
end
```

```
Download message_queues/02/features/step_definitions/account_steps.rb
Given /^my account has been credited with (#\{CAPTURE_CASH_AMOUNT}\)$/
  my_account.credit(amount)
end

Then /^the balance of my account should be (#\{CAPTURE_CASH_AMOUNT}\)$/
  sleep 1
  my_account.balance.should eq(amount)
end
```

We've moved the sleeps around so that Cucumber loses the race every time, meaning the backend processor will always finish first. Run cucumber now, and you should see the scenario pass.

Catching Exceptions from External Services

When you run the code you're testing in the same Ruby process as Cucumber is running in, any exception raised by the code being tested will be raised right up and displayed by Cucumber in the console. When the error occurs in an out-of-process component, however, Cucumber can't see the error and won't report it to you. This can be really confusing because your test fails but you have no idea why.

If you use Service Manager to orchestrate your backend services, and those services report their errors to their own console output, then Service Manager will print them into Cucumber's console output too. Otherwise, an easy route is to write the errors to a log file and then check that log file from your Cucumber tests, perhaps in an After block.

This illustrates just how fragile tests for asynchronous systems can be to timing issues. We could just leave it like this, but for one thing, it's unnecessarily slow. Even though the transaction processor takes much less than a second to update the balance, our step will always pause for the full second. Also, if we make a change in the future to the transaction processor that means it takes more than one second, the test will start to fail again.

Try This

To experience the true randomness of a flickering scenario, remove both sleeps and run cucumber several times over:

```
$ ruby -e "30.times { system 'cucumber -f progress' }"
```

You should find that, from that sample of several runs, you'll get a mixture of passing and failing results.

Using Sampling to Fix the Flickering

Now that you've seen a flickering test at firsthand, let's fix it properly. What we need to do is change the way our Then step behaves so that rather than simply checking the balance once, it checks it repeatedly until it reaches the value we expect.

Let's imagine we have a method eventually in our World that will keep trying to run a block of code until either it stops raising an error or it reaches a time limit. Here's how we could use it:

`Download message_queues/03/features/step_definitions/account_steps.rb`

```
Then /^the balance of my account should be (#\{CAPTURE_CASH_AMOUNT}\)$/
  do |amount|
    eventually { my_account.balance.should eq(amount) }
  end
```

**Joe asks:**

Should I Show Synchronization Points in My Features?

In this chapter, we've used the `eventually` method to introduce an implicit synchronization point into our scenario without changing the Gherkin itself. This means that the fact that the behavior is actually asynchronous is hidden from anyone reading the scenario. In many situations, asynchronous behavior is a technical detail that doesn't need to be surfaced in the features, but sometimes it's something your readers will want to see in the scenarios.

The key question is whether the nontechnical stakeholders on your project *care* about the asynchronous behavior of the system. Ask them. Perhaps they would prefer to read this:

```
Given my account has been credited with $100
When I withdraw $20
Then $20 should be dispensed
> When I wait for all the transactions to be processed
  Then the balance of my account should be $80
```

Remember, writing Cucumber features is about communicating the behavior of the system. The right amount of detail will be different for every team and every project, so ask your team what they think.

Because the `should` assertion will raise an error if the balance isn't the expected amount, `eventually` will keep trying until the assertion passes or the timeout is reached. We can implement this method by adding a new module of code to our support layer:

[Download message_queues/03/features/support/async_support.rb](#)

```
module AsyncSupport
  def eventually
    timeout = 2
    polling_interval = 0.1
    time_limit = Time.now + timeout
    loop do
      begin
        yield
      rescue Exception => error
      end
      return if error.nil?
      raise error if Time.now >= time_limit
      sleep polling_interval
    end
  end
end
World(AsyncSupport)
```

This code uses Ruby's `begin..rescue` structure to run the block of code passed to the method and catch any error. If there was no error, we return from the method, meaning the step can carry on successfully. Remember that Cucumber will assume the step has passed unless it raises an error. If there was an error and the time limit has been reached, we raise the error so the step will fail. Finally, we take a little sleep to give the system a chance to do other things before we go around the loop again.

To really test this, put the `sleep 1` line back into the transaction processor and run cucumber. You should see the scenario pause for a second as our `eventually` method waits for the system to update the balance and then pass.

Try This

To satisfy yourself that our new method would fail if there were a problem with the transaction processor, try putting the sleep in the transaction processor up to three seconds. Because the timeout in our `eventually` is only two seconds, the scenario should fail.

Testing That Nothing Happens

One last thing to consider: suppose you have a scenario where you credit the account with \$100, withdraw \$100, and then check that the balance is zero. If the test passes, how can you be sure that the tests have seen the balance *after* the transactions have been processed? It's possible that the transaction processor might not have started processing the transactions or even failed altogether. You'd never know, because the outcome you're looking for to indicate success looks just like the state of the system when you started the scenario.

This kind of problem illustrates the disadvantage of using sampling to synchronize. If you could listen for events from the Transaction Processor, you could wait until you were sure it had finished its work before checking the balance. Using sampling, you're still left with a potential timing problem.

In these circumstances, you need to introduce an intermediate synchronization point into the scenario to make sure that the system has moved away from the default state. After crediting the account with \$100, you need to wait for that credit to have an effect, sampling for the changed balance, before proceeding with the next step.

9.5 What We Just Learned

When you add asynchronous behavior to a system, you need to make a concerted effort to tame the random effects that it can have on your tests. Build



Matt says:

Other Asynchronous Assertions

There are alternatives to rolling your own eventually as we've done in this chapter. Josh Chisholm's `Anticipate`^a gem gives you a fluent interface for expressing the same kind of assertion:

```
trying_every(0.1).seconds.failing_after(20).tries do
  my_account.balance.should eq(amount)
end
```

The `eventually` itself has been added into the `Wrong`^b assertion library.

-
- a. <http://rubygems.org/gems/anticipate>
 - b. <http://rubygems.org/gems/wrong>

your tests with a knowledge of how the system works and introduce synchronization points where timing issues are likely to arise.

Using sleeps in your steps is not a good way to tackle these timing issues, because it makes your tests slow and doesn't solve the reliability problem: if the system changes and becomes slower, your sleep may not be long enough and the test will start to break again. The best solution is to listen for events broadcast by the system and pause at the appropriate points in the scenario until those events have been received. That way, you minimize the amount of time the tests waste waiting for the system.

The next best solution is to use sampling to repeatedly poll the system, looking for an expected change of state. This approach works in most circumstances, but you need to take care, especially when the outcome you're looking for at the end of the scenario looks just the same as at an earlier time in the scenario.

Try This

Do some more experiments with this code, adding sleeps into the code in different places to make sure you understand how the timing issues affect the reliability of the scenario. Try to draw a sequence diagram⁵ of what's going on as the scenario runs.

Think about your own system. Are there any places where there is asynchronous behavior? How do you work around that in the tests at the moment? Could you improve on that using what you've learned in this chapter?

5. http://en.wikipedia.org/wiki/Sequence_diagram

CHAPTER 10

Databases

Back in [Chapter 6, When Cucumbers Go Bad, on page 85](#) we described the risks associated with *leaky scenarios*, where data left behind by one scenario affects the results of the next. In this chapter we're going to illustrate this problem with an example, and we'll describe the two methods for dealing with it, along with their advantages and disadvantages.

For this final installment of the worked example, we're going to introduce a relational database into our ATM system. Almost every system you'll write Cucumber tests for will have a database of some kind, and it helps to know how to talk to it directly from your test code. You can use ActiveRecord to connect to almost any kind of database and set up or inspect its data from your tests. We're going to show you how to use the ActiveRecord¹ gem to connect to existing databases from just a few lines of Ruby code.

Right now, our file-based database can only store the balance of a single customer's account. Let's introduce the capacity to store the balance for several different accounts, using a database table to store each customer's account balance in a separate row.

We'll use an sqlite3² database, although the choice of database is not all that important to this example. That's because we'll be talking to it through ActiveRecord, which provides the same consistent interface to every relational database that it talks to.³ It's a useful library for testers to know about, so we'll start with an introduction to that library before we plug it into our example.

-
1. <http://rubygems.org/gems/activerecord>
 2. <http://www.sqlite.org/>
 3. At the time of this writing, ActiveRecord has drivers for MySQL, PostgreSQL, SQLite, SQL Server, Sybase, DB2, and Oracle databases.

10.1 Introducing ActiveRecord

ActiveRecord was born in the Ruby on Rails framework, but it's easy to use it from stand-alone Ruby code to talk to any existing database. For example, if we had an accounts table with the following data in it:

id	number	balance
1	1765	80
2	2214	250

then we could query that database table using the following Ruby code:

```
>> account = Account.find_by_number(2214)
>> puts account.balance
=> 250
```

Similarly, to add a new row to the table, we could do this:

```
new_account = Account.create!(:number => 1234, :amount => 0)
```

We still need to define this Account class, but the clever thing is, *we don't need to define the database columns in the code*. Here's how the class is defined:

```
class Account < ActiveRecord::Base
end
```

That's it! ActiveRecord examines the database schema and sprinkles a little bit of Ruby's metaprogramming magic to dynamically extend the Account class with attribute accessor methods, each representing a column in the database table. It even creates special class methods like `find_by_number` for searching for rows by a particular field. All we have to do is inherit from `ActiveRecord::Base`, and ActiveRecord does the rest.

How does it know which table to look at? ActiveRecord espouses a principle of *convention over configuration*. This means that the expected name for the database table represented by the Account ActiveRecord class is `accounts`; if the table were named `wIDGETS`, we'd call the class `Widget`. As long as we stick to those conventions, we get a lot of very useful behavior for very little code.

Managing Schema Changes with Migrations

ActiveRecord also provides a mechanism for dealing with changes to your database schema, which is great when you're developing a database. Each incremental change to the schema is defined in a *migration* script. Each migration has a unique version number, and ActiveRecord stores these version numbers in a special table (`schema_migrations`) as they are applied to the database,

Tweaking ActiveRecord for Nonconventional Databases

ActiveRecord's default conventions are great if you're building a database from scratch, but what if you're working with a legacy system? If your database doesn't fit ActiveRecord's conventions, you can override the defaults like this:

```
class Account < ActiveRecord::Base
  set_table_name 'tblAccount'
  alias_attribute :number, 'intNumber'
  alias_attribute :balance, 'moneyBalance'
end
```

The `set_table_name` method allows you to tell ActiveRecord the name of the table to use, while `alias_attribute` tells ActiveRecord you want to map the ugly name of the database column to a more conventional method name on the Ruby class.

meaning it can always tell which migrations need to be applied to bring a schema up-to-date. For more about migrations, check the documentation.⁴

Now you have some grounding in ActiveRecord, let's get our hands dirty and start adding it to our banking system.

10.2 Refactoring to Use a Database

In our current system, the balance of the single account is stored in a file and read and written by the `BalanceStore` class. In our new design, we will make the `Account` responsible for reading the balance straight out of a database instead. It's refactoring time again!

We'll start by moving the `Account` class into its own `lib/account.rb` file and make it an ActiveRecord class:

[Download databases/00/lib/account.rb](#)

```
require 'active_record'
ActiveRecord::Base.establish_connection(:adapter => 'sqlite3',
                                       :database => 'db/bank.db')
ActiveRecord::Migrator.migrate("db/migrate/")
class Account < ActiveRecord::Base
  validates_uniqueness_of :number
  def queue
    @queue ||= TransactionQueue.new
  end

  def credit(amount)
    queue.write("+#{amount},#{number}")
  end
```

4. <http://api.rubyonrails.org/classes/ActiveRecord/Migration.html>

```

def debit(amount)
  queue.write("-#{amount},#{number}")
end
end

```

At the top of the file we set up ActiveRecord to use a SQLite database in db/bank.db. We are also telling ActiveRecord to run migrations in the db/migrate/ directory. This will force it to update our database schema, if necessary, as soon as we load this file. Then we create our Account, inheriting from ActiveRecord::Base so that it will act on the accounts table we're about to create.

We'll create an accounts table shortly with three columns:

- id The primary key, which is generated automatically by ActiveRecord.
- number The account number, which serves as a secondary key. This has to be unique. We've used ActiveRecord's validates_uniqueness_of method within our Account class to enforce this.
- balance The current balance of the account.

ActiveRecord will make all three fields available as attributes of each Account instance. This is why we've removed the :balance field that we had before we refactored the class. We've also changed the credit and debit methods to write out the account number as well as the amount so that the TransactionProcessor can know what account a message refers to.

Let's set up a database migration that will create the accounts table if it doesn't exist. We'll store the migration script in db/migrate/01_create_accounts.rb:

[Download databases/00/db/migrate/01_create_accounts.rb](#)

```

class CreateAccounts < ActiveRecord::Migration
  def change
    create_table :accounts do |t|
      t.string :number
      t.integer :balance
    end
  end
end

```

Now that we have an Account that knows how to persist itself along with the balance it's time to get rid of the BalanceStore class, so we will simply delete the lib/balance_store.rb file. We also need to modify features/support/hooks.rb, which no longer needs to reference it:

[Download databases/00/features/support/hooks.rb](#)

```

Before do
  TransactionQueue.clear
end

```

The next step is to update our require_relative statements. The top of our lib/nice_bank.rb now looks like this:

```
Download databases/00/lib/nice_bank.rb
require_relative 'transaction_queue'
require_relative 'account'

class Teller
```

And here is lib/transaction_processor.rb:

```
Download databases/00/lib/transaction_processor.rb
require_relative 'transaction_queue'
require_relative 'account'

transaction_queue = TransactionQueue.new

puts "transaction processor ready"
```

Before we run Cucumber again, we'll add the activerecord and sqlite3 gems to our Gemfile:

```
Download databases/00/Gemfile
source :rubygems

gem 'sinatra', '1.3.1'
gem 'service_manager', '0.6.2'
➤ gem 'activerecord', '3.1.3'
➤ gem 'sqlite3', '1.3.5'

group :development do
  gem 'rspec', '2.7.0'
  gem 'cucumber', '1.1.3'
  gem 'capybara', '1.1.2'
  gem 'launchy', '2.0.5'
end
```

Now let's run bundle install followed by cucumber:

```
== CreateAccounts: migrating =====
-- create_table(:accounts)
  -> 0.0010s
== CreateAccounts: migrated (0.0011s) =====

Starting transaction_processor in ~/message_queues/01 with
'ruby lib/transaction_processor.rb'
transaction processor ready
Server transaction_processor (94557) is up.
Feature: Cash Withdrawal

  Scenario: Successful withdrawal from an account in credit
    Given my account has been credited with $100
```

```

When I withdraw $20
Then $20 should be dispensed
lib/transaction_processor.rb:13:in `block (2 levels) in <main>':
undefined local variable or method `balance_store' for main:Object (NameError)
    from ~/databases/00/lib/transaction_queue.rb:21:in `read'
    from lib/transaction_processor.rb:10:in `block in <main>'
    from lib/transaction_processor.rb:9:in `loop'
    from lib/transaction_processor.rb:9:in `<main>'
And the balance of my account should be $80

expected: 80
      got: nil

(compared using ==)
(RSpec::Expectations::ExpectationNotMetError)
./features/step_definitions/account_steps.rb:7
./features/support/async_support.rb:8
./features/support/async_support.rb:6
./features/support/async_support.rb:6
./features/step_definitions/account_steps.rb:7
features/cash_withdrawal.feature:6

Failing Scenarios:
cucumber features/cash_withdrawal.feature:2

1 scenario (1 failed)
4 steps (1 failed, 3 passed)
0m2.284s
Shutting down transaction_processor (94557)
Server transaction_processor (94557) is shut down

```

The first thing we see is that the database is automatically migrated—the accounts table is created. This technique of letting the application automatically migrate the database when it starts is a handy trick. But then we get a failure further down.

It looks like we have some more work to do. Not to worry—Cucumber tells us exactly what we need to fix. We no longer have a `balance_store`, so we need to load the balance in a different way.

10.3 Reading and Writing to the Database

When we modified the Account earlier, we made it write out each transaction message as `amount,account_number`. The TransactionProcessor needs the `account_number` to find an account in the database and update its balance:

[Download databases/01/lib/transaction_processor.rb](#)

```

require_relative 'transaction_queue'
require_relative 'account'

```

```

transaction_queue = TransactionQueue.new
puts "transaction processor ready"
loop do
  transaction_queue.read do |message|
    sleep 1
  >  transaction_amount, number = message.split(/,/)
  >  account = Account.find_by_number!(number.strip)
  >  new_balance = account.balance + transaction_amount.to_i
  >  account.balance = new_balance
  >  account.save
end
end

```

However, looking at the message files in the messages directory, we see that the account numbers are missing. That's because we haven't assigned an account number anywhere! Let's go back to the code where we instantiate an Account and make it assign a number, and while we're at it, let's save it to the database using ActiveRecord's create! method:

```

Download databases/02/features/support/world_extensions.rb
def my_account
  @my_account ||= Account.create!(:number => "test", :balance => 0)
end

```

The step that failed is now passing, and we're left with a last failing step:

```

Starting transaction_processor in ~/message_queues/01 with
'ruby lib/transaction_processor.rb'
transaction processor ready
Server transaction_processor (94557) is up.
Feature: Cash Withdrawal

  Scenario: Successful withdrawal from an account in credit
    Given my account has been credited with $100
    When I withdraw $20
    Then $20 should be dispensed
./.rvm/gems/ruby-1.9.3-p0@hwcuC/gems/activerecord-3.1.3/
lib/active_record/relation/finder_methods.rb:266:in `find_by_attributes':
Couldn't find Account with number =  (ActiveRecord::RecordNotFound)
from ~/.rvm/gems/ruby-1.9.3-p0@hwcuC/gems/activerecord-3.1.3/
lib/active_record/base.rb:1063:in `method_missing'
from lib/transaction_processor.rb:12:in `block (2 levels) in <main>'
from ~/databases/02/lib/transaction_queue.rb:21:in `read'
from lib/transaction_processor.rb:8:in `block in <main>'
from lib/transaction_processor.rb:7:in `loop'
from lib/transaction_processor.rb:7:in `<main>'
And the balance of my account should be $80

expected: 80
got: 0

```

```
(compared using ==)
(RSpec::Expectations::ExpectationNotMetError)
./features/step_definitions/account_steps.rb:7
./features/support/async_support.rb:8
./features/support/async_support.rb:6
./features/support/async_support.rb:6
./features/step_definitions/account_steps.rb:7
features/cash_withdrawal.feature:6
```

Failing Scenarios:

```
cucumber features/cash_withdrawal.feature:2
```

```
1 scenario (1 failed)
4 steps (1 failed, 3 passed)
0m2.389s
Shutting down transaction_processor (94557)
Server transaction_processor (94557) is shut down
```

We expected the balance to be \$80, but it looks like it's still \$0. Where did the money go? Let's look in the database.

```
$ sqlite3 db/bank.db
sqlite> select * from accounts;
1|test|80
sqlite> .quit
```

Phew! The money is safe in our account. It turns out the step failed because we are still looking at the original instance of the account record—stored in @my_account—from when we created it with a zero balance. Even though the underlying database row has been modified by the TransactionProcessor, ActiveRecord doesn't know that the record we have is out-of-date, so we need to tell it to reload the record from the database:

[Download databases/03/features/step_definitions/account_steps.rb](#)

```
Then /^the balance of my account should be #{$CAPTURE_CASH_AMOUNT}$/ do |amount|
  eventually { my_account.reload.balance.should eq(amount) }
end
```

Let's run the scenario again:

```
Starting transaction_processor in ~/message_queues/01 with
'ruby lib/transaction_processor.rb'
transaction processor ready
Server transaction_processor (94557) is up.
Feature: Cash Withdrawal
```

```
Scenario: Successful withdrawal from an account in credit
  Given my account has been credited with $100
    Validation failed: Number has already been taken
```

```
(ActiveRecord::RecordInvalid)
./features/support/world_extensions.rb:20
./features/step_definitions/account_steps.rb:2
features/cash_withdrawal.feature:3
When I withdraw $20
Then $20 should be dispensed
And the balance of my account should be $80

Failing Scenarios:
cucumber features/cash_withdrawal.feature:2

1 scenario (1 failed)
4 steps (1 failed, 3 skipped)
0m0.077s
Shutting down transaction_processor (94557)
Server transaction_processor (94557) is shut down
```

This time the first step in our scenario, which ran perfectly fine the last time we run the scenario, has failed. We have stumbled upon one of the most common problems of automated tests for a system using a database. The previous test run left data in the database, and running it again makes it fail. We can't create an account because the Account number has to be unique, and a row for that Account was left behind by the previous test run. We have a *leaky scenario!*

The good thing about common problems is that there often is a common solution. We have to make sure each scenario starts with a clean database. There are two strategies to achieve this—*transaction* and *truncation*. We'll explore both, starting with transactions.

10.4 Cleaning the Database with Transactions

This is a clever approach that uses the database's transaction support in a somewhat unusual way. Before the scenario starts, we start a new database transaction in a Before hook. Then, our step definitions and application insert and modify data in the database. However, since this is happening in an uncommitted transaction, nothing actually gets changed in the database until the transaction is committed. Then, when the scenario is over (in an After hook), we do the opposite. We roll the transaction back! All the data that was modified during the scenario gets lost, and the database is back in its original state. You can see how this works in [Figure 11, Database transactions, on page 182](#).

This is actually what we want. It means that the next scenario that comes along starts with a blank slate, and we don't need to worry about leftovers from the previous scenario.

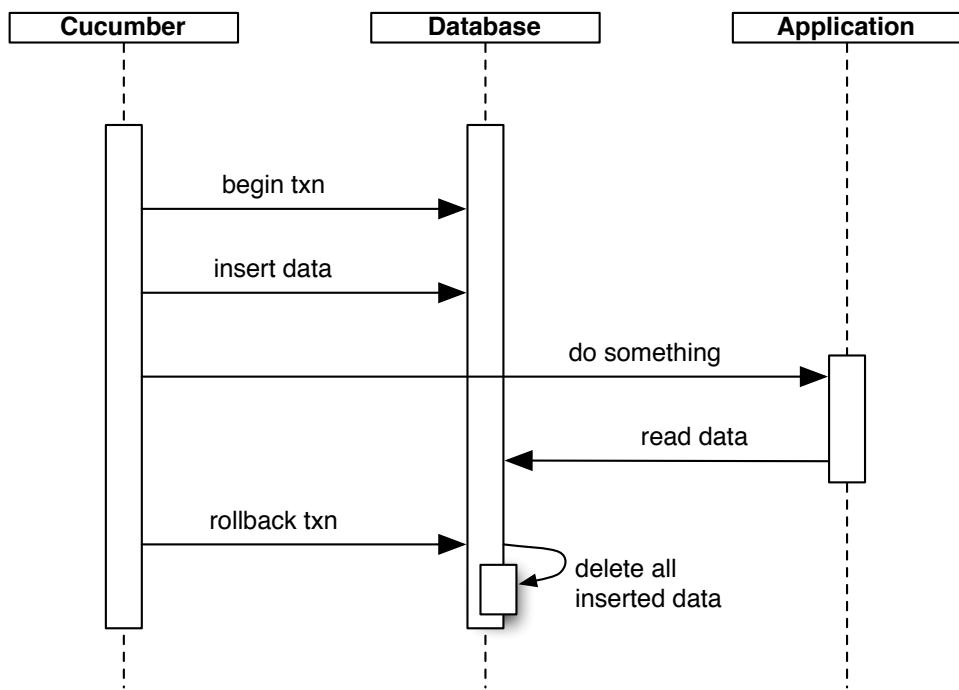


Figure 11—Database transactions

Unfortunately, the transaction approach doesn't work for our application. Understanding why this doesn't work is essential, so we are going to try it anyway. Seeing it fail and understanding why will save you hours of problem solving in the future. It will also help you understand when to use transactions and when not to use them.

Our previous run would have left a row in the accounts table, so let's remove that before we try to run with transactions:

```
$ sqlite3 db/bank.db
sqlite> delete from accounts;
sqlite> .quit
```

Now that we have a clean database, let's configure Cucumber to begin and roll back a transaction. DatabaseCleaner⁵ is a Ruby gem that provides an easy way to ensure the database is in a clean state before each scenario starts. Let's add it to our Gemfile:

5. DatabaseCleaner (https://github.com/bmabey/database_cleaner) was written by Ben Mabey, who was also one of the first contributors to Cucumber.

Download databases/04/Gemfile

```
source :rubygems

gem 'sinatra', '1.3.1'
gem 'service_manager', '0.6.2'
gem 'activerecord', '3.1.3'
gem 'sqlite3', '1.3.5'

group :development do
  gem 'rspec', '2.7.0'
  gem 'cucumber', '1.1.3'
  gem 'capybara', '1.1.2'
  gem 'launchy', '2.0.5'
  > gem 'database_cleaner', '0.7.0'
end
```

Now we have to define a Before hook that starts a transaction and an After hook that rolls it back. Add the following to features/support/database.rb:

Download databases/04/features/support/database.rb

```
require 'database_cleaner'

DatabaseCleaner.strategy = :transaction

Before do
  DatabaseCleaner.start
end

After do
  DatabaseCleaner.clean
end
```

That's all you need. In transaction mode, DatabaseCleaner.start opens a transaction, and DatabaseCleaner.clean rolls it back. Let's see how that works:

```
Starting transaction_processor in ~/message_queues/01 with
'ruby lib/transaction_processor.rb'
transaction processor ready
Server transaction_processor (94557) is up.
Feature: Cash Withdrawal

  Scenario: Successful withdrawal from an account in credit
    Given my account has been credited with $100
    When I withdraw $20
    Then $20 should be dispensed
~/rvm/gems/ruby-1.9.3-p0@hwcu/gems/activerecord-3.1.3/
lib/active_record/relation/finder_methods.rb:266:in `find_by_attributes':
Couldn't find Account with number = test (ActiveRecord::RecordNotFound)
      from ~/rvm/gems/ruby-1.9.3-p0@hwcu/gems/activerecord-3.1.3/
lib/active_record/base.rb:1063:in `method_missing'
      from lib/transaction_processor.rb:12:in `block (2 levels) in <main>'
```

```

from ~/databases/04/lib/transaction_queue.rb:21:in `read'
from lib/transaction_processor.rb:8:in `block in <main>'
from lib/transaction_processor.rb:7:in `loop'
from lib/transaction_processor.rb:7:in `<main>'

And the balance of my account should be $80

```

```

expected: 80
got: 0

(compared using ==)
(RSpec::Expectations::ExpectationNotMetError)
./features/step_definitions/account_steps.rb:7
./features/support/async_support.rb:8
./features/support/async_support.rb:6
./features/support/async_support.rb:6
./features/step_definitions/account_steps.rb:7
features/cash_withdrawal.feature:6

```

Failing Scenarios:

```
cucumber features/cash_withdrawal.feature:2
```

```

1 scenario (1 failed)
4 steps (1 failed, 3 passed)
0m2.380s
Shutting down transaction_processor (94557)
Server transaction_processor (94557) is shut down

```

We told you it wasn't going to work. It's time to find out why. The TransactionProcessor couldn't find the account we created in our first step. It throws an exception, and our scenario fails. Why can't the TransactionProcessor find the account?

One of the properties of database transactions is that they are *isolated*. This means that whatever database activity happens inside a transaction cannot be seen by *any* other database connections. Don't forget that we have two database connections.

The first database connection is made by the process that runs Cucumber. Cucumber begins a database transaction and creates an account.

The second database connection is made by the TransactionProcessor, which is started in a separate process by ServiceManager. The TransactionProcessor pops credit and debit messages off a queue, looks up accounts, and updates the balance. However, since the database transaction that Cucumber started hasn't been committed, the TransactionProcessor can't see the account we created in our first step.



Aslak says: Browser Testing and Databases

The problem with transactional cleaning and multiple database connections often occurs when we test web applications. This happens when the web application has a different database connection than Cucumber. The problem typically manifests itself in two situations:

- Cucumber inserts some data, but it isn't displayed in the browser.
- A browser action causes some data to be inserted, but Cucumber can't see it.

If this happens, you need to make sure you are not starting a transaction anywhere. You have to use the *truncation* cleaning strategy instead. If you're logging your database calls, reading the logs can be very useful when diagnosing a problem like this.

In [Chapter 15, Using Capybara to Test Ajax Web Applications, on page 245](#), you will learn how to drive a browser from Cucumber. Capybara actually runs the web server in a separate thread within the same process, so you might think Cucumber and the web server share the same database connection. In fact, they don't—each thread also gets its own database connection, which means transactional cleaning won't work. It obscures data from the other connection.

Cucumber-Rails allows you to override this behavior in `features/support/env.rb` so that the Cucumber thread shares the connection with the web server. This means you can use the faster *transaction* cleaning strategy. Be warned that this works only on *some* databases. It may cause intermittent unexpected results on others!

Of course, we could have committed the transaction after creating the account, but that would defeat our goal of rolling back to get a clean database when the scenario is done. We can't roll back a committed transaction. The rule is simple: when the application has a different database connection than Cucumber, we cannot use transactions to clean the database. We have to use the other strategy instead: *truncation*.

10.5 Cleaning the Database with Truncation

Truncating the database before each scenario is a brute-force technique, and the main drawback is that it's generally slower than rolling back a transaction. This is why the transactional approach is generally preferable if you can get away with it. The advantage of truncation is that it's a cleaning strategy that works reliably when we have more than one database connection, since no transactions are used. Let's modify our cleaning strategy and see how it compares. This is a simple change in our `database.rb` file:

```
Download databases/05/features/support/database.rb
require 'database_cleaner'
```

```
DatabaseCleaner.strategy = :truncation
```

```
Before do
  DatabaseCleaner.clean
end
```

Notice how we removed the After hook and made the Before hook call DatabaseCleaner.clean instead. In truncation mode, DatabaseCleaner.start does nothing, and DatabaseCleaner.clean truncates all tables. In other words, it deletes all rows.

Running Cucumber again makes the scenario pass again:

```
Starting transaction_processor in ~/message_queues/01 with
  'ruby lib/transaction_processor.rb'
transaction processor ready
Server transaction_processor (94557) is up.
Feature: Cash Withdrawal

  Scenario: Successful withdrawal from an account in credit
    Given my account has been credited with $100
    When I withdraw $20
    Then $20 should be dispensed
    And the balance of my account should be $80

1 scenario (1 passed)
4 steps (4 passed)
0m2.207s
Shutting down transaction_processor (94557)
Server transaction_processor (94557) is shut down
```

That brings us to the end of the worked example. What follows in the next part are a series of recipes for using Cucumber in the wild. Good luck out there!

10.6 What We Just Learned

Testing applications that use databases can be difficult. You have to make sure each scenario starts with a clean slate, but cleaning the slate can have its own pitfalls. In this chapter, we learned about the benefits and drawbacks of two strategies—transaction and truncation. To sum it up:

- ActiveRecord is a useful library for quickly connecting to SQL databases.
- Resetting state between scenarios is vital; otherwise, you get weird failures.
- Transaction-based cleaning is preferred because it is fast, but it works only when there is one single-threaded process.



Joe asks:

Why Not Truncate the Database in an After Hook?

Truncating the database in an After hook usually works just as well as doing it in a Before hook, but there is a subtle difference.

First, if the process gets killed before it has time to clean up in the After hook, it might cause the next test run to fail.

Second, when a scenario fails it might not be evident why it failed, and having the ability to peek inside the database as a post-mortem often helps us understand why a scenario is failing.

As you will also see in [Isolating Scenarios, on page 285](#), this post-mortem technique can be used in other situations as well.

- Truncation-based cleaning is a slower, brute-force technique that works in multiprocess and multithreaded environments.

Try This

Find an existing database, perhaps somewhere on your network at work, and see if you can connect ActiveRecord to it. It should only take you a few lines of Ruby code to set up an ActiveRecord::Base subclass that can talk to one of the tables in the schema. Read up on ActiveRecord's API documentation⁶ and try running some queries against the database. See how easy it is to use Ruby to talk to any database!

6. <http://api.rubyonrails.org/classes/ActiveRecord/Base.html>

Part III

Cucumber Applied

The Cucumber Command-Line Interface

Cucumber can be used to test almost any computer system. In this final part of the book, we're going to give you a set of recipes that you can pick and choose from depending on your circumstances. Most of those recipes are technical: you'll find guidance on testing web services, command-line programs, and Ajax-heavy websites, but there's also a chapter about testing legacy systems that doesn't contain much code at all. We'll start by giving you an in-depth tour of Cucumber's user interface: the command line.

We have done our best to make Cucumber work out of the box without complex configuration. It doesn't require you to provide a configuration file or pass a lot of complex command-line options to work. However, there are times when we want to tweak how Cucumber behaves.

Sometimes the default output can be too verbose, or perhaps we need it in a format that is easier to share with others. We might want to run just a subset of scenarios or perhaps organize the location of our Gherkin features and step definitions a little differently.

In this chapter, we are going to take a closer look at what command-line options Cucumber has to offer and how they can help you achieve some of these things.

11.1 Cucumber's Command-Line Options

Let's start by taking a look at the command-line options. Like most command-line tools, Cucumber provides a `--help` option. The following is a shortened version of what that would print:

```
$ cucumber --help
-r, --require LIBRARY|DIR      Require files before executing the features.
--i18n LANG                    List keywords for in a particular language.
```

<code>-f, --format FORMAT</code>	Run with " <code>--i18n help</code> " to see all languages. How to format features (Default: pretty).
<code>-o, --out [FILE DIR]</code>	Write output to a file/directory instead of STDOUT.
<code>-t, --tags TAG_EXPRESSION</code>	Only execute the features or scenarios with tags matching TAG_EXPRESSION.
<code>-n, --name NAME</code>	Only execute the feature elements which match part of the given name.
<code>-e, --exclude PATTERN</code>	Don't run feature files or require ruby files matching PATTERN
<code>-p, --profile PROFILE</code>	Pull command line arguments from cucumber.yml.
<code>-P, --no-profile</code>	Disables all profile loading to avoid using the 'default' profile.
<code>-c, --[no]color</code>	Whether or not to use ANSI color in the output.
<code>-d, --dry-run</code>	Invokes formatters without executing the steps.
<code>-a, --autoformat DIR</code>	Reformats (pretty prints) feature files and writes them to DIR.
<code>-m, --no-multiline</code>	Don't print multiline strings and tables under steps.
<code>-s, --no-source</code>	Don't print the file and line of the step definition with the steps.
<code>-i, --no-snippets</code>	Don't print snippets for pending steps.
<code>-q, --quiet</code>	Alias for <code>--no-snippets --no-source</code> .
<code>-b, --backtrace</code>	Show full backtrace for all errors.
<code>-S, --strict</code>	Fail if there are any undefined or pending steps.
<code>-w, --wip</code>	Fail if there are any passing scenarios.
<code>-v, --verbose</code>	Show the files and features loaded.
<code>-g, --guess</code>	Guess best match for Ambiguous steps.
<code>-l, --lines LINES</code>	Run given line numbers. Equivalent to FILE:LINE syntax
<code>-x, --expand</code>	Expand Scenario Outline Tables in output.
<code>--drb</code>	Run features against a DRb server. (i.e. with the spork gem)
<code>--port PORT</code>	Specify DRb port. Ignored without <code>--drb</code>
<code>--version</code>	Show version.
<code>-h, --help</code>	You're looking at it.

That's a lot of options, and it may not be obvious when each one of them is useful. Let's look at some situations when you would use some of them.

11.2 Running a Subset of Scenarios

As the number of feature and scenarios grows, we'll frequently run into situations where we want to run only a single (or maybe a couple) of scenarios to get faster feedback. This can be if we are working on a new scenario or if we have broken an existing one. Let's take a closer look at how this works.

Filtering with Tag Expressions

The simplest way to use the `--tags` option is to give it a single tag to run, for example:

```
$ cucumber --tags @focus
```

This would cause Cucumber to run just the scenarios tagged with `@focus`. It's quite common to temporarily tag a scenario or feature with a unique tag like this to filter what to run.

In some situations, we may want a little more control over the tag filtering. Imagine that we have made some changes to a piece of our application that sends out emails. What if we want to run all scenarios tagged with `@focus` or `@email`? Here is how:

```
$ cucumber --tags @focus,@email
```

The comma is interpreted as a logical OR statement. It's not uncommon to have scenarios that run fast and others that run slowly. If we have tagged the fast scenarios or features with a tag, for example `@fast`, we can apply a logical AND to say we want to run scenarios that are tagged as `@fast` and either `@focus` or `@email`:

```
$ cucumber --tags @fast --tags @focus,@email
```

Here we are using `--tags` twice. Each of them will be logically ANDed together. Ideally, most of our scenarios are fast, and only a few are slow. So, instead of tagging most of our scenarios with `@fast`, it makes more sense to tag the few that are slow with `@slow`. If we want to run the scenarios that are *not* slow (and at the same time `@focus` or `@email`), we can use negation (tilde):

```
$ cucumber --tags ~@slow --tags @focus,@email
```

Let's look at another way to run a subset of scenarios—line numbers.

Filtering on Lines

You don't need to tag your scenarios or features to run just a subset of them. Cucumber also provides a convenient way to specify the line number of the scenario you want to run. Here is an example:

```
$ cucumber features/something.feature --line 45
```

If the feature file has more than one scenario, it will run only the one on line 45. The line number can be anywhere in the feature, ranging from a comment line to the last step (including a step table or doc string).

You have probably noticed that Cucumber prints out the location of a feature file with the line number in error messages.

```
features/something.feature:45
```

If you want to rerun the scenario that failed, just select the text and paste it:

```
$ cucumber features/something.feature:45
```

This is equivalent to using the `--line` option. The colon notation also allows you to specify several line numbers, allowing you to specify several scenarios to run:

```
$ cucumber features/something.feature:45:89:107
```

If you prefer, you can also do this using the `--lines` argument. Cucumber doesn't mind whether you say `--line` or `--lines`.

Filtering on Names

If tag and line filtering still doesn't meet your needs, you can filter scenarios based on name. Say, for example, that you want to run all scenarios that have the text `logout` in their name and that they are scattered around several feature files without a particular tag to identify them. Running them is just a matter of the following:

```
$ cucumber --name logout
```

In a similar fashion, you can also `--exclude` scenarios with a certain name.

11.3 Changing Cucumber's Output

Cucumber's default behavior is to output results similar to the Gherkin source text—with a little extra information such as colors, locations of matched step definitions, highlighting of arguments, and so on. This isn't the only way to output results. Cucumber lets you use a different output format. For example, if you want a minimal report with a single character per step, you can use the progress formatter:

```
$ cucumber --format progress
..U---F..
```

Each character represents the status of each step:

- `.` means passing.
- `U` means undefined.
- `-` means skipped (or a Scenario Outline step).
- `F` means failing.

Cucumber also has built-in formatters that output html, json, and junit. The latter is handy if you are running Cucumber in a continuous integration (CI) environment since most CI servers know how to interpret JUnit reports. More on that in a minute.

Special Formatters

In addition to the formatters we have seen so far, Cucumber also bundles a few formatters that create output that isn't really meant to be read as a report of the run. These formatters rather serve as a development aid.

The usage formatter lists all of the step definitions in your project, as well as the steps that are using it. It shows you unused step definitions, and it sorts the step definitions by their average execution time. The output from the usage formatter is great for quickly finding slow parts in your code but also a great way to get an overview of your step definitions.

The stepdefs formatter is similar to the usage formatter but with a little less information.

Finally, there is the rerun formatter. This is a special formatter that creates output like this:

```
$ cucumber -f rerun
features/one.feature:367 features/another.feature:91:117
```

Does this look familiar? This is the same kind of format we used to run a subset of scenarios using line filtering, which we described earlier.

If all scenarios are passing, the rerun formatter doesn't output anything at all. However, if one or more scenarios fail, it will output their location so that you can copy and paste the output to rerun just those scenarios. Being able to do this saves a lot of time when fixing broken scenarios.

Formatting to File and Using Multiple Formatters

By default, all formatters will print their output to STDOUT. So, what do we do if we want to see the usual pretty output but also have a html and rerun report? This is where you have to use the --out option. It tells Cucumber to write the report to a file instead of STDOUT. Here is an example:

```
$ cucumber -f pretty -f html --out cukes.html -f rerun --out rerun.txt
```

This tells Cucumber to write the HTML report to the file cukes.html, then rerun output to rerun.txt, and finally display the pretty formatter's output in the console.

Displaying Full Backtraces

Using the `--backtrace` option causes Cucumber to print a full backtrace for each failure instead of a shortened one. This is often useful when you are hunting bugs.

11.4 Specifying the Location of Step Definitions

Have you ever wondered how Cucumber finds your step definitions? There is nothing in Cucumber that expects them to be under `features/step_definitions`. In fact, if you renamed these two directories to `jolly/jumper` and ran `cucumber jolly`, things would work just as well.

Cucumber recursively scans *some* directories for `.rb` files to load, and if there are step definitions in them (or hooks or transforms), they get loaded. It's important to understand *what* directories Cucumber scans. Once you understand this, you will save yourself from a common source of confusion and frustration. Let's illustrate that with an example. Assume you have organized your features and step definitions as follows:

```
features
└── billing
    └── credit_card.feature
└── scoring
    ├── multi_player.feature
    └── single_player.feature
└── step_definitions
    ├── billing_steps.rb
    └── scoring_steps.rb
```

If you run all of the features with `cucumber features` or just `cucumber`, everything will work great. Then, one day you add a new scenario to your `credit_card.feature` file to support a new credit card. Then you run it:

```
$ cucumber features/billing/credit_card.feature:104 -f progress
UUUU
```

All of your steps show up as undefined, even though your new scenarios were reusing some existing step definitions. The directories scanned for step definitions and support code are determined by the feature files (or directories) you give it. If you give it a feature file, it will look underneath its directory. If you give it a directory, it will look under that directory. To see where Cucumber is looking for code, you can pass the `--verbose` option.

The solution to this common problem is to use `--require` to tell Cucumber explicitly where to load code from. The correct way to run the previous feature would be as follows:

Setting Tag Limits

Sometimes you need to keep a check on the number of scenarios with a particular tag. You can ask Cucumber to fail if the number of scenarios matching a tag exceeds a certain number, like this:

```
# Execute all scenarios tagged with @javascript, failing if
# more than ten are found.
$ cucumber --tags @javascript:10
```

```
$ cucumber features/billing/credit_card.feature:104 -f progress -r features
...F
```

Our code is now found and loaded, and we can carry on as we had intended, working on our new, failing scenario. You can configure this to happen by default using a custom profile, which we'll show you in a few pages.

11.5 Managing Your Work in Progress (WIP)

Having a lot of pending or half-implemented features and scenarios is something you should try to avoid. It makes it hard to stay focused, and it also has a negative effect on the lead time—the time from starting to releasing—of each feature.¹ A common term for started but unfinished work is *work in progress*, often called just *WIP*. If the team keeps a low number of WIP, the lead time of each feature will also be lower.

Cucumber can help you manage your WIP with the help of tags and the `--wip` option. Try to make it a habit to tag each scenario that is being worked on with `@wip` and remove the tag when it passes. Let's say your team has agreed to always keep the WIP at or below three scenarios. To enforce this, you could run Cucumber with the following option:

```
$ cucumber --wip @wip:3
```

This has some interesting side effects. First, if Cucumber finds more than three scenarios tagged with `@wip`, it will fail immediately without even running any scenarios. This is an indication that too much is being worked on and that some scenarios should be removed until the other ones are passing (and the `@wip` tag is removed).

The other side effect is that if *any scenarios* pass (and remember, you are running only the `@wip` ones), Cucumber will also fail. This kind of failure is a reminder to remove the `@wip` tag.

1. *Kanban* [And10]

11.6 Using Profiles

As you get comfortable with the various command-line options Cucumber has to offer and start to use more of them, it can be tedious to type it all on the command line over and over again. Cucumber lets you store the command-line options to a cucumber.yml YAML file. This file must be placed either in your project's root directory or underneath a directory called config. Here is a sample cucumber.yml file:

```
default: --tags ~@wip --require features
wip: --tags @wip:3 --wip --require features
```

Now, if you run cucumber --profile wip, the command-line options keyed with wip: will be transparently appended to your command line. If you don't specify a --profile argument (and you have a cucumber.yml file), Cucumber will pick the default: one.

Notice that we've added --require features to both profiles. This is because we want Cucumber to load step definitions and support code from the root directory automatically, no matter what folder we run the features from.

11.7 Running Cucumber from Rake

Many projects use a build tool to perform common tasks. For Ruby projects, Rake is the de facto build tool. Cucumber has a Rake task that makes it easy to run Cucumber from Rake. If you are using Cucumber-Rails, which we describe in [Chapter 14, Bootstrapping Rails, on page 229](#), you will have Rake tasks set up automatically.

If you are not using Rake, it's still easy to set it up yourself. Try adding this to a Rakefile in the root of your project:

```
require 'cucumber/rake/task'
```

```
Cucumber::Rake::Task.new # defines a task named cucumber
```

Now you can run your Cucumber features from Rake:

```
$ rake cucumber
```

Profiles from cucumber.yml will be picked up when you run Cucumber from Rake, but you can also define Cucumber command-line options from the Rakefile:

```
Cucumber::Rake::Task.new('cucumber_progress') do |t|
  t.cucumber_opts = %w{--format progress}
end
```

Now that you know how to run Cucumber both from its native command-line interface and from Rake, you can also make your continuous integration server run it. Cucumber (or Rake) exits with a status code of 1 if any scenarios fail, so your continuous integration server can properly detect failure.

11.8 Running Cucumber in Continuous Integration

Many teams set up their continuous integration² (CI) server to run Cucumber every time someone shares changes with the rest of the team. Since it's a command-line tool, there is nothing fancy you need to do—just plop the cucumber command into the CI project's configuration.

You may want to use slightly different command-line options for CI than you use on your own workstation. You can do this by defining a special profile for the CI build (for example ci).

Being Strict

It's common to have missing or pending steps while we're working on a new scenario. Some teams strive to keep the mainline (trunk) of the source repository pristine, without any pending or missing steps.

CI systems detect failure by inspecting the exit status of the processes it's running, and by default Cucumber exits only with an error status (a nonzero value) if there is one or more *failing* steps. If you are running Cucumber in a CI environment, you may want to set it up to fail if anyone checks in *missing* or *pending* steps.

If we pass the --strict option to cucumber, it will exit with a nonzero value if there are any *missing* or *pending* steps as well as *failing* ones. You can inspect the exit status yourself after running cucumber:

```
# OS X or Linux
$ echo $?

# Windows
$ echo %ERRORLEVEL%
```

Sharing Reports

Earlier in this chapter we saw how you can use the --format option to change Cucumber's output. If you pass --out JUNIT_DIR --format junit to Cucumber in your ci profile, you can configure your CI server to pick up those reports from JUNIT_DIR and analyze them. Some CI servers can generate a trend chart

2. <http://martinfowler.com/articles/continuousIntegration.html>

showing you and your team how healthy your build stays over time. This is a great way to get an indicator of whether the project is improving or decaying.

You can also pass the `--out HTML_FILE --format html` to get a full HTML report of the results. This is particularly handy if you are using a browser automation tool, because you can embed screenshots of the browser right into the HTML report. You can learn more about this in [Section 15.4, “Taking Screenshots, on page 271.](#)

11.9 What We Just Learned

Cucumber has a rich set of command-line options that give you full control over how it behaves. You can do the following:

- Run a subset of scenarios with `--tags`, `--name`, `--line/-lines`, or `path:line`.
- Use the `--format` option to get progress, HTML, or JUnit reports.
- List step definition statistics with `--format usage` and `--format stepdefs`.
- Use the `--require` option to control where step definitions are loaded from.
- Limit the work in progress with `--wip` and tag limits.
- Define profiles in `cucumber.yml` and run them with a single `--profile` or `-p` option.
- Integrate Cucumber into your Rake scripts.

Try This

We left out a couple of command-line options so you can experiment a little yourself. Here are some exercises you can try:

- Use the command line to list the Gherkin keywords in another language, for example Pirate or LOLCAT. Just for fun, try to write a Cucumber scenario in one of those languages.
- Run a scenario with a step that is matched by two different step definitions, one defining one argument and another defining two. After you get an *Ambiguous* error, run the scenario again, this time with `--guess`.

Testing a REST Web Service

Sometimes the user of the system we need to test isn't a human being but another computer program. For systems like these, the user interface is often a REST¹ web service. To automate tests against it, we need to make our Cucumber step definitions talk to the web service as though they were a regular client application.

There are two main approaches for testing a web application with Cucumber —*in-process* and *out-of-process*. Figure 12, *Two main approaches for testing REST APIs with Cucumber*, on page 202 gives a high-level overview of these two approaches.

The preferred approach is usually *in-process*, which means that Cucumber is running in the same Ruby process as your own application. There is no running web server or HTTP traffic, and this is both fast to run and easy to set up. Being able to easily reset state (usually in a database) between scenarios is another strength of the *in-process* approach, as we explained in Chapter 10, *Databases*, on page 173.

Your application doesn't have to be written in Ruby in order to be testable with Cucumber. Applications written in Java, .NET, PHP, or any other programming language can be tested using the *out-of-process* approach, which means starting your application before Cucumber runs and then having Cucumber talk to it using an HTTP client library.

This is obviously a very technical chapter. We'll use the Ruby web framework Sinatra to create a very simple web service from scratch. We've chosen Sinatra because it allows us to build the web service in just a few lines of code, meaning we can focus our attention on tests rather than implementation.

1. REST stands for Representational State Transfer. See http://en.wikipedia.org/wiki/Representational_state_transfer.

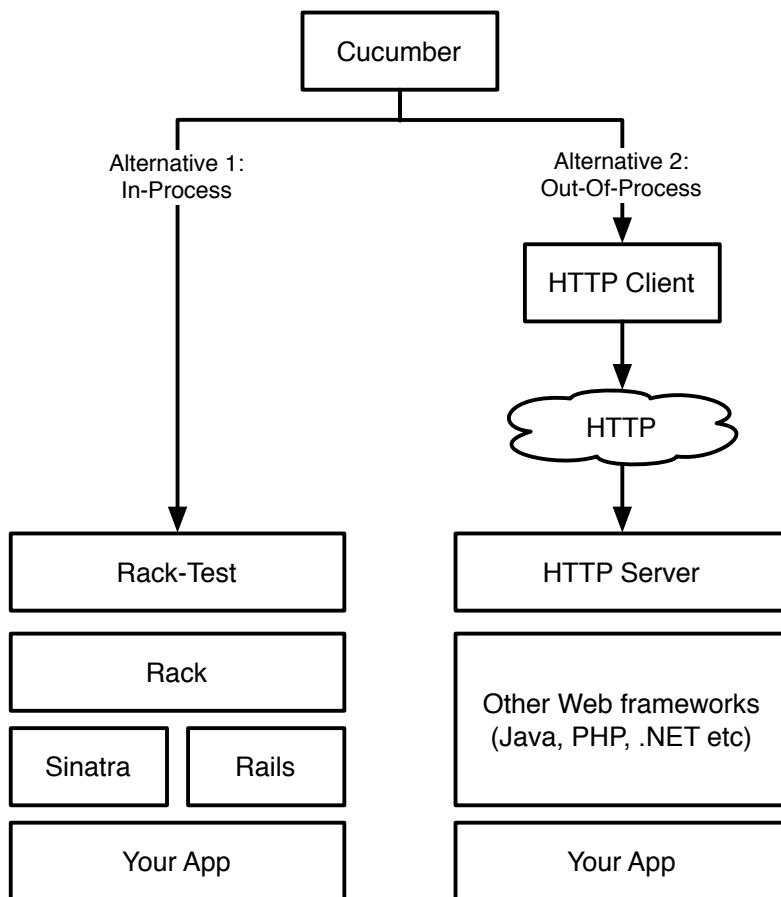


Figure 12—Two main approaches for testing REST APIs with Cucumber

12.1 In-Process Testing of Rack-Based REST APIs

Let's start with the first approach, where both Cucumber and your application run in the same Ruby process. We'll be dealing with a very simple REST API for storing and retrieving fruit. It's probably not the most useful system in the world but should allow us to illustrate the fundamentals of REST and Cucumber.

We're going to build our web service from scratch. Create a new project with a features directory in it, and create this file:

[Download rest_web_services/00/features/fruit_list.feature](#)

Feature: Fruit list

In order to make a great smoothie
I need some fruit.

```

Scenario: List fruit
  Given the system knows about the following fruit:
    | name      | color |
    | banana    | yellow |
    | strawberry | red   |
  When the client requests GET /fruits
  Then the response should be JSON:
    .....
    [
      {"name": "banana", "color": "yellow"},
      {"name": "strawberry", "color": "red"}
    ]
    .....

```

In the rest of this section we are going to implement this feature. We'll see how Cucumber can issue HTTP requests to our application.

Standing Up a Skeleton App—and Storing Some Fruit

When we run our List fruit feature with cucumber, we'll see the usual snippets:

```

Given /^the system knows about the following fruit:$/
  do |table|
    # table is a Cucumber::Ast::Table
    pending # express the regexp above with the code you wish you had
end

When /^the client requests GET \/fruits$/
  do
    pending # express the regexp above with the code you wish you had
end

Then /^the response should be JSON:$/
  do |string|
    pending # express the regexp above with the code you wish you had
end

```

Before we go ahead and paste the previous snippets into a Ruby file and fill in the blanks, let's look at each one. The first snippet is a domain-specific step definition—it's about our fruit system. The last two seem like more general-purpose step definitions for HTTP REST operations. As discussed in [Section 7.4, “Organizing the Code,” on page 128](#), we'll put each of them in a file that describes the domain concept they are related to. This makes it easier in the future to find our step definitions. Paste the first one in a `features/step_definitions/fruit_steps.rb` file and the last two in a `features/step_definitions/rest_steps.rb` file.

Let's start by making the first step pass. To keep things simple, we are not going to use a database to store our fruit—we can store them in memory. This means that in our Given step, we'll just store our fruit in a (class) variable:

[Download rest_web_services/01/features/step_definitions/fruit_steps.rb](#)

```
Given /^the system knows about the following fruit:$/
  do |fruits|
    FruitApp.data = fruits.hashes
  end
```

Of course, this is going to cause the Given step to go from pending to failing, because we don't yet have a `FruitApp` class:

```
Feature: Fruit list
  In order to make a great smoothie
  I need some fruit.
```

Scenario: List fruit

```
  Given the system knows about the following fruit:
```

name	color
banana	yellow
strawberry	red

```
  uninitialized constant FruitApp (NameError)
  ./features/step_definitions/fruit_steps.rb:2
  features/fruit_list.feature:6
```

When the client requests GET /fruits

Then the response should be JSON:

```
"""
[
  {"name": "banana", "color": "yellow"},
  {"name": "strawberry", "color": "red"}
]
"""
```

Failing Scenarios:

cucumber features/fruit_list.feature:5

```
1 scenario (1 failed)
3 steps (1 failed, 2 skipped)
0m0.004s
```

We're going to implement our little REST web service with the Sinatra web framework because it is so simple. If you were to use a different Rack-based web framework such as Rails, the Cucumber features and Ruby files under the `features` directory would be exactly the same, but there would be much more application code for us to include in the book. Let's start by installing all the gems we're going to need. Create a `Gemfile` in the root of the project:

[Download rest_web_services/01/Gemfile](#)

```
source 'http://rubygems.org'

gem 'sinatra', '1.3.1'
gem 'json', '1.6.3'

group :test do
```

```
gem 'cucumber', '1.1.3'
gem 'rspec', '2.7.0'
gem 'rack-test', '0.6.1'
end
```

We need Sinatra and JSON to run our web service; we need Cucumber to run the tests, RSpec to make assertions in the step definitions, and Rack-Test to automate the web service from our step definitions. Now tell Bundler to install the gems:

```
$ bundle
```

With that out of the way, we can get on with building our web server. Create a file named `fruit_app.rb` in the directory above the `features` directory. Add the following code to that file:

```
Download rest_web_services/02/fruit_app.rb
require 'sinatra'

class FruitApp < Sinatra::Base
  set :data, ''
end
```

We've created a class `FruitApp` that inherits from Sinatra's base class; this will become our web server. We need to tell Cucumber to load that file. We'll create a `features/support/env.rb` file to do that:

```
Download rest_web_services/02/features/support/env.rb
require File.join(File.dirname(__FILE__), '..', '..', 'fruit_app')
```

The first line loads the `fruit_app.rb` file from the root directory of our project—two directories up from this file. Try to run cucumber now. It should make our first step pass. It's time to get into the REST business!

Poking Our Application with Rack-Test

Since we are running Cucumber in the same Ruby process as the application itself, we have the luxury of being able to talk to it directly, without going through HTTP or a web server. This is possible thanks to a nifty little Ruby gem called `rack-test`. Rack-Test is a library that is designed to be used in tests and that acts as a façade² to your web application, very much in the same way as a regular web server does. Rack-Test has methods for mimicking HTTP requests from code. As shown in [Figure 13, Testing an in-process Rack application with Cucumber, on page 207](#), we'll be talking to our application through

2. http://en.wikipedia.org/wiki/Facade_pattern

Sinatra

Sinatra is a lightweight web framework for creating web applications. The simplest Sinatra web application looks like this:

```
require 'sinatra'

get '/' do
  'Hello world!'
end
```

Sinatra, like most other Ruby web application frameworks, is based on the low-level web server interface Rack. In this chapter, we are using Rack-Test, a testing library that can talk to any Rack-based web application. This means we could implement the application in this chapter in a different Rack-based web framework, like Rails or Ramaze—using the same Cucumber features and step definitions.

You can read more about Sinatra at <http://sinatrarb.com>.

Rack-Test, which exposes Rack's API for talking to our application just like an HTTP server would.

Let's put Rack-Test to use in the `/^the client requests GET /fruits$/` step definition in the `rest_steps.rb` file, and while we're at it, let's turn the request path into a variable with a capture group—this makes our step definition usable for other paths too.

```
When /^the client requests GET (.*)$/ do |path|
  get(path)
end
```

We're calling Rack-Test's `get` method here, but how exactly does Cucumber know about this method? Well, it doesn't actually, so we have to wire it up. The `get` method is defined in a module called `Rack::Test::Methods`, which gets loaded when we require `'rack/test'`. To wire it all up, we need to modify our `env.rb` file a little more:

```
Download rest_web_services/03/features/support/env.rb
require File.join(File.dirname(__FILE__), '..', '..', 'fruit_app')
require 'rack/test'

module AppHelper
  # Rack-Test expects the app method to return a Rack application
  def app
    FruitApp
  end
end

World(Rack::Test::Methods, AppHelper)
```

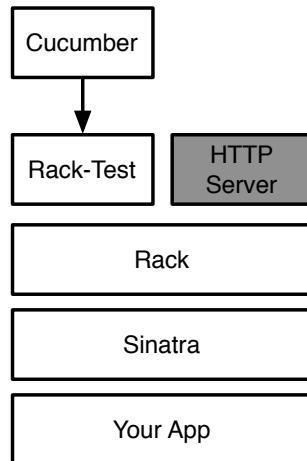


Figure 13—Testing an in-process Rack application with Cucumber

Let's take a look at this code. The last line with the `World` call registers two Ruby modules within Cucumber: first `Rack::Test::Methods`, which defines the HTTP simulation methods like `get`, and then our own helper module that we define right above. Both these modules will be mixed into Cucumber's `World`, as we've explained in [Chapter 8, Support Code, on page 133](#). The reason we need our own little `AppHelper` is that `Rack-Test` needs to know which Rack application it should talk to, and it expects an `app` method to be defined in the same object that mixes in the other `Rack-Test` methods. This method must return the class of our web app.

Now that we have `Rack-Test` properly wired up, let's see what Cucumber has to say:

```

Feature: Fruit list
  In order to make a great smoothie
  I need some fruit.

Scenario: List fruit
  Given the system knows about the following fruit:
    | name      | color |
    | banana    | yellow |
    | strawberry | red   |
  When the client requests GET /fruits
  Then the response should be JSON:
  """
  [
    {"name": "banana", "color": "yellow"},
    {"name": "strawberry", "color": "red"}
  ]

```

```
"""
TODO (Cucumber::Pending)
./features/step_definitions/rest_steps.rb:6
features/fruit_list.feature:11

1 scenario (1 pending)
3 steps (1 pending, 2 passed)
0m0.017s
```

Our second step is passing, so it looks like our GET request worked well. But wait a minute! How is that possible? We haven't implemented anything in our fruit_app.rb yet. Why is the step passing? We'll find out in the next section.

Comparing JSON

Although it might seem a little surprising that our second step is passing at this point, let's not worry about why. Instead, let's continue to make our final step pass, and everything will become clear. In this step, we want to compare the response from the GET request with our expected JSON. Rack-Test always lets us access the response of the last request by invoking the `last_response` method, which returns an object that holds various information about the response. We're interested in the body, which is where we expect our web app to write the JSON:

```
Then /^the response should be JSON:$/
  |json|
  last_response.body.should == json
end
```

Let's run Cucumber again. We're getting an error message this time, and it's a quite long one. Let's take some time to study it.

```
Feature: Fruit list
  In order to make a great smoothie
  I need some fruit.

  Scenario: List fruit
    Given the system knows about the following fruit:
      | name      | color |
      | banana    | yellow |
      | strawberry | red   |
    When the client requests GET /fruits
    Then the response should be JSON:
    """
    [
      {"name": "banana", "color": "yellow"},
      {"name": "strawberry", "color": "red"}
    ]
    """
    expected: "[\n      {\\"name\\": \\"banana\\", \\"color\\": \\"yellow\\"}, \n      {\\"name\\": \\"strawberry\\", \\"color\\": \\"red\\"}\n    ]"
  
```

```

\"name\": \"strawberry\", \"color\": \"red\"}\n]"
got: "<!DOCTYPE html>\n<html>\n<head>\n  <style type=\"text/css\">\nbody { text-align:center;font-family:helvetica,arial;font-size:22px;\n  color:#888;margin:20px}\n  #c {margin:0\n  auto;width:500px;text-align:left}\n  </style>\n</head>\n<body>\n<h2>Sinatra doesn't know this ditty.</h2>\n<img\nsrc='http://example.org/_sinatra_/404.png'>\n<div id=\"c\"\>\n  Try\nthis:\n<pre>get '/fruits' do\n    \"Hello World\"\nend</pre>\n</div>\n</body>\n</html>\n" (using ==)
Diff:
@@ -1,5 +1,21 @@
-[
-  {"name": "banana", "color": "yellow"},
-  {"name": "strawberry", "color": "red"}
-]
+<!DOCTYPE html>
+<html>
+<head>
+  <style type="text/css">
+    body { text-align:center;font-family:helvetica,arial;font-size:22px;
+      color:#888;margin:20px}
+    #c {margin:0 auto;width:500px;text-align:left}
+  </style>
+</head>
+<body>
+  <h2>Sinatra doesn't know this ditty.</h2>
+  <img src='http://example.org/_sinatra_/404.png'>
+  <div id="c">
+    Try this:
+    <pre>get '/fruits' do
+      \"Hello World"
+    end</pre>
+  </div>
+</body>
+</html>
(RSpec::Expectations::ExpectationNotMetError)
./features/step_definitions/rest_steps.rb:6
features/fruit_list.feature:11

```

Failing Scenarios:
cucumber features/fruit_list.feature:5

```

1 scenario (1 failed)
3 steps (1 failed, 2 passed)
0m0.018s

```

Cucumber is telling us that it expected to get back the JSON we specified in our feature, but instead it got some HTML back. What is going on here?

What we're seeing here is actually Sinatra's 404 Page Not Found page, which it shows for any HTTP request it doesn't know how to serve. This begins to explain why the second step was passing. Cucumber and Rack-Test were able to make a GET request, but since we haven't yet defined a route for /fruits, the response of that request was a 404 Page Not Found. We actually see this in the HTML in the error message if we look carefully.

What might surprise you is that Rack-Test didn't raise an exception since it got a 404 Page Not Found response, instead of the 200 Successful that we were expecting. In fact, no HTTP error responses in the 4xx and 5xx response code ranges will cause Ruby errors to be raised by Rack-Test. This is a deliberate design decision by the creator of Rack-Test, and it has the benefit that you can easily write tests for edge cases where an error is the expected behavior, as well as happy-path scenarios like this one.

Now that we understand what is going on, let's implement the /fruits route in our web service. Back in your fruit_app.rb, edit it so it looks like the following:

```
Download rest_web_services/05/fruit_app.rb
require 'sinatra'
require 'json'

class FruitApp < Sinatra::Base
  set :data, ''

  get '/fruits' do
    content_type :json
    FruitApp.data.to_json
  end
end
```

Running Cucumber again brings us just a few yards from the goal.

```
Feature: Fruit list
  In order to make a great smoothie
  I need some fruit.

  Scenario: List fruit
    Given the system knows about the following fruit:
      | name      | color |
      | banana    | yellow |
      | strawberry | red   |
    When the client requests GET /fruits
    Then the response should be JSON:
      """
      [
        {"name": "banana", "color": "yellow"},
        {"name": "strawberry", "color": "red"}
      ]
```

// Joe asks:

I Thought Scenarios Should Avoid Technical Terms like JSON

It's always important to make a scenario readable by a stakeholder. However, for a REST interface, the stakeholder is going to be another programmer writing a client for the REST interface. In such cases, it's fine to expose technical details. For a human user interface, such as an HTML web application, we should use nontechnical terms. There's more about this in [Chapter 15, Using Cucumber to Test Ajax Web Applications, on page 245](#).

```
"""
expected: "[\n  {"name": "banana", "color": "yellow"},\n  {"name": "strawberry", "color": "red"}]\n"
got:
[{"name": "banana", "color": "yellow"},\n  {"name": "strawberry", "color": "red"}]"
(using ==)
Diff:
@@ -1,5 +1,2 @@
-[{"name": "banana", "color": "yellow"},\n  {"name": "strawberry", "color": "red"}]
+[{"name": "banana", "color": "yellow"}, {"name": "strawberry", "color": "red"}]
(RSpec::Expectations::ExpectationNotMetError)
./features/step_definitions/rest_steps.rb:6
features/fruit_list.feature:11
```

Failing Scenarios:
cucumber features/fruit_list.feature:5

```
1 scenario (1 failed)
3 steps (1 failed, 2 passed)
0m0.018s
```

Now the scenario is failing simply because the actual JSON has no spaces and newlines, while the JSON from our scenario does. Of course, we could make the scenario pass by removing whitespace and newlines from the scenario too, but that would make it very hard to read. Remember, our features are supposed to serve as documentation too, not only automated tests. In any case, it's the data that's significant in this scenario, not the whitespace. What we really want to do is to parse the two JSON strings into an object structure and compare those. This is just a small change:

```
Then /^the response should be JSON:$/
  |json|
  JSON.parse(last_response.body).should == JSON.parse(json)
end
```

Finally, our first REST feature is passing!

```
Feature: Fruit list
  In order to make a great smoothie
  I need some fruit.
```

```
Scenario: List fruit
```

```
  Given the system knows about the following fruit:
```

name	color
banana	yellow
strawberry	red

```
  When the client requests GET /fruits
```

```
  Then the response should be JSON:
```

```
"""
[
  {"name": "banana", "color": "yellow"},
  {"name": "strawberry", "color": "red"}
]
"""
```

```
1 scenario (1 passed)
3 steps (3 passed)
0m0.018s
```

Using Rack-Test makes it easy to test a REST interface with Cucumber. If we were to write another scenario—perhaps testing how the application deals with duplicate fruit—we would have to make sure that any data stored in the previous scenario is cleaned away before the next one starts. With our simple application, we could easily do this in a `Before` hook, which we described in detail in [Section 8.4, Using Hooks, on page 147](#). Since Cucumber is running in the same process as the application, cleaning away previous data would be simple—just call `FruitApp.data = []` in a `Before` block. If our application was using a proper database, we would delete all of the rows/records that were created in the scenario instead, as described in [Chapter 10, Databases, on page 173](#).

Testing in-process is great when you have the option available to you, but there are many more web services written in other languages than Ruby that you'll still want to be able to test. In the next section, we'll decouple ourselves from the Sinatra application and run it out of process so you can see how to test a stand-alone web service.



Matt says:

Comparing Data Is Less Brittle Than Comparing Strings

The lesson we learn in this chapter about parsing both strings into JSON documents before comparing them is a useful one to remember for other situations. When you compare two strings in a test, you're often leaving yourself open to brittle failures that aren't telling you anything useful. Parsing the string from the Gherkin feature into more meaningful data can often mean you have more robust tests that fail only when they should.

12.2 Out-of-Process Testing of Any REST API

In the previous section, we tested a REST API by running Cucumber in the same Ruby process as our application. In this section, we are going to explore how to test a REST API when Cucumber is running in a different process than the application. [Figure 14, Testing an out-of-process web application with Cucumber, on page 215](#) illustrates how this fits together. Cucumber and the HTTP Client library run in one process, and the web application (the dotted boxes) runs in another. We'll be using the same application that we created in the previous section, but you could just as well use an application written in a different programming language. This is thanks to a handy little library called *HTTParty*, which can issue HTTP requests to a web server.

We'll start by changing our Gemfile. We're going to add the *HTTParty* library and another library called *ChildProcess*, which we'll use to start the web server process:

```
Download rest_web_services/07/Gemfile
source 'http://rubygems.org'

gem 'sinatra', '1.3.1'
gem 'json', '1.6.3'

group :test do
  gem 'cucumber', '1.1.3'
  gem 'rspec', '2.7.0'
  gem 'httparty', '0.8.1'
  gem 'childprocess', '0.2.3'
end
```

We've also removed Rack-Test from the Gemfile since we're no longer going to be using it to talk to the web server.

Run bundle to install the new gems.

HTTParty

Ruby's standard library ships with an HTTP client called `Net::Http`. Many developers have found this library is a little cumbersome to work with, so there is a host of HTTP client libraries with simpler APIs available as Ruby gems.

One of these libraries is `HTTParty`, which we'll use in this section. There are several other similar libraries—`RestClient`, `em-http-request`, `Curb`, `Restfulie`, and some more. They each have their strength and weaknesses, and it doesn't matter much which one you choose.

You can read more about `HTTParty` at <http://httparty.rubyforge.com>.

We never actually fired up our fruit application, so let's put in place a missing piece so we can do that.

We're going to start the application via `rackup`, which is the Rack framework's command-line tool for starting Rack-based applications. The `rackup` command-line tool will look for a `config.ru` file in the current directory. We won't go into details about Rack in this book, so we'll just provide a simple `config.ru` file so we can start the application. Put the following file in the root directory, next to your `fruit_app.rb` file:

```
Download rest_web_services/07/config.ru
require File.dirname(__FILE__) + '/fruit_app'
run FruitApp
```

Now we can start up our app. Let's run it on port 9999:

```
$ rackup --port 9999
```

If you point your web browser to <http://localhost:9999/fruits>, you should get a JSON response containing `{}`—we have no fruit in our system. We're going out of process now, so let's remove all of the Rack-Test code from the previous section. This boils down to deleting all the code in the `features/support/env.rb` file. Go ahead and clean out that file, and then let's run the feature again and see where that leaves us:

```
Feature: Fruit list
  In order to make a great smoothie
  I need some fruit.
```

```
Scenario: List fruit
  Given the system knows about the following fruit:
    | name      | color |
    | banana    | yellow |
    | strawberry| red   |
  uninitialized constant FruitApp (NameError)
```

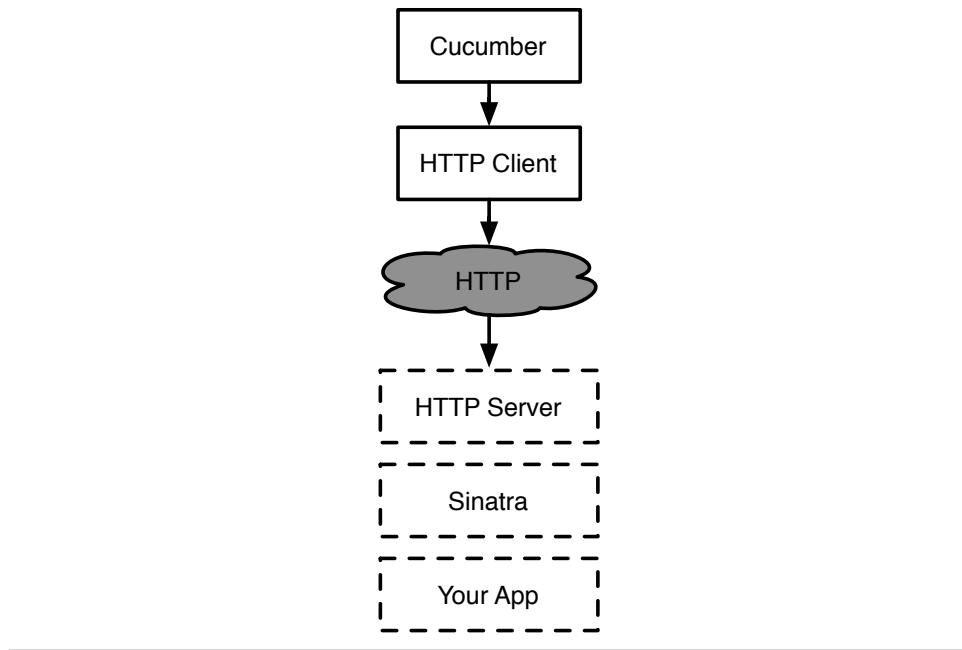


Figure 14—Testing an out-of process web application with Cucumber

```

./features/step_definitions/fruit_steps.rb:2
features/fruit_list.feature:6
When the client requests GET /fruits
Then the response should be JSON:
"""
[
  {"name": "banana", "color": "yellow"},
  {"name": "strawberry", "color": "red"}
]
"""

Failing Scenarios:
cucumber features/fruit_list.feature:5

1 scenario (1 failed)
3 steps (1 failed, 2 skipped)
0m0.004s
  
```

Ouch! It looks like we broke something here. We get a uninitialized constant Object::FruitApp error in the first step. This is because when we cleared out our features/support/env.rb file, we also removed the first two lines that load our application code. Should we add them back? It might be tempting, but this is precisely what we are *not* going to do. Remember, we are now in a situation

where Cucumber and our application are meant to run in two different processes. For all Cucumber cares, our application may be an application that isn't even written in Ruby. We did the right thing by removing everything from `features/support/env.rb`—we don't want Cucumber to load a single line of code from our web application!

This brings up an interesting question: how can we make the first step pass? Our app is using an in-memory data store (a simple class variable). We need some way to access this data store from Cucumber, which is going to be running in a different process. There are many ways to make this happen, but the essence of it is that we need to make our application's data store accessible from a different process.

One option would be to expose a new method over our HTTP API that allowed a client (like our Cucumber tests) to add new fruit to the system with an HTTP POST. POSTing to a special URL to reset a database is not an uncommon approach. Just make sure that if you do this, the URL is disabled in your production environment. Another route would be to refactor our application to use a data store such as MySQL or MongoDB and use a database library to insert data. If we did that, our step definition could put fruit in the system by talking directly to the same database as our web application.

For the sake of simplicity, let's use the second simplest data store after memory—a file. This takes only a couple of small modifications to `fruit_app.rb`. We'll read the fruit from a JSON file and validate that it's valid JSON while we're at it:

```
Download rest_web_services/08/fruit_app.rb
require 'sinatra'
require 'json'

class FruitApp < Sinatra::Base
  set :data do
    JSON.parse(File.read('fruits.json'))
  end

  get '/fruits' do
    content_type :json
    FruitApp.data.to_json
  end
end
```

The `fruits.json` file will be read every time a request is made and asks for `FruitApp.data`. Now it's easy to do the right thing in the first step by modifying `features/step_definitions/fruit_steps.rb` to write to our new file database.

[Download rest_web_services/08/features/step_definitions/fruit_steps.rb](#)

```
Given /^the system knows about the following fruit:$/
  File.open('fruits.json', 'w') do |io|
    io.write(fruits.hashes.to_json)
  end
end
```

We've used another method on Cucumber::Ast::Table, hashes, which returns the data in the table as an array of hashes: one hash per row in the table. The column headings in the first row of the table are used as keys in the hash.

Now that you have made changes to your application, you need to restart it again so that our code changes are picked up. It quickly becomes tedious to keep two shells open and switch back and forth to restart our server. Let's tell Cucumber to start and stop the server instead. Let's add some code back to our features/support/env.rb file:

[Download rest_web_services/08/features/support/env.rb](#)

```
require 'childprocess'
require 'timeout'
require 'httpparty'

server = ChildProcess.build("rackup", "--port", "9999")
server.start
Timeout.timeout(3) do
  loop do
    begin
      HTTParty.get('http://localhost:9999')
      break
    rescue Errno::ECONNREFUSED => try_again
      sleep 0.1
    end
  end
end

at_exit do
  server.stop
end
```

If you need to start a different kind of web service, just substitute the rackup command for the necessary command to start up your web server. The rest of the pattern, polling the service until it's up, should work perfectly whatever platform your web service runs on.

The code starts the server in the background and waits up to three seconds until it responds. The at_exit hook shuts it down just before Cucumber exits when the tests have been run. With this in place, let's run cucumber again:



Joe asks:

What's The Difference Between ChildProcess and ServiceManager? How Should I Decide Which One To Use?

In [Chapter 9, Dealing with Message Queues and Asynchronous Components, on page 157](#), we showed you how to use the Service Manager gem to start and stop an external process from your tests. Service Manager would also be a good fit for this situation because it ships with the functionality to poll an external service until it is up, something we've written by hand here.

ChildProcess is a more low-level library, which means it's more flexible. That flexibility comes at the cost of needing more code to achieve common tasks, like the code to start a web server in this chapter. We wanted to show you how to use both libraries, but we'd suggest Service Manager is the better choice in most situations.

Feature: Fruit list

```
In order to make a great smoothie
I need some fruit.
```

Scenario: List fruit

```
Given the system knows about the following fruit:
```

name	color
banana	yellow
strawberry	red

```
When the client requests GET /fruits
```

```
undefined method 'get' for #<Object:0x63756b65> (NoMethodError)
./features/step_definitions/rest_steps.rb:2
features/fruit_list.feature:10
```

```
Then the response should be JSON:
```

```
"""
[
```

```
  {"name": "banana", "color": "yellow"},  
  {"name": "strawberry", "color": "red"}  
]
```

```
"""

```

Failing Scenarios:

```
cucumber features/fruit_list.feature:5
```

```
1 scenario (1 failed)
```

```
3 steps (1 failed, 1 skipped, 1 passed)
```

```
0m0.005s
```

This is a bit like opening up a set of Russian dolls: each time a new step passes, we learn more about what we need to do next. Our first step is passing

again, but now the second step is failing. We no longer have a get method available to make our request since we removed our Rack-Test code.

This is the first time we'll put HTTParty—our HTTP client—to use. We need to modify the steps in `features/step_definitions/rest_steps.rb` to use HTTParty instead of Rack-Test:

```
Download rest_web_services/09/features/step_definitions/rest_steps.rb
require 'httparty'

When /^the client requests GET (.*)$/ do |path|
  @last_response = HTTParty.get('http://localhost:9999' + path)
end

Then /^the response should be JSON:$/ do |json|
  JSON.parse(@last_response.body).should == JSON.parse(json)
end
```

First, we load HTTParty with `require 'httparty'`. Then, we call `HTTParty.get` to issue an HTTP request. Here we have to pass a full URL, which is why we tack on '`http://localhost:9999`' before the path passed to the step definition. We store the response in a `@last_response` instance variable, because HTTParty's API is a little different from Rack-Test. Finally, in the last step definition we refer to that variable to get the body from the HTTP request. Let's run the feature again:

```
Feature: Fruit list
  In order to make a great smoothie
  I need some fruit.

  Scenario: List fruit
    Given the system knows about the following fruit:
      | name      | color |
      | banana   | yellow |
      | strawberry | red   |
    When the client requests GET /fruits
    Then the response should be JSON:
      """
      [
        {"name": "banana", "color": "yellow"},
        {"name": "strawberry", "color": "red"}
      ]
      """

1 scenario (1 passed)
3 steps (3 passed)
0m0.065s
```

Great! If you're following along, you'll probably notice that the scenario feels slower to run than the equivalent in-process test. This is the added overhead

of waiting for the web server to start up before the scenario starts. Remember, you pay this penalty only once per test run. Running Cucumber *out-of-process* with HTTParty is just as successful as our *in-process* approach that used Rack-Test. We didn't change a single line of our feature files—all the changes are nicely tucked away in our step definitions. The main difference was that this time we executed real HTTP requests, and the tests took a little bit longer to fire up.

12.3 What We Just Learned

When testing a REST web service with Cucumber, it's generally preferable to run Cucumber *in-process*—in the same process as your application. In this chapter, we covered the two main approaches—*in-process* and *out-of-process*. Although our example used the Sinatra web framework, the techniques used in this chapter can be applied to other web frameworks too.

- Rack-based applications such as Ruby on Rails and Sinatra can be tested *in-process* using Rack-Test, completely bypassing HTTP, making them fast.
- Non-Rack-based applications run in a separate process from Cucumber and have to be started before Cucumber runs.
- You can use indented JSON documents in your scenarios, as long as you ignore whitespace when you compare them to documents produced by your API.

Try This

Our Fruit application is very simple. It doesn't provide an API to store fruit, and the file-based database is very simplistic. Here are some new features you can try to express with Cucumber and then implement:

- Add a scenario that starts with one fruit in the database, adds another one via an HTTP POST, and then verifies that you now have two fruit.
- Refactor your code to use a proper database, like MySQL or MongoDB, to store fruit. Make sure all scenarios are still passing and that you can run each of them individually. Hint: use DatabaseCleaner to make sure each scenario starts with a blank slate.

Adding Tests to a Legacy Application

In the real world, you don't always get the luxury of working on shiny new code. Reading a book like this can be a little bit frustrating because the examples all deal with nice, simple problems, usually in a new codebase.

We all know that software development isn't really like that.

Even in the best teams we've worked on, there have always been a few dark and ugly corners of the codebase where nobody really wanted to go. People would sometimes disappear into there for days at a time and emerge exhausted and confused, blinking in the bright sunlight.

Like an old abandoned mine, those areas of the codebase are dangerous. The code is fragile, and the slightest change can cause other parts of the code to collapse and stop working. Everyone knows this, and that's why people are reluctant to go in there: it's stressful work.

If you had to go down an old mine tunnel, what would you do to make it safer? You'd probably build some scaffolding to hold the roof of the tunnel up so that it didn't collapse—just enough to help you get in and get out again safely.

You can think of automated tests like this. When you have to make a change to an area of the code that you know is brittle, it's scary. What is the thing that you're most afraid of?

Breaking something.

Automated tests can help you keep this fear at bay. If you make a change and other parts of the code start to collapse, the tests will give you a warning while you still have a chance to do something to correct it. But what if you don't have any tests?

Even if you have recognized that automated testing can help your team write better code, the prospect of adding tests to a large legacy codebase can seem overwhelming. In this chapter, we'll show you some simple techniques you can use to help to solve the problem gradually, giving you added benefit each step of the way.

This isn't a technical recipe, but it will give you some useful techniques and encouragement if you're facing this situation.

Let's start with the first tool you'll need when you enter the mine: a flashlight.

13.1 Characterization Tests

In his excellent book *Working Effectively with Legacy Code* [Fea04], Michael Feathers talks about two different types of tests, which he calls *specification tests* and *characterization tests*.

Specification tests are the ones we've been talking about in the rest of this book. They check that the code does what it's supposed to. Ideally you write them before you write the code itself and use them as a guide to help you get the code into the right shape.

Characterization tests are different. You can think of them more like a science experiment, where you test the properties of a mysterious substance by boiling it or mixing it with other substances to see how it reacts. With characterization tests, the aim is just to understand what the system currently does.

Characterizations tests apply perfectly to legacy code that has limited or no tests and where the code is hard to read and understand. Here is our recipe, adapted from Michael Feathers', for creating a Cucumber characterization test:

1. Write a scenario that exercises some interesting—but mysterious—behavior of your system.
2. Write a Then step that you know will fail.
3. Wire up the step definitions and run the scenario. Let the failure in the Then step tell you what the actual behavior is.
4. Change the failing Then step so that it describes the actual behavior of the system.
5. Repeat.

As an example, let's suppose we're making some changes to the checkout system for a supermarket. We've been asked to add a new special offer to the

system, which was developed a couple of years ago by a firm of expensive management consultants, who are sadly no longer with us. We've grabbed the source code, managed to get the system to spin up on our development machine, and have been poking around through the user interface. The code is pretty gnarly and it's hard to tell exactly what's going on, but we noticed something about shampoo. It looks like there might be an existing special offer on shampoo bottles already baked into the code, but it's hard to tell exactly what the rules are. Let's write a test:

```
Scenario: Buy Two Bottles of Shampoo
  Given the price of a bottle of shampoo is $1.99
  When I scan 2 bottles of shampoo
  Then the price should be $0
```

We know that's going to fail: they're not going to give us the shampoo for free, are they? (Steps 1 and 2.)

So, we wire up the step definitions to the system and run the scenario. Here's what happens:

Feature:

```
Scenario: Buy Two Bottles of Shampoo
  Given the price of a bottle of shampoo is $1.99
  When I scan 2 bottles of shampoo
  Then the price should be $0
    expected: 0
      got: 1.99 (using ==) (RSpec::Expectations::ExpectationNotMetError)
    ./features/step_definitions/steps.rb:8
  features/bogof.feature:5
```

Failing Scenarios:
cucumber features/bogof.feature:2

```
1 scenario (1 failed)
3 steps (1 failed, 2 passed)
0m0.003s
```

A-ha! Now we know what price the system is charging for two \$1.99 bottles of shampoo: \$1.99. (Step 3.)

Now we update the Then step of our scenario to reflect our new understanding of what the system does (step 4):

```
Scenario: Buy Two Bottles of Shampoo
  Given the price of a bottle of shampoo is $1.99
  When I scan 2 bottles of shampoo
  Then the price should be $1.99
```

We run the scenario again, and this time it passes. Great! We've added our first characterization test. It's not much, but it's a start, and we know that whatever we do to the code from now on, we'll always have this scenario to tell us whether we break this particular aspect of its behavior.

It looks like there's a buy-one-get-one-free offer on shampoo, but we can't be sure from this single example. Following the recipe, we now need to repeat steps 1 to 4 and add some more scenarios to give us some more clues as to what the system is doing.

We can refactor the scenario into a scenario outline (see [Chapter 5, Expressive Scenarios, on page 61](#)) to allow us to try different amounts and prices:

Feature: Special Offers

Scenario Outline: Shampoo

Given the price of a bottle of shampoo is \$1.99

When I scan <number> bottles of shampoo

Then the price should be <total>

Examples:

number	total
1	\$0
2	\$1.99

We can now add examples into the table, one at a time. Each time we start with a silly value for the total price and then run the scenario and let it tell us what the real total is. Then we update the scenario to document that behavior.

13.2 Squashing Bugs

Our legacy application is big and mysterious, and we could spend an awfully long time writing characterization tests if we just started adding them aimlessly. So, assuming that we do want to grow our suite of automated tests, where should we start?

One of the best ways to start practicing with Cucumber is when you have a bug to fix. Bug reports generally come to you in the form of an example, so they're nice and easy to translate into Cucumber scenarios.

1. Translate the bug report into a Cucumber scenario. Show the scenario to the person who reported the bug, and ask them whether it accurately describes what they were doing.

2. Wire up the step definitions and run the scenario. It should fail in the same way as the real system did when the bug was first discovered. The bug is trapped!
3. Examine the defective code, and think about what you'll need to change. If you're unsure or worried about breaking some existing behavior, write a characterization test scenario for it.
4. Fix the code so that the bug report scenario passes.
5. Run the characterization tests to check you didn't break anything.

You'll find that the bug report scenario acts as a driving force, helping you focus on the code instead of having to keep running a manual test to see whether the fix has worked.

Despite your new characterization tests, you may still find that you missed something and introduced a new bug. That happens sometimes, and it would have happened just the same if you hadn't used any automated tests. If you use the same recipe to fix that new bug and each new bug that comes along, then gradually, over time, you'll build up a solid suite of Cucumber scenarios. Not only will those scenarios prevent any of these bugs from recurring, but they'll start to document the behavior of the system for anyone doing maintenance on it in the future.

13.3 Adding New Behavior

When you think about it, the process of adding new behavior to a system isn't so different from fixing a bug. The overall goal is to change some aspect of how the system behaves, without breaking anything else.

Just as with bug fixing, we can use characterization tests to pin down the surrounding behavior of the system to make sure it isn't dislodged by our work. Characterization tests can also be useful before that, while the new feature is first being considered. You can use them to help understand how much work is involved in the new feature by clarifying exactly what the system currently does.

Here's our recipe for adding new behavior to a legacy system:

1. Examine the new feature. If necessary, write a few characterization scenarios to examine and clarify the current behavior of the system in that area.
2. Now, with the new feature in mind, modify those scenarios or write new ones to specify the desired new behavior.



Matt says: Features for Bug Fixes

We've just told you to use Cucumber to help you reproduce and trap bugs, so it's quite likely that you might end up writing a new feature and calling it something like `features/verify_bugfix_52553.feature`. We've done this ourselves, and trust us—it doesn't make for great documentation!

There are two ways around this problem. If you judge that the scenario is relevant enough to keep as business-facing documentation, then just talk to your team about where to file it away tidily in your features. If you're adding features to a legacy system, that might will mean creating a new empty feature file about a whole big area of the system and then just adding a single scenario to describe one aspect of its behavior. That's OK; you've created a space where other people can add more scenarios as they come up. Try not to mention the bug itself as you write the scenario—just describe the behavior you want as though it's always been there.

On the other hand, if you decide that it's such an obscure edge case that it isn't interesting enough to remain as business-facing documentation, you can just delete the scenario once you've fixed the bug. That's assuming you've used a unit test to cover the changes you've made to make the fix, so you can be safe in the knowledge that the bug won't reappear.

3. Run through the scenarios with your team's stakeholder representative to check that you're about to build the right thing. Correct the scenarios with them if necessary.
4. Run the scenarios. If more than one fails, pick one, and examine the code you'll need to change to make it pass.
5. Write any extra characterization tests you need to give you the confidence to change that code.
6. Change the code to make the scenario pass.
7. Repeat from step 4 until all the scenarios pass.

Sometimes we find that when we're about to implement the change (step 6), we see that the code we're about to change is responsible for doing things that we hadn't anticipated when we wrote the original characterization tests in step 1. At this point, we'll stop and add some new characterization tests (step 5) first if we think there's a significant risk we could break something.

Just as with bug fixing, following this process means that you quickly get the benefits of automated testing in the areas of the system where you most need them: the ones that are most prone to change and instability.

As you gradually build up your suite of Cucumber scenarios for your legacy application, you'll find you have more confidence to refactor and clean up the code. This becomes a virtuous cycle, with cleaner code causing less bugs.

13.4 Code Coverage

Code coverage tools allow us to discover which specific lines of code in the system were executed during a test. When you are starting to add tests to a legacy application, it can be really useful to know your code coverage:

- If you know that a line of code isn't covered by your tests yet, you can see more clearly what kind of a characterization test you need to write.
- If you know that a line of code is covered by your tests, you can be more confident in refactoring or changing it.

Provided you're testing a Ruby application, recording your code coverage from Cucumber tests is pretty easy. For Ruby 1.8 applications, you can use Cucumber's built-in Rake task:

```
Download legacy_applications/Rakefile
require 'cucumber/rake/task'
```

```
Cucumber::Rake::Task.new do |t|
  t.rcov = true
end
```

If you now run `rake -T` in your project's root directory, you'll see a new Rake task that you can run with `rake cucumber`. When you run it, a new folder is created called `coverage` that contains a pretty HTML report showing exactly which lines of code were executed during the test.

At the time of writing, there was no built-in support in Cucumber for collecting code coverage statistics under Ruby 1.9. Instead, you need to install the `SimpleCov`¹ gem and add a couple of lines to your `env.rb`:

```
require 'simplecov'
SimpleCov.start
```

When your features finish running, you should find a code coverage report in `coverage/index.html`.

13.5 What We Just Learned

Working with legacy code is always a challenge, but you can use Cucumber tests to help make it a much more enjoyable challenge.

1. <http://rubygems.org/gems/simplecov>

Track Your Code Coverage for Team Encouragement

Code coverage is a much maligned metric of code quality. We've heard of managers who've demanded that their teams achieve 100 percent code coverage. We've also heard of teams that, under those circumstances, simply wrote a load of tests with no assertions in them. The tests were exercising all the code, but they weren't testing anything!

As a source of internal feedback for the team, though, code coverage can be useful, especially when you're embarking on a long-term project to bring a legacy system under the control of automated tests. Start measuring your code coverage today, and come back every few weeks to measure it again. You should be pleased to see the number is climbing all the time.

Another great related metric to track, which your manager might be more impressed with, is the defect rate (number of new bugs discovered per week). If your Cucumber tests are going to add any value to your customers, this is the most likely place it will surface. As you add more test coverage, your team will make fewer and fewer mistakes, and you should see your defect rate drop.

- Be pragmatic! Don't exhaust yourself trying to retrofit a complete set of Cucumber features for everything the system already does. Instead, add them gradually, one at a time, as you need them.
- Use characterization tests to help you understand what the system is already doing and to give you some security before you make a change.
- Every time you discover a bug, trap it with a new Cucumber scenario. Every time you add new behavior to the system, start by describing it with a Cucumber feature.
- Use code coverage to give you and your team feedback and encouragement about your progress in getting the system under test.
- Enjoy the newfound confidence with which you can refactor and clean up the code that's covered by the tests.

Bootstrapping Rails

Cucumber was born in the Ruby on Rails community, and there are some well-worn patterns for setting up a Rails application to work with Cucumber. This chapter walks you through those steps, from creating a new Rails application right to making your first scenario pass. If you're about to embark on a Rails project, this chapter was written for you. Even if you don't work with Rails in your day job, you'll learn how to work with libraries like FactoryGirl and ActiveRecord that you can put to wider use, even on non-Ruby projects.

Picture the scene.

It's Monday morning. The coffee is still steaming on your desk, too hot to drink. Your boss bursts into the room.

I've got an amazing idea! We're going to build a blogging platform, a micro-blogging platform. It's like blogging, right, but your messages are limited to 140 characters. So, you could send them from a phone even! This is going to change the world! We're going to call it...Squeaker.

You look up, skeptically. He often does this on Mondays—he should get out more, you think to yourself. This does seem like one of his better ideas, though.

So, you open up your text editor and start to sketch out the first Cucumber feature:

Download rails/01/features/see_messages.feature

Feature: See Messages

Scenario: See another user's messages

Given there is a User

And the User has posted the message "*this is my message*"

When I visit the page for the User

Then I should see "*this is my message*"

Yeah, that's it! says your boss. That's definitely the first thing we should focus on. Can you make it do that?

14.1 Running the Generators

Rails uses *generators* to help you set up your application with typical boilerplate code that you can then customize to your needs. First we're going to generate the Rails application, and then we'll add the cucumber-rails plug-in and use its generator to configure your application so it's ready for you to start driving it with Cucumber.

We start by creating our Rails application. We're going to use the rails new command with a single option:

```
$ gem install rails --version 3.1.3
$ rails new squeaker --skip-test-unit
```

The option doesn't matter too much—those are parts of Rails that we're not going to use, so they'd just be cluttering up our code if we left them in.

Now we need to edit our new Rails application's Gemfile to include the cucumber-rails gem. We're also including the rspec-rails gem, which we'll use to make assertions later, and finally we're adding database_cleaner, a gem that will make sure our database is cleaned up between each scenario:

```
Download rails/01/Gemfile
source 'http://rubygems.org'

gem 'rails', '3.1.3'

# Bundle edge Rails instead:
# gem 'rails',      :git => 'git://github.com/rails/rails.git'
gem 'sqlite3'

# Gems used only for assets and not required
# in production environments by default.
group :assets do
  gem 'sass-rails',    '~> 3.1.5'
  gem 'coffee-rails',  '~> 3.1.1'
  gem 'uglifier',     '>= 1.0.3'
end

gem 'jquery-rails'

# To use ActiveModel has_secure_password
# gem 'bcrypt-ruby',  '~> 3.0.0'

# Use unicorn as the web server
# gem 'unicorn'
```

```
# Deploy with Capistrano
# gem 'capistrano'

# To use debugger
# gem 'ruby-debug19', :require => 'ruby-debug'

group :test do
  gem 'cucumber-rails', '1.2.1'
  gem 'rspec-rails', '2.7.0'
  gem 'database_cleaner', '0.7.0'
end
```

We've tucked the Cucumber gems away in a test group so they won't be installed when we deploy to production.

With that done, we can move our working directory into the new application's folder and run the Cucumber generator:

```
$ cd squeaker
$ rails generate cucumber:install
```

Let's take a quick look at the files that the generator has given us:

config/cucumber.yml

Contains some profiles that you can use to run Cucumber with different sets of options.

features/support/env.rb

This is the script Cucumber runs first when the tests start. It loads the Rails environment and requires the various Cucumber and Capybara libraries needed to test your Rails application.

lib/tasks/cucumber.rake

Gives your application a cucumber Rake task. You can edit this file to change the default options for the Rake task or create new ones.

Now that we have a Rails application with Cucumber set up, let's put the feature into features/see_messages.feature:

[Download rails/01/features/see_messages.feature](#)

Feature: See Messages

Scenario: See another user's messages

Given there is a User

And the User has posted the message "*this is my message*"

When I visit the page for the User

Then I should see "*this is my message*"

Let's see if we can put just a little flesh on the bones.

14.2 Creating a User

To get our first step to pass, we need to be able to create a user in our system. We could do that by walking around the UI, filling in the user registration forms and clicking buttons, but actually that's not the point of this feature. Our goal is to build the page where you can see a user's messages so we don't need to worry about all that user registration stuff.

Let's imagine we have a fast-forward button, and we skip past all the user registration to the point where the user has registered. What state is the system in now?

Well, we don't know for sure because we haven't built those screens yet, but it seems safe to assume that there would be a new row in the users table in our database. To get the system into that state, we can use the Rails models to add a User record to the system, just as though they had registered. A great library to help us work with ActiveRecord, the library Rails uses to talk to the database, is called FactoryGirl.¹ Let's add her to the test group in our Gemfile:

[Download rails/02/Gemfile](#)

```
group :test do
  gem 'cucumber-rails', '1.2.1'
  gem 'rspec-rails', '2.7.0'
  gem 'database_cleaner', '0.7.0'
  gem 'factory_girl', '2.3.2'
end
```

Now we can implement the step definition for our first step. We'll create it in a file called `step_definitions/user_steps.rb`:

[Download rails/02/features/step_definitions/user_steps.rb](#)

```
Given /^there is a User$/ do
  Factory(:user)
end
```

We're calling FactoryGirl's special helper method `Factory`, telling it to create us a standard instance of the User model. If you run the feature now, you should see the following error:

Feature: See Messages

```
Scenario: See another user's messages
  Given there is a User
    Factory not registered: user (ArgumentError)
      ./features/step_definitions/user_steps.rb:2
      features/see_messages.feature:3
```

1. http://rubygems.org/gems/factory_girl

```

And the User has posted the message "this is my message"
  Undefined step: "the User has posted the message "this is my message"""
    (Cucumber:::Undefined)
      features/see_messages.feature:4
When I visit the page for the User
  Undefined step: "I visit the page for the User" (Cucumber:::Undefined)
    features/see_messages.feature:5
Then I should see "this is my message"
  Undefined step: "I should see "this is my message"" (Cucumber:::Undefined)
    features/see_messages.feature:6

Failing Scenarios:
cucumber features/see_messages.feature:2

1 scenario (1 failed)
4 steps (1 failed, 3 undefined)
0m0.811s

```

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

```

Given /^the User has posted the message "([^"]*)"$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end

```

```

When /^I visit the page for the User$/ do
  pending # express the regexp above with the code you wish you had
end

```

```

Then /^I should see "([^"]*)"$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end

```

Why? Well, we need to tell FactoryGirl how to create a user. Create a file called `features/support/factories.rb`, and give it some very basic sample data:

```

Download rails/03/features/support/factories.rb
require 'factory_girl'

FactoryGirl.define do
  factory :user do |f|
    f.username 'testuser'
  end
end

```

We haven't given much thought to the attributes for our user yet. That's OK, because Rails will let us change the columns in the `users` table easily using migrations, and our Cucumber features will always tell us whether things are still working. Right now we can concentrate on doing something simple and getting it to work.



Joe asks:

Why Do I Need FactoryGirl When I Can Just Create Records with ActiveRecord Directly?

It's really easy to create rows in your database just using ActiveRecord directly. For our User class, we could use this code:

```
User.create! :username => 'testuser'
```

The problem with doing this is that as the shape of your User evolves, more attributes are added, and validation rules are implemented, the hash of data that you need to pass will get longer and longer. Pretty soon it could easily look like this:

```
User.create! :username => 'testuser',
            :password => 'password',
            :password_confirmation => 'password',
            :email => 'test@test.com',
            :first_name => 'Test',
            :last_name => 'User'
```

Now you have a bunch of problems. First, you have a duplication problem: each time you add a new mandatory field to User, you have to hunt for every place where you've called User.create! and add the new field to the hash, along with a sample value. Second, you have an incidental detail problem. The precise attributes of the user are totally irrelevant to our See messages scenario, yet there's all this noise in the step definition where we need to create a user. That's distracting for anyone who has to come along and modify this code later, especially in another scenario where one of those attributes is actually relevant to what's being tested.

One solution, which is still baked into Rails at the time of writing, is to use fixtures. We explain the problems with using fixtures in [Chapter 6, When Cucumbers Go Bad, on page 85](#).

FactoryGirl is an implementation of the Test Data Builder pattern as first described by Nat Pryce.^a It allows you to configure what a standard instance of any ActiveRecord database object should look like in a central place so that your test code is clutter-free. Remember that you can use ActiveRecord and FactoryGirl to set up data in any SQL database, whether or not the application you're testing is written in Ruby.

a. <http://www.natpryce.com/articles/000714.html>

If we run the features again, we should have made some progress:

Feature: See Messages

```
Scenario: See another user's messages
Given there is a User
  uninitialized constant User (NameError)
  ./features/step_definitions/user_steps.rb:2
features/see_messages.feature:3
```

```

And the User has posted the message "this is my message"
  Undefined step: "the User has posted the message "this is my message"""
    (Cucumber::Undefined)
  features/see_messages.feature:4
When I visit the page for the User
  Undefined step: "I visit the page for the User" (Cucumber::Undefined)
  features/see_messages.feature:5
Then I should see "this is my message"
  Undefined step: "I should see "this is my message"" (Cucumber::Undefined)
  features/see_messages.feature:6

```

Failing Scenarios:

```
cucumber features/see_messages.feature:2
```

```

1 scenario (1 failed)
4 steps (1 failed, 3 undefined)
0m0.816s

```

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

```
Given /^the User has posted the message "([^\"]*)"$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end
```

```
When /^I visit the page for the User$/ do
  pending # express the regexp above with the code you wish you had
end
```

```
Then /^I should see "([^\"]*)"$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end
```

Ah yes, a new error. So, working our way from the outside-in, we've hit the point where we need to actually create the User model. Don't worry about the undefined steps for now—we'll get to those later. We know that the user needs to have a username, so let's use the Rails model generator to create our very simple User model:

```
$ rails generate model User username:string
```

If we run the features now, we'll see that—hooray!—FactoryGirl can now find the User model, but there's no users table in our test database. Let's fix that:

```
$ rake db:migrate db:test:prepare
```

Run the features now, and you should see that the first step has passed.

14.3 Posting a Message

To implement our second step, we're going to cheat again and simulate the posting of a message by calling the models directly. This allows us to focus on building the messages page without getting too distracted. Let's implement the second step definition:

[Download rails/05/features/step_definitions/user_steps.rb](#)

```
Given /^the User has posted the message "([^"]*)"$/ do |message_text|
  User.count.should == 1
  Factory(:message, :content => message_text, :user => User.first)
end
```

Notice that we've made an assertion that there should be only one user in the system. We know that in the feature we're writing right now there's only one user, but it's possible that this step definition could be reused in the future by another scenario involving more than one and behave in surprising ways. Since the step definition talks about *the user*, it's implying that there's only one user in the system. So, we use the assertion to make sure that's true. Now when we run the features, of course poor old FactoryGirl complains that she doesn't know what a message should look like:

Feature: See Messages

```
Scenario: See another user's messages
  Given there is a User
  And the User has posted the message "this is my message"
    Factory not registered: message (ArgumentError)
    ./features/step_definitions/user_steps.rb:8
    features/see_messages.feature:4
  When I visit the page for the User
    Undefined step: "I visit the page for the User" (Cucumber::Undefined)
    features/see_messages.feature:5
  Then I should see "this is my message"
    Undefined step: "I should see \"this is my message\"" (Cucumber::Undefined)
    features/see_messages.feature:6
```

Failing Scenarios:

cucumber features/see_messages.feature:2

```
1 scenario (1 failed)
4 steps (1 failed, 2 undefined, 1 passed)
0m0.842s
```

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

```
When /^I visit the page for the User$/ do
  pending # express the regexp above with the code you wish you had
end
```



Joe asks:

When Do I Use the UI and When Do I Use Models Directly to Create Records?

A Given means that something has happened in the past. If a Given describes stuff that should be in the database, it is usually preferable to create those records directly using the database (ActiveRecord)—without going through the UI. This gives us the freedom to implement certain functionality without depending on other UI screens to be developed first. It also runs faster and makes our step definitions simpler. The *RSpec Book* [CADH09] calls this pattern *DMA*—Direct Model Access.

```
Then /^I should see "([^\"]*)"$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end
```

Let's fix that, again by creating a very simple factory definition in `features/support/factories.rb`:

```
Download rails/06/features/support/factories.rb
factory :message do |f|
  f.association :user
  f.content 'Test message content'
end
```

Again, when we run Cucumber, we're told we need to create a Message model:

Feature: See Messages

```
Scenario: See another user's messages
  Given there is a User
  And the User has posted the message "this is my message"
    uninitialized constant Message (NameError)
    ./features/step_definitions/user_steps.rb:8
    features/see_messages.feature:4
  When I visit the page for the User
    Undefined step: "I visit the page for the User" (Cucumber::Undefined)
    features/see_messages.feature:5
  Then I should see "this is my message"
    Undefined step: "I should see \"this is my message\"" (Cucumber::Undefined)
    features/see_messages.feature:6
```

Failing Scenarios:

```
cucumber features/see_messages.feature:2
```

```
1 scenario (1 failed)
4 steps (1 failed, 2 undefined, 1 passed)
0m0.891s
```

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

```
When /^I visit the page for the User$/ do
  pending # express the regexp above with the code you wish you had
end
```

```
Then /^I should see "([^"]*)"$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end
```

We'll fix that once more by generating our very basic Message model and then running the database migrations:

```
$ rails g model Message user_id:integer content:string
$ rake db:migrate db:test:prepare
```

That got us past the previous error, but now we have uncovered another missing piece:

Feature: See Messages

Scenario: See another user's messages

```
Given there is a User
And the User has posted the message "this is my message"
undefined method 'user=' for #<Message:0x63756b65> (NoMethodError)
./features/step_definitions/user_steps.rb:8
features/see_messages.feature:4
```

When I visit the page for the User

```
Undefined step: "I visit the page for the User" (Cucumber::Undefined)
features/see_messages.feature:5
```

Then I should see "this is my message"

```
Undefined step: "I should see "this is my message"" (Cucumber::Undefined)
features/see_messages.feature:6
```

Failing Scenarios:

```
cucumber features/see_messages.feature:2
```

```
1 scenario (1 failed)
4 steps (1 failed, 2 undefined, 1 passed)
0m0.957s
```

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

```
When /^I visit the page for the User$/ do
  pending # express the regexp above with the code you wish you had
end
```

```
Then /^I should see "([^"]*)"$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end
```

Each new error means we've made a little bit of progress, but we want to get to passing! Our step is now failing where we've asked FactoryGirl to create the message and given her a user to associate with the message. She hasn't been able to do that because we haven't told Rails yet that the two objects have a relationship.

14.4 Associating a Message with a User

We haven't defined our user association on the Message model yet. Let's do that:

```
Download rails/08/app/models/message.rb
class Message < ActiveRecord::Base
  belongs_to :user
end
```

Cucumber now outputs this:

Feature: See Messages

```
Scenario: See another user's messages
  Given there is a User
  And the User has posted the message "this is my message"
  When I visit the page for the User
    Undefined step: "I visit the page for the User" (Cucumber::Undefined)
    features/see_messages.feature:5
  Then I should see "this is my message"
    Undefined step: "I should see \"this is my message\"" (Cucumber::Undefined)
    features/see_messages.feature:6

1 scenario (1 undefined)
4 steps (2 undefined, 2 passed)
0m0.908s
```

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

```
When /^I visit the page for the User$/ do
  pending # express the regexp above with the code you wish you had
end
```

```
Then /^I should see "([^"]*)"$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end
```

Great! Our two Given steps are passing. Notice how nicely our Cucumber steps have driven us to create just the right code in the models. Now we have an undefined step that we need to implement. This will be the first time we actually interact with the web interface of our application. We will do this with Capybara, which is installed as part of Cucumber-Rails. Capybara will be

fully explained in [Chapter 15, Using Capybara to Test Ajax Web Applications, on page 245](#). For now, let's just use the `visit(path)` method that Capybara gives us. Add another step definition to `features/step_definitions/user_steps.rb`:

[Download rails/09/features/step_definitions/user_steps.rb](#)

```
When /^I visit the page for the User$/ do
  User.count.should == 1
  visit(user_path(User.first))
end
```

Here we are using the Rails route `user_path`, which takes a `User` instance. We're just passing `User.first` since we know there is only one user in our database. Running Cucumber again now tells us that Rails doesn't know about `user_path`:

Feature: See Messages

```
Scenario: See another user's messages
  Given there is a User
  And the User has posted the message "this is my message"
  When I visit the page for the User
    undefined method 'user_path' for
      #<Cucumber::Rails::World:0x63756b65> (NoMethodError)
      ./features/step_definitions/user_steps.rb:15
      features/see_messages.feature:5
  Then I should see "this is my message"
    Undefined step: "I should see \"this is my message\""
      (Cucumber::Undefined)
      features/see_messages.feature:6
```

Failing Scenarios:

`cucumber features/see_messages.feature:2`

```
1 scenario (1 failed)
4 steps (1 failed, 1 undefined, 2 passed)
0m0.890s
```

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

```
Then /^I should see "([^\"]*)"$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end
```

That's not surprising—we haven't created a `UsersController` yet, and we haven't mapped it in our Rails application's routes.

14.5 Creating a Controller by Hand

It may be tempting at this point to run a Rails generator to generate the controller, views, helpers, tests, and who knows what other files. We'll resist this temptation and implement the bare minimum by hand. This way, we

don't end up with a lot of code that isn't tested and that we might not even need.

Let's map our controller to config/routes.rb:

```
Download rails/10/config/routes.rb
Squeaker::Application.routes.draw do
  resources :users
end
```

Add app/controllers/users_controller.rb:

```
Download rails/10/app/controllers/users_controller.rb
class UsersController < ApplicationController
  def show
  end
end
```

And a view in app/views/users/show.html.erb with some temporary content:

```
Download rails/10/app/views/users/show.html.erb
Hello
```

Admittedly, we did take a couple of shortcuts here. A fundamentalist BDD practitioner would have run Cucumber between each little code addition, getting guidance at every little step on the way. We recommend doing this in the beginning until you find your own comfort level regarding how much code is safe to add, remove, or modify between each run. This is a trade-off between development speed and confidence.

Many people set up their environment so that Cucumber runs in the background every time you save a file. Explaining how do this is beyond the scope of this book, but we recommend you check out Guard³ or Watchr.⁴ They are both easy to set up and fun to use!

With the route, controller, and view in place, let's run Cucumber once more to see where we're at:

Feature: See Messages

```
Scenario: See another user's messages
  Given there is a User
  And the User has posted the message "this is my message"
  When I visit the page for the User
  Then I should see "this is my message"
    Undefined step: "I should see \"this is my message\"" (Cucumber::Undefined)
    features/see_messages.feature:6
```

-
- 3. <https://github.com/guard/guard>
 - 4. <https://github.com/mynymyl/watchr>

```
1 scenario (1 undefined)
4 steps (1 undefined, 3 passed)
0m0.999s
```

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

```
Then /^I should see "([^"]*)"$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end
```

Whoa! We successfully made a request to our user page. The last step is still undefined, so we need to implement that as well.

14.6 Implementing the View

Let's make the last few changes to make our scenario pass. The next thing we need to do is to implement the last missing step. Add the following to `features/step_definitions/view_steps.rb`:

[Download rails/11/features/step_definitions/view_steps.rb](#)

```
Then /^I should see "([^"]*)"$/ do |text|
  page.should have_content(text)
end
```

We're using Capybara's `page` object and `have_content` RSpec matcher here to verify that a snippet of text is on the page. We'll explain more about that in the next chapter. This makes our step go from undefined to failing since the view doesn't show what we want it to show. Remember—we just hard-coded a message in it.

We'll work outside-in, starting with the `app/views/users/show.html.erb` view. Let's assume the controller will assign an `@user` variable:

[Download rails/11/app/views/users/show.html.erb](#)

```
<% @user.messages.each do |message| %>
<p><%= message.content %></p>
<% end %>
```

If you want, run Cucumber again. That will tell us that, no, there isn't a `@user` variable. Let's fix our controller so that it loads the `@user`:

[Download rails/11/app/controllers/users_controller.rb](#)

```
class UsersController < ApplicationController
  def show
    @user = User.find(params[:id])
  end
end
```

This isn't enough to make the scenario pass quite yet. Now we get a new error because we haven't added the other side to the one-to-many relationship between User and Message. Adding that is simple, since we already created our Message class with a `user_id` field:

```
Download rails/11/app/models/user.rb
class User < ActiveRecord::Base
  has_many :messages
end
```

And finally—our feature is passing:

Feature: See Messages

```
Scenario: See another user's messages
  Given there is a User
  And the User has posted the message "this is my message"
  When I visit the page for the User
  Then I should see "this is my message"
```

```
1 scenario (1 passed)
4 steps (4 passed)
0m1.046s
```

Excellent. We haven't actually seen the code in action in a browser yet, but we've done all the hard work. And what's more, we know that whatever we do to the application in the future, this scenario will always guarantee that this core functionality is working. It's time to call the boss over for a demo.

14.7 What We Just Learned

Cucumber-Rails integrates Cucumber with Rails.

- We installed Cucumber-Rails in our Rails project by running the `cucumber:install` generator. This created several files that wire up Cucumber to run scenarios against our Rails application.
- We wrote a feature about displaying data without creating any user interface to enter it. Being able to prioritize features this way makes us flexible—we are not dictated by a strict implementation order.
- We saw how we can use FactoryGirl to populate our database in `Given` steps.
- We saw how we can access our web interface using Capybara's `visit` method.

Now that you have learned how to set up Cucumber in a Rails application, it's time to learn more about the full workflow. In the next chapter, you'll learn more as we add Ajax behavior to Squeaker.

14.8 Try this

We never fired up our browser to see whether our Squeaker application actually works. Try to insert some sample users and messages in your development database using the Rails console. Or better yet, implement a feature for entering messages through the user interface, driven by Cucumber.

CHAPTER 15

Using Capybara to Test Ajax Web Applications

One of the most popular ways to use Cucumber is to automate the testing of web applications. Here is a dirty little secret:

Cucumber has no idea how to talk to a web application!

That's right—it's completely useless for that. Still, people keep using it to test web applications. How come?

Remember, Cucumber isn't much more than a tool that can parse Gherkin feature files and execute step definitions. It doesn't know how to talk to databases, web apps, or any external system. People install other libraries for this and use them in their step definitions and support code to connect to those external systems.

At the time of writing, the most popular Ruby library for interacting with a web application is Capybara.¹ It provides an API for accessing web pages and interacting with them in a way that is very similar to how a real user would—entering text in text fields and text areas, checking checkboxes, clicking links and buttons, and so on. Capybara allows you to plug in different *drivers* that run those interactions in several different ways—using a real browser like Firefox, IE, or Chrome, or a number of different browser simulators. In this chapter, you'll learn how to use Capybara and when to use the various available drivers it provides.

Have you ever come to a website and thought: *If only this site had a search function...* Or worse:

1. <http://rubygems.org/gems/capybara>



Joe asks:

What About Watir and Webrat?

Watir is the oldest web browser automation tool for Ruby. It was created in 2003 by Bret Pettichord. The original Watir supported only Internet Explorer, but the more recent Watir-WebDriver supports most mainstream web browsers.

Webrat was created in 2008 by Bryan Helkamp and is heavily inspired by Watir. Webrat introduced a browser simulator for quickly running tests and integrated with Selenium WebDriver for real browser tests.

Capybara was created in 2009 by Jonas Nicklas and is essentially an improved implementation of Webrat with a slightly simpler design and is more up-to-date with other frameworks and libraries.

We are not covering Watir or Webrat in this book because they are so similar to Capybara—and Capybara is the most widely used of those libraries.

They might as well remove this search field—this thing never finds what I'm looking for!

People have become so accustomed to search that they expect it to be in every system they use, and they expect it to actually find what they are looking for. In this chapter, we'll explore Capybara's most important features through the development of a search function for the Squeaker application we started in [Chapter 14, Bootstrapping Rails, on page 229](#). We won't get into detail about advanced search engines, but you will get a good understanding about how to *specify* how a search function should work.

Let's start by implementing a simple search that consists of a text field, a button, and a list underneath where the search results are displayed. The napkin sketch in [Figure 15, Search design, on page 247](#) shows what we want.

We'll start off by implementing this feature without any Ajax. This will show you the basics of working with Capybara, using Capybara's Rack driver to send commands directly to the Rails application. This works very similarly to the *in-process* approach we showed you in [Chapter 12, Testing a REST Web Service, on page 201](#), where we used the Rack-Test gem to send web requests to our web service running in the same process as our Cucumber tests (in fact, Capybara's Rack driver uses Rack-Test under the hood). When that's done, we'll make our search more responsive using Ajax by switching to Capybara's Selenium driver. You'll see how Capybara makes it easy to run your tests against different browsers.

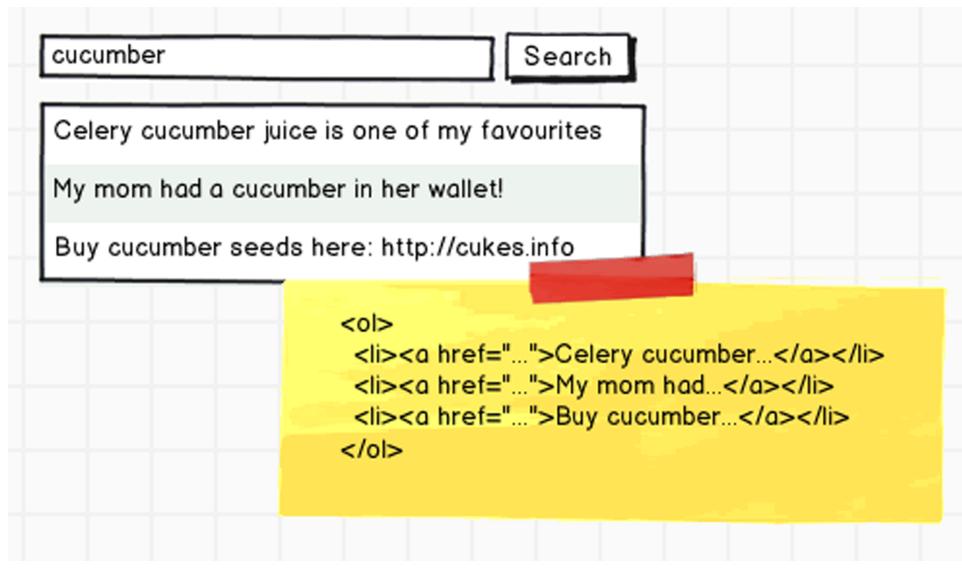


Figure 15—Search design

15.1 Implementing a Simple Search Without Ajax

In [Chapter 14, Bootstrapping Rails, on page 229](#), we dabbled a little bit in Capybara's API. We used the `visit(path)` method to go to a user's page, and we verified that messages were displayed on the screen using Capybara's `have_content` matcher. Let's add a feature that will allow users to search for messages. This will expose us to more of the goodness Capybara has to offer.

Start with the code we left off with at the end of the previous chapter and add a new search feature to `features/search.feature`:

[Download capybara/00/features/search.feature](#)

Feature: Search

Scenario: Find messages by content

Given a User has posted the following messages:

content	
I am making dinner	
I just woke up	
I am going to work	

When I search for "*I am*"

Then the results should be:

content	
I am making dinner	
I am going to work	

If you skipped the previous chapter, you can download the code to jump right in at this point.

We've used a simple example where two out of the three messages that the user has posted will match the search. Notice that we haven't referred specifically to any button clicks or other user interface details in this scenario. Instead, we've been careful to use the language of our own domain: words like *posted*, *search*, and *results*.

When you run this feature, you will see that all the steps are undefined.

Preparing Something to Search For

Our search isn't going to find anything unless some messages already exist, so before we get to do anything with Capybara, we have to add some messages to the database. Knowing how to set up the database before each scenario is essential to testing web applications, so we'll show you how here. Add the following step definition to `features/step_definitions/user_steps.rb`:

```
Download capybara/00/features/step_definitions/user_steps.rb
Given /^a User has posted the following messages:$/
  do |messages|
    user = Factory(:user)
    messages_attributes = messages.hashes.map do |messageAttrs|
      messageAttrs.merge({:user => user})
    end
    Message.create!(messages_attributes)
  end
```

This step definition is a little complicated. Let's examine it, starting at the bottom. ActiveRecord's `create!` method is invoked with an Array of Hash objects, which lets us create several Message records in one fell swoop. Cucumber's `Table`² class (our `messages` object) conveniently provides the `hashes` method to convert it to just that—an Array of Hash. To link those messages to a user, we modify the Array by adding a `:user` entry in each Hash. The first line of the step definition creates that User. The `messages_attributes` variable will contain something like this:

```
[{"content"=>"I am making dinner", :user=>#<User id: 1>},
 {"content"=>"I just woke up", :user=>#<User id: 1>},
 {"content"=>"I am going to work", :user=>#<User id: 1>}]
```

2. See [Chapter 4, Step Definitions: From the Outside, on page 39](#) to learn about tables.

Navigating, Filling in Fields, and Clicking Buttons

With the first Given out of the way, let's get to work with our first custom Capybara step definition.

We decided to write our steps in a declarative style. When I search for "I am" is a great example of a declarative step. What widgets the user interacts with will be nicely hidden inside a single step definition. This makes our Gherkin much simpler and expressive, but it also makes our scenarios much easier to maintain. In the future, we might have many scenarios for search, and if we decide to change the user interface, we only have to update our single step definition instead of having to update a whole lot of scenarios.

To do this, we have to break the declarative step into concrete user interaction steps that Capybara can deal with:

1. Navigate to the search page.
2. Fill in the search criteria in the search field.
3. Click the Search button.

All of these actions are easy to perform with Capybara. We'll implement this in a new file called features/step_definitions/search_steps.rb:

```
Download capybara/00/features/step_definitions/search_steps.rb
When /^I search for "([^"]*)"$/ do |query|
  visit('/search')
  fill_in('query', :with => query)
  click_button('Search')
end
```

We are making several assumptions here that can be used to guide the implementation of the search. First, we expect the web app to respond to the /search path. Second, we expect an input field named query to be on the rendered page. Third, there should be a Search button we can click. None of this stuff exists yet, so we have set ourselves up for failure—the good kind of failure that tells us what to do next:

Fixing the Controller Code

Let's run Cucumber and examine the output:

Feature: Search

```
Scenario: Find messages by content
  Given a User has posted the following messages:
    | content           |
    | I am making dinner |
    | I just woke up   |
    | I am going to work|
```

```

When I search for "I am"
  No route matches [GET] "/search" (ActionController::RoutingError)
    ./features/step_definitions/search_steps.rb:2
      features/search.feature:8
Then the results should be:
  | content           |
  | I am making dinner |
  | I am going to work |
Undefined step: "the results should be:" (Cucumber::Undefined)
features/search.feature:9

```

Failing Scenarios:

```
cucumber features/search.feature:2
```

```

1 scenario (1 failed)
3 steps (1 failed, 1 undefined, 1 passed)
0m0.862s

```

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

```

Then /^the results should be:$/ do |table|
  # table is a Cucumber::Ast::Table
  pending # express the regexp above with the code you wish you had
end

```

Our second step is failing, but this doesn't mean there is something wrong with our step definition. It's our application that is wrong. In [Chapter 14, "Bootstrapping Rails, on page 229](#), we described the overall flow you go through when making a step pass in a Rails application, but let's repeat it quickly here.

The exception from Rails tells us that we need to add a route:

```
Download capybara/01/config/routes.rb
Squeaker::Application.routes.draw do
  resources :users
  resource :search, :only => :show, :controller => :search
end
```

Now, Cucumber will complain about a missing controller, so we'll add that:

```
Download capybara/01/app/controllers/search_controller.rb
class SearchController < ApplicationController
  def show
  end
end
```

We have no view, so we'll add that as well. Let's just enter some hard-coded text for now:

[Download capybara/01/app/views/search/show.html.erb](#)

Search page

With the route, controller, and dummy view, we are able to get a little further.

Let's Give Capybara Something to Work On

Cucumber now fails with the following output:

Feature: Search

```
Scenario: Find messages by content
  Given a User has posted the following messages:
    | content      |
    | I am making dinner |
    | I just woke up   |
    | I am going to work |
  When I search for "I am"
    cannot fill in, no text field, text area or password field with id, name,
    or label 'query' found (Capybara::ElementNotFound)
    (eval):2
    ./features/step_definitions/search_steps.rb:3
    features/search.feature:8
  Then the results should be:
    | content      |
    | I am making dinner |
    | I am going to work |
  Undefined step: "the results should be:" (Cucumber::Undefined)
    features/search.feature:9
```

Failing Scenarios:

cucumber features/search.feature:2

```
1 scenario (1 failed)
3 steps (1 failed, 1 undefined, 1 passed)
0m0.961s
```

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

```
Then /^the results should be:$/
  do |table|
    # table is a Cucumber::Ast::Table
    pending # express the regexp above with the code you wish you had
  end
```

Capybara is telling us that it can't find the search field where we're entering the search criteria. As you see from the error message, Capybara says it knows how to locate a text field based on id, name, or label. We don't need a label for our input field, so all we need to add is a field with a name="criteria" attribute.

We also need a Search button, so we'll put everything in a form that issues a GET request to the same page.³ Our modified view now looks like this:

```
Download capybara/02/app/views/search/show.html.erb
<form id="search" method="get" action="/search">
  <fieldset>
    <input type="text" id="query" name="query" />
    <input type="submit" value="Search" />
  </fieldset>
</form>
```

Now Cucumber passes the search step, and the last verification step remains undefined:

Feature: Search

```
Scenario: Find messages by content
  Given a User has posted the following messages:
    | content      |
    | I am making dinner |
    | I just woke up   |
    | I am going to work |
  When I search for "I am"
  Then the results should be:
    | content      |
    | I am making dinner |
    | I am going to work |
  Undefined step: "the results should be:" (Cucumber::Undefined)
  features/search.feature:9
```

```
1 scenario (1 undefined)
3 steps (1 undefined, 2 passed)
0m1.082s
```

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

```
Then /^the results should be:$/
  do |table|
    # table is a Cucumber::Ast::Table
    pending # express the regexp above with the code you wish you had
end
```

Verifying Page Content

Our last step definition will require some extra thought. We are going to use the value of the query URL parameter to run a database query and display the results in a list (an `` element with `` elements for each result).

3. Using GET instead of POST makes searches easier to share—users can paste the search URL into emails and other systems, or they can bookmark them. Here is an example search URL: <http://localhost:3000/search?query=I%20am>.

To compare the contents of the list on the search page with the expected values in the Gherkin table, we need to extract the interesting bits from the markup. This is easy once you have done it a few times, but at first it can be a little daunting because it involves Capybara's Capybara::Node::Finders API and the use of CSS selectors.

This leaves us in a little bit of a tricky situation. We're about to write nontrivial step definition code to test nontrivial application code. Where do we start? A useful technique in situations like this is to create a *sketch* of the solution. We are going to hard-code the markup we expect to see in our view, without any application logic at all. This will serve as scaffolding to support our test. Let's just modify our app/views/search/show.html.erb view so it contains exactly what we want:

```
Download capybara/03/app/views/search/show.html.erb
<form id="search" method="get" action="/search">
  <fieldset>
    <input type="text" id="query" name="query" />
    <input type="submit" value="Search" />
  </fieldset>
  <ol class="results">
    <li><a href="/users/2/messages/20">I am making dinner</a></li>
    <li><a href="/users/2/messages/21">SHOULD NOT BE HERE</a></li>
    <li><a href="/users/2/messages/22">I am going to work</a></li>
  </ol>
</form>
```

Actually, this isn't *exactly* what we want. The view has the *markup structure* we want but not the correct *data!* (There is a row too many.) This is a very powerful technique. We are deliberately making our hard-coded view be *slightly* wrong. This allows us to verify that our step definition will detect the incorrect (hard-coded) result and display a proper error message. Then we can focus on writing the step definition code correctly without worrying that we have made a mistake in either the application logic or the step definition logic, not knowing where the mistake was.

Extracting Data from the Page

Let's start by constructing an Array of Array to hold the expected search results from the page. (We'll see in a minute how this can be used to compare against a Gherkin table):

```
Download capybara/03/features/step_definitions/search_steps.rb
Then /^the results should be:$/
  do |expected_results|
    results = [['content']] + page.all('ol.results li').map do |li|
      [li.text]
    end
```

```

    puts results.join("\n")
    pending
end

```

The `page.all('ol.results li')` call will return an Array of Capybara::Element objects, each one representing an `` element that contains a single `<a>` element. We're calling `map` on the Array to convert each Capybara::Element to another Array of Strings. Finally, we're using Cucumber's `puts` method to print it out the console so we can verify that our conversion is what we intended.

The `[[['content']]` in the beginning is there to ensure our first row is the same as the one we're sending in from Gherkin—having the same values in the top row is necessary in order to compare a Gherkin table with actual values. Running Cucumber prints the following:

Feature: Search

```

Scenario: Find messages by content
  Given a User has posted the following messages:
    | content           |
    | I am making dinner |
    | I just woke up     |
    | I am going to work |
  When I search for "I am"
  Then the results should be:
    content
    I am making dinner
    SHOULD NOT BE HERE
    I am going to work
    | content           |
    | I am making dinner |
    | I am going to work |
  TODO (Cucumber::Pending)
./features/step_definitions/search_steps.rb:14
features/search.feature:9

1 scenario (1 pending)
3 steps (1 pending, 2 passed)
0m1.002s

```

That looks about right! We are successfully extracting data from our web page.

Using Table Diffs

Now that we are confident that we are properly extracting data from the page, let's compare it to our Gherkin table that contains the expected results. The `Table` object that gets passed to your step definition is an instance of Cucumber::Ast::Table, and one of the lesser known but extremely powerful methods on

this class is the `diff!` method. It takes a single argument that it expects to be an Array of Array representing rows and columns. If all the values are equal, the step definition passes. If not, the step definition fails, and a diff is printed out. Let's put it to use:

```
Download capybara/04/features/step_definitions/search_steps.rb
Then /^the results should be:$/
  do |expected_results|
    results = [['content']] + page.all('ol.results li').map do |li|
      [li.text]
    end

    expected_results.diff!(results)
  end
```

Will Cucumber give us a proper error message?

Feature: Search

```
Scenario: Find messages by content
  Given a User has posted the following messages:
    | content           |
    | I am making dinner |
    | I just woke up   |
    | I am going to work |
  When I search for "I am"
  Then the results should be:
    | content           |
    | I am making dinner |
    | SHOULD NOT BE HERE |
    | I am going to work |
Tables were not identical (Cucumber::Ast::Table::Different)
./features/step_definitions/search_steps.rb:13
features/search.feature:9
```

Failing Scenarios:

cucumber features/search.feature:2

```
1 scenario (1 failed)
3 steps (1 failed, 2 passed)
0m0.975s
```

It looks like it does. If you have colors enabled in your command window, you will see the surplus row colored gray. If there had been missing rows, they would have been yellow. Likewise for columns. We have built up confidence that our somewhat complicated step definition will properly detect errors. Now let's get rid of the hard-coded parts and implement the search properly. We'll start by changing the view to loop over an array of `Message` and produce the same markup as our hard-coded markup:

```
Download capybara/05/app/views/search/show.html.erb
<form id="search" method="get" action="/search">
  <fieldset>
    <input type="text" id="query" name="query" />
    <input type="submit" value="Search" />
  </fieldset>

  <ol class="results">
    <% @messages.each do |message| %>
      <li><%= link_to(message.content, [message.user, message]) %></li>
    <% end %>
  </ol>
</form>
```

If we run Cucumber again, we will get an error message from the view. This is because we haven't assigned anything to the `@messages` variable in our controller.

In our controller we are just going to take the query parameter and ask the model class to do the search for us:

```
Download capybara/05/app/controllers/search_controller.rb
class SearchController < ApplicationController
  def show
    @messages = Message.like(params[:query])
  end
end
```

We could have implemented the search logic in the controller, but it is a wiser design decision to put that in the model. That allows us to unit test it in isolation later if we want. Here is what it looks like:

```
Download capybara/05/app/models/message.rb
class Message < ActiveRecord::Base
  belongs_to :user

  def self.like(content)
    content.nil? ? [] : where(['content LIKE ?', "%#{content}%"])
  end
end
```

We're making sure the `like` method returns an empty array in case the argument is `nil`, which it will be if the request doesn't include a `query` parameter.

Now that the query logic is implemented, we will get an error message from Rails saying it can't generate the link. This is because we haven't defined a route for the messages yet. That's easy to do:

Download capybara/05/config/routes.rb

```
Squeaker::Application.routes.draw do
  resources :users
  resources :messages
end
resource :search, :only => :show, :controller => :search
end
```

And finally Cucumber is happy:

Feature: Search

```
Scenario: Find messages by content
  Given a User has posted the following messages:
    | content           |
    | I am making dinner |
    | I just woke up   |
    | I am going to work |
  When I search for "I am"
  Then the results should be:
    | content           |
    | I am making dinner |
    | I am going to work |
```

```
1 scenario (1 passed)
3 steps (3 passed)
0m1.018s
```

Try This

Before we head on to learn about Ajax, here are some small exercises you can try, to learn more about Capybara and Cucumber's table diffing:

- Add an extra column in the results table.
- Add a feature for posting messages through the UI so you can try the app —we never ran it yet!

15.2 Searching with Ajax

Our users are ecstatic about the search feature we released, and Squeaker has taken over most of the world, leaving millions of messages in our system. Some of the users have complained that despite the new search functionality, it takes too long to find what they are looking for. During usability testing sessions, we also discovered that many of the users don't even know what they are looking for!

We have come up with an idea that we think might improve this. We're going to try to make search results more incremental by displaying them as the

user types—even before they hit the Search button. This is described by a new scenario:

[Download capybara/06/features/search.feature](#)

```
Scenario: Find messages by content using auto-search
  Given a User has posted the following messages:
    | content           |
    | I am making dinner |
    | I just woke up   |
    | I am going to work |
  When I enter "I am" in the search field
  Then the results should be:
    | content           |
    | I am making dinner |
    | I am going to work |
```

There is a subtle but essential difference in our When step here. We're only *entering* text, and unlike the When step in our previous scenario, we are not submitting the form by clicking the button or hitting the Enter/Return key. We have an idea about how to implement this feature—we'll issue an Ajax request whenever the user types a character, and when the response comes back, we'll display the results on the same page without doing a full page load.

Using Selenium

When we implemented our basic search feature, we used Capybara to access our web interface using Capybara's *Rack* driver, which is what Capybara uses by default. Capybara's Rack driver is a browser simulator implemented in Ruby that connects directly to the web server interface of your Rails application, making it nice and fast. While it knows how to parse the HTML on our page and create a simplified DOM⁴ that we can interact with and query, it completely ignores any JavaScript on the page because it doesn't have a JavaScript engine built in. So, how can we use Capybara to test the JavaScript-based autosearch we're about to write?

We have to tell Capybara to use a driver that supports JavaScript instead of the more limited Rack driver. Capybara, being one of Cucumber's best friends, defines a couple of *tagged hooks*, as described in [Tagged Hooks, on page 147](#), to make this easy. One of those tagged hooks will run for each scenario tagged with `@javascript` and set up Capybara to use a JavaScript-aware driver instead of the default one for any scenario carrying the same tag. Capybara's default

4. The Document Object Model is a tree-structure representing elements in a HTML page: <http://www.w3.org/DOM/>.

Capybara Drivers: Pros and Cons

Capybara makes it possible to plug in different *drivers*, which are different strategies for connecting to a web application. They all sit behind the public Capybara API you will be using (see [Section 15.3, The Capybara API, on page 268](#)), and trying a different driver is usually a one-line change.

Some drivers ship with Capybara itself, while others are third-party libraries. They each have their strengths and weaknesses, which we have tried to summarize here:

Driver	Good	Bad
Rack	Fast and simple.	No JavaScript support. Works only with Ruby/Rack applications.
Selenium	Real browser with JavaScript support. Works with web applications written in PHP, Java, .NET, and so on.	Slow. Needs a desktop operating system to run.
Capybara-Webkit ^a	Fast headless browser with JavaScript support.	Runs only on Linux and OS X. Hard to install.

a. <http://rubygems.org/gems/capybara-webkit>

JavaScript driver uses Selenium⁶ to interact with our application. Let's add that @javascript tag to our scenario to see how it works:

```
Download capybara/07/features/search.feature
@javascript
Scenario: Find messages by content using auto-search
  Given a User has posted the following messages:
    | content      |
    | I am making dinner |
    | I just woke up   |
    | I am going to work |
  When I enter "I am" in the search field
  Then the results should be:
    | content      |
    | I am making dinner |
    | I am going to work |
```

Running this feature will make a browser appear, and all of the interaction now happens via the real browser. If your web application is based on the Rack API (most Ruby-based web frameworks are), Capybara also takes care

6. Selenium is a web browser automation tool that will open a real browser to interact with a web server. You can learn more about Selenium at <http://seleniumhq.org/>.

of starting your application's web server in a separate thread before opening the browser.

Feature: Search

```
Scenario: Find messages by content
  Given a User has posted the following messages:
    | content           |
    | I am making dinner |
    | I just woke up   |
    | I am going to work |
  When I search for "I am"
  Then the results should be:
    | content           |
    | I am making dinner |
    | I am going to work |

@javascript
Scenario: Find messages by content using auto-search
  Given a User has posted the following messages:
    | content           |
    | I am making dinner |
    | I just woke up   |
    | I am going to work |
  When I enter "I am" in the search field
    Undefined step: "I enter "I am" in the search field" (Cucumber::Undefined)
      features/search.feature:22
  Then the results should be:
    | content           |
    | I am making dinner |
    | I am going to work |

2 scenarios (1 undefined, 1 passed)
6 steps (1 skipped, 1 undefined, 4 passed)
0m1.058s
```

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

```
When /^I enter "([^\"]*)" in the search field$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end
```

Unsurprisingly, Cucumber exits with a missing step definition. We need to tell it how to enter text in a field without submitting the form. That's easy! We'll implement the undefined step definition almost like the one we used in our previous scenario, except this time we won't click the Search button—we are only going to fill in the field:

Running Capybara Against Non-Ruby Applications

Capybara isn't just limited to testing Ruby on Rails applications. You can very easily configure Capybara to automate any Ajax web application, but you'll need to add a few lines of configuration to a Ruby file in your features/support directory. Here's what you need to do:

```
Capybara.run_server = false
Capybara.current_driver = :selenium
Capybara.app_host = 'http://www.your-test-server.com'
```

This tells Capybara not to try to start a Rack server and configures it to use the Selenium driver by default. The last setting, `app_host`, means that when you call `visit('/login')`, Capybara will actually visit the URL `http://www.your-test-server.com/login`, saving you from having to duplicate the hostname throughout your test code.

You'll need to start the web server yourself, which you can do automatically using the ServiceManager^a gem or using the technique explained in [Section 12.2, Out-of-Process Testing of Any REST API, on page 213](#).

a. http://rubygems.org/gems/service_manager

[Download capybara/08/features/step_definitions/search_steps.rb](#)

```
When /^I enter "([^"]*)" in the search field$/ do |query|
  visit('/search')
  fill_in('query', :with => query)
end
```

If you run the feature again and look carefully, you will see that the search field is populated with the search text from our Gherkin scenario before the browser is closed when the scenario is done. Neat! This leaves us with only one more pending step—the one where we need to verify that intermediate search results are coming back. It's time to take a little break and think about how we are going to implement this with Ajax and dynamic HTML.

Designing Our Live Search

When the user types in the search field and waits a short time, the browser will issue an Ajax GET request identical to the one we used for our non-JavaScript version, with one small modification: the request will have an additional `Accepts` HTTP header, indicating to the server that we want the response as JSON (instead of HTML). When the response comes back to the browser, we'll iterate over the search results in the JSON we get back and create those `` elements dynamically with JavaScript.

Implementing this in JavaScript, even with the help of jQuery, will require somewhere between twenty and sixty lines of code, depending on your style.



Matt says: Pausing Capybara

It can often be quite frustrating watching Capybara's browser flash past something that you want to take a closer look at. An easy trick to get Cucumber, and therefore Capybara, to pause during a step is to call Cucumber's `ask` method in the step definition. You can use it like this:

```
When /^I enter "([^\"])*" in the search field$/ do |query|
  visit('/search')
  fill_in('query', :with => query)
  ask('does that look right?')
end
```

An alternative is to use a debugger statement to pause the code, which will mean installing a debugger. For Ruby 1.9, you need `ruby-debug19`.^a This gives you the added flexibility of being able to actually try commands on the fly. I especially like to use this when I need to compose a tricky selector in Capybara.

a. <http://rubygems.org/gems/ruby-debug19>

This is not an unsurmountable amount of code, but running Cucumber every time you make a small change to this code is something we recommend against because the feedback loop will be too slow. So, how do we proceed now? There are two main directions you can take from here: TDD or not.

JavaScript TDD

Developing JavaScript with TDD used to be hard to do, because of a lack of good tools, but this is a thing of the past. Unfortunately, Cucumber is not one of those tools—it's too high level. Low-level TDD with JavaScript is beyond the scope of this book, but if you want to give it a try, we can recommend QUnit⁹ or Jasmine.¹⁰ *Test-Driven JavaScript Development* [Joh10] is an excellent book that treats the topic really well.

Without JavaScript TDD

This is how we all started programming. Just write the code and test it manually! Since JavaScript TDD is a big topic that we are not going to cover in this book, we are going to cheat a little and “just write the code.” We are going to add a couple of messages to the development database using the rails console, start up the web server, and code until we have something that works, testing

9. <http://docs.jquery.com/QUnit/>
 10. <http://pivotal.github.com/jasmine/>

our live search manually in the browser. When we think we have something that works, we can run Cucumber to get a final verification.

This approach is not ideal—on real projects we would use TDD for our JavaScript as well. Still, our Cucumber scenario both will serve as validation and will protect us against regressions should someone change the JavaScript in the future. So, without further ado, we'll just give you some JavaScript code that we developed exactly this way. Create a new file in `app/assets/javascripts/search.js` with the following content:

```
Download capybara/09/app/assets/javascripts/search.js
function Search(form) {
  this.form = form;
}

Search.prototype.queue = function (query) {
  if (this.timer) {
    clearTimeout(this.timer);
  }
  var self = this;
  this.timer = setTimeout(function () {
    self.search(query);
  }, 150);
};

Search.prototype.search = function (query) {
  var self = this;
  jQuery.ajax({
    url: this.form.action,
    type: this.form.method,
    data: {'query': query},
    success: function(results) {self.render(results)},
    contentType: 'application/json',
    dataType: 'json'
  });
}

Search.prototype.render = function (results) {
  var html = "";
  for (var i = 0, l = results.length; i < l; ++i) {
    var url = '/users/' + results[i].user_id + '/messages/' + results[i].id;
    html += '<li><a href="' + url + '">' + results[i].content + '</a></li>';
  }
  jQuery(this.form).find('ol.results').html(html);
}

jQuery.fn.search = function () {
  this.each(function () {
    var search = new Search(this);
```

```

var input = jQuery(this).find("input[type=text]");
input.bind('keyup', function () {
  search.queue(this.value);
});
});

jQuery(function() {
  jQuery('#search').search();
});

```

This code defines a jQuery search plug-in and activates it on the input element with a DOM ID of search. Whenever a user (or Selenium!) enters a character, the browser fires the keyup event, and this starts a timer. If a delay of 150ms elapses without any more characters being typed, it issues an Ajax request with the search. We use the delay so that we don't hammer our server with search requests if the user is a really fast typist. This request also indicates that it would like to have the response back as JSON. When the response successfully comes back with the search results in a JSON array, we iterate over them and render our search results.

Making the Web App Return JSON

As we mentioned earlier, we want the response from the search request to contain JSON when the client explicitly says it wants JSON. This is just a little tweak to the SearchController:

```

Download capybara/09/app/controllers/search_controller.rb
class SearchController < ApplicationController
  respond_to :html, :json
  def show
    @messages = Message.like(params[:query])
    respond_with(@messages)
  end
end

```

Here we have added a `respond_to` call at the class level to indicate that we can serve both HTML and JSON. We have also added a `respond_with` call at the bottom of our `show` method so that Rails knows *what* it should turn into JSON —the search results. Rails makes this very easy for us.

Finally, we'll add an `autocomplete="off"` attribute to our input field to prevent the browser itself from suggesting values as we type:

```

Download capybara/09/app/views/search/show.html.erb
<form id="search" method="get" action="/search">
  <fieldset>
    <input type="text" id="query" name="query" autocomplete="off" />
  </fieldset>
</form>

```

```

<input type="submit" value="Search" />
</fieldset>
<ol class="results">
  <% @messages.each do |message| %>
    <li><%= link_to(message.content, [message.user, message]) %></li>
  <% end %>
</ol>
</form>

```

That's a lot of code all of a sudden, and it's high time we ran Cucumber again!

Feature: Search

Scenario: Find messages by content

Given a User has posted the following messages:

content	
I am making dinner	
I just woke up	
I am going to work	

When I search for "I am"

Then the results should be:

content	
I am making dinner	
I am going to work	

@javascript

Scenario: Find messages by content using auto-search

Given a User has posted the following messages:

content	
I am making dinner	
I just woke up	
I am going to work	

When I enter "I am" in the search field

Then the results should be:

content	
I am making dinner	
I am going to work	

Tables were not identical (Cucumber::Ast::Table::Different)

./features/step_definitions/search_steps.rb:20

features/search.feature:23

Failing Scenarios:

cucumber features/search.feature:16

2 scenarios (1 failed, 1 passed)

6 steps (1 failed, 5 passed)

0m4.473s

Our first feature is still passing, but our new one testing the Ajax isn't yet. The table diffing indicates that the two expected lines did not show up on the

page. If you were running this in a console with colors enabled, you would see that the rows I am making dinner and I am going to work are yellow, indicating that they did not show up on the page.

What's missing?

It can be a little hard to tell at this point. Opening up the search page in a browser and doing a little bit of manual testing will help us understand what is going on here. If the search results do indeed show up in the browser, we might have to tweak our step definition a little. Before we bring up the search page, we need to make sure we have something to search for. Cucumber (using DatabaseCleaner) would have cleaned out any data, so we can't use that. And besides, Cucumber uses the test database, whereas the running server we are about to start will use the development one.

Since we haven't yet created a user interface for posting messages, let's stick some messages in the database manually:

```
$ bundle exec rails console

u = User.create! :username => "_why"
u.messages.create! :content => "I am making dinner"
u.messages.create! :content => "I just woke up"
u.messages.create! :content => "I am going to work"
```

Let's start the application:

```
$ bundle exec rails server
```

Now we can open a browser, go to <http://localhost:3000/search>, and start to poke around. Try to enter *I am* in the search field without hitting Enter. You should see two results show up automatically. The app seems to be behaving as expected. So, why was our scenario failing? The reason for this is timing.

Dealing with the Asynchronous Nature of Ajax

In our couple of steps, we are typing the search term and then verifying that we have results in our list. The problem is—we are looking for the results too soon, before the Ajax query has a chance to complete! As we discussed in [Chapter 9, Dealing with Message Queues and Asynchronous Components, on page 157](#), we need to introduce a *synchronization point* so that we can be sure the search has completed before we proceed in checking what it has returned.

Luckily, Capybara knows how to deal with this situation in a simple way. If we add an explicit call to `find`, passing a CSS selector for a DOM element on the page that doesn't yet exist, Capybara will wait a little (50ms) and try again



Matt says:

The Progressive Enhancement Pattern

When I first started using Cucumber, Capybara’s predecessor Webrat was the new kid on the testing block. Although using Selenium for full-stack JavaScript testing was possible with Webrat, it wasn’t easy to set up, and we quickly made a decision: we would build the first iteration of every feature to work without JavaScript. This was mainly driven by pragmatism; we could easily use Webrat to test the app through Rails integration testing stack (similar to how Capybara’s rack mode works), and these tests ran quickly rather than waiting for a browser to start up.

What happened surprised us. First, we would receive fancy designs from our designers that could be implemented only with fairly complex JavaScript. Because of our rule, we had to figure out how to deliver the same behavior using basic HTTP posts and gets of forms and URLs—no JavaScript. This required us to simplify the designs for our first iteration. We got a little pushback at first, but we were a tight team, and the designers quickly caught on, especially when we started shipping these first-iteration features quickly and reliably. And funny enough, it turned out that often these simple HTTP-only versions of the features were actually fine, and the designers decided we should move onto other stuff, instead of building all that complex JavaScript that had been implied by their earlier designs.

So, this rule had helped us to do the simple thing. We shipped features and moved on.

When we did have to add JavaScript, we added it on top of the existing working HTTP-only implementation, and we got another surprise: it was easy! Nearly every time, because we’d built the server-side infrastructure to a clean, standard pattern, the JavaScript we needed was clean and simple to write.

This technique is known as *progressive enhancement*.^a

a. http://en.wikipedia.org/wiki/Progressive_enhancement

until the element appears. If it doesn’t appear after a while (two seconds by default, though this is configurable), it will raise an exception, causing the step definition to fail. Let’s add that find:

```
Download capybara/10/features/step_definitions/search_steps.rb
Then /^the results should be:$/
  do |expected_results|
    # Wait until a matching element is found on the page
    find('ol.results li')
    results = [['content']] + page.all('ol.results li').map do |li|
      [li.text]
    end

    expected_results.diff!(results)
  end
```

This will make the scenario pass, and what's more—we have covered the essential parts of the Capybara API. Capybara has more to offer, but it's mostly variations of what we have already seen.

15.3 The Capybara API

Even though we have used only four of Capybara's methods at this point, we have actually explored the most essential parts already. Let's take a look at more of Capybara's API.

The methods in the Capybara API can be grouped in six categories: navigating, clicking links and buttons, interacting with forms, querying, finding, and scoping. The following sections list the most common methods. Make sure you consult Capybara's own documentation for more comprehensive details.

Navigating

`visit(path)` Navigates to the page at path, issuing a GET request.

Clicking Links and Buttons

<code>click_link(locator)</code>	Finds a link by ID or text and clicks it. This may cause Capybara to load a new page.
<code>click_button(locator)</code>	Finds a button by ID, text, or value and clicks it. This may cause Capybara to submit a form.
<code>click_on(locator)</code>	Finds a button or link by ID, text, or value and clicks it.

Interacting with Forms

<code>fill_in(locator, {:with => text})</code>	Locates a text field or text area and fills it in with the given text. The field can be found via its name, ID, or label text.
<code>choose(locator)</code>	Finds a radio button and marks it as checked. The radio button can be found via name, ID, or label text.
<code>check(locator)</code>	Finds a checkbox and marks it as checked. The checkbox can be found via name, ID, or label text.
<code>uncheck(locator)</code>	Finds a checkbox and marks it as unchecked. The checkbox can be found via name, ID, or label text.
<code>attach_file(locator, path)</code>	Finds a file field on the page and attaches a file given its path. The file field can be found via its name, ID, or label text.

<code>select(value, {:from => locator})</code>	Finds a select box on the page and selects a particular option from it. If the select box is a multiple select, select can be called multiple times to select more than one option. The select box can be found via its name, ID, or label text.
---	--

Querying

<code>page.has_css?(css_selector)</code>	Checks whether an element identified by CSS or XPath is on the page or inside the current element.
<code>page.has_button?(locator)</code>	Checks whether an element identified by locator is on the page or inside the current element.
<code>page.has_checked_field?(locator)</code>	
<code>page.has_unchecked_field?(locator)</code>	
<code>page.has_field?(locator, options = {})</code>	
<code>page.has_link?(locator, options = {})</code>	
<code>page.has_select?(locator, options = {})</code>	
<code>page.has_table?(locator, options = {})</code>	
<code>page.has_content?(text)</code>	Checks whether the page or current node has the given text content, ignoring any HTML tags and normalizing whitespace.

All of the `has_xxx?` methods return true or false, and they also have an opposite `has_no_xxx?` that can be used to verify the absence of elements.

These `has_xxx?` and `has_no_xxx?` methods should always be used in conjunction with assertions—either using Test::Unit or RSpec. For example, to check that the page has the text *orange juice*, you would write the following using Test::Unit:

```
assert(page.has_content?("orange juice"))
```

Or using RSpec:

```
page.should have_content("orange juice")
```

Finding

<code>find_field(locator)</code>	Finds a form field on the page.
<code>find_link(locator)</code>	Finds a link on the page.
<code>find(selector)</code>	Finds the first matching element on the page using a CSS selector or raises an error.

find(:xpath, selector)	Uses XPath to find the first matching element on the page. You can tell Capybara to use XPath by default with Capybara.default_selector = :xpath.
all(selector)	Returns all elements (as an Array) matching the given selector.

A full reference can be found at <http://rubydoc.info/github/jnicklas/capybara/master/Capybara/Node/Finders>.

All `find*` methods will poll for a maximum of two seconds until an element is found before it gives up and raises an exception. At first glance, they might seem similar to the `has_xxx?` and `has_xxx?` Query methods, but the `find*` methods do not return true or false. Instead, they return a `Capybara::Element` object on which you can invoke methods. For example:

```
page.find('#main-nav').click_link('Login')
```

If you are familiar with the jQuery JavaScript library, you will find some similarities here.

Scoping

within(locator, &block)	Executes the given block for a particular scope on the page. Finds the first element matching the given locator and executes the block scoped to that element.
within_fieldset(locator, &block)	
within_frame(frame_id, &block)	
within_table(locator, &block)	
within_window(handle, &block)	

Using scoping can sometimes be useful when we want to click, fill in, query, or find elements in a specific area of a page. This can help avoid picking the wrong element if there are several similar elements on the page. For example, if we had more than one Search button on our page and wanted to make sure we were clicking the one inside our form with `id="search"`, we could have scoped it like so:

```
within('#search') do
  click_button('Search')
end
```

Scoping is a powerful technique, especially for complex pages that have a lot of markup. For simpler pages you may not need it at all—just search globally on the page as we did in our Squeaker examples.

15.4 Taking Screenshots

Taking a screenshot of the browser when a step fails can help you diagnose failures quicker. These screenshots can be embedded directly into an HTML report. The trick is to set up an After hook, as described in [Section 8.4, Using Hooks, on page 147](#). Try adding the following code to a new file in features/support/screenshots.rb:

```
Download capybara/11/features/support/screenshots.rb
After('@javascript') do |scenario|
  if(scenario.failed?)
    page.driver.browser.save_screenshot("html-report/#{$scenario._id_}.png")
    embed("#{$scenario._id_}.png", "image/png", "SCREENSHOT")
  end
end
```

We can take screenshots only when there is a browser, which is why we tagged the hook with @javascript. A hook can receive an optional argument that gives us a reference to the current scenario object, and we ask it if it's failed so that we don't take screenshots for passing scenarios.

Then we get an access to Selenium's browser object by calling page.driver.browser. Keep in mind that it's only Capybara's Selenium driver that has a browser accessor. We're safe since the tag makes sure this happens only when we're running Selenium. We're calling the save_screenshot method from Selenium to save the screenshot, and we're using the scenario object's `_id_` as an identifier for the image file. (In Ruby, all objects have a unique `_id_`.)

After saving the object, we are calling the embed method, which takes three arguments—`file_name`, `mime_type`, and `label`. Cucumber will forward this to the active formatters, which will embed a reference to the file into the report. It's actually only the HTML formatter that will take any action here. The other formatters will just ignore it.

Let's make a little change to our code to provoke a failure so we can see the screenshot. Let's reverse the results from the search in app/model/message.rb:

```
Download capybara/11/app/models/message.rb
class Message < ActiveRecord::Base
  belongs_to :user

  def self.like(content)
    content.nil? ? [] : where(['content LIKE ?', "%#{content}%"]).reverse
  end
end
```

Before we run Cucumber again, we have to create the output folder where the images will be written:

```
$ mkdir html-report
```

Now we can run Cucumber with the HTML formatter:

```
$ bundle exec cucumber --format html --out html-report/index.html
```

Try opening the generated HTML file in your browser. You should see a *SCREENSHOT* link for one of the two failing scenarios. The other failing scenario doesn't have a screenshot, since it was running with the headless Rack driver.

15.5 What We Just Learned

Testing web applications can be done in different ways. How to do it depends on what programming language your application is written in and whether it's using JavaScript and Ajax. Let's summarize what we've covered in this chapter:

- Capybara has a simple DSL for navigating around URLs, filling in forms values, and clicking buttons and links.
- Capybara has several drivers—some of them built-in, and some of them available as separate gems—that allow the same code to automate different browsers or browser simulators.
- Web applications written in Ruby can use Capybara's Rack driver to test functionality without JavaScript or Ajax. This is the fastest possible way to do an end-to-end test.
- For Ruby/Rack applications, the `@javascript` tag tells Capybara to use the Selenium driver to test pages that use JavaScript, allowing you to mix scenarios that use the faster Rack driver with scenarios that need a browser.
- Web applications that use Ajax (or that are not based on Ruby/Rack) cannot be tested with Capybara's Rack driver, but it's easy to switch over to the Selenium driver, which brings up a browser.
- We learned to extract multiple text values from the page using Capybara's `all` method and then compare them with a Gherkin table using `diff!`.
- The A in Ajax stands for asynchronous, which means you sometimes need to think about where to put synchronization points into your tests.

Capybara will wait for Ajax content to appear when you call the `find` method.

Try This

Capybara can be used outside of Cucumber too. A handy way to practice getting to grips with Capybara is to automate some dull repetitive task that you have to do on the Web, such as filling in timesheets or logging in to your Internet banking.

Create a plain old Ruby script like this one, and flesh it out with whatever you'd like it to automate:

```
Download capybara/try_this/web_robot.rb
require 'capybara/dsl'

Capybara.run_server = false
Capybara.default_driver = :selenium
Capybara.app_host = 'http://www.google.com'

class WebRobot
  include Capybara::DSL

  def go
    visit '/'
    # ... etc

    puts('press any key to finish')
    gets
  end
end

WebRobot.new.go
```

Save that file as `web_robot.rb` and run it with this:

```
$ bundle exec ruby web_robot.rb
```

Now flesh out the robot to do your bidding!

Testing Command-Line Applications with Aruba

Command-line applications are the bread and butter of many programming tasks. Any programmer is familiar with common command-line applications like `ls/dir`, `cp/copy`, and `cd`. If you are a Linux or OS X user, you probably use them a lot—`ps` and `kill` for process management; `grep`, `sed`, and `awk` for file manipulation; and so on. Even cucumber is a command-line application.

You have probably written a couple of command-line applications or scripts yourself. Even a Rake task is an example of a command-line script. A command-line application usually starts its life as a small script, but often it evolves into a more elaborate program. While many people end up testing the internal logic of a command-line application with unit tests, it is less common to test the whole stack of the application by interacting with it as a user would, through the command-line interface—the user interface of the application.

In this chapter, you'll learn how a Cucumber plug-in called Aruba lets you test your own (and other people's) command-line applications. Before we dive in, let's first look at some characteristics that all command-line applications have in common.

16.1 Simple Interfaces

The beauty of command-line applications is that they all have the same simple interface, as shown in [Figure 16, Command-line applications all use the same simple interface, on page 277](#). They can read from standard input (`STDIN`) and write to standard output (`STDOUT`). When something goes wrong, they sometimes write to standard error (`STDERR`). In addition to using those three streams,



Aslak says: How Aruba Was Born

The Aruba gem was born on the island of Aruba in the Caribbean while David Chelimsky (project lead for RSpec) and I were at a conference called SpeakerConf. Open source contributors get to go to warm places!

RSpec and Cucumber are both command-line applications themselves, and both projects have a set of Cucumber tests that their developers use when they're working on the code. There were a lot of similarities between the two sets of Cucumber tests, so we extracted a library of common step definitions and called it Aruba. Both frameworks now use Aruba for their own acceptance tests.

command-line applications can accept arguments on the command line, which in Ruby end up in the ARGV constant. Consider the following command:

```
cp foo.txt bar.txt
```

Here, the two command-line arguments foo.txt and bar.txt are passed to the cp command-line application.

When a command-line application is done and exits, it reports how the execution went with an *exit status*. On Linux and OS X, the exit status is available in the shell variable named \$? . On Windows, it's available in %ERRORLEVEL%. A nonzero exit status is used to indicate that something went wrong.

Command-line applications can be written in any programming language, and they can be combined with pipes, like Lego, to perform more advanced tasks.

Seen from the outside, all command-line applications are the same. The only way to interact with them is via streams and command-line arguments. This also means that as far as Cucumber is concerned, the approach for interacting with a command-line application is the same regardless of what the application does or what programming language it's written in. Because the high-level interaction is the same for all command-line applications, there is no need to reinvent the wheel every time we want to test a new command-line application. Aruba has everything we need.

16.2 Our First Aruba Feature

Aruba is a Ruby gem that bundles several useful Cucumber step definitions that you can use in your own scenarios. To stay focused on what Aruba has to offer, we're not going to build our own command-line application right now. Instead, we'll just use Aruba to test the ruby command-line application. We're

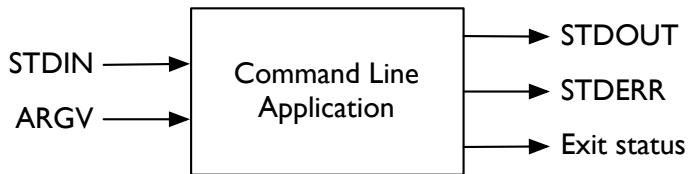


Figure 16—Command-line applications all use the same simple interface.

going to start off by writing our feature without installing the aruba gem. Just bear with us; you will soon see it in action. Let's start with something basic.

Streams and Exit Status

The ruby command has a number of handy command-line options. One of them is the -e option, which lets us execute a one-liner of Ruby code.

Let's implement a scenario that verifies that the following command works as expected:

```
$ ruby -e 'puts "hello"'
```

We'll start by creating a simple project structure. Create an empty folder with a features directory and a single ruby.feature file:

```
features/
+-- ruby.feature
```

Add the following code to features/ruby.feature:

```
Download command_line_applications/01/features/ruby.feature
Feature: ruby -e
  Scenario: print something
    When I run `ruby -e "puts 'hello'"`  

    Then it should pass with:  

      ....  

      hello  

      ....
```

That's all we need. Let's run the feature and let Cucumber lead us to the next step:

```
Feature: ruby -e

  Scenario: print something
    When I run 'ruby -e "puts 'hello'''  

    Then it should pass with:  

      ....  

      hello  

      ....
```

```
1 scenario (1 undefined)
2 steps (2 undefined)
0m0.001s
```

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

```
When /^I run 'ruby \-e "([^\"]*)"$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end
```

```
Then /^it should pass with:$/ do |string|
  pending # express the regexp above with the code you wish you had
end
```

If you want snippets in a different programming language, just make sure a file with the appropriate file extension exists where cucumber looks for step definitions.

As usual, Cucumber is trying to be helpful by suggesting step definition snippets for us. This time, however, we'll do things a little differently than usual. Instead of copying those step definition snippets to our own files, we're going to install the aruba gem.

Installing Aruba

There is nothing fancy about installing Aruba. First, we will install the gem. Create a Gemfile like this and then run bundle to install the gems:

[Download command_line_applications/02/Gemfile](#)

```
source :rubygems

group :test do
  gem 'cucumber', '1.1.3'
  gem 'aruba', '0.4.8'
end
```

Now that we have Aruba installed, we can tell Cucumber to use it simply by adding a features/support/aruba.rb file with a single line:

[Download command_line_applications/02/features/support/aruba.rb](#)

```
require 'aruba/cucumber'
```

This will load and register several step definitions that are packaged inside the Aruba gem. When we run the feature again, we get a different result:

Feature: ruby -e

```
  Scenario: print something
    When I run 'ruby -e "puts 'hello'''
    Then it should pass with:
```

```
"""
hello
"""

1 scenario (1 passed)
2 steps (2 passed)
0m0.314s
```

Wow! The steps no longer show up as undefined; in fact, they're passing. This is because Aruba comes bundled with some generic step definitions for testing command-line applications. It just so happens that the steps we wrote in our scenario have matched perfectly with a few of those generic step definitions. Aruba has helped get our scenario up and running in next to no time, without having to write a single line of Ruby code. Let's dig a little deeper to find out how that works.

Examining Aruba's Step Definitions

When you have a large library of step definitions, especially one provided by a third-party library like Aruba, it can be useful to ask Cucumber for a complete list of all the step definitions. You can do that using the `stepdefs` formatter, like so:

```
$ cucumber --format stepdefs
```

What you should see is a long list of step definitions. If you have colors enabled in your terminal, you'll see that two of the step definitions are highlighted in green, indicating that they've been used and have passed. Those step definitions are as follows:

<code>/^I run `([^\`]*` \$/</code>	Executes the command between the back-ticks in a child process, waits for it to finish, and remembers the STDOUT, STDERR, and exit status.
<code>/^it should (pass fail) with:\$/</code>	Checks that the exit status indicated pass or fail as expected.
	Verifies that the string you passed in between triple quotes appears in either STDOUT or STDERR.
	If either of these checks fail, Aruba raises an exception, and the step fails.

Let's try to provoke a failure. Change the feature to say `Then it should fail with:` instead of `Then it should pass with:`. That should give you the following output:

```
Feature: ruby -e

  Scenario: print something
    When I run 'ruby -e "puts 'hello'''
    Then it should fail with:
      """
      hello
      """
    Exit status was 0 which was not expected. Output:
      hello

      (RSpec::Expectations::ExpectationNotMetError)
      features/ruby.feature:4
```

Failing Scenarios:
cucumber features/ruby.feature:2

1 scenario (1 failed)
2 steps (1 failed, 1 passed)
0m0.313s

We expected the Ruby process to fail, which technically speaking means exiting with a nonzero exit status. However, ruby executed our simple program quite perfectly, so Aruba correctly failed our step.

Now that you understand the fundamentals of how Aruba and Cucumber work together to test command-line tools, we'll look at how to work with files.

Try This

Change the feature back to the original Then it should pass with: step.

- Can you make the test fail by changing the Ruby command to produce a nonzero exit status?
- What happens if you change the expected output to something different from hello, for example bonjour?

16.3 Working with Files and Executables

Command-line applications often deal with files on the file system. The most trivial example is the cat command (type on Windows), which prints the contents of a file to STDOUT. Command-line applications will read files, and sometimes they'll write them too. So when we test them, we need to be able to create files or examine their contents as part of our scenarios.

Remember way back at the beginning of the book in [Chapter 2, First Taste, on page 11](#) where we created a simple command-line calculator? We're going

What If I Care Only About the Exit Status?

The Then `/^it should pass with:$/` step definition looks for output in both STDOUT and STDERR and then looks at exitstatus. While this is a common use case to want to make all these checks at once, there are situations where we want to check only one or the other, but not both.

The following step definitions let you do just that:

- `/^the exit status should be (\d+)$/`
- `/^the exit status should not be (\d+)$/`
- `/^the stdout should contain "([^\"]*)"$/`
- `/^the stderr should contain "([^\"]*)"$/`
- `/^the stdout should not contain "([^\"]*)"$/`
- `/^the stderr should not contain "([^\"]*)"$/`

to revisit that example now, this time using Aruba to take care of interacting with the calculator. We'll build a slightly different calculator that uses a file as input this time so that we can show you more of Aruba's features. By using Aruba, we'll be able to keep our step definition code from getting too complex and focus on the behavior we want from the calculator instead.

Create a new project, and we'll start again from scratch with the feature we wrote for adding two numbers together:

[Download command_line_applications/05/features/adding.feature](#)

Feature: Adding

```
Scenario: Add two numbers
  Given the input "2+2"
  When the calculator is run
  Then the output should be "4"
```

Of course, we didn't know about Aruba when we wrote this scenario, so we're not going to match any of Aruba's step definitions. We could reword the scenario to use steps that Aruba understands, but that would make the scenario harder to read. Instead, we'll start by using Cucumber's built-in steps method to delegate the work to Aruba from the body of our step definitions. Later we'll show you how to refactor the step definitions to call Aruba's Ruby API directly, but this is a nice simple first step.

First, add Aruba to the project just as before by creating a Gemfile and a file `features/support/aruba.rb` with just a single line:

[Download command_line_applications/05/Gemfile](#)
 source :rubygems

```
group :test do
  gem 'cucumber', '1.1.3'
  gem 'aruba', '0.4.8'
end
```

[Download command_line_applications/05/features/support/aruba.rb](#)
 require 'aruba/cucumber'

As usual, run bundle to install the gems if you haven't already.

Now, run cucumber to get the stub step definition snippets and paste them into features/step_definitions/calculator_steps.rb. Let's implement the first step by delegating to an Aruba step definition:

[Download command_line_applications/05/features/step_definitions/calculator_steps.rb](#)

```
Given /^the input "([^"]*)"$/ |input|
steps %{
  Given a file named "input.txt" with:
  """
  #{input}
  """
}

end
```

We're using steps inside our step definition to give Cucumber some more Gherkin to execute, as was first described in [Section 5.4, Nesting Steps, on page 75](#). We could have written this Gherkin inline in our scenario, but it would have been coupling our calculator features to be relevant only to the command-line version. We have big plans for our calculator, so our stakeholders have asked us to try to keep our feature abstract from the nitty-gritty details of interacting with this particular version. Let's carry on and implement the rest of the step definitions using the same pattern:

[Download command_line_applications/05/features/step_definitions/calculator_steps.rb](#)

```
When /^the calculator is run$/ |do|
steps %{
  When I run `ruby calculator.rb input.txt`
}

Then /^the output should be "([^"]*)"$/ |output|
steps %{
  Then it should pass with:
  """
  #{output}
  """
}
```

With all that done, let's run our scenario. We know it's going to fail because we don't have our calculator.rb program in place yet, but it will be useful to get some feedback nonetheless:

Feature: Adding

```
Scenario: Add two numbers
  Given the input "2+2"
  When the calculator is run
  Then the output should be "4"
  Exit status was 1 but expected it to be 0. Output:
```

```
/usr/bin/ruby: No such file or
directory - calculator.rb (LoadError)
```

```
(RSpec::Expectations::ExpectationNotMetError)
features/adding.feature:6
```

Failing Scenarios:
cucumber features/adding.feature:3

```
1 scenario (1 failed)
3 steps (1 failed, 2 passed)
0m0.191s
```

Out first two steps are passing already, but the last one is failing. The Aruba step definition we've called is checking the exit status of our process. We told Aruba to expect the process to pass, but it's found an exit code of 1, indicating that it failed. As part of the error message, Aruba has printed the output generated by running the command, which seems to indicate that our calculator.rb program could not be found. That makes sense, since we haven't added it yet.

Let's add the calculator program, putting it in the root of our project as we did before. Here's a reminder of what that file should look like:

```
Download command_line_applications/06/calculator.rb
input = File.read(ARGV[0])
numbers_to_add = input.split('+').map { |n| n.to_i }

total = 0
numbers_to_add.each do |number|
  total += number
end

print(total)
```

When you've added that file, your project should look like this:

```
calculator.rb
features/
  +- adding.feature
  +- step_definitions
    +- calculator_steps.rb
  +- support/
    +- aruba.rb
```

Even now, with `calculator.rb` in place, you'll still get the same error: Aruba just can't see our `calculator.rb` file. We need some more clues as to what's going on. Fortunately, Aruba has a mechanism to help us out.

Using `@announce` to See What Aruba Sees

Aruba gives you a quick and easy way to see exactly what's happening as it interacts with your command-line application. All you have to do is tag the scenario with `@announce`, and Aruba will ask Cucumber to report everything that goes on. Let's give it a try. Open `features/adding.feature` and add an `@announce` tag at the top of the file. Now run `cucumber` again. Here's what you should see:

```
@announce
Feature: Adding

  Scenario: Add two numbers      # features/adding.feature:4
    Given the input "2+2"        # features/step_definitions/calculator_steps.rb:2
    When the calculator is run   # features/step_definitions/calculator_steps.rb:13
      $ cd ~/command_line_applications/06/tmp/aruba
      $ /usr/bin/ruby calculator.rb input.txt

      /usr/bin/ruby: No such file or directory – calculator.rb (LoadError)
      Then the output should be "4" # features/step_definitions/calculator_steps.rb:19

      /usr/bin/ruby: No such file or directory – calculator.rb (LoadError)

      /usr/bin/ruby: No such file or directory – calculator.rb (LoadError)
      Exit status was 1 but expected it to be 0. Output:

      /usr/bin/ruby: No such file or
      directory – calculator.rb (LoadError)

      (RSpec::Expectations::ExpectationNotMetError)
      features/adding.feature:7

Failing Scenarios:
cucumber features/adding.feature:4 # Scenario: Add two numbers

  1 scenario (1 failed)
  3 steps (1 failed, 2 passed)
  0m0.118s
```

Now we can see each of the commands that Aruba has run and the response (in this case, in the STDERR stream) from running each command. The first thing that happened was a cd command, changing the directory into tmp/aruba. No wonder our calculator.rb program can't be found: we're in the wrong directory! This illustrates an important difference about how Aruba executes commands compared to the hand-rolled way we did it in [Chapter 2, First Taste, on page 11](#), which we'd better explain.

Isolating Scenarios

When you're testing a command-line application, files on disk are like databases. Just as when we test database-driven applications, we want to avoid state leaking between scenarios, so Aruba provides for this by running each scenario in a temporary directory. The temporary directory is deleted at the beginning of each scenario so that it starts with a clean slate. A happy benefit of cleaning the slate at the beginning rather than the end of a scenario is that you can examine the files and do a post-mortem if you need to debug your scenario. Take a look for yourself: you'll find a tmp/aruba directory in the root of your project:

```
$ ls tmp/aruba
```

You should see the input.txt file that was written the last time you ran your scenario. If we add another scenario to our feature that also writes to input.txt, we can be confident that we'll get a fresh copy of the file for that scenario.

Telling Aruba to Leave Files Alone

By default, Aruba will clear the contents of the tmp/aruba directory before each scenario. If, for some reason, you want a particular scenario to run but leave the files on disk alone, you can tag it with @no-clobber. This is often used when you want to run Aruba in a large, complex folder structure that you're confident won't be modified during the scenario.

In that situation, it's often also useful to tell Aruba to run its tests in a different directory.

Setting Aruba's Working Directory

You can tell Aruba to work in a specific directory other than the default tmp/aruba by setting the @dirs instance variable at the beginning of each scenario. The best way to do that is in a Before block:

```
Before { @dirs = ['path/to/somewhere/else'] }
```

Aruba Logging

Sometimes it can be handy to see exactly what command Aruba executes and what the actual output of that command is. This is easily enabled by adding a tag to your scenario or feature. The tags that Aruba understands are as follows:

@announce-cmd	prints the executed command
@announce-stdout	prints the STDOUT of the executed command
@announce-stderr	prints the STDERR of the executed command
@announce	all of the above

It's tempting to think about changing the working directory to point to the root of our project, but we'd also need to set @no-clobber to keep Aruba from destroying the rest of our source code! What else can we do?

A quick hack to make our scenario work is to specify the full path to calculator.rb. We can work it out in the step definition relative to that file:

```
Download command_line_applications/07/features/step_definitions/calculator_steps.rb
When /^the calculator is run$/ do
  path = File.expand_path("#{File.dirname(__FILE__)}../../../../calculator.rb")
  steps %{
    When I run `ruby #{path} input.txt`
  }
end
```

But this is a rather ugly solution: previously this step definition was just a simple Gherkin to Gherkin delegator; now it has Ruby code all mixed up in there too. When we use steps to delegate to other Gherkin steps, we like to make sure that it is the only code in the step definition so it stays at the same level of abstraction throughout, making it easy to read.

Still, it should have made our scenario pass:

Feature: Adding

```
Scenario: Add two numbers      # features/adding.feature:3
  Given the input "2+2"        # features/step_definitions/calculator_steps.rb:1
  When the calculator is run   # features/step_definitions/calculator_steps.rb:11
  Then the output should be "4" # features/step_definitions/calculator_steps.rb:19

1 scenario (1 passed)
3 steps (3 passed)
0m0.316s
```

Great. It's good to have our scenario working. We've removed the @announce tag since we don't need the debugging information anymore now that the

scenario is passing. Now we'll make our program into a proper binary command so that we can clean up that ugly code in the step definition.

Setting \$PATH

What we'd really like is for the calculator to be a full-grown command-line application with its own binary command. That way, we'll be able to distribute it as a Ruby gem, so it will be easy for our users to install and upgrade it. Creating an actual Ruby gem is out of the scope of this book, but we'll use the same conventional structure. First, we need to create a `bin` directory in the root of our project.

```
$ mkdir bin
```

Now move the `calculator.rb` file into the `bin` directory, then rename it to simply `calculator`:

```
$ mv calculator.rb bin/calculator
```

We need to add a *shebang* to the file so that the operating system knows that it's a Ruby program:

[Download command_line_applications/08/bin/calculator](#)

```
#!/usr/bin/env ruby
input = File.read(ARGV[0])
numbers_to_add = input.split('+').map { |n| n.to_i }

total = 0
numbers_to_add.each do |number|
  total += number
end

print(total)
```

Finally, we need to tell the operating system that the file is executable:

```
$ chmod +x bin/calculator
```

At this point, you should be able to run the `calculator` command manually. You'll still need to provide an input file and provide the path to the `calculator` command, like this:

```
$ bin/calculator input.txt
```

If we were to go through all the steps to turn this into a Ruby gem and install it on a user's machine, the `bin` directory will be added to the operating system path so that the `calculator` command can be run from any directory. Let's simulate that situation in our tests and modify the step definition to call our new command:

Special note for Windows Users

On Windows you can't make a script without a file extension executable. What we can do instead is create a .bat script that will run our Ruby script. Create a bin/calculator.bat file with the following content:

```
Download command_line_applications/08/bin/calculator.bat
@"ruby.exe" "%~dpn0" %*
```

The %~dpn0 will automatically expand to the full path of the .bat script—*without* the .bat extension, and that's our Ruby script. In the examples that follow, make sure you use the calculator.bat filename instead of calculator so that Windows executes your .bat file.

If you are developing a Ruby gem, a similar .bat file will be generated automatically by Rubygems when the gem is installed on Windows.

```
Download command_line_applications/08/features/step_definitions/calculator_steps.rb
```

```
When /^the calculator is run$/ do
  steps %{{
    When I run `calculator input.txt`
  }}
end
```

Aruba will automatically add the project's bin directory to the PATH while the tests execute, so we don't need to do anything else. Our scenario should be passing again:

Feature: Adding

```
Scenario: Add two numbers      # features/adding.feature:3
  Given the input "2+2"        # features/step_definitions/calculator_steps.rb:1
  When the calculator is run  # features/step_definitions/calculator_steps.rb:11
  Then the output should be "4" # features/step_definitions/calculator_steps.rb:18
```

```
1 scenario (1 passed)
3 steps (3 passed)
0m0.316s
```

You'll find many more step definitions in Aruba for working with files and executables, but now you have a good idea of the fundamentals. Another common use case for command-line applications is where the user is asked for input—to enter a password, for example—and that's what we'll see next.

16.4 Interacting with User Input

Sometimes a command-line application needs to ask the user a question. Aruba supports this, allowing you to write scenarios that include a user responding to requests for information from the command-line application.

We've been getting feedback from user testing of our calculator that a significant number of our users want to be able to perform additions without saving them to a file first. Let's write another scenario to describe that use case:

[Download command_line_applications/09/features/adding.feature](#)

```
Scenario: Add two numbers interactively
  When the calculator is run with no input
  And I enter the calculation "2+2"
  Then the output should be "4"
```

Run cucumber to generate the new step definition snippets, and then paste them into features/step_definitions/calculator_steps.rb:

[Download command_line_applications/09/features/step_definitions/calculator_steps.rb](#)

```
When /^the calculator is run with no input$/ do
  pending # express the regexp above with the code you wish you had
end
When /^I enter the calculation "([^\"]*)"$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end
```

We'll implement the first step definition by delegating to a variation of the Aruba step definition we used earlier. This time, we'll tell Aruba we're going to interact with the child process:

[Download command_line_applications/10/features/step_definitions/calculator_steps.rb](#)

```
When /^the calculator is run with no input$/ do
  steps %{
    When I run `calculator` interactively
  }
end
When /^I enter the calculation "([^\"]*)"$/ do |calculation|
  steps %{
    When I type "#{calculation}"
  }
end
```

We've used two new Aruba step definitions here. You can probably guess how they work, but let's explain them:

/^I run `([^\"]*)` interactively\$/	Executes the command between the backticks in a child process and connects to the STDIN, STDOUT, and STDERR streams, ready for subsequent steps to interact with the child process.
-------------------------------------	---

/^I type "([^\"]*)"\$/	Sends the given text to the running process' STDIN stream.
------------------------	--

Run cucumber, and you should have a failing test, ready to drive out our new behavior:

Feature: Adding

```
Scenario: Add two numbers
  Given the input "2+2"
  When the calculator is run
  Then the output should be "4"
```

```
Scenario: Add two numbers interactively
  When the calculator is run with no input
  And I enter the calculation "2+2"
  Then the output should be "4"
  Exit status was 1 but expected it to be 0. Output:
```

```
~/command_line_applications/10/bin/calculator:2:in
'read': can't convert nil into String (TypeError)
  from
~/command_line_applications/10/bin/calculator:2:in
'<main>'

(RSpec::Expectations::ExpectationNotMetError)
features/adding.feature:12
```

Failing Scenarios:
cucumber features/adding.feature:9

2 scenarios (1 failed, 1 passed)
6 steps (1 failed, 5 passed)
0m0.638s

We've had an error from line 2 of our calculator program, where we try to read the contents of the input file passed as a command-line argument. That's because we haven't changed our application to wait for user input yet, so it's still expecting to be told what file to read. We need to change our calculator program to ask for input if no file is specified:

```
Download command_line_applications/11/bin/calculator
#!/usr/bin/env ruby
input = if ARGV.any?
  File.read(ARGV[0])
else
  puts "Please enter the calculation you'd like me to perform"
  gets
end

numbers_to_add = input.split('+').map { |n| n.to_i }

total = 0
```

```

numbers_to_add.each do |number|
  total += number
end

print(total)

```

Now if there are no command-line arguments, our calculator will ask the user to enter the calculation and then sit and wait for them to type something. Let's see how this plays out when we run our scenario:

Feature: Adding

```

Scenario: Add two numbers
  Given the input "2+2"
  When the calculator is run
  Then the output should be "4"

```

```

Scenario: Add two numbers interactively
  When the calculator is run with no input
  And I enter the calculation "2+2"
  Then the output should be "4"

```

```

2 scenarios (2 passed)
6 steps (6 passed)
0m1.065s

```

Excellent! We've modified the calculator, and not only is our new scenario passing, but so is the first one, which means we haven't broken the existing behavior. Let's make one last change and refactor our step definitions to make them even cleaner. Instead of delegating to Aruba's step definitions, let's use the Aruba API directly.

16.5 Using Aruba's Ruby DSL

Aruba's step definitions are a great example of how step definitions should be written. Each of them is hardly ever more than a single line long, delegating all its work to Ruby methods in Aruba's support layer module Aruba::API. We can call these methods directly from our own step definitions, removing an unnecessary layer of indirection in our code. Here's how the features/step_definitions/calculator_steps.rb file looks with each of the Aruba step definitions translated to their underlying method calls:

[Download command_line_applications/12/features/step_definitions/calculator_steps.rb](#)

```

Given /^the input "([^"]*)"$/ do |input|
  write_file 'input.txt', input
end
When /^the calculator is run$/ do
  run 'calculator input.txt'
end

```

```

When /^the calculator is run with no input$/ do
  run_interactive 'calculator'
end
When /^I enter the calculation "([^"]*)"$/ do |calculation|
  type calculation
end
Then /^the output should be "([^"]*)"$/ do |output|
  assert_passing_with output
end

```

Run cucumber, and you should see the feature is still passing just as before. We find this preferable to leaving the calls to steps in our step definitions. Once we're down into the step definitions layer, we like to keep things simple and work in pure Ruby rather than mixing Gherkin and Ruby together.

16.6 What We Just Learned

With these few examples, you should know enough to start automating tests for your own command-line applications. Let's review what we have learned:

- Command-line applications all look the same from the outside, whatever language they're written in.
- We can verify that a command-line application prints what we expect it to print.
- We know how to check that a command-line application exits with a specific exit status.
- You can create temporary files that the command-line application you are testing can use.
- Aruba uses an empty, temporary directory for each scenario.
- Aruba recognizes special tags that let us see more informative output.
- We can use Aruba to interact with simple command-line applications that ask for user input.

We haven't been through every single step definition of Aruba in this chapter, but we have shown you the most important ones. If you want to learn more about the step definitions we haven't covered, the best place to look is Aruba's own Cucumber features on GitHub.¹ They provide examples of how to use Aruba.

Aruba's built-in step definitions provide a useful starting point for testing your own command applications, but you shouldn't let them have too much

1. <http://github.com/cucumber/aruba/tree/master/features>

influence on how you write your scenarios. Make sure that your scenarios are written in *your project's* domain language so that they make the most sense to you and your team.

Try This

Create your own simple command-line application: a script that takes all its command-line arguments, changes them to uppercase, and prints them to a file called result.txt.

Verify that the file result.txt contains the arguments you passed on the command line, all in uppercase. You should be able to do that using only Aruba step definitions, without writing a single step definition yourself.

APPENDIX 1

Using Cucumber with Other Platforms

Although Cucumber started as a Ruby tool, it has grown to embrace many other platforms. This means you're able to write your Cucumber step definitions in the same language you use for your production code and reach directly into your domain model if necessary. Explaining in detail how to use Cucumber to drive all of those platforms is beyond the scope of this book, but this appendix should give you some pointers where to look for more information.

Platform	Tool	Further Information
iOS	Frank	https://github.com/moredip/Frank
Flash/ActionScript	Cuke4AS3	https://github.com/flashquartermaster/Cuke4AS3
PHP	Cuke4PHP	https://github.com/olbrich/cuke4php
JavaScript/Node.js	cucumber-js	https://github.com/cucumber/cucumber-js
Java and other JVM languages	cucumber-jvm	https://github.com/cucumber/cucumber-jvm
Microsoft .NET/Mono	SpecFlow	http://specflow.org/
C++	CukeBins	https://github.com/paoloambrosio/cukebins

If you're interested in understanding how these tools work or writing your own, you could look further into two main things: the Gherkin parser and the wire protocol.

Gherkin is the language you use to write Cucumber specifications, and it's also a library¹ you can use to parse them. It uses Ragel² to generate its code.

1. <https://github.com/cucumber/gherkin>
2. <http://www.complang.org/ragel/>

At the time of writing, the Gherkin parser is released for the following platforms:

- Ruby 1.8.6-1.9.3 (MRI, JRuby, REE, Rubinius)
- Pure Java (JAR file)
- JavaScript (NPM package; tested with V8/node.js/Chrome, but might work on other JavaScript engines)
- .NET (DLL file)

This allows you to build your own version of Cucumber on top of any of those platforms, and that's how Cucumber-JVM, Cucumber-JS, and SpecFlow all work.

The second piece of the puzzle is Cucumber's wire protocol. This allows Cucumber to invoke step definitions running in another process through a TCP socket. That process could be running on any machine and be written in any language. As we write this, Cuke4PHP and Cuke4AS3 are the main users of the wire protocol, allowing developers to write step definitions in PHP and ActionScript, respectively, and invoke them from Cucumber itself. Cucumber's wire protocol is well documented, with examples, in Cucumber's own features. See the Cucumber source code for details.

Installing Cucumber

A2.1 Installing Ruby

Before you can install Cucumber, you need to install Ruby. The code in this book was tested with *ruby 1.9.3-p0*, and we recommend you use this version or a more recent one. Each operating system has different ways to install Ruby. This section walks you through the most common installation procedures for OS X, Linux, and Windows.

OS X and Linux

First, you are going to need GCC installed to build Ruby and gems that use C extensions, including Cucumber. On OS X you will have to install Xcode. Each Mac ships with a DVD that contains Xcode, and it can also be downloaded from <http://developer.apple.com/xcode/>.

Some Linux distributions come with GCC pre-installed, but many (such as Ubuntu) do not. Just to be on the safe side, Linux users should install some packages. The following commands should work on Ubuntu and other Debian distributions:

```
$ sudo apt-get install build-essential curl zlib1g-dev libreadline5-dev  
$ sudo apt-get install libssl-dev libxml2-dev libxslt-dev sqlite3 libsqlite3-dev
```

That will install GCC and some other tools you'll need to build Ruby and the various gems used in the book. You should now be ready to install RVM.

We recommend you use RVM (Ruby Version Manager) to install Ruby. RVM is a command-line tool that allows you to easily install, manage, and work with multiple Ruby environments from interpreters to sets of gems. That means that even if you already have another version of Ruby installed, you can use the same one as these code examples have been tested with.

To install and use RVM, you'll first need to install Git, which you can download from <http://git-scm.com/download>. On Ubuntu you can install Git with sudo apt-get install git-core.

Follow the installation instructions on the RVM website (<http://beginrescueend.com/>). Make sure to follow the post-installation instructions—including restarting your terminal—to set up a simple rvm command you can run from a shell. When you have installed RVM, use it to install Ruby 1.9.3-p0 and make that your default Ruby interpreter:

```
$ rvm install ruby-1.9.3-p0  
$ rvm --default ruby-1.9.3-p0
```

Windows

We recommend you use Ruby Installer to install Ruby on Windows. You can get it from <http://rubyinstaller.org/>.

You'll then need to install the Ruby Installer Development Kit after you have installed Ruby. This installs a GCC compiler tool chain that will allow you to install gems that use C extensions later. (You cannot use Visual Studio for this.) You can get the Ruby Installer Development Kit from <http://rubyinstaller.org/add-ons/devkit/>.

If you have any problems during installation, please refer to the troubleshooting page for Ruby Installer: <https://github.com/oneclick/rubyinstaller/wiki/Troubleshooting>.

Console Colors on Windows

Cucumber uses ANSI escape codes to print colored output to the console. This isn't supported natively in Windows, so you have to install a tool called ANSICON to see colors.

Download and unzip the latest version from <https://github.com/adoxa/ansicon/downloads>. Open a command prompt and cd to the folder where you unzipped it. Now, cd into either x86 or x64 (depending on your machine's processor) and install it globally on your machine:

```
C:\somewhere\ansi140\x64> ansicon -i
```

Any program that prints ANSI colors will now display properly on your machine.

A2.2 HTTP Proxy Settings

If you are behind a corporate firewall or HTTP proxy, you may have to tell RubyGems how to get through it so it can download gems from the Internet.

This can be done by defining the environment variable `HTTP_PROXY=http://user:pass@host:port`.

A2.3 Installing Bundler

Bundler is a tool that makes it easier to keep track of what gems (and what versions of those gems) an application depends upon. It does so by installing all the gems in your application's Gemfile. All of the code examples in this book use a Gemfile.

Bundler itself is a gem and can be installed with the following:

```
$ gem install bundler
```

A2.4 Installing Cucumber (and RSpec)

Cucumber is packaged as a gem and can be installed by running `gem install cucumber`. However, most of the examples in this book assume you install it via Bundler, which means having a Gemfile in the root directory of your project. Most of the examples also assume you'll use RSpec,¹ so you'll need that in your Gemfile too:

[Download installation/Gemfile](#)

```
source :rubygems

group :test do
  gem 'cucumber', '1.1.3'
  gem 'rspec-expectations', '2.7.0'
end
```

Cucumber will then install with the following command:

```
$ bundle install
```

To verify that you have installed it properly, run `cucumber --help`.

A2.5 Installing Other Gems

Various chapters in this book uses other gems, such as rails or capybara. These are installed by adding them to your Gemfile and running `bundle update`. In most places, we will list what you need to put in your Gemfile, and you can always find the Gemfile by downloading an example's source.

If you just want to install a complete set of all the gems used in the book in one go, you can use the master Gemfile listed in [Appendix 3, Ruby Gem Versions, on page 303](#).

1. <http://rubygems.org/gems/rspec>

A2.6 Choosing a Text Editor

You can use any text editor to work with Cucumber. If you don't have a favorite editor, we recommend one of the following free ones:

- *Windows*: Notepad++: <http://notepad-plus-plus.org/>
- *OS X*: TextWrangler <http://www.barebones.com/products/TextWrangler/>
- *Linux*: GEdit: <http://projects.gnome.org/gedit/>

All of these editors have plug-ins or extensions that will syntax highlight Cucumber's feature files.

APPENDIX 3

Ruby Gem Versions

This book uses several Ruby gems for the code examples. Things move quickly in the open source world, and the versions of many of the gems we've used will have been superseded by the time you read this. You'll find the code examples will work best if you use the same versions of the gems as we did, so there's a complete list of them in this appendix.

To install these gems, create a working directory where you'll be writing the code examples, and download this Gemfile:

Download Gemfile

```
# This Gemfile lists all Gems used throughout the book - with versions.
source :rubygems

# Build stuff
gem 'bundler', '1.0.21'
gem 'rake', '0.9.2.2'
gem 'watchr', '0.7'
gem 'bcat', '0.6.2'

# General stuff
gem 'aruba', '0.4.8'
gem 'cucumber', '1.1.3'
gem 'rake', '0.9.2.2'
gem 'rspec', '2.7.0'

# Rails stuff
gem 'activerecord', '3.1.3'
gem 'capybara', '1.1.2'
gem 'coffee-rails', '3.1.1'
gem 'cucumber-rails', '1.2.1'
gem 'database_cleaner', '0.7.0'
gem 'factory_girl', '2.3.2'
gem 'jquery-rails', '1.0.19'
gem 'launchy', '2.0.5'
```

```
gem 'rails', '3.1.3'  
gem 'rspec-rails', '2.7.0'  
gem 'sass-rails', '3.1.5'  
gem 'sqlite3', '1.3.5'  
gem 'uglifier', '1.1.0'  
  
# Non Rails stuff  
gem 'childprocess', '0.2.3'  
gem 'httparty', '0.8.1'  
gem 'json', '1.6.3'  
gem 'rack-test', '0.6.1'  
gem 'service_manager', '0.6.2'  
gem 'sinatra', '1.3.1'
```

Now, once you've followed the instructions in [Appendix 2, Installing Cucumber](#), [on page 299](#), you can run the bundle command in the directory where you saved the Gemfile, and exactly the right versions of all the gems should install on your machine.

To ensure that these gem versions don't clash with other gems on your machine, we recommend either using a feature of RVM called *gemsets*¹ or prefixing all commands with bundle exec. For example, to run cucumber, you'd run this:

```
$ bundle exec cucumber
```

That way, you'll ensure that Bundler picks the right set of gem versions using your Gemfile and ignores any other versions you may have already installed on your machine.

1. <http://beginrescueend.com/gemsets/>

APPENDIX 4

Bibliography

- [Adz11] Gojko Adzic. *Specification by Example*. Manning Publications Co., Greenwich, CT, 2011.
- [And10] David J. Anderson. *Kanban*. Blue Hole Press, <http://www.e-junkie.com/129573>, 2010.
- [Bec00] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Longman, Reading, MA, 2000.
- [Bec02] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley, Reading, MA, 2002.
- [Bro95] Frederick P. Brooks Jr.. *The Mythical Man Month: Essays on Software Engineering*. Addison-Wesley, Reading, MA, Anniversary, 1995.
- [CADH09] David Chelimsky, Dave Astels, Zach Dennis, Aslak Hellesøy, Bryan Helmkamp, and Dan North. *The RSpec Book*. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2009.
- [CG08] Lisa Crispin and Janet Gregory. *Agile Testing: A Practical Guide for Testers and Agile Teams*. Addison-Wesley, Reading, MA, 2008.
- [Coc04] Alistair Cockburn. *Crystal Clear: A Human-Powered Methodology for Small Teams*. Addison-Wesley Professional, Boston, MA, 2004.
- [Coh05] Mike Cohn. *Agile Estimating and Planning*. Prentice Hall, Englewood Cliffs, NJ, 2005.
- [Dee08] Ian Dees. *Scripted GUI Testing with Ruby*. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2008.
- [Eva03] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Longman, Reading, MA, First, 2003.

- [FBBO99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Reading, MA, 1999.
- [FP09] Steve Freeman and Nat Pryce. *Growing Object-Oriented Software, Guided by Tests*. Addison-Wesley Longman, Reading, MA, 2009.
- [Fea04] Michael Feathers. *Working Effectively with Legacy Code*. Prentice Hall, Englewood Cliffs, NJ, 2004.
- [HT00] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, Reading, MA, 2000.
- [Joh10] Christian Johansen. *Test-Driven JavaScript Development*. Addison-Wesley Professional, Boston, MA, 2010.
- [Mar07] Brian Marick. *Everyday Scripting with Ruby*. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2007.
- [Mes07] Gerard Meszaros. *xUnit Test Patterns*. Addison-Wesley, Reading, MA, 2007.
- [Ohn88] Taiichi Ohno. *Toyota Production System - Beyond Large Scale Production*. Productivity Press, New York, NY, USA, 1st, 1988.
- [TFH08] David Thomas, Chad Fowler, and Andrew Hunt. *Programming Ruby: The Pragmatic Programmer's Guide*. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, Third, 2008.

Index

SYMBOLS

language: comment, 34
\$ anchor, 52, 117
\$ character, escaping, 41, 47
\$? variable, 17
%{ ... } construct, 76
* keyword, 32
* modifier, 46
+ modifier, 48, 50
- character, 18, 194
. character
 progress formatter, 18,
 194
 wildcard, 46
. notation, 194
? modifier, 51
@ tag, 80
^ anchor, 52, 117
| character, 65

A

abstraction, 92–93, 128
acceptance criteria, 26
acceptance testing
 about, 4–5
 collaboration and, 5
 concrete examples, 27
ActionScript, 295
ActiveRecord
 creating records, 234,
 237, 248
 data builders and, 98
 database interfacing,
 174–187
 database schema manage-
 ment, 102

legacy databases, 175
migration, 102, 174, 176
Adzic, Gojko, 6, 73, 89
After hook, 147, 181, 183,
 187, 271
AfterConfiguration hook, 150
AfterStep hook, 150
*Agile Testing: A Practical
Guide for Testers and Agile
Teams*, 4, 101
Ajax
 asynchronicity, 266
 searching with, 257–268
alternation, 46, 51
anchors, regular expressions,
 52, 117
And keyword, 31
AND statements, 193
@announce tag, 284, 286
ANSI colors, 300
ANSICON, 300
Anticipate gem, 172
app_host, 261

- database transactions, 175–187
 overdraft exercise, 155
 PIN authentication and change, 62–64, 75
 teller function, 120–128
 user interface, 133, 141–155
 withdrawals, 70–78, 111–128, 133–137, 152–154
- authentication application examples description, 74
 incidental details, 90
 PIN, 62–64, 75
- autocomplete="off" attribute, 264
- automated testing about, 4
 collaboration and, 4–5
 concrete examples, 25, 27
 legacy applications, 221–227
 maintenance, 105
 reliability problems, 97–104
 skills needed, 140
- automation libraries, *see* libraries
-
- B**
- \b shorthand, 49
 back-end services, message queues, 157, 161–169
 Background element, 61–65
 --backtrace option, 196
 balance exercise, 155
 .bat file, 288
 BDD (Behaviour-Driven Development), 4
 Beck, Kent, 22, 137
 Before hook, 147, 183, 212
 begin..rescue structure, 171
 Behaviour-Driven Development (BDD), 4
 Big Ball of Mud, 103–104
 booting up, support code, 129
 bored stakeholders, 86, 88–97
 brittle features, 86–87
 brittle scenarios coupling, 133
 doc strings, 79
 duplication and, 22, 93
 fixture data, 102
 imperative style, 92
- leaking states, 97
 state and, 32
 symptoms, 86–87
 tester collaboration, 101
- Brooks, Fred, 25
- browsers databases and, 185
 screenshots, 200, 271
- bugs defect rate, 228
 legacy applications, 224
 missing scenarios, 135
 naming, 226
- Bundler, installing, 301
- business stakeholders, *see also* collaboration acceptance testing and, 4–5
 bored, 86, 88–97
 concrete examples, 26
- But keyword, 31
- buttons, 268
-
- C**
- C#, xix
- C++, 295
- calculator application Aruba, 280–292
 simple, 11–23
- capture groups, 45, 119
- capturing arguments, 45–50, 117–120
- Capybara about, 245
 associating messages with users, 239–243
 custom step definitions, 249
 database cleaning, 185
 drivers, 245, 259
 extracting markup, 253
 hooks, 150
 methods list, 268–270
 non-Ruby applications, 261
 pausing, 262
 search, Ajax, 257–268
 search, simple, 246–257
 synchronization, 266
 user interface wrapping, 141–146, 150–154
- character classes, 47, 49
- characterization tests, 222–225
- characters, escaping, 41, 47
- Chemlinsky, David, 276
 child process, Aruba interaction, 289
- ChildProcess, poor man's Service Manager, 166
- Chisolm, Josh, 172
- classes, defining, 113
- cleaning databases, 181–187, 212, 230
- clicking methods in Capybara, 268
- Cockburn, Alistair, 22, 104
- code coverage, 227–228
- collaboration, *see also* communication; ubiquitous language automated testing and, 4–5
 bugs, 135
 problems, 88
 testers, 101
 unit testing, 104
- colon in command-line interface, 194
- colors, console, 300
- combinatorial explosion, 73
- command-line applications about, 275
 arguments in, 275
 Aruba testing, basic, 276–280
 exit status, 276, 281
 files and Aruba, 280–288
 user input, 288–291
- command-line interface, *see also* Aruba continuous integration, 199
 filtering, 192–194
 location specifying, 196
 managing WIP, 197
 options list, 191
 outputs, 194–196
 profiles, 198
 Rake, 198
- commas, 193
- comments, 33, 35
- communication, *see also* collaboration; ubiquitous language concrete examples, 25
 importance of, 3
 precision, 26
 problems, 89–97

- comparing
 data vs. strings, 213
 tables with diff, 68
- concrete examples, 25, 27
- conditions of satisfaction, 26
- console colors, 300
- continuous improvement, 105
- continuous integration
 command-line options, 199
 formatters, 195, 199
 strict mode, 57
- controllers
 creating by hand, 240
 search function, 249–251, 264
- converting
 markup, 254
 strings into numbers, 113, 117–120
- coupling, 133, 149
- create! method, 248
- credit card payment application, 26–28
- Crispin, Lisa, 4
- Crystal Clear: A Human-Powered Methodology for Small Teams*, 22
- CSS selectors, 253, 266
- Cucumber
 benefits, xvii, 89
 example tests, 5
 installing, 299–302
 layers, 14
 non-Ruby platforms, 295
 overview, 7
 problem symptoms, 85–89
 resources, xx
 why use, 3–9
- Cucumber-JS, 295
- Cucumber-JVM, 295
- Cucumber-Rails, *see* Rails
- Cuke4AS3, 295
- Cuke4PHP, 295
- CukeBins, 295
- custom helper methods, 120–128
- CWhildProcess, out-of-process REST testing, 213
-
- D**
- \d shorthand, 49
- DAMP principle, 94
- data
 builders, 98, 102, 234
 fixture, 101, 234
 stores, 203, 216
 vs. strings, 213
- Data tables, 77
- data tables, 64–69, 71, 217, 254, *see also* tables
- DatabaseCleaner gem, 182–185, 230
- databases
 ActiveRecord, 102, 174–187, 234, 237, 248
 browser testing, 185
 cleaning, 181–187, 212, 230
 FactoryGirl, 98, 102, 232–235
 migrations, 102, 174–178, 233, 238
 one-click systems setup, 102
 querying, 252, 256
 reading and writing to, 178–181
 resetting, 216
 setting, 248
 shared environments, 100, 102
- debugger statements, 262
- declarative style, 91–93, 249
- defect prevention, 106
- defect rate, 228
- definitions, step, *see* step definitions
- Delayed Job, 160
- deleting, scenarios, 226
- dependencies
 avoiding, 32, 97
 files, 129
- dependency injection, 126
- descriptions
 Background elements, 63
 documentation, 35
 examples tables, 74
 features, 29, 31
 rules, 74
 scenarios, 32, 74
- Descriptive and Meaningful Phrases (DAMP), 94
- design criteria, 137
- details, incidental, 90, 107, 234
- diff! method, 68, 254
- digit character shorthand, 49
- Direct Model Access (DMA), 237
- directories
 creating, 12
 domain model, 129
 isolating in Aruba, 285
 lib, 129
 scanning, 196
- @dirs instance variable, 285
- discussion forums for Cucumber, xx
- DMA (Direct Model Access), 237
- doc strings, 78
- documentation
 comments and descriptions, 35
 living, 6, 89, 97
 tags, 82
 user stories, 81
- DOM elements, synchronization, 266
- Domain Driven Design*, 5
- domain model
 directory, 129
 organizing, 130
 sketching, 112–114
 user interface wrapping, 133, 141–155
- Don't Repeat Yourself (DRY) principle, 93–94, *see also* duplication, avoiding
- dot character
 progress formatter, 18, 194
 wildcard, 46
- drivers, Capybara, 245, 259
- DRY principle, 93–94, *see also* duplication, avoiding
- dry-run, 29, 130
- DSL module, 146
- duplication, avoiding
 in ActiveRecord, 234
 design criteria, 137, 140
 examples tables, 72
 steps and examples, 93
 transforms, 117–120
 walking skeleton, 22
-
- E**
- e option, 277
- eXtreme Programming, 4
- embed method, 271
- env.rb file
 --dry-run and, 130

booting up, 129
 deleting code, 214
 Rails, 231
 environments, shared, 100, 102
 errors, *see* exceptions
 escaping special characters, 41, 47
 eval method, 23
 Evans, Eric, 5
 eventually method, 170, 172
Everyday Scripting with Ruby, xviii, 163
 examples
 duplication, 94
 key, 73
 number of, 73
 tables, 71–75
 writing, 6, 25, 27
 Examples tables, 71–75
 exceptions
 assertions and, 55, 57
 confidence in, 253, 255
 external services, 169
 failed steps, 56
 executable specifications, 6, 27
 exit hook, 150, 166, 217
 exit status, 276, 281
 --expand option, 71
 exploratory testing, 101
Extreme Programming Explained, 137

F

F character in progress formatter, 18, 194
 Factory method, 232
FactoryGirl, 98, 102, 232–235
 false positives, 44
 façades, 205
 Feathers, Michael, 104, 222
 Feature Injection template, 31
 Feature keyword, 13, 29
 features
 adding to legacy applications, 225
 brittle, 86–87
 comments, 33, 35
 coupled steps, 149
 creating, 12
 descriptions, 29, 31
 naming, 29, 226

organizing, 79–82, 103
 overview, 7
 partitioning, 89
 sharing, 96
 siloed, 95
 slow, 86–87, 103
 subfolders, 80
 synchronization points, 170
 templates, 31
 feedback
 code coverage as, 228
 loops, 97
 files
 Aruba and, 280–288
 organizing, 128–131
 filtering
 lines, 193
 names, 194
 tags, 82, 193
 find, 266, 270
 finding methods in Capybara, 270
 firewalls, 300
 Fixnum integer, 117
 fixtures
 data, 101, 234
 terminology, 99
 Flash, 295
 flickering scenarios, 86, 99–100, 102, 159, 165–171
 flight code exercise, 48
 folders
 organizing with, 79–80, 103
 partitioning with, 89
 formatters
 changing, 18, 194–196
 HTML, 19, 271
 html, 195, 199
 json, 195
 JUnit, 19
 junit, 195, 199
 listing, 19
 progress, 18, 194
 rerun, 195
 screenshots, 271
 stepdefs, 195, 279
 usage, 195
 forms, HTML, 150, 268
 Fowler, Martin, 94
 Frank, 295
 Freeman, Steve, 159

fruit retrieval application
 in-process testing, 202–212
 out-of-process testing, 213–220

G

Gemfile
 Bundler installation, 301
 creating, 141
 versions list, 303
 gemsets, 304
 generators, Rails, 230–231
 get method, 206
 GET requests, 208, 210, 252, 261
 Gherkin
 basics, 25–36
 comments, 33, 35
 data tables, 64–69
 doc strings, 78
 format and syntax, 28
 imperative style, 91, 93
 keywords, 13, 28–31, 43, 45
 languages support, 28, 34, 36, 45
 nested steps, 75–78
 overview, 7
 parser, 295
 readability, 27
 scenario outlines, 70–75
 tags, 79–82
 Given keyword, 13, 30, 42
 goals, features, 31
 Gregory, Janet, 4
Growing Object-Oriented Software, Guided by Tests, 137, 159
 Guard, 241

H

Hash objects, 248
 hashes method, 217
 have_content, 242
 Helkamp, Bryan, 246
 --help, 19, 191
 helper methods
 custom, 120–128
 listing, 124
 hierarchy, stack, 7
 hooks
 After, 147, 181, 183, 187, 271
 AfterConfiguration, 150

- AfterStep, 150
 Around, 148
`at_exit`, 150, 166, 217
 Before, 147, 183, 212
`@javascript`, 258–261
 state, 147, 164
 tagged, 82, 147, 149,
 258–261
 using, 147–150
- hostnames, non-Ruby applications, 261
- HTML**
 formatters, 19, 195, 200,
 271
 forms, 150, 268
- HTTP**
 error responses, 210
 HTTParty, 213–214, 219
 proxy settings, 300
 REST mimicking methods, 205
- Hydra, 89
-
- I**
- `--i18n` switch, 35
 iOS, 295
 imperative style, 76, 91, 93
 improvement, continuous, 105
 in-process testing, 201–212
 incidental details, 90, 107,
 234
 indentation, doc strings, 78
 independent scenarios, 98
 intention, revealing, 138
 isolation
 in Aruba, 285
 databases and, 184
-
- J**
- jQuery, auto-search testing,
 261–264
 jargon, *see* ubiquitous language
 Jasmine, 262
 Java, xix, 295
 JavaScript, xix
 auto-search testing, 258–
 268
 TDD, 262
 tools, 295
 Webrat and, 267
`@javascript` tag, 82, 258–261
 Jenkins, 89
- JSON**
 formatters, 195
 REST web service testing,
 205, 208–212, 216
 search function, 261, 264
 whitespaces, 211
- JUnit, formatters, 19, 195,
 199
- JVM**, xix
-
- K**
- Keogh, Liz, 31
 key examples, 73
 keyup event, 264
 keywords, Gherkin
 `*` (asterisk), 32
 And, 31
 Background, 61–64
 But, 31
 comments, 33
 Feature, 13, 29
 Given, 13, 30, 42
 list, 28, 45
 Scenario, 13, 30–31
 Scenario Outline, 22, 70–75
 structure, 13
 Then, 13, 30, 43
 When, 13, 30, 43
- KnowsTheDomain module, 129–
 130
-
- L**
- language, *see also* communication; ubiquitous language
 concrete examples, 25
 Gherkin support, 28, 34,
 36, 45
 readability, 5, 27, 211
 revealing intention, 138
 search function, 249
- `last_response` method, 208
`@last_response` variable, 219
 launchy gem, 150
 Lawrence, Richard, 149
 leaky scenarios, 97, 102
 leaning on the compiler, 139
 legacy applications
 adding new behavior, 225
 adding tests to, 221–227
 breaking up, 104
 bugs, 224
 characterization tests,
 222–224
 code coverage, 227–228
 fixture data, 101
- lib** directory, 129
- libraries**
 about, 40
 assertions, 60
 Net::Http, 214
 overview, 7
 testers and, 101
- `--line` option, 193
`--lines` option, 194
 lines, filtering on, 193
 links, 268
- Linux installation, 299
 listening, synchronization by,
 159
- living documentation, 6, 89,
 97
- loads and shared environments, 100
 location specification, 196
 logging, Aruba, 286
-
- M**
- maintenance, test, 105
 markup, extracting, 253
 matching
 capture groups, 45
 RSpec, 124, 242
 steps, 40–50, 117
 transforms, 117
- Matts, Chris, 31
- memcache daemons, one-click systems setup, 102
- memory stores, 203, 216
- message posting application,
 236–243
- message queues
 one-click systems setup,
 102
 using, 157–172
- metacharacters
 \$ sign, 41, 47, 52
 anchors, 52
 dot, 18, 46, 194
 escaping, 47
- metrics, 228
- Microsoft .NET, 295
- migrating databases, 174–
 178, 233, 238
- MiniTest, 60
- missing scenarios, 135
- missing steps, 199
- models, Rails
 creating users, 232–235
 message posting, 236–
 243

- modifiers
 plural, 51
 repetition, 46, 48
- modifiers, regular expressions
 plus, 48, 50
 question mark, 51
 star, 46
- modules, World, 124, 129–130
- Mono, 295
- multiple captures, 49
-
- N**
- name option, 194
- naming
 Background elements, 63–64
 bugs, 226
 features, 29, 226
 filtering by name, 194
 lib directory, 129
 revealing intention, 138
 scenarios, 32, 34
- navigating methods in Capybara, 268
- negation, 49, 193
- nested steps, 75–78
- Net::Http library, 214
- Nicklas, Jonas, 246
- nightly builds, 103
- No Silver Bullet, 25
- @no-clobber tag, 285
- Node.js, 295
- noncapturing groups, 51
- North, Dan, 94, 149
-
- O**
- Ohno, Taiichi, 105
- one-click system setup, 102
- operator, asking the, 126
- OR statements, 193
- organizing
 files, 128–131
 step definitions, 130
 with tags and sub-folders, 103
 with tags and subfolders, 79–82
 transforms and modules, 130
- OS X installation, 299
- out option, 195
- out-of-process testing, 201, 213–220
- outputs, changing, 194–196
- outside-in development
 benefits, xviii, 17
 bugs, 135
 placeholders, 21
 sketching, 115
 TDD and, 4
 what vs how, 115
- overdraft exercise, 155
-
- P**
- page content verification, 252
- page object, 242
- pairwise gem, 73
- parallel testing, 89, 103
- parsing
 Gherkin parser, 295
 strings, 211, 213
- partitioning
 features, 89
 steps, 129
 tests, 103
- \$PATH, 287
- pauses, *see* sleeps
- payroll system application, 98
- pending marker, 16
- pending steps, 16, 55, 199
- Pettichord, Bret, 246
- PHP, 295
- PID (process identifier) file, 166
- PIN authentication, 62–64, 75
- pipe character, 65
- placeholders, 21, 71
- plurals modifier, 51
- plus modifier, 48, 50
- polling, 160–161
- ports and adapters architecture, 104
- POST, 216, 252
- post, HTML forms, 151
- posting Squeaker messages, 236–243
- precision, 26, 44
- Premdas, Andrew, 77
- problems, *see also* brittle scenarios
 bored stakeholders, 86, 88–97
 communication, 89–97
 flickering scenarios, 86, 99–100, 159, 165–171
 leaky scenarios, 97, 102
 race conditions, 99–100
- reliability, 97–104
- root-cause analysis, 105–106
- shared environments, 100, 102
- slow features, 86–87, 103
- symptoms, 85–89
- too many scenarios, 103
- process identifier (PID) file, 166
- profile option, 198
- profiles, command line, 198
- Programming Ruby: The Pragmatic Programmer's Guide*, xviii, 163
- progress formatter, 18, 194
- progressive enhancement, 267
- proxy settings, 300
- Pryce, Nat, 159, 234
- puts method, 254
- puts self, 124
-
- Q**
- queries
 database, 252, 256
 list of Capybara methods, 269
- question mark modifier, 51
- queues, *see* message queues
- JUnit, 262
-
- R**
- race conditions, 99–100
- Rack
 pros and cons, 259
 Rack-Test, 205–208
 search, simple, 246–257
- rackup, 214, 217
- Ragel, 295
- Rails
 building applications, 229–235
 creating controllers by hand, 240
 creating users, 232–235
 database cleaning, 185
 generators, 230–231
 message posting, 236–243
 Rake, 198, 231
 Rake, 198, 227, 231
 rake-T, 227
 raw method, 67

- readability
 - collaboration, 5
 - examples tables, 72–73
 - need for, 27
 - REST interfaces, 211
 - scenarios, 61
 - ubiquitous language building, 50
- refactoring
 - Background, 65
 - long test runs, 88
 - revealing intention, 138
- Refactoring: Improving the Design of Existing Code*, 139
- regressions, 116
- regular expressions
 - anchors, 52, 117
 - avoiding, 42
 - capturing arguments, 45–50, 117–120
 - character classes, 47, 49
 - matching steps, 40–48
 - modifiers, 46, 48, 50–51
 - noncapturing groups, 51
 - transforms, 117–120
 - wording flexibility, 50–52
- reliability problems, 97–104
- Relish, 96
- repetition modifiers, 46, 48
- repetition, avoiding
 - Background elements, 64
 - data tables, 64–69
 - scenario outlines, 70–75
- reports, sharing, 199
- require option, 196, 198
- require_relative, 162, 177
- rerun formatter, 195
- respond_to, 264
- REST testing
 - in-process, 201–212
 - out-of-process, 213–220
- revealing intention, 138
- reviewing code, 137–141
- robot exercise, 36, 273
- root-cause analysis, 105–106
- routes, adding, 250
- RSpec
 - Aruba and, 276
 - assertions, 20, 60, 230
 - installation, 301
 - matchers, 124, 242
 - Rails assertions, 230
- REST web service testing
 - 205
 - Story Runner, 81
 - The RSpec Book*, 137
 - RSpec Story Runner, 81
 - rspec-rails gem, 230
- Ruby, *see also* step definitions
 - gems list, 303
 - installing, 299
 - overview, 7
 - resources, xviii
 - version, 299
- ruby
 - Aruba testing, 276–280
 - options, 277
- Ruby Installer Development Kit, 300
- Ruby Version Manager (RVM), 299, 304
- ruby-debug19, 262
- rules, describing, 74
- run times
 - sleeps and, 100
 - slow, 87
 - too many scenarios, 103
- RVM (Ruby Version Manager), 299, 304
- S**
- \s shorthand, 49
- sampling, synchronization by, 159–160, 169–171
- satisfaction, conditions of, 26
- scanning directories, 196
- Scenario keyword, 13, 30–31
- Scenario object, 148
- Scenario Outline keyword, 22, 70–75
- scenarios, *see also* brittle scenarios; hooks
 - Background, 61–64
 - bugs, 226
 - comments, 33, 35
 - creating, 12–17
 - data tables, 64–69, 77
 - deleting, 226
 - dependent, 97
 - descriptions, 32, 74
 - doc strings, 78
 - example tables, 71–75
 - execution overview, 52
 - filtering, 192–194
 - fixture data and, 102
- flickering, 86, 99–100, 102, 159, 165–171
- imperative, 76, 91, 93
- implementing, 17–23
- incidental details, 90, 107
- independent, 98
- isolating in Aruba, 285
- leaky, 97, 102
- missing, 135
- naming, 32, 34
- nested steps, 75–78
- nightly builds, 103
- outlines, 70–75
- overview, 7
- placeholders, 21, 71
- results states, 52–60
- slow features, 86–87, 103
- state and, 32, 97, 102, 201
- synchronization points, 159, 171
- tags, 79–82, 197
- Three Amigos, 95–96
- too many, 103
- translating bugs into, 224
- World and, 125
- writing, 30–33, 61, 89–97
- scoping, 270
- screenshots, 200, 271
- search
 - Ajax, 257–268
 - simple, 246–257
- search plug-in, 264
- Selenium, 246, 258–268, 271
- separation layer, 133
- serializing, 162
- Service Manager gem, 165–169, 261
- shared environments, 100, 102
- sharing
 - features, 96
 - reports, 199
- shebang, adding, 287
- shorthand character classes, 48–49
- siloed features, 95
- SimpleCov, 227
- Sinatra
 - 404 Page Not Found page, 210
 - about, 206
 - REST web service testing, 201, 204–212

- test doubles, 154
 - user interface wrapping, 141–146, 150–154
 - skeleton
 - REST web services, 203
 - walking, 22
 - sketching, 112–114, 253
 - skills, testing, 140
 - slave machines, 89
 - sleeps, 99–100, 168, 171
 - slow features, 86–87, 103
 - snippets, 14, 54, 278
 - source of truth, 7
 - SpecFlow, 295
 - special characters, escaping, 41
 - Specification by Example*, 6
 - specification tests, 222
 - specifications, executable, 6, 27
 - splitting, *see* partitioning
 - spoken languages support, 28, 34, 36, 45
 - sqlite3 database, using, 175–187
 - Squeaker application
 - creating users, 232–235
 - generators, 230–243
 - message posting, 236–243
 - search, Ajax, 257–268
 - search, simple, 246–257
 - stack, hierarchy, 7
 - stakeholders, *see* business stakeholders
 - star modifier, 46
 - starting web services, 217
 - state
 - Background elements and, 64
 - hooks, 147, 164
 - in-process testing, 201
 - independence, 32
 - leaky scenarios, 97, 102
 - message queues and, 164
 - scenarios and, 201
 - World and, 122
 - step definitions, *see also* Aruba
 - about, 39
 - capturing arguments, 45–50
 - creating, 14–17, 42–45
 - custom Capybara, 249
 - data tables, 64–69, 77
 - doc strings, 78
 - domain model, 112–114
 - exit status, 281
 - failing, 56
 - global scope, 149
 - implementing, 16–20
 - location specifying, 196
 - matching, 40–50, 117
 - nested steps, 75–78
 - organizing, 130
 - outlines, 70–75
 - overview, 7
 - pending, 55, 199
 - precision, 44
 - snippets, 14
 - splitting, 129
 - state passing, 122
 - vs.* steps, 40
 - user interface connection, 142–146
 - wording, 44, 50–52, 114, 128, 137–141
 - stepdefs formatter, 195, 279
 - steps
 - failing, 56
 - feature-coupled, 149
 - imperative, 91, 93
 - matching, 40–50, 117
 - nested, 75–78
 - overview, 7
 - pending, 16, 55, 199
 - sleepy, 99–100
 - splitting, 129
 - vs.* step definitions, 40
 - undefined, 54
 - steps method, 77, 281
 - stopping
 - Capybara, 262
 - the line, 105
 - web services, 217
 - storing in memory, 203, 216
 - .story extension, 81
 - strict, 57, 199
 - strings
 - converting into numbers, 113, 117–120
 - doc strings, 78
 - interpolation, 77
 - multiple lines, 76
 - parsing, 211, 213
 - vs.* regular expressions, 42
 - serializing, 162
 - stubs, 21
 - sub-folders, organizing with, 103
 - subfolders, organizing with, 79–80
 - supermarket checkout application, 222–224
 - support code
 - booting up, 129
 - hooks, 147–150
 - user interface wrapping, 141–155
 - symptoms of problems, 85–89
 - synchronization, 158–160, 169–171, 266
 - syntax rules, *see* Gherkin
 - system setup, one-click, 102
 - system test environment, 100
-
- T**
- tables
 - comparing with diff, 68
 - data tables, 64–69, 71, 77, 217, 254
 - Examples, 71–75
 - Rails, 233
 - Scenario Outline, 22
 - turning into arrays, 67
 - tags
 - @announce, 284, 286
 - filtering, 82, 193
 - hooks, 82, 147, 149, 258–261
 - limiting, 197
 - @no-clobber, 285
 - organizing with, 79–82, 103
 - partitioning with, 89, 103
 - WIP management, 197
 - tags option, 193
 - TCP sockets and wire protocol, 297
 - TDD (Test-Driven Development)
 - automated acceptance testing, 4
 - JavaScript, 262
 - teller application, *see* ATM application
 - templates, feature descriptions, 31
 - terminology, *see* ubiquitous language
 - Test Data Builders, 98, 102, 234
 - test doubles, 125, 154
 - test maintenance, 105

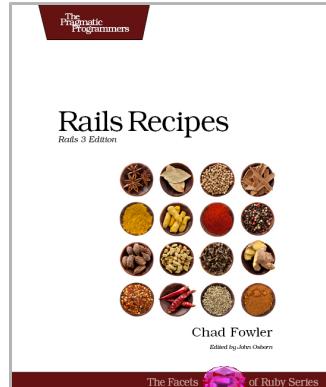
- Test-Driven Development,
see TDD (Test-Driven Development)
- Test-Driven JavaScript Development*, 262
- Testbot, 89
- testers, collaboration and, 101
- testing, *see* acceptance testing; automated testing; characterization tests; exploratory testing; unit testing
- testing skills, 140
- text editors, 302
- Then keyword, 13, 30, 43
- Three Amigos, 95–96
- tic-tac-toe game, 66–69
- timers, 264
- Toyota, 105
- transactions, cleaning databases with, 181–185
- transforms, 117–120, 129–130
- triangulation, 22
- troubleshooting, *see* problems
- truncation, cleaning databases with, 181, 185, 187
- truth, source of, 7
-
- U**
- U character in progress formatter, 194
- ubiquitous language
- advantages, 5
 - benefits, 95
 - context, 94
 - declarative style, 92
 - feature-coupled steps, 149
 - readability and, 50
 - revealing intention, 138
-
- V**
- verbose option, 196
- verifying page content, 252
- versions list, 303
- visit(path) method, 239
-
- W**
- \w shorthand, 49–50
- walking skeleton, 22
- Watchr, 241
- Watir, 246
- web applications
- browsers and databases, 185
 - in-process testing, 201–212
 - out-of-process testing, 201, 213–220
- web interface, *see* user interface
- WebKit, 259
- Webrat, 246, 267
- When keyword, 13, 30, 43
- whitespace
- character shorthand, 49
 - JSON, 211
- wildcards, 45–50
- Wilk, Joe, 73
- Windows
- .bat file, 288
 - command prompt symbol, xx
 - console colors, 300
 - Ruby installation, 300
 - Service Manager, 166
- wip option, 197
- WIP (Work in Progress), managing, 197
- wire protocol, 295
- wireframes, user interface, 145
- withdrawals application, *see* ATM application
- word boundary shorthand, 49
- word character shorthand, 49–50
- Work in Progress (WIP), managing, 197
- Working Effectively with Legacy Code*, 104, 222
- World
- about, 124
 - custom helper methods, 120–128
 - modules list, 124
 - Rack-Test, 207
 - state and, 122
- Wrong, 60, 172
-
- X**
- Xcode, 299
-
- Y**
- YAML files, 198, 231

Go Beyond with Rails and NoSQL

There's so much new to learn with Rails 3 and the latest crop of NoSQL databases. These titles will get you up to speed on the latest.

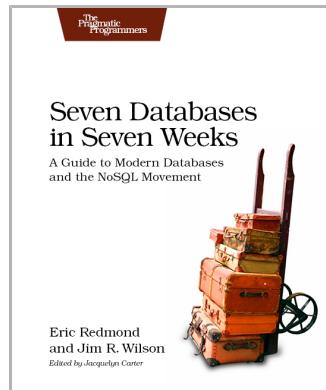
Thousands of developers have used the first edition of *Rails Recipes* to solve the hard problems. Now, five years later, it's time for the Rails 3.1 edition of this trusted collection of solutions, completely revised by Rails master Chad Fowler.

Chad Fowler
(350 pages) ISBN: 9781934356777. \$35
<http://pragprog.com/titles/rr2>



Data is getting bigger and more complex by the day, and so are your choices in handling it. From traditional RDBMS to newer NoSQL approaches, *Seven Databases in Seven Weeks* takes you on a tour of some of the hottest open source databases today. In the tradition of Bruce A. Tate's *Seven Languages in Seven Weeks*, this book goes beyond a basic tutorial to explore the essential concepts at the core of each technology.

Eric Redmond and Jim Wilson
(330 pages) ISBN: 9781934356920. \$35
<http://pragprog.com/titles/rwdata>

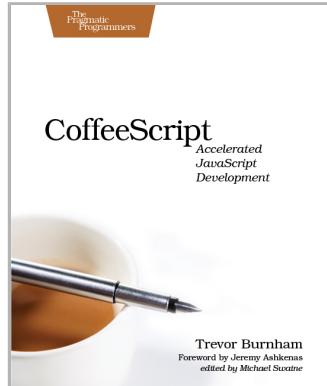


Welcome to the New Web

You need a better JavaScript and more expressive CSS and HTML today. Start here.

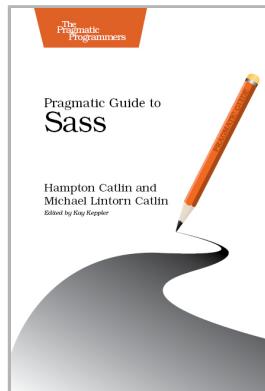
CoffeeScript is JavaScript done right. It provides all of JavaScript's functionality wrapped in a cleaner, more succinct syntax. In the first book on this exciting new language, CoffeeScript guru Trevor Burnham shows you how to hold onto all the power and flexibility of JavaScript while writing clearer, cleaner, and safer code.

Trevor Burnham
(160 pages) ISBN: 9781934356784. \$29
<http://pragprog.com/titles/tbcoffee>



CSS is fundamental to the web, but it's a basic language and lacks many features. Sass is just like CSS, but with a whole lot of extra power so you can get more done, more quickly. Build better web pages today with *Pragmatic Guide to Sass*. These concise, easy-to-digest tips and techniques are the shortcuts experienced CSS developers need to start developing in Sass today.

Hampton Catlin and Michael Lintorn Catlin
(128 pages) ISBN: 9781934356845. \$25
http://pragprog.com/titles/pg_sass



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

This Book's Home Page

<http://pragprog.com/titles/hwcuc>

Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates

<http://pragprog.com/updates>

Be notified when updates and new books become available.

Join the Community

<http://pragprog.com/community>

Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy

<http://pragprog.com/news>

Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: <http://pragprog.com/titles/hwcuc>

Contact Us

Online Orders: <http://pragprog.com/catalog>

Customer Service: support@pragprog.com

International Rights: translations@pragprog.com

Academic Use: academic@pragprog.com

Write for Us: <http://pragprog.com/write-for-us>

Or Call: +1 800-699-7764