



BDD IN ACTION

Behavior-Driven Development for
the whole software lifecycle

John Ferguson Smart

FOREWORD BY Dan North

 MANNING

BDD in Action

BDD in Action

*Behavior-Driven Development for
the whole software lifecycle*

JOHN FERGUSON SMART



MANNING

SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

©2015 by Manning Publications Co. All rights reserved.

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

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

☺ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

 Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor: Dan Maharry
Technical development editor: Ray Lugo
Copyeditor: Benjamin Berg
Proofreaders: Andy Carroll, Melody Dolab
Typesetter: Dennis Dalinnik
Cover designer: Marija Tudor

ISBN: 9781617291654

Printed in the United States of America

1 2 3 4 5 6 7 8 9 10 – EBM – 19 18 17 16 15 14

Deliberate Discovery—A “Sonnet”

*Uncertainty's the muse of all that's new,
And ignorance the space in which she plays;
A year's enough to prove a vision true,
But we could prove it false in only days.
We dream, and chase our dream, and never fear
To fail, and fail. Up, up! And on again,
But ask us to pursue another's goals
And failure makes us mice where we were men.
Ah, best laid plans! Where were you at the end
Who chained us and constrained us from the start?
We knew you made a fickle, fragile friend;
You tricked us when you claimed you had a heart!
We thought less travelled roads would see us winning
In places other fools had feared to stray—
If only we had known from the beginning
The ignorance we found along the way.
And yet, a list of dangers and disasters
Filled out, and scanned, and added to some more
Would still have left out some of what we mastered—
We didn't know we didn't know before.*

*We planned our way with maps we'd made already
Assuming the terrain would be the same,
Expecting well-paved roads to keep us steady
And any local creatures to be tame.
We loaded up our caravans and wagons
With good advice, best practices and tools
But didn't spot the legend—"Here be dragons!"
So we got burnt, again. They say that fools
Rush in, and yet we count ourselves as wise,
We praise each other's skill and raise a glass
To intellect—ignoring the demise
Of expeditions just as skilled as ours.
When they return, worn out, their pride in shreds,
We laugh and say, "A death march! You expect
Such things to fail." And in our clever heads
It's obvious—at least in retrospect.
The dragons of our ignorance will slay us
If we don't slay them first. We could be brave
And work for kings who don't refuse to pay us
When we're delayed because we found their cave.*

*They say that matter cannot be created,
A fundamental principle and law,
While dragons keep emerging, unabated;
As many as you slay, there's still one more.
Our ignorance is limitless—be grateful,
Or else we'd find we've nothing left to learn;
To be surprised by dragons may be fateful,
But truth be told, it's best laid plans that burn.
We could seek out the dragons in their dungeons
And tread there softly, ready to retreat;
We could seek other roads, postponing large ones,
And only fight the ones we might defeat.
The world could be a world of dragon slayers
And stand as men and women, not as mice;
The joy that comes from learning more should sway us;
The fiercest dragons won't surprise us twice.
Discover tiny dragons, be they few,
And all the mightiest, with equal praise—
Uncertainty's our muse of all that's new,
And ignorance the space in which she plays.*

—Liz Keogh

brief contents

PART 1	FIRST STEPS	1
1	■ Building software that makes a difference	3
2	■ BDD—the whirlwind tour	32
PART 2	WHAT DO I WANT? DEFINING REQUIREMENTS USING BDD.....	59
3	■ Understanding the business goals: Feature Injection and related techniques	61
4	■ Defining and illustrating features	87
5	■ From examples to executable specifications	114
6	■ Automating the scenarios	140
PART 3	HOW DO I BUILD IT? CODING THE BDD WAY	179
7	■ From executable specifications to rock-solid automated acceptance tests	181
8	■ Automating acceptance criteria for the UI layer	201
9	■ Automating acceptance criteria for non-UI requirements	236
10	■ BDD and unit testing	260

PART 4 TAKING BDD FURTHER299

- 11 ■ Living Documentation: reporting and project management 301
- 12 ■ BDD in the build process 321

contents

foreword xvii
preface xxi
acknowledgements xxiii
about this book xxv
about the cover illustration xxix

PART 1 FIRST STEPS.....1

1 *Building software that makes a difference* 3

- 1.1 BDD from 50,000 feet 5
- 1.2 What problems are you trying to solve? 7
 - Building the software right* 7 ■ *Building the right software* 9
 - The knowledge constraint—dealing with uncertainty* 10
- 1.3 Introducing Behavior-Driven Development 12
 - BDD was originally designed as an improved version of TDD* 12
 - BDD also works well for requirements analysis* 14
 - BDD principles and practices* 15
- 1.4 Benefits of BDD 28
 - Reduced waste* 28 ■ *Reduced costs* 29 ■ *Easier and safer changes* 29 ■ *Faster releases* 29

- 1.5 Disadvantages and potential challenges of BDD 29
 - BDD requires high business engagement and collaboration* 29
 - BDD works best in an Agile or iterative context* 30
 - *BDD doesn't work well in a silo* 30
 - *Poorly written tests can lead to higher test-maintenance costs* 30
- 1.6 Summary 30

2 **BDD—the whirlwind tour** 32

- 2.1 Introducing the train timetable application 33
- 2.2 Determining the value proposition of the application 35
- 2.3 Requirements analysis: discovering and understanding features 35
 - Describing features* 35
 - *Breaking features down into stories* 37
 - Illustrating the stories with examples* 38
- 2.4 Implementation: building and delivering features 38
 - Going from examples to acceptance criteria* 39
 - *Setting up Maven and Git* 40
 - *Executable specifications: automating the acceptance criteria* 42
 - *Automated tests: implementing the acceptance criteria* 46
 - *Tests as living documentation* 55
- 2.5 Maintenance 56
- 2.6 Summary 58

PART 2 WHAT DO I WANT? DEFINING REQUIREMENTS USING BDD.....59

3 **Understanding the business goals: Feature Injection and related techniques** 61

- 3.1 Introducing Flying High Airlines 63
- 3.2 Feature Injection 63
 - Hunt the value* 64
 - *Inject the features* 65
 - Spot the examples* 65
 - *Putting it all together* 66
- 3.3 What do you want to achieve? Start with a vision 67
 - The vision statement* 68
 - *Using vision statement templates* 69
- 3.4 How will it benefit the business?
 - Identify the business goals 70
 - Writing good business goals* 71
 - *Show me the money—business goals and revenue* 72
 - *Popping the “why stack”: digging out the business goals* 73
 - *Impact Mapping: a visual approach* 76

- 3.5 Who will benefit? Identify stakeholders and their needs 80
- 3.6 What do you need to build? Identify capabilities 82
- 3.7 What features will provide the most ROI?
 - The Purpose-Based Alignment Model 84
 - Differentiating features* 85 ▪ *Parity features* 86
 - Partner features* 86 ▪ *Minimum impact* 86
- 3.8 Summary 86

4 ***Defining and illustrating features*** 87

- 4.1 What is a “feature”? 88
 - Features deliver capabilities* 90 ▪ *Features can be broken down into more manageable chunks* 93 ▪ *A feature can be described by one or more user stories* 95 ▪ *A feature is not a user story* 98
 - Epics are really big user stories* 99 ▪ *Not everything fits into a hierarchy* 100
- 4.2 Illustrating features with examples 100
- 4.3 Real Options: don’t make commitments before you have to 107
 - Options have value* 108 ▪ *Options expire* 109
 - Never commit early unless you know why* 110
- 4.4 Deliberate Discovery 110
- 4.5 From examples to working software: the bigger picture 111
- 4.6 Summary 113

5 ***From examples to executable specifications*** 114

- 5.1 Turning concrete examples into executable scenarios 115
- 5.2 Writing executable scenarios 119
 - A feature file has a title and a description* 119 ▪ *Describing the scenarios* 120 ▪ *The “Given ... When ... Then” structure* 121
 - Ands and buts* 122 ▪ *Comments* 123
- 5.3 Using tables in scenarios 124
 - Using tables in individual steps* 125 ▪ *Using tables of examples* 125
- 5.4 Expressive scenarios: patterns and anti-patterns 128
 - Writing expressive Given steps* 129 ▪ *Writing expressive When steps* 130 ▪ *Writing expressive Then steps* 131
 - Providing background and context* 132 ▪ *Avoid dependencies between scenarios* 133

- 5.5 Organizing your scenarios using feature files and tags 134
 - The scenarios go in a feature file 135* ■ *A feature file can contain one or more scenarios 136* ■ *Organizing the feature files 136*
 - Annotating your scenarios with tags 137*
- 5.6 Summary 138

6 Automating the scenarios 140

- 6.1 Introduction to automating scenarios 143
 - Step definitions interpret the steps 143* ■ *Keep the step definition methods simple 145*
- 6.2 Implementing step definitions: general principles 146
 - Installing BDD tools 147* ■ *Implementing step definitions 147*
 - Passing parameters to step implementations 148*
 - Maintaining state between steps 149* ■ *Using table data from step definitions 150* ■ *Implementing example-based scenarios 151*
 - Understanding scenario outcomes 152*
- 6.3 Implementing BDD more effectively with Thucydides 154
- 6.4 Automating scenarios in Java with JBehave 154
 - Installing and setting up JBehave 154* ■ *JBehave step definitions 156* ■ *Sharing data between steps 157*
 - Passing tables to steps 158* ■ *Step definitions for tables of examples 158* ■ *Pattern variants 159* ■ *Failures and errors in the scenario outcomes 160*
- 6.5 Automating scenarios in Java using Cucumber-JVM 161
 - Cucumber-JVM project setup and structure 162* ■ *Cucumber-JVM step definitions 163* ■ *Pattern variants 163* ■ *Passing tables to steps 164* ■ *Step definitions for tables of examples 165*
 - Sharing data between steps 165* ■ *Pending steps and step outcomes 166*
- 6.6 Automating scenarios in Python with Behave 166
 - Installing Behave 167* ■ *The Behave project structure 167*
 - Behave step definitions 167* ■ *Combining steps 168*
 - Step definitions using embedded tables 168* ■ *Step definitions for tables of examples 169* ■ *Running scenarios in Behave 169*
- 6.7 Automating scenarios in .NET with SpecFlow 170
 - Setting up SpecFlow 170* ■ *Adding feature files 170*
 - Running scenarios 171* ■ *SpecFlow step definitions 172*
 - Sharing data between steps 173* ■ *Step definitions using example tables 173*

- 6.8 Automating scenarios in JavaScript with Cucumber-JS 174
 - Setting up Cucumber-JS* 175 ■ *Writing feature files in Cucumber-JS* 175 ■ *Implementing the steps* 176
 - Running the scenarios* 177
- 6.9 Summary 178

PART 3 HOW DO I BUILD IT? CODING THE BDD WAY ...179

7 *From executable specifications to rock-solid automated acceptance tests* 181

- 7.1 Writing industrial-strength acceptance tests 183
- 7.2 Automating your test setup process 185
 - Initializing the database before each test* 186 ■ *Initializing the database at the start of the test suite* 186 ■ *Using initialization hooks* 186 ■ *Setting up scenario-specific data* 190
 - Using personas and known entities* 192
- 7.3 Separating the what from the how 194
 - The Business Rules layer describes the expected outcomes* 195
 - The Business Flow layer describes the user's journey* 196
 - The Technical layer interacts with the system* 198
 - How many layers?* 199
- 7.4 Summary 200

8 *Automating acceptance criteria for the UI layer* 201

- 8.1 When and how should you test the UI? 203
 - The risks of too many web tests* 203 ■ *Web testing with headless browsers* 204 ■ *How much web testing do you really need?* 205
- 8.2 Automating web-based acceptance criteria using Selenium WebDriver 206
 - Getting started with WebDriver in Java* 207 ■ *Identifying web elements* 210 ■ *Interacting with web elements* 218
 - Working with asynchronous pages and testing AJAX applications* 219 ■ *Writing test-friendly web applications* 221
- 8.3 Using page objects to make your tests cleaner 222
 - Introducing the Page Objects pattern* 223 ■ *Writing well-designed page objects* 227 ■ *Using libraries that extend WebDriver* 233
- 8.4 Summary 235

9 *Automating acceptance criteria for non-UI requirements* 236

- 9.1 Balancing UI and non-UI acceptance tests 238
- 9.2 When to use non-UI acceptance tests 240
- 9.3 Types of non-UI automated acceptance tests 243
 - Testing against the controller layer* 243 ▪ *Testing business logic directly* 247 ▪ *Testing the service layer* 251
- 9.4 Defining and testing nonfunctional requirements 255
- 9.5 Discovering the design 257
- 9.6 Summary 259

10 *BDD and unit testing* 260

- 10.1 BDD, TDD, and unit testing 261
 - BDD is about writing specifications, not tests, at all levels* 263
 - BDD builds on established TDD practices* 264 ▪ *BDD unit-testing tools are there to help* 264
- 10.2 Going from acceptance criteria to implemented features 264
 - BDD favors an outside-in development approach* 264
 - Start with a high-level acceptance criterion* 266
 - Automate the acceptance criteria scenarios* 267
 - Implement the step definitions* 268 ▪ *Understand the domain model* 268 ▪ *Write the code you'd like to have* 269
 - Use the step definition code to specify and implement the application code* 269 ▪ *How did BDD help?* 274
- 10.3 Exploring low-level requirements, discovering design, and implementing more complex functionality 274
 - Use step definition code to explore low-level design* 275
 - Working with tables of examples* 277 ▪ *Discover new classes and services as you implement the production code* 279
 - Implement simple classes or methods immediately* 280
 - Use a minimal implementation* 281 ▪ *Use stubs and mocks to defer the implementation of more complex code* 281
 - Expand on low-level technical specifications* 283
- 10.4 Tools that make BDD unit testing easier 285
 - Practicing BDD with traditional unit-testing tools* 285
 - Writing specifications, not tests: the RSpec family* 287
 - Writing more expressive specifications using Spock or Spec2* 291

- 10.5 Using executable specifications as living documentation 293
 - Using fluent coding to improve readability* 294 ■ *Fluent assertions in JavaScript* 295 ■ *Fluent assertions in static languages* 295
- 10.6 Summary 297

PART 4 TAKING BDD FURTHER299

11 *Living Documentation: reporting and project management* 301

- 11.1 Living documentation: a high-level view 302
- 11.2 Are we there yet? Reporting on feature readiness and feature coverage 304
 - Feature readiness: what features are ready to deliver* 304
 - Feature coverage: what requirements have been built* 305
- 11.3 Integrating a digital product backlog 308
- 11.4 Organizing the living documentation 309
 - Organizing living documentation by high-level requirements* 310
 - Organizing living documentation using tags* 311
 - Living documentation for release reporting* 312
- 11.5 Providing more free-form documentation 314
- 11.6 Technical living documentation 315
 - Unit tests as living documentation* 316 ■ *Living Documentation for legacy applications* 319
- 11.7 Summary 319

12 *BDD in the build process* 321

- 12.1 Executable specifications should be part of an automated build 322
 - Each specification should be self-sufficient* 322 ■ *Executable specifications should be stored under version control* 324
 - You should be able to run the executable specifications from the command line* 325
- 12.2 Continuous integration speeds up the feedback cycle 325
- 12.3 Continuous delivery: any build is a potential release 328
- 12.4 Continuous integration used to deploy living documentation 330
 - Publishing living documentation on the build server* 331
 - Publishing living documentation to a dedicated web server* 332

12.5	Faster automated acceptance criteria	332
	<i>Running parallel acceptance tests within your automated build</i>	332
	▪ <i>Running parallel tests on multiple machines</i>	335
	<i>Running parallel web tests using Selenium Grid</i>	337
12.6	Summary	342
12.7	Final words	342
	<i>index</i>	345

foreword

Since its modest beginnings as a coaching experiment over a decade ago, Behavior-Driven Development (BDD) has taken off in a way I never would have imagined. I'd like to tell you how it started, and why I believe it is more relevant today than it has ever been.

When I first started working on BDD back in 2003, the software industry was going through a revolution. It was a time of possibility, with everything I knew about enterprise software delivery open to question. Two years earlier, a group of software professionals had come together to formulate the Agile Manifesto, and the year after that, in 2002, I was fortunate enough to join some of the pioneers of this new movement. In April of that year, I was hired in the newly opened London office of ThoughtWorks, Inc., a software consulting company at the forefront of the agile movement. Their chief scientist Martin Fowler was one of the signatories of the Agile Manifesto and a few years earlier had written *Refactoring* (Addison-Wesley Professional, 1999), which had a profound effect on how I thought about software. He was also on the first XP project at Chrysler with Kent Beck and Ward Cunningham. It is safe to say his agile credentials are pretty solid, and he had joined the team at ThoughtWorks because they worked in a way that resonated with those values.

At this point I was over ten years into my career and I thought I was a decent programmer. That didn't last very long. I was constantly amazed by the talent of the people I was working with and the game-changing nature of the things I was encountering, things we now take for granted. My first tech lead turned out to have invented continuous integration, which would have been even more intimidating if he hadn't

been such a thoroughly nice guy. That seemed to be a common theme among the early agilists. They were without exception generous with their knowledge, generous with their time, and humble about their discoveries and achievements.

I had some great mentors in my early years at ThoughtWorks. I was encouraged to try things I'd never done before—coaching skeptical programmers, reaching out to anxious testers, and engaging with suspicious business folks. This, then, was the context in which BDD was born. It was a response to a triple conundrum: programmers didn't want to write tests; testers didn't want programmers writing tests; and business stakeholders didn't see any value in anything that wasn't production code. I didn't think I was inventing a methodology—I was just trying to teach TDD to a bunch of programmers without scaring off the testers.

I started by changing the words, moving from “tests” to “behavior.” This seemed to have a profound effect. Then I wrote some software to explore the idea (I registered the `jbehave.org` domain on Christmas Eve 2003, much to my wife's dismay), and, the following year, developed the Given-When-Then vocabulary of scenarios with business analyst Chris Matts.

Now it's over a decade later and BDD has gone in some surprising directions. Liz Keogh integrated it with complexity theory; Chris Matts and Olav Maassen evolved it into RealOptions; and, of course, it has spawned a plethora of tools in multiple languages. It even has its own conference!

Given all this, I'm delighted to see the arrival of a comprehensive BDD book. John Smart has taken on a large, fast-moving space and produced a book that is everything I would want it to be. He has managed to capture the essence of what I was driving at with BDD while acknowledging the many others who have been part of the journey. As well as a solid theoretical foundation, *BDD in Action* delivers a thorough treatment of the current state of BDD tools, along with general structural and automation guidance that will most likely outlive many of them.

Other authors and practitioners have covered various aspects of BDD—the RSpec and Cucumber books provide a good background but are necessarily tool-oriented. Gojko Adzic's *Specification by Example* (Manning, 2011) is a masterful treatment of the living documentation aspect of BDD and its intersection with acceptance test-driven development. But this is the first time I've seen all the pieces brought together into a cohesive whole.

The last decade has seen agile methods move into the mainstream, which means the relationship between delivery teams and their stakeholders, and the feedback and communication involved in that, can be the difference between success and failure. BDD provides a means to develop and support that relationship, and *BDD in Action* is a great handbook for technical and non-technical participants alike.

I would like to finish on a cautionary note. BDD is a mechanism for fostering collaboration and discovery through examples. The goal isn't generating acceptance criteria or feature files; it isn't automation; it isn't testing; although BDD contains elements of all of these. I see teams obsessing about the number or density of feature

files, or the degree of automation, or debating the merits of one tool over another. All of this is beside the point. The examples and scenarios of BDD are simply a form of documentation. More documentation isn't better—better documentation is better! Don't lose sight of the real goal, which is to use software to create business impact. BDD is just one way of staying focused on that goal, and is only useful to the extent that it helps with this. It isn't an end in itself.

It is exciting to see an idea you had brought to life articulated in someone else's words, and I am grateful to John for his comprehensive analysis of the many facets of BDD in this book. I hope you enjoy reading it as much as I did, and that it helps you in your efforts to deliver software that matters.

DAN NORTH
CREATOR OF BDD

preface

When I finally sat down to write about the origins of this book, I realized that my involvement with Behavior-Driven Development goes back much further than I thought.

My first foray into the world of executable specifications was in 2001, though at the time, I didn't know it by that name. In fact, at the time, I didn't know it had a name. I was working on a project to write a pricing algorithm for an insurance company. Along with a hefty 200-page Word document, the client kindly provided us with an Excel spreadsheet containing 6,000 or so rows of sample data that he used to check pricing changes. It was an afterthought: the idea was that we might find the spreadsheet handy to check our pricing application manually. I found it more convenient to set up a test harness to load the Excel data, run it against our pricing algorithm, and check the outcomes. While I was at it, I got the test harness to write an updated version of the spreadsheet—but with extra columns for the calculated values, highlighted in green if they matched, and red if they didn't.

At first nothing passed, and the spreadsheet was solid red. Then, I implemented the application bit by bit, following the large formal specifications document provided by our client, and, gradually, more and more of the spreadsheet went green. We even found a number of errors in the specifications along the way! The resulting code had a near-zero defect rate, and a delighted customer, and my first experiment with what would now be called Acceptance Test Driven Development and Living Documentation was a hit.

I didn't actually hear the term “Behavior-Driven Development” for another six years or so. While I was working on the Java Power Tools book in 2007, my good friend

Andy Glover added an offhand question in one of his emails: “Quick question—what are your thoughts on BDD?” “Bee what?” I thought to myself, before running off to read Dan North’s seminal article on the topic.¹ And it immediately clicked. Dan had crystalized what I thought the next steps in testing practices should be.

I soon became involved in the BDD community, contributing to Andy’s BDD tool for Groovy, easyb,² and experimenting with other BDD tools and approaches, and BDD soon became an invaluable part of my tool kit when helping teams out with their agile adoption.

Fast-forward four years: on a sunny summer day in January 2011, I met Andrew Walker in a cafe opposite Hyde Park in Sydney to discuss helping to get his teams started with TDD, BDD, and test automation. Andrew became a good friend, and we had many fascinating discussions, particularly around the ideas of Agile requirements management, executable specifications, and living documentation.

The Thucydides³ project was born of these conversations and from an idea—would it be possible to produce meaningful living documentation by observing the execution of automated acceptance tests as they run?

Andrew was also the one to point out that I should write a book on the topic. In early 2013, I finally got in touch with the folks at Manning with the idea of a book on BDD. They were interested, and so a new book project was launched!

This book is a labor of love about a topic I hold close to my heart. I regularly see confusion and misunderstanding in the community about what BDD is (and isn’t). One of the goals of this book is to give the broader software development community a better understanding of what BDD is, where it can be used, and how it can help. My hope is that you will find as many benefits in practicing BDD as I have.

¹ <http://dannorth.net/introducing-bdd/>, first published in March 2006

² <http://easyb.org/>

³ <http://thucydides.info>

acknowledgements

A little like a film, a book has a cast of hundreds, from minor roles to people whose contributions made this book possible.

My thanks go to the dedication, professionalism, and attention to detail of all the folks at Manning: Michael Stephens, Maureen Spencer, Ozren Harlovic, Andy Carroll, Rebecca Rinehart, and many others. Dan Maharry, my development editor, was unflagging, courteous, and helpful—all the way to the final chapter in his drive to push this book into production. And thanks to Doug Warren, who did an exemplary job as a technical proofreader and came up with some great suggestions along the way.

The reviewers also deserve special mention—this book would not be what it is without their help: David Cabrero, Dror Helper, Ferdinando Santacroce, Geraint Williams, Giuseppe Maxia, Gregory Ostermayr, Karl Metivier, Marc Bluemner, Renzo Kottmann, and Stephen Abrams.

Thanks to Andrew Gibson, whose thorough and well-thought-out criticism was a major driver in making the book a much better one than it would have been otherwise. And special thanks to Liz Keogh, who provided invaluable feedback on many key parts of the book. Her “Deliberate Discovery” sonnet at the start of the book manages to capture many of the essential ideas behind BDD.

I owe much of what I know about BDD to the BDD community: Gojko Adzic, Nigel Charman, Andrew Glover, Liz Keogh, Chris Matts, Dan North, Richard Vowles, and many others. Not to mention the broader Agile and open source communities: Dan Allen, John Hurst, Paul King, Aslak Knutsen, Bartosz Majsak, Alex Soto, Renee Troughton,

and more. Thanks for so many fruitful conversations, email exchanges, pair coding sessions, and Skype chats! And special thanks to Dan North for contributing the foreword to the book.

Much of the contents of the book is inspired by work done and conversations held over the years with clients, friends, and colleagues in many different organizations: Parikshit Basrur, Tom Howard, Ray King, Ian Mansell, Peter Merel, Michael Rembach, Simeon Ross, Trevor Vella, Gordon Weir, and many others.

Finally, a very special thanks to my dedicated spouse, Chantal, and my boys, James and William, without whose patience, endurance, support, and encouragement this book would simply not have been possible.

about this book

The goal of this book is to help get teams up and running with effective BDD practices. It aims to give you a complete picture of how BDD practices apply at all levels of the software development process, including discovering and defining high-level requirements, implementing the application features, and writing executable specifications in the form of automated acceptance and unit tests.

Audience

This book has a broad audience. It's aimed both at teams who are completely new to BDD, and at teams who are already trying to roll out BDD or related practices, like Acceptance-Test-Driven Development or Specification by Example. It's for teams who struggle with misaligned and changing requirements, time wasted due to defects and rework, and product quality. It's for practitioners whose job is to help these teams, and it's for everyone who shares a passion for discovering better ways to build and deliver software.

Different people will get different things out of this book:

- *Business analysts and testers* will learn more effective ways of discovering requirements in collaboration with users, and of communicating these requirements to development teams.
- *Developers* will learn how to write higher-quality, more maintainable code with fewer bugs, how to focus on writing code that delivers real value, and how to build automated test suites that provide documentation and feedback for the whole team.

- *Project managers and business stakeholders* will learn how to help teams build better, more valuable software for the business.

How the book is organized

The book is divided into four parts, each addressing different aspects of BDD:

- *Part 1* presents the motivations, origins, and general philosophy of BDD, and concludes with a quick practical introduction to what BDD looks like in the real world.
- *Part 2* discusses how BDD practices can help teams analyze requirements more effectively in order to discover and describe what features will deliver real value to the organization. This section lays the conceptual foundation for the rest of the book and presents a number of important requirements-analysis techniques.
- *Part 3* provides more technical coverage of BDD practices. We'll look at techniques for automating acceptance tests in a robust and sustainable way, study a number of BDD tools for different languages and frameworks, and see how BDD helps developers write cleaner, better-designed, higher-quality code. This section is hands-on and practical.
- *Part 4* looks the broader picture of BDD in the context of project management, product documentation, reporting, and integration into the build process.

Many of the practical examples in the book will use JVM-based languages and tools, but we'll also look at examples of BDD tools for .NET, Ruby, Python, and JavaScript. The approaches I'll describe will be generally applicable to any language.

Because of the broad focus of the book, you may find different sections more or less applicable to your daily work. For example, business analysts might find the material on requirements analysis more relevant than the chapters on coding practices. Table 1 presents a (very) rough guide to the sections various readers might find particularly useful.

Table 1 A rough indicator of the target audience for each section of this book

	Business analyst	Tester	Developer	Project manager
Part 1	✓ ✓ ✓	✓ ✓ ✓	✓ ✓ ✓	✓ ✓ ✓
Part 2	✓ ✓ ✓	✓ ✓ ✓	✓ ✓	✓ ✓ ✓
Part 3	✓	✓ ✓	✓ ✓ ✓	✓
Part 4	✓ ✓ ✓	✓ ✓ ✓	✓ ✓	✓ ✓

Prerequisites

The prerequisites for *BDD in Action* will vary depending on the parts of the book being read.

Parts 1 and 2 (high-level BDD)—These sections require little technical knowledge—they are aimed at all team members, and introduce general principles of BDD. A basic understanding of Agile development practices will be helpful.

Part 3 (low-level BDD/TDD)—This section requires programming knowledge. Many examples are Java-based, but there are also examples in C#, Python, and JavaScript. The general approach is to illustrate concepts and practices with working code, rather than to document any one technology exhaustively. Different technology sections will benefit from a working knowledge of the following technologies:

- *Maven*—The Java/JVM code samples use Maven, though only a superficial knowledge (the ability to build a Maven project) is required.
- *HTML/CSS*—The sections on UI testing using Selenium/WebDriver need a basic understanding of how HTML pages are built, what a CSS selector looks like, and, optionally, some familiarity with XPath.
- *Restful web services*—The sections on testing web services need some understanding of how web services are implemented, and in particular how web service clients are implemented.
- *JavaScript*—The section on testing JavaScript and JavaScript applications requires a reasonable understanding of JavaScript programming.

Part 4 (Taking BDD further)—This section is general, and has no real technical requirements.

Code conventions

This book contains many source code examples illustrating the various tools and techniques discussed. Source code in listings or in the text appears in a fixed-width font like this. Other related concepts that appear in the text, such as class or variable names, also appear in this font.

Because this book discusses many languages, I've made a special effort to keep all of the listings readable and easy to follow, even if you're not familiar with the language being used. Most of the listings are annotated to make the code easier to follow, and some also have numbered cueballs, indicating particular lines of code that are discussed in the text that follows.

Source code downloads

This book contains many source code examples in a variety of languages. The source code for these examples is available for download on GitHub at <https://github.com/bdd-in-action>, with a separate subdirectory for each chapter. Some examples are discussed across several chapters—in these cases, each chapter contains the version of the source code discussed in that chapter. The source code can also be downloaded from the publisher's website at www.manning.com/BDDinAction.

The source code projects don't contain IDE-specific project files, but they're laid out in such a way as to make it easy to import them into an IDE.

Solutions to exercises and other resources

There are a number of exercises throughout this book. While they're primarily designed to get you thinking about the topics under discussion, sample solutions are available on the book's website at <http://bdd-in-action.com>. This site also contains links to the tools and libraries used in the book, and other useful related resources

Author Online

Purchase of *BDD in Action* includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the author and from other users. To access the forum and subscribe to it, point your web browser to www.manning.com/smart.

This page provides information on how to get on the forum once you're registered, what kind of help is available, and the rules of conduct on the forum. Manning's commitment to our readers is to provide a venue where a meaningful dialog between individual readers and between readers and the author can take place. It's not a commitment to any specific amount of participation on the part of the author, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the author some challenging questions lest his interest stray!

The Author Online forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

about the cover illustration

The figure on the cover of *BDD in Action* is captioned “A Medieval Knight.” The illustration by Paolo Mercuri (1804–1884) is taken from “Costumes Historiques,” a multi-volume compendium of historical costumes from the twelfth, thirteenth, fourteenth, and fifteenth centuries assembled and edited by Camille Bonnard and published in Paris in the 1850s or 1860s. The nineteenth century saw an increased interest in exotic locales and in times gone by, and people were drawn to collections such as this one to explore the world they lived in—as well as the world of the distant past.

The colorful variety of Mercuri’s illustrations in this historical collection reminds us vividly of how culturally apart the world’s towns and regions were a few hundred years ago. In the streets or in the countryside people were easy to place—sometimes with an error of no more than a dozen miles—just by their dress. Their station in life, as well as their trade or profession, could be easily identified. Dress codes have changed over the centuries, and the diversity by region, so rich at one time, has faded away. Today, it is hard to tell apart the inhabitants of one continent from another, let alone the towns or countries they come from, or their social status or profession. Perhaps we have traded cultural diversity for a more varied personal life—certainly a more varied and faster-paced technological life.

At a time when it is hard to tell one computer book from another, Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional life of many centuries ago, brought back to life by Mercuri’s pictures.

Part 1

First steps

Welcome to the world of BDD! The aim of part 1 of this book is to give you both a 50,000-foot-high view of the world of Behavior-Driven Development, and also a first taste of what BDD looks like in the field.

In chapter 1, you'll learn about the motivations and origins of Behavior-Driven Development and where it sits with regard to Agile and other software development approaches. You'll discover the broad scope of BDD, learning how it applies at all levels of software development, from high-level requirements discovery and specification to detailed low-level coding. And you'll learn how important it is not only to build the software right, but also to build the right software.

As practitioners, we like to keep things grounded in real-world examples, so in chapter 2 you'll see what BDD looks like in a real project, all the way from discovering the requirements and automating the high-level acceptance criteria to building and verifying the design and implementation, through to producing accurate and up-to-date technical and functional documentation.

By the end of part 1, you should have a good grasp of the motivations and overall broad scope of BDD, as well as an idea of what it looks like in practice at different levels of the software development process.

Building software that makes a difference

This chapter covers

- The problems that Behavior-Driven Development addresses
- General principles and origins of Behavior-Driven Development
- Activities and outcomes seen in a Behavior-Driven Development project
- The pros and cons of Behavior-Driven Development

This book is about building and delivering better software. It's about building software that works well and is easy to change and maintain, but more importantly, it's about building software that provides real value to its users. We want to build software well, but we also need to build software that's worth building.

In 2012, the U.S. Air Force decided to ditch a major software project that had already cost over \$1 billion USD. The Expeditionary Combat Support System was designed to modernize and streamline supply chain management in order to save billions of dollars and meet new legislative requirements. But after seven years of development, the system had still “not yielded any significant military capability.”¹

¹ Chris Kanaracus, “Air Force scraps massive ERP project after racking up \$1 billion in costs,” *CIO*, November 14, 2012, <http://www.cio.com/article/2390341>.

The Air Force estimated that an additional \$1.1 billion USD would be required to deliver just a quarter of the original scope, and that the solution could not be rolled out until 2020, three years after the legislative deadline of 2017.

This happens a lot in the software industry. According to a number of studies, around half of all software projects fail to deliver in some significant way. The 2011 edition of the Standish Group's annual *CHAOS Report* found that 42% of projects were delivered late, ran over budget, or failed to deliver all of the requested features,² and 21% of projects were cancelled entirely. Scott Ambler's annual survey on IT project success rates uses a more flexible definition of success, but still found a 30–50% failure rate, depending on the methodologies used.³ This corresponds to billions of dollars in wasted effort, writing software that ultimately won't be used or that doesn't solve the business problem it was intended to solve.

What if it didn't have to be this way? What if we could write software in a way that would let us discover and focus our efforts on what really matters? What if we could objectively learn what features will really benefit the organization and the most cost-effective way to implement them? What if we could see beyond what the user asks for and build what the user actually needs?

There are organizations discovering how to do just that. Many teams are successfully collaborating to build and deliver more valuable, more effective, and more reliable software. And they're learning to do this faster and more efficiently. In this book, you'll see how—we'll explore a number of methods and techniques, grouped under the general heading of *Behavior-Driven Development* (BDD).

BDD helps teams focus their efforts on identifying, understanding, and building valuable features that matter to businesses, and it makes sure that these features are well designed and well implemented.

BDD practitioners use conversations around concrete examples of system behavior to help understand how features will provide value to the business. BDD encourages business analysts, software developers, and testers to collaborate more closely by enabling them to express requirements in a more testable way, in a form that both the development team and business stakeholders can easily understand. BDD tools can help turn these requirements into automated tests that help guide the developer, verify the feature, and document what the application does.

BDD isn't a software development methodology in its own right. It's not a replacement for Scrum, XP, Kanban, RUP, or whatever methodology you're currently using. As you'll see, BDD incorporates, builds on, and enhances ideas from many of these methodologies. And no matter what methodology you're using, there are ways that BDD can help make your life easier.

² Whether these figures reflect more on our ability to build and deliver software or on our ability to plan and estimate is a subject of some debate in the Agile development community—see Jim Highsmith's book *Agile Project Management: Creating Innovative Products*, second edition (Addison-Wesley Professional, 2009).

³ Scott Ambler, *Surveys Exploring the Current State of Information Technology Practices*, <http://www.ambysoft.com/surveys/>.

1.1 BDD from 50,000 feet

So what does BDD bring to the table? Here's a (slightly oversimplified) perspective. Let's say Chris's company needs a new module for its accounting software. When Chris wants to add a new feature, the process goes something like this (see figure 1.1):

- 1 Chris tells a business analyst how he would like the feature to work.
- 2 The business analyst translates Chris's requests into a set of requirements for the developers, describing what the software should do. These requirements are written in English and stored in a Microsoft Word document.
- 3 The developer translates the requirements into code and unit tests—written in Java, C#, or some other programming language—in order to implement the new feature.
- 4 The tester translates the requirements in the Word document into test cases, and uses them to verify that the new feature meets the requirements.
- 5 Documentation engineers then translate the working software and code back into plain English technical and functional documentation.

There are many opportunities for information to get lost in translation, be misunderstood, or just be ignored. Chances are that the new module itself may not do exactly what was required and that the documentation won't reflect the initial requirements that Chris gave the analyst.

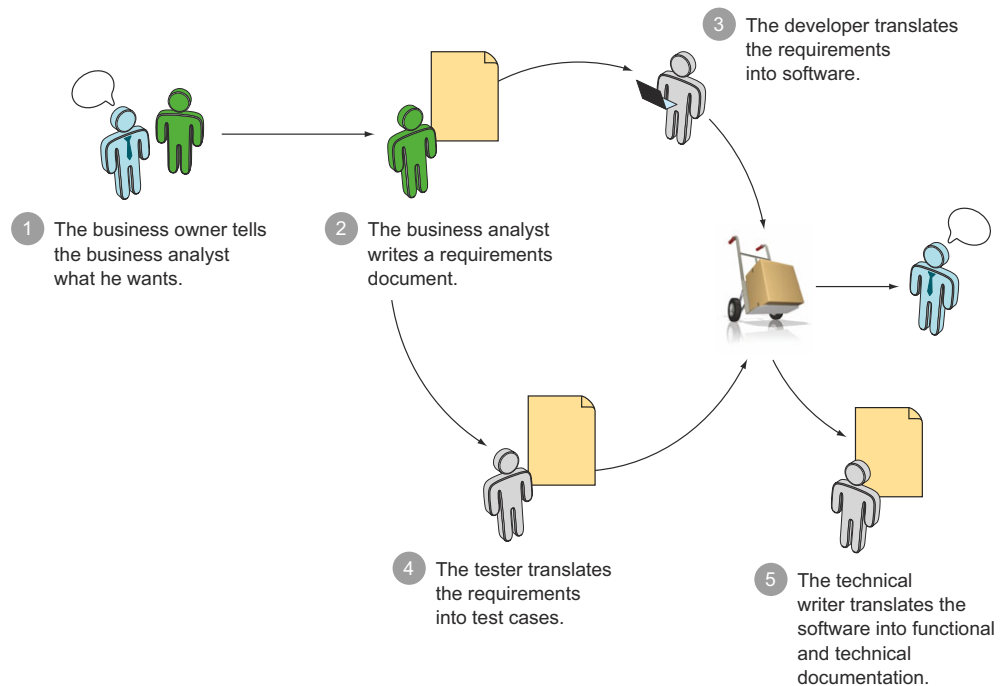


Figure 1.1 The traditional development process provides many opportunities for misunderstandings and miscommunication.

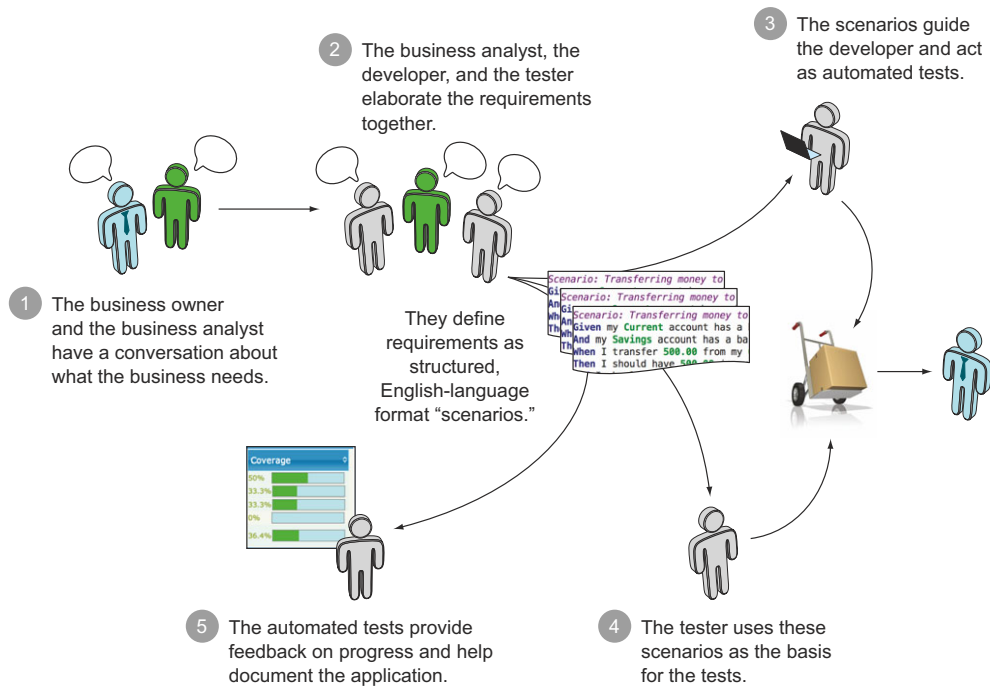


Figure 1.2 BDD uses conversations around examples, expressed in a form that can be easily automated, to reduce lost information and misunderstandings.

Chris's friend Sarah runs another company that just introduced BDD. In a team practicing BDD, the business analysts, developers, and testers collaborate to understand and define the requirements together (see figure 1.2). They use a common language that allows for an easy, less ambiguous path from end-user requirements to usable, automatable tests. These tests specify how the software should behave, and they guide the developers in building working software that focuses on features that really matter to the business.

- 1** Like Chris, Sarah talks to a business analyst about what she wants. To reduce the risk of misunderstandings and hidden assumptions, they talk through concrete examples of what the feature should do.
- 2** Before work starts on the feature, the business analyst gets together with the developer and tester who will be working on it, and they have a conversation about the feature. In this conversation, they discuss and translate key examples of how the feature should work into a set of requirements written in a structured, English-language format often referred to as Gherkin.
- 3** The developer uses a BDD tool to turn these requirements into a set of automated tests that run against the application code and help objectively determine when a feature is finished.

- ④ The tester uses the results of these tests as the starting point for manual and exploratory tests.
- ⑤ The automated tests act as low-level technical documentation, and provide up-to-date examples of how the system works. Sarah can review the test reports to see what features have been delivered, and whether they work the way she expected.

Compared to Chris's scenario, Sarah's team makes heavy use of conversations and examples to reduce the amount of information lost in translation. Every stage beyond step 2 starts with the specifications written in Gherkin, which are based on concrete examples provided by Sarah. In this way, a great deal of the ambiguity in translating the client's initial requirements into code, reports, and documentation is removed.

We'll discuss all of these points in detail throughout the rest of the book. You'll learn ways to help ensure that your code is of high quality, solid, well tested, and well documented. You'll learn how to write more effective unit tests and more meaningful automated acceptance criteria. You'll also learn how to ensure that the features you deliver solve the right problems and provide real benefit to the users and the business.

1.2 What problems are you trying to solve?

Software projects fail for many reasons, but the most significant causes fall into two broad categories:

- Not building the software right
- Not building the right software

Figure 1.3 illustrates this in the form of a graph. The vertical axis represents *what* you're building, and the horizontal axis represents *how* you build it. If you perform poorly on the *how* axis, not writing well-crafted and well-designed software, you'll end up with a buggy, unreliable product that's hard to change and maintain. If you don't do well on the *what* axis, failing to understand what features the business really needs, you'll end up with a product that nobody needs.

1.2.1 Building the software right

Many projects suffer or fail because of software quality issues. Although internal software quality is mostly invisible to nontechnical stakeholders, the consequences of poor-quality software can be painfully visible. In my experience, applications that are poorly designed, badly written, or lack well-written, automated tests tend to be buggy, hard to maintain, hard to change, and hard to scale.

I've seen too many applications where simple change requests and new features take too long to deliver. Developers spend more and more time fixing bugs rather than working on new features, which makes it harder to deliver new features quickly. It takes longer for new developers to get up to speed and become productive, simply because the code is hard to understand. It also becomes harder and harder to add new features without breaking existing code. The existing technical documentation (if there is any) is inevitably out of date, and teams find themselves incapable of

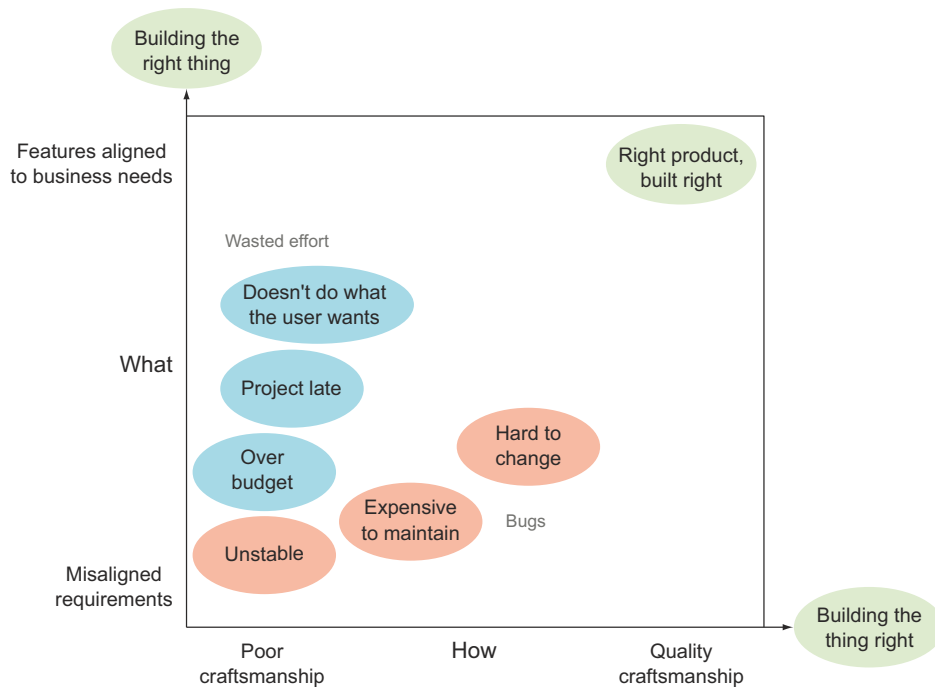


Figure 1.3 Successful projects must both build features well and build the right features.

delivering new features quickly because each release requires a lengthy period of manual testing and bug fixes.

Organizations that embrace high-quality technical practices have a different story to tell. I've seen many teams that adopt practices such as Test-Driven Development, Clean Coding, Living Documentation, and Continuous Integration regularly reporting low to near-zero defect rates, as well as code that's much easier to adapt and extend as new requirements emerge and new features are requested. These teams can also add features at a more consistent pace, because the automated tests ensure that existing features won't be broken unknowingly. They implement the features faster and more precisely than other teams because they don't have to struggle with long bug-fixing sessions and unpredictable side effects when they make changes. And the resulting application is easier and cheaper to maintain.

Note that there is no magic formula for building high-quality, easily maintainable software. Software development is a complex field, human factors abound, and techniques such as Test-Driven Development, Clean Coding, and Automated Testing don't automatically guarantee good results. But studies do suggest a strong correlation between lean and Agile practices and project success rates⁴ when compared to more

⁴ See, for example, Ambysoft, "2013 IT Project Success Rates Survey Results," <http://www.ambysoft.com/surveys/success2013.html>.

traditional approaches. Other studies have found a correlation between Test-Driven Development practices, reduced bug counts,⁵ and improved code quality.⁶ Although it's certainly possible to write high-quality code without practicing techniques such as Test-Driven Development and Clean Coding, teams that value good development practices do seem to succeed in delivering high-quality code more often.

But building high-quality software isn't in itself enough to guarantee a successful project. The software must also benefit its users and business stakeholders.

1.2.2 Building the right software

Software is never developed in a vacuum. Software projects are part of a broader business strategy, and they need to be aligned with business goals if they're to be beneficial to the organization. At the end of the day, the software solution you deliver needs to help users achieve their goals more effectively. Any effort that doesn't contribute to this end is waste.

In practice, there's often a lot of waste. In many projects, time and money are spent building features that are never used or that provide only marginal value to the business. According to the Standish Group's CHAOS studies,⁷ on average some 45% of the features delivered into production are never used. Even apparently predictable projects, such as migrating software from a mainframe system onto a more modern platform, have their share of features that need updating or that are no longer necessary. When you don't fully understand the goals that your client is trying to achieve, it's very easy to deliver perfectly functional, well-written features that are of little use to the end user.

On the other hand, many software projects end up delivering little or no real business value. Not only do they deliver features that are of little use to the business, but they fail to even deliver the minimum capabilities that would make the projects viable.

The consequences of not building it right, and not building the right thing

The impact of poorly understood requirements and poor code realization isn't just a theoretical concept or a "nice to have;" on the contrary, it's often painfully concrete. In December 2007, the Queensland Health Department kicked off work on a new payroll system for its 85,000 employees. The initial budget for the project was around \$6 million, with a delivery date of August 2008.

⁵ See, for example, Nachiappan Nagappan, E. Michael Maximilien, Thirumalesh Bhat, and Laurie Williams, "Realizing quality improvement through test driven development: results and experiences of four industrial teams," http://research.microsoft.com/en-us/groups/ese/nagappan_tdd.pdf.

⁶ Rod Hilton, "Quantitatively Evaluating Test-Driven Development by Applying Object-Oriented Quality Metrics to Open Source Projects" (PhD thesis, Regis University, 2009), http://www.rodhilton.com/files/tdd_thesis.pdf.

⁷ The Standish Group's *CHAOS Report 2002* reported a value of 45% and I've seen more recent internal studies where the figure is around 50%.

(continued)

When the solution was rolled out in 2010, some 18 months late, it was a disaster.⁸ Tens of thousands of public servants were underpaid, overpaid, or not paid at all. Since the go-live date, over 1,000 payroll staff have been required to carry out some 200,000 manual processes each fortnight to ensure that staff salaries are paid.

In 2012, an independent review found that the project had cost the state over \$416 million since going into production and would cost an additional \$837 million to fix. This colossal sum included \$220 million just to fix the immediate software issues that were preventing the system from delivering its core capability of paying Queensland Health staff what they were owed each month.

Building the right software is made even trickier by one commonly overlooked fact: early on in a project, you usually don't know what the right features are.

1.2.3 The knowledge constraint—dealing with uncertainty

One fact of life in software development is that there will be things you don't know. Changing requirements are a normal part of every software project. Knowledge and understanding about the problem at hand and about how best to solve it increases progressively throughout the project.

In software development, each project is different. There are always new business requirements to cater to, new technological problems to solve, and new opportunities to seize. As a project progresses, market conditions, business strategies, technological constraints, or simply your understanding of the requirements will evolve, and you'll need to change your tack and adjust your course. Each project is a journey of discovery where the real constraint isn't time, the budget, or even programmer hours, but your lack of knowledge about what you need to build and how you should build it. When reality doesn't go according to plan, you need to adapt to reality, rather than trying to force reality to fit into your plan. "When the terrain disagrees with the map, trust the terrain" (Swiss Army proverb).

Users and stakeholders will usually know what high-level goals they want to achieve and can be coaxed into revealing these goals if you take the time to ask. They'll be able to tell you that they need an online ticketing system or a payroll solution that caters to 85,000 different employees. And you can get a feel for the scope of the application you might need to build early on in the project.

But the details are another matter entirely. Although users are quick to ask for specific technical solutions to their problems, they're not usually the best-placed to know what solution would serve them best, or even, for that matter, what solutions exist. Your team's collective understanding of the best way to deliver these capabilities, as

⁸ See KPMG, "Review of the Queensland Health Payroll System" (2012), http://delimenter.com.au/wp-content/uploads/2012/06/KPMG_audit.pdf.

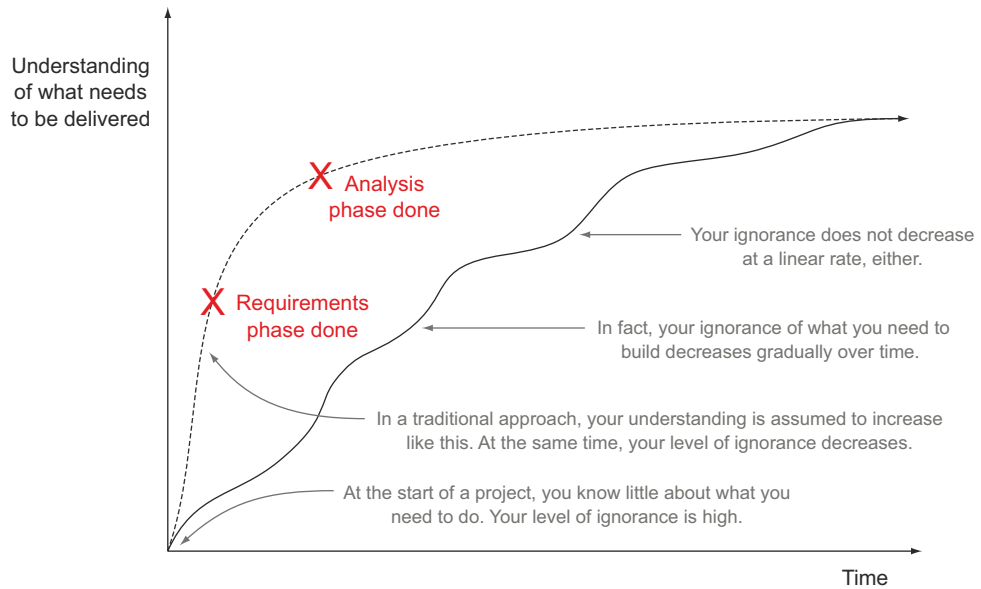


Figure 1.4 At the start of a project, there are many unknowns. You reduce these unknowns as the project progresses, but not in a linear or very predictable way.

well as the optimal feature set for achieving the underlying business goals, will grow as the project progresses.

As illustrated in figure 1.4, the more prescriptive, plan-based requirements-analysis techniques suppose that you can learn almost all there is to know about a project's requirements, as well as the optimal solution design, very quickly in the early phases of the project. By the end of the analysis phase, the specifications are signed-off on and locked down, and all that remains to do is code.

Of course, reality doesn't always work this way. At the start of the project, a development team will often have only a superficial understanding of the business domain and the goals the users need to achieve. In fact, the job of a software engineering team isn't to know how to build a solution; it's to know how to discover the best way to build the solution.

The team's collective understanding will naturally increase over the duration of the project. You become less ignorant over time. Toward the end of the project, a good team will have built up a deep, intimate knowledge of the user's needs and will be able to proactively propose features and implementations that will be better suited to the particular user base. But this learning path is neither linear nor predictable. It's hard to know what you don't know, so it's hard to predict what you'll learn as the project progresses.

For the majority of modern software development projects, the main challenge in managing scope isn't to eliminate uncertainty by defining and locking down requirements as early as possible. The main challenge is to manage this uncertainty in a way

that will help you progressively discover and deliver an effective solution that matches up with the underlying business goals behind a project. As you'll see, one important benefit of BDD is that it provides techniques that can help you manage this uncertainty and reduce the risk that comes with it.

1.3 *Introducing Behavior-Driven Development*

Behavior-Driven Development (BDD) is a set of software engineering practices designed to help teams build and deliver more valuable, higher quality software faster. It draws on Agile and lean practices including, in particular, Test-Driven Development (TDD) and Domain-Driven Design (DDD). But most importantly, BDD provides a common language based on simple, structured sentences expressed in English (or in the native language of the stakeholders) that facilitate communication between project team members and business stakeholders.

To better understand the motivations and philosophy that drive BDD practices, it's useful to understand where BDD comes from.

1.3.1 *BDD was originally designed as an improved version of TDD*

BDD was originally invented by Dan North⁹ in the early to mid-2000s as an easier way to teach and practice Test-Driven Development (TDD). TDD, invented by Kent Beck in the early days of Agile,¹⁰ is a remarkably effective technique that uses unit tests to specify, design, and verify application code.

When TDD practitioners need to implement a feature, they first write a failing test that describes, or specifies, that feature. Next, they write just enough code to make the test pass. Finally, they refactor the code to help ensure that it will be easy to maintain (see figure 1.5). This simple but powerful technique encourages developers to write cleaner, better-designed, easier-to-maintain code¹¹ and results in substantially lower defect counts.¹²

Despite its advantages, many teams still have difficulty adopting and using TDD effectively. Developers often have trouble knowing where to start or what tests they should write next. Sometimes TDD can lead developers to become too detail-focused, losing the broader picture of the business goals they're supposed to implement. Some teams also find that the large numbers of unit tests can become hard to maintain as the project grows in size.

In fact, many traditional unit tests, written with or without TDD, are tightly coupled to a particular implementation of the code. They focus on the method or function they're testing, rather than on what the code should do in business terms.

⁹ Dan North, "Introducing BDD," <http://dannorth.net/introducing-bdd/>.

¹⁰ Kent Beck, *Test-Driven Development: By Example* (Addison-Wesley Professional, 2002).

¹¹ Rod Hilton, "Quantitatively Evaluating Test-Driven Development by Applying Object-Oriented Quality Metrics to Open Source Projects" (PhD thesis, Regis University, 2009), http://www.rodhilton.com/files/tdd_thesis.pdf.

¹² Nachiappan Nagappan et al., "Realizing Quality Improvement through Test Driven Development" (2008), http://research.microsoft.com/en-us/groups/ese/nagappan_tdd.pdf.

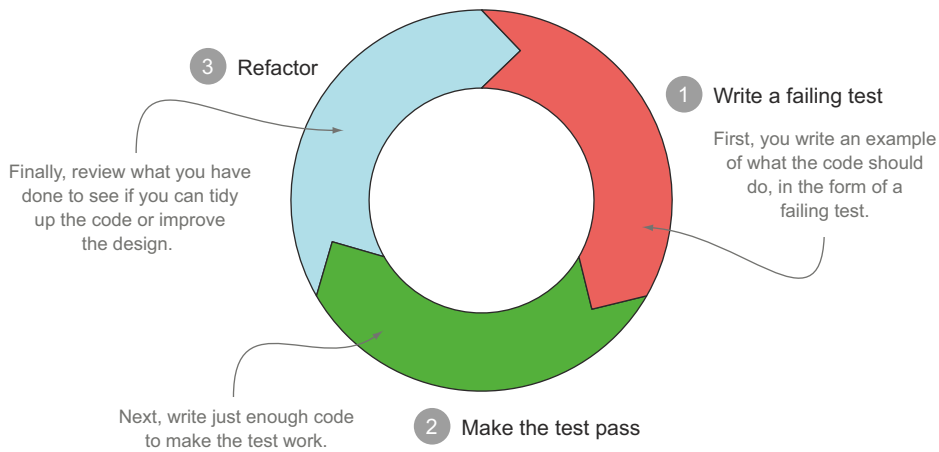


Figure 1.5 Test-Driven Development relies on a simple, three-phase cycle.

For example, suppose Paul is a Java developer working on a new financial trading application in a large bank. He has been asked to implement a new feature to transfer money from one account to another. He creates an `Account` class with a `transfer()` method, a `deposit()` method, and so on. The corresponding unit tests are focused on testing these methods:

```
public class BankAccountTest {
    @Test
    public void testTransfer() {...}

    @Test
    public void testDeposit() {...}
}
```

Tests like this are better than nothing, but they can limit your options. For example, they don't describe what you expect the `transfer()` and `deposit()` functions to do, which makes them harder to understand and to fix if they break. They're tightly coupled to the method they test, which means that if you refactor the implementation, you need to rename your test as well. And because they don't say much about what they're actually testing, it's hard to know what other tests (if any) you need to write before you're done.

North observed that a few simple practices, such as naming unit tests as full sentences and using the word "should," can help developers write more meaningful tests, which in turn helps them write higher quality code more efficiently. When you think in terms of what the class *should* do, instead of what method or function is being tested, it's easier to keep your efforts focused on the underlying business requirements.

For example, Paul could write more descriptive tests along the following lines:

```
public class WhenTransferringInternationalFunds {
    @Test
    public void should_transfer_funds_to_a_local_account() {...}
```



```

@Test
public void should_transfer_funds_to_a_different_bank() {...}
...

@Test
public void should_deduct_fees_as_a_separate_transaction() {...}
...
}

```

Tests that are written this way read more like specifications than unit tests. They focus on the behavior of the application, using tests simply as a means to express and verify that behavior. North also noted that tests written this way are much easier to maintain because their intent is so clear. The impact of this approach was so significant that he started referring to what he was doing no longer as Test-Driven Development, but as *Behavior-Driven Development*.

1.3.2 **BDD also works well for requirements analysis**

But describing a system's behavior turns out to be what business analysts do every day. Working with business analyst colleague Chris Matts, North set out to apply what he had learned to the requirements-analysis space. Around this time, Eric Evans introduced the idea of Domain-Driven Design,¹³ which promotes the use of a ubiquitous language that business people can understand to describe and model a system. North and Matts's vision was to create a ubiquitous language that business analysts could use to define requirements unambiguously, and that could also be easily transformed into automated acceptance tests. To implement this vision, they started expressing the acceptance criteria for user stories in the form of loosely structured examples, known as "scenarios," like this one:

```

Given a customer has a current account
When the customer transfers funds from this account to an overseas account
Then the funds should be deposited in the overseas account
And the transaction fee should be deducted from the current account

```

A business owner can easily understand a scenario written like this. It gives clear and objective goals for each story in terms of what needs to be developed and of what needs to be tested.

This notation eventually evolved into a commonly used form often referred to as Gherkin. With appropriate tools, scenarios written in this form can be turned into automated acceptance criteria that can be executed automatically whenever required. Dan North wrote the first dedicated BDD test automation library, JBehave, in the mid-2000s, and since then many others have emerged for different languages, both at the unit-testing and acceptance-testing levels.

¹³ Eric Evans, *Domain Driven Design* (Addison-Wesley Professional, 2003).

BDD by any other name

Many of the ideas around BDD are not new and have been practiced for many years under a number of different names. Some of the more common terms used for these practices include Acceptance-Test-Driven Development, Acceptance Test-Driven Planning, and Specification by Example. To avoid confusion, let's clarify a few of these terms in relation to BDD.

Specification by Example describes the set of practices that have emerged around using examples and conversation to discover and describe requirements. In his seminal book of the same name,¹⁴ Gojko Adzic chose this term as the most representative name to refer to these practices. Using conversation and examples to specify how you expect a system to behave is a core part of BDD, and we'll discuss it at length in the first half of this book.

Acceptance-Test-Driven Development (ATDD) is now a widely used synonym for Specification by Example, but the practice has existed in various forms since at least the late 1990s. Kent Beck and Martin Fowler mentioned the concept in 2000,¹⁵ though they observed that it was difficult to implement acceptance criteria in the form of conventional unit tests at the start of a project. But unit tests aren't the only way to write automated acceptance tests, and since at least the early 2000s, innovative teams have asked users to contribute to executable acceptance tests and have reaped the benefits.¹⁶

Acceptance-Test-Driven Planning is the idea that defining acceptance criteria for a feature leads to better estimates than doing a task breakdown.

1.3.3 BDD principles and practices

Today BDD is successfully practiced in a large number of organizations of all sizes around the world, in a variety of different ways. In *Specification by Example*, Gojko Adzic provides case studies for over 50 such organizations. In this section, we'll look at a number of general principles or guidelines that BDD practitioners have found useful over the years.

Figure 1.6 gives a high-level overview of the way BDD sees the world. BDD practitioners like to start by identifying business goals and looking for features that will help deliver these goals. Collaborating with the user, they use concrete examples to illustrate these features. Wherever possible, these examples are automated in the form of executable specifications, which both validate the software and provide automatically updated technical and functional documentation. BDD principles are also used at the coding level, where they help developers write code that's of higher quality, better tested, better documented, and easier to use and maintain.

¹⁴ Gojko Adzic, *Specification by Example* (Manning, 2011).

¹⁵ Kent Beck and Martin Fowler, *Planning Extreme Programming* (Addison-Wesley Professional, 2000).

¹⁶ Johan Andersson et al., "XP with Acceptance-Test Driven Development: A Rewrite Project for a Resource Optimization System," *Lecture Notes in Computer Science Volume 2675* (2003). Available at <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.108.6097&rep=rep1&type=pdf>.

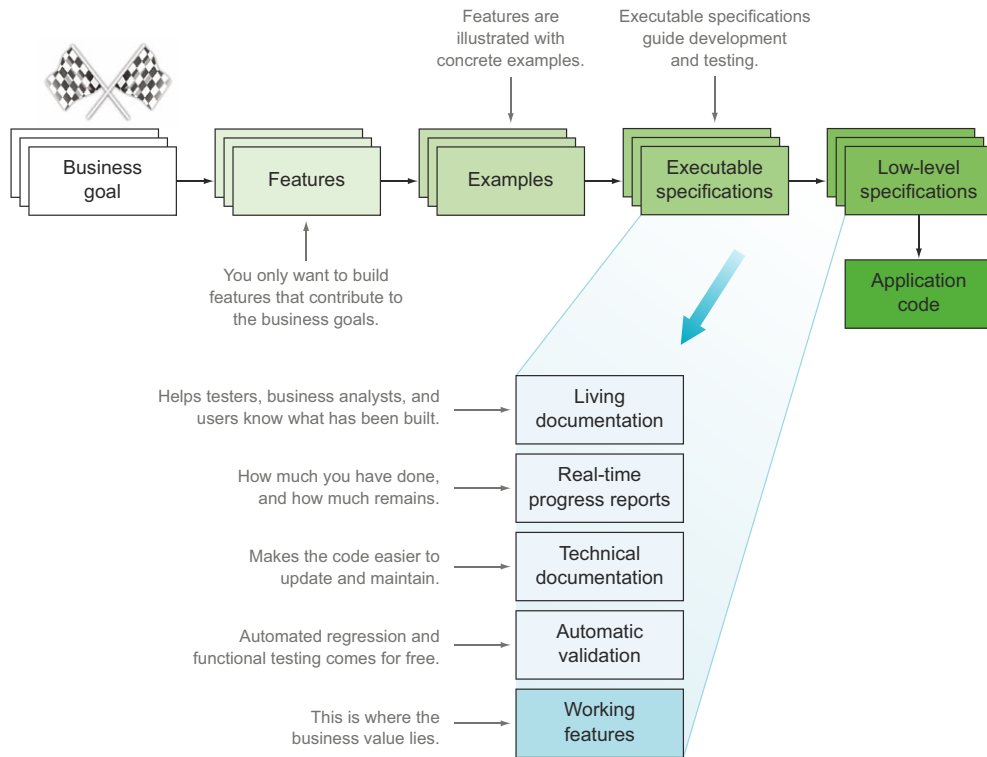


Figure 1.6 The principal activities and outcomes of BDD. Note that these activities occur repeatedly and continuously throughout the process; this isn't a single linear Waterfall-style process, but a sequence of activities that you practice for each feature you implement.

In the following sections, we'll look at how these principles work in more detail.

FOCUS ON FEATURES THAT DELIVER BUSINESS VALUE

As you've seen, uncertainty about requirements is a major challenge in many software projects, and heavy upfront specifications don't work particularly well when confronted with a shifting understanding of what features need to be delivered.

A *feature* is a tangible, deliverable piece of functionality that helps the business to achieve its business goals. For example, suppose you work in a bank that's implementing an online banking solution. One of the business goals for this project might be "to attract more clients by providing a simple and convenient way for clients to manage their accounts." Some features that might help achieve this goal could be "Transfer funds between a client's accounts," "Transfer funds to another national account," or "Transfer funds to an overseas account."

Rather than attempting to nail down all of the requirements once and for all, teams practicing BDD engage in ongoing conversations with the end users and other stakeholders to progressively build a common understanding of what features they should create. Rather than working upfront to design a complete solution for

the developers to implement, users explain what they need to get out of the system and how it might help them achieve their objectives. And rather than accepting a list of feature requests from the users with no questions asked, teams try to understand the core business goals underlying the project, proposing only features that can be demonstrated to support these business goals. This constant focus on delivering business value means that teams can deliver more useful features earlier and with less wasted effort.

WORK TOGETHER TO SPECIFY FEATURES

A complex problem, like discovering ways to delight clients, is best solved by a cognitively diverse group of people that is given responsibility for solving the problem, self-organizes, and works together to solve it.

Stephen Denning, *The Leader's Guide to Radical Management*
(Jossey-Bass, 2010)

BDD is a highly collaborative practice, both between users and the development team, and within the team itself. Business analysts, developers, and testers work together with the end users to define and specify features, and team members draw ideas from their individual experience and know-how. This approach is highly efficient.

In a more traditional approach, when business analysts simply relay their understanding of the users' requirements to the rest of the team, there is a high risk of misinterpretation and lost information.

If you ask users to write up what they want, they'll typically give you a set of detailed requirements that matches how they envisage the solution. In other words, users will not tell you *what they need*; rather, *they'll design a solution for you*. I've seen many business analysts fall into the same trap, simply because they've been trained to write specifications that way. The problem with this approach is twofold: not only will they fail to benefit from the development team's expertise in software design, but they're effectively binding the development team to a particular solution, which may not be the optimal one in business or technical terms. In addition, developers can't use their technical know-how to help deliver a technically superior design, and testers don't get the opportunity to comment on the testability of the specifications until the end of the project.

For example, the "Transfer funds to an overseas account" feature involves many user-experience and technical considerations. How can you display the constantly changing exchange rates to the client? When and how are the fees calculated and shown to the client? For how long can you guarantee a proposed exchange rate? How can you verify that the right exchange rate is being used? All of these considerations will have an impact on the design, implementation, and cost of the feature and can change the way the business analysts and business stakeholders originally imagined the solution.

When teams practice BDD, on the other hand, team members build up a shared appreciation of the users' needs, as well as a sense of common ownership and engagement in the solution.

EMBRACE UNCERTAINTY

A BDD team knows that they won't know everything upfront, no matter how long they spend writing specifications. As we discussed earlier, the biggest thing slowing developers down in a software project is understanding what they need to build.

Rather than attempting to lock down the specifications at the start of the project, BDD practitioners assume that the requirements, or more precisely, their *understanding* of the requirements, will evolve and change throughout the life of a project. They try to get early feedback from the users and stakeholders to ensure that they're on track, and change tack accordingly, instead of waiting until the end of the project to see if their assumptions about the business requirements were correct.

Very often, the most effective way to see if users like a feature is to build it and show it to them as early as possible. With this in mind, experienced BDD teams prioritize the features that will deliver value, will improve their understanding of what features the users really need, and will help them understand how best to build and deliver these features.

ILLUSTRATE FEATURES WITH CONCRETE EXAMPLES

When a team practicing BDD decides to implement a feature, they work together with users and other stakeholders to define stories and scenarios of what users expect this feature to deliver. In particular, the users help define a set of concrete examples that illustrate key outcomes of the feature (see figure 1.7).

These examples use a common vocabulary and can be readily understood by both end users and members of the development team. They're usually expressed using the *Given ... When ... Then* notation you saw in section 1.3.2. For instance, a simple example that illustrates the “Transfer funds between a client's accounts” feature might look like this:

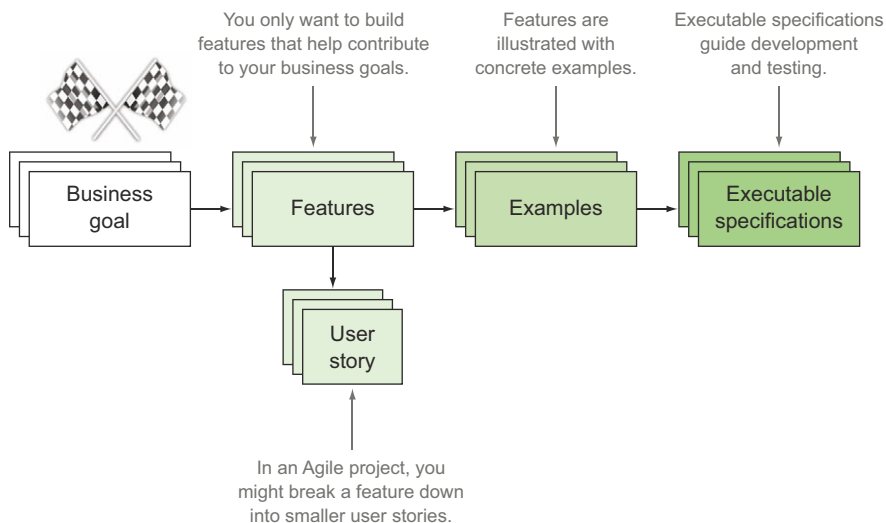


Figure 1.7 Examples play a primary role in BDD, helping everyone understand the requirements more clearly.

```

Scenario: Transferring money to a savings account
  Given I have a current account with 1000.00
  And I have a savings account with 2000.00
  When I transfer 500.00 from my current account to my savings account
  Then I should have 500.00 in my current account
  And I should have 2500.00 in my savings account

```

Examples play a primary role in BDD, simply because they're an extremely effective way of communicating clear, precise, and unambiguous requirements. Specifications written in natural language are, as it turns out, a terribly poor way of communicating requirements, because there's so much space for ambiguity, assumptions, and misunderstandings. Examples are a great way to overcome these limitations and clarify the requirements.

Examples are also a great way to explore and expand your knowledge. When a user proposes an example of how a feature should behave, project team members often ask for extra examples to illustrate corner cases, explore edge cases, or clarify assumptions. Testers are particularly good at this, which is why it's so valuable for them to be involved at this stage of the project.

A Gherkin primer

Most BDD tools that we'll look at in this book use a format generally known as Gherkin, or a very close variation on this format used by JBehave.¹⁷ This format is designed to be both easily understandable for business stakeholders and easy to automate using dedicated BDD tools such as Cucumber and JBehave. This way, it both documents your requirements and runs your automated tests.

In Gherkin, the requirements related to a particular feature are grouped into a single text file called a *feature file*. A feature file contains a short description of the feature, followed by a number of scenarios, or formalized examples of how a feature works.

```

Feature: Transferring money between accounts
  In order to manage my money more efficiently
  As a bank client
  I want to transfer funds between my accounts whenever I need to

Scenario: Transferring money to a savings account
  Given my Current account has a balance of 1000.00
  And my Savings account has a balance of 2000.00
  When I transfer 500.00 from my Current account to my Savings account
  Then I should have 500.00 in my Current account
  And I should have 2500.00 in my Savings account
Scenario: Transferring with insufficient funds
  Given my Current account has a balance of 1000.00
  And my Savings account has a balance of 2000.00
  When I transfer 1500.00 from my Current account to my Savings account
  Then I should receive an 'insufficient funds' error
  Then I should have 1000.00 in my Current account
  And I should have 2000.00 in my Savings account

```

¹⁷ Strictly speaking, Gherkin refers to the format recognized by the Cucumber family of BDD automation tools (see <http://cukes.info>). For simplicity, we'll use the term *Gherkin* to refer to both variations, and I'll indicate any differences as we come across them.

(continued)

As can be seen here, Gherkin requirements are expressed in plain English, but with a specific structure. Each *scenario* is made up of a number of *steps*, where each step starts with one of a small number of keywords (*Given*, *When*, *Then*, *And*, and *But*).

The natural order of a scenario is *Given ... When ... Then*:

- *Given* describes the preconditions for the scenario and prepares the test environment.
- *When* describes the action under test.
- *Then* describes the expected outcomes.

The *And* and *But* keywords can be used to join several *Given*, *When*, or *Then* steps together in a more readable way:

Given I have a current account with \$1000

And I have a savings account with \$2000

Several related scenarios can often be grouped into a single scenario using a table of examples. For example, the following scenario illustrates how interest is calculated on different types of accounts:

Scenario Outline: Earning interest

Given I have an account of type <account-type> with a balance of
<initial-balance>

When the monthly interest is calculated

Then I should have earned at an annual interest rate of <interest-rate>

And I should have a new balance of <new-balance>

Examples:

initial-balance	account-type	interest-rate	new-balance
10000	current	1	10008.33
10000	savings	3	10025
10000	supersaver	5	10041.67

This scenario would be run three times in all, once for each row in the Examples table. The values in each row are inserted into the placeholder variables, which are indicated by the <...> notation (<account-type>, <initial-balance>, and so forth). This not only saves typing, but also makes it easier to understand the whole requirement at a glance.

You can also use the following tabular notation within the steps themselves in order to display test data more concisely. For example, the previous money-transfer scenario could have been written like this:

Scenario: Transferring money between accounts within the bank

Given I have the following accounts:

account	balance
current	1000
savings	2000

When I transfer 500.00 from current to savings

Then my accounts should look like this:

account	balance
current	500
savings	2500

We'll look at this notation in much more detail in chapter 5.

DON'T WRITE AUTOMATED TESTS, WRITE EXECUTABLE SPECIFICATIONS

These stories and examples form the basis of the specifications that developers use to build the system. They act as both acceptance criteria, determining when a feature is done, and as guidelines for developers, giving them a clear picture of what needs to be built.

Acceptance criteria give the team a way to objectively judge whether a feature has been implemented correctly. But checking this manually for each code change would be time-consuming and inefficient. It would also slow down feedback, which would in turn slow down the development process. Wherever feasible, teams turn these acceptance criteria into automated acceptance tests or, more precisely, into *executable specifications*.

An executable specification is an automated test that illustrates and verifies how the application delivers a specific business requirement. These automated tests run as part of the build process and run whenever a change is made to the application. In this way, they serve both as acceptance tests, determining which new features are complete, and as regression tests, ensuring that new changes haven't broken any existing features (see figure 1.8).

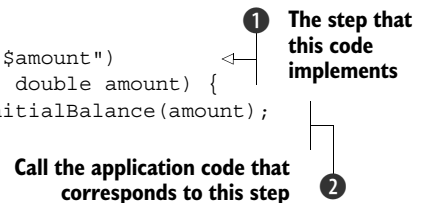
You can automate an executable specification by writing test code corresponding to each step. BDD tools like Cucumber and JBehave will match the text in each step of your scenario to the appropriate test code.

For example, this is the first step of the scenario in figure 1.8:

Given my Current account has a balance of 1000.00

You might automate this step in Java using JBehave with code like this:

```
@Given("my $accountType account has a balance of $amount")
public void setupInitialAccount(AccountType type, double amount) {
    Account account = Account.ofType(type).withInitialBalance(amount);
    accountService.create(account);
}
```



When JBehave runs the scenario, it'll execute each step of the scenario, using basic pattern matching to find the method associated with this step ①. Once it knows what method to call, it'll extract variables like `type` and `amount` and execute the corresponding application code ②.

Unlike conventional unit or integration tests, or the automated functional tests many QA teams are used to, executable specifications are expressed in something close to natural language. They use precisely the examples that the users and development team members proposed and refined earlier on, using exactly the same terms and vocabulary. Executable specifications are about communication as much as they are about validation, and the test reports they generate are easily understandable by everyone involved with the project.

These executable specifications also become a single source of truth, providing reference documentation for how features should be implemented. This makes

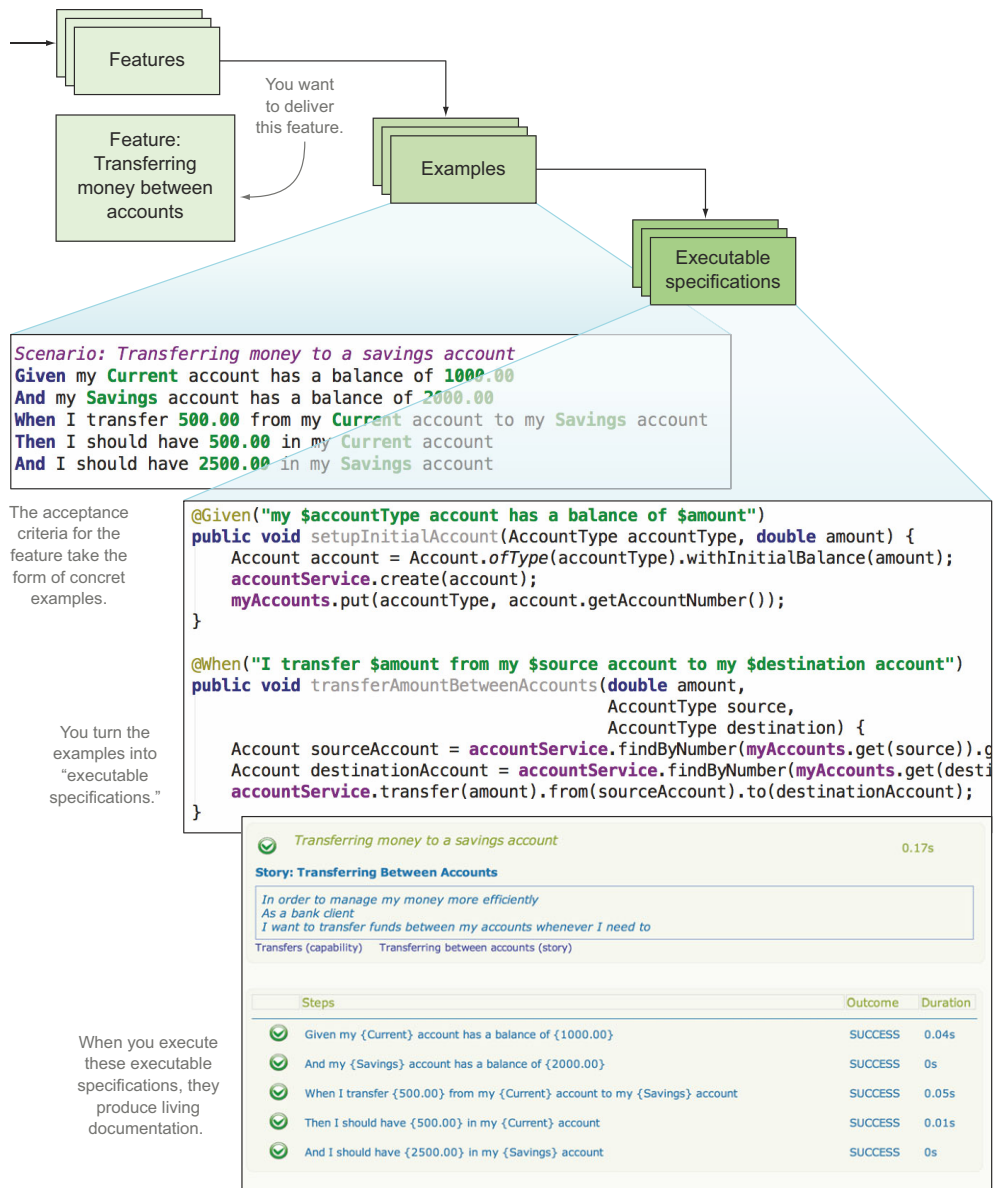


Figure 1.8 Executable specifications are expressed using a common business vocabulary that the whole team can understand. They guide development and testing activities and produce readable reports available to all.

maintaining the requirements much easier. If specifications are stored in the form of a Word document or on a Wiki page, as is done for many traditional projects, any changes to the requirements need to be reflected both in the requirements document and in the acceptance tests and test scripts, which introduces a high risk of

inconsistency. For teams practicing BDD, the requirements and executable specifications are the same thing; when the requirements change, the executable specifications are updated directly in a single place. We'll look at this in detail in chapter 9.

DON'T WRITE UNIT TESTS, WRITE LOW-LEVEL SPECIFICATIONS

BDD doesn't stop at the acceptance tests. BDD also helps developers write higher quality code that's more reliable, more maintainable, and better documented.

Developers practicing BDD typically use an *outside-in* approach. When they implement a feature, they start from the acceptance criteria and work down, building whatever is needed to make those acceptance criteria pass. The acceptance criteria define the expected outcomes, and the developer's job is to write the code that produces those outcomes. This is a very efficient, focused way of working. Just as no feature is implemented unless it contributes to an identified business goal, no code is written unless it contributes to making an acceptance test pass, and therefore to implementing a feature.

But it doesn't stop there. Before writing any code, a BDD developer will reason about what this code should actually do and express this in the form of a *low-level executable specification*. The developer won't think in terms of writing unit tests for a particular class, but of writing technical specifications describing how the application should behave, such as how it should respond to certain inputs or what it should do in a given situation. These low-level specifications flow naturally from the high-level acceptance criteria, and help developers design and document the application code in the context of delivering high-level features (see figure 1.9).

For example, the step definition code in figure 1.9 involves creating a new account:

```
@Given("my $accountType account has a balance of $amount")
public void setupInitialAccount(AccountType type, double amount) {
    Account account = Account.ofType(type)
                           .withInitialBalance(amount);
    ...
}
```

Create a new account of given type and with given initial balance.

This leads the developer to write a low-level specification to design the Account class. This example uses Spock, a BDD unit testing library for Java and Groovy. The corresponding specification takes the following form:

```
class WhenCreatingANewAccount extends Specification {

    def "account should have a type and an initial balance" () {
        when:
            Account account = Account.ofType(Savings)
                                   .withInitialBalance(100)

        then:
            account.accountType == Savings
            account.balance == 100
    }
}
```

High-level acceptance criteria in the form of *executable specifications*.

Scenario: Transferring money to a savings account

Given my **Current** account has a balance of **1000.00**

And my **Savings** account has a balance of **2000.00**

When I transfer **500.00** from my **Current** account to my **Savings** account

Then I should have **500.00** in my **Current** account

And I should have **2500.00** in my **Savings** account

Step definitions
call application
code to
implement steps
in the
acceptance
criteria.

```
@Given("my $accountType account has a balance of $amount")
public void setupInitialAccount(AccountType accountType, double amount) {
    Account account = Account.ofType(accountType).withInitialBalance(amount);
    accountService.create(account);
    myAccounts.put(accountType, account.getAccountNumber());
}

@When("I transfer $amount from my $source account to my $destination account")
public void transferAmountBetweenAccounts(double amount,
    AccountType source,
    AccountType destination) {
    Account sourceAccount = accountService.findByNumber(myAccounts.get(source)).getAccount();
    Account destinationAccount = accountService.findByNumber(myAccounts.get(destination)).getAccount();
    accountService.transfer(amount).from(sourceAccount).to(destinationAccount);
}
```

Low-level executable specifications (*unit tests*) help design the detailed implementation.

```
class WhenCreatingANewAccount extends Specification {

    def "account should have a number, a type and an initial balance"() {
        when:
            Account account = Account.ofType(Savings)
                .withInitialBalance(100)

        then:
            account.accountType == Savings
            account.balance == 100
    }
}
```

Figure 1.9 Low-level specifications, written as unit tests, flow naturally from the high-level specifications.

You could also write this specification using conventional unit-testing tools, such as JUnit or NUnit, or more specialized BDD tools such as RSpec (see figure 1.10).

Executable specifications like this are similar to conventional unit tests, but they're written in a way that both communicates the intent of the code and provides a worked example of how the code should be used. Writing low-level executable specifications this way is a little like writing detailed design documentation, with lots of examples, but using a tool that's easy and even fun for developers.

Unit testing the space shuttle

Good developers have known the importance of unit testing for a very long time. The IBM Federal Systems Division team was fully aware of their importance when they wrote the central avionics software for NASA's space shuttle in the late seventies. The approach they took to unit testing, where the unit tests were designed using the requirements and with examples provided by the business, has a surprisingly modern feel to it:

(continued)

“During the development activity, specific testing was done to ensure that the mathematical equations and logic paths provided the results expected. These algorithms and logic paths were checked for accuracy and, where possible, compared against results from external sources and against the system design specification (SDS).”¹⁸

At a more technical level, this approach encourages a clean, modular design with well-defined interactions (or APIs, if you prefer a more technical term) between the modules. It also results in code that’s reliable, accurate, and extremely well tested.

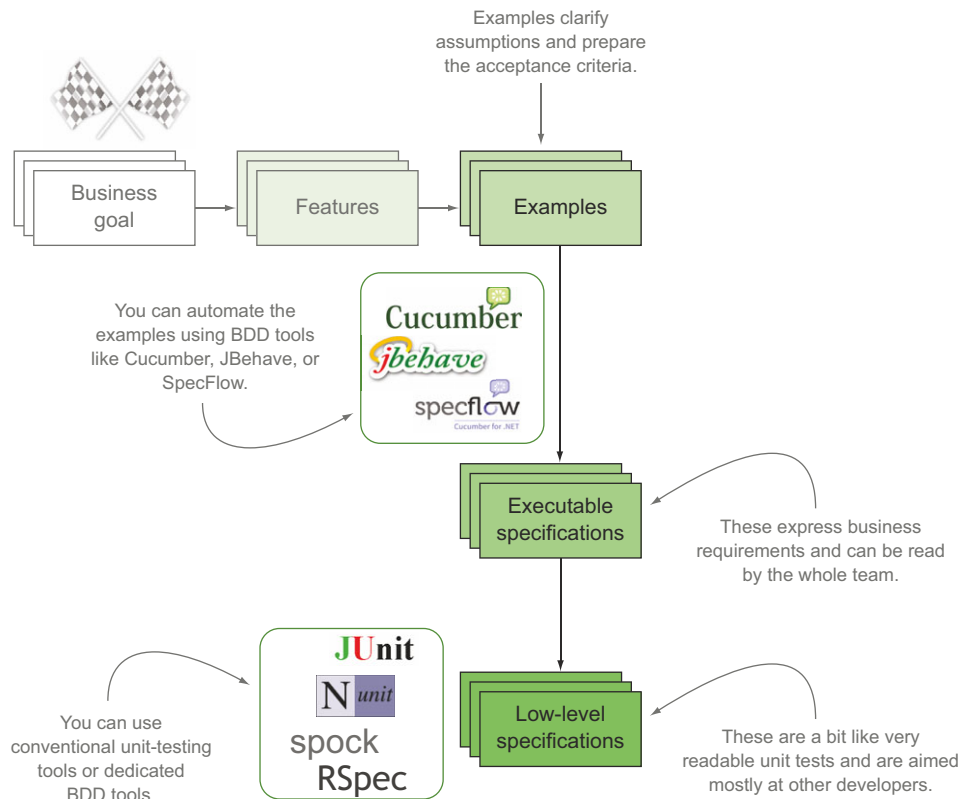


Figure 1.10 High-level and low-level executable specifications are typically implemented using different tool sets.

¹⁸ William A. Madden and Kyle Y. Rone, “Design, Development, Integration: Space Shuttle Primary Flight Software System,” *Communications of the ACM* (September 1984).

DELIVER LIVING DOCUMENTATION

The reports produced by executable specifications aren't simply technical reports for developers but effectively become a form of product documentation for the whole team, expressed in a vocabulary familiar to users (see figure 1.11). This documentation is always up to date and requires little or no manual maintenance. It's automatically produced from the latest version of the application. Each application feature is described in readable terms and is illustrated by a few key examples. For web applications, this sort of living documentation often also includes screenshots of the application for each feature.

Experienced teams organize this documentation so that it's easy to read and easy for everyone involved in the project to use (see figure 1.12). Developers can consult it to see how existing features work. Testers and business analysts can see how the features they specified have been implemented. Product owners and project managers can use summary views to judge the current state of the project, view progress, and decide what features can be released into production. Users can even use it to see what the application can do and how it works.

Just as automated acceptance criteria provide great documentation for the whole team, low-level executable specifications also provide excellent technical documentation for other developers. This documentation is always up to date, is cheap to maintain, contains working code samples, and expresses the intent behind each specification.

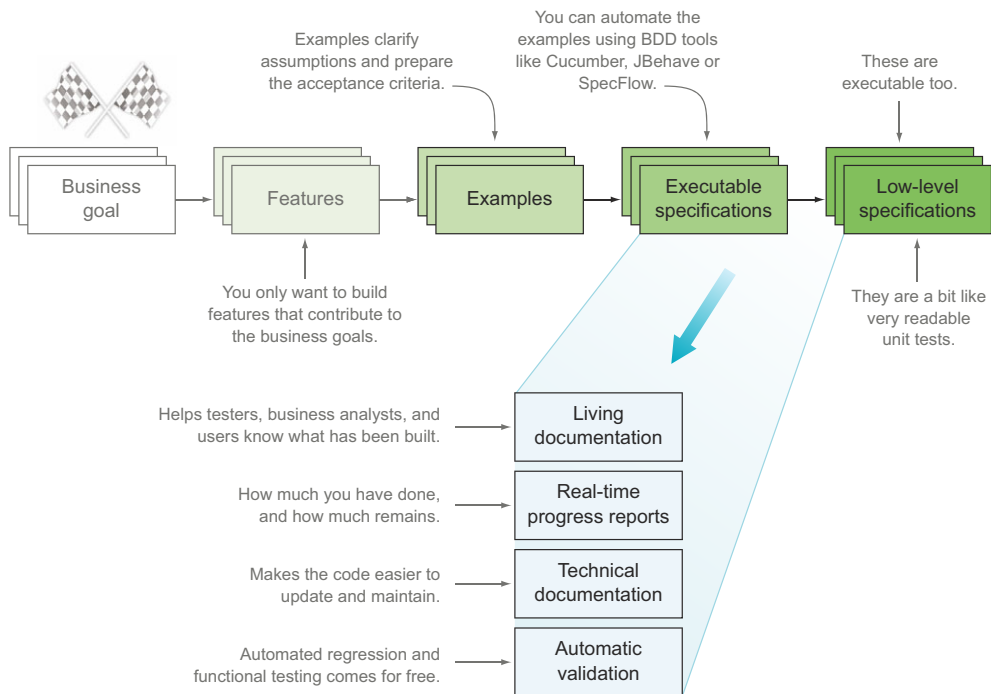


Figure 1.11 High-level and low-level executable specifications generate different sorts of living documentation for the system.

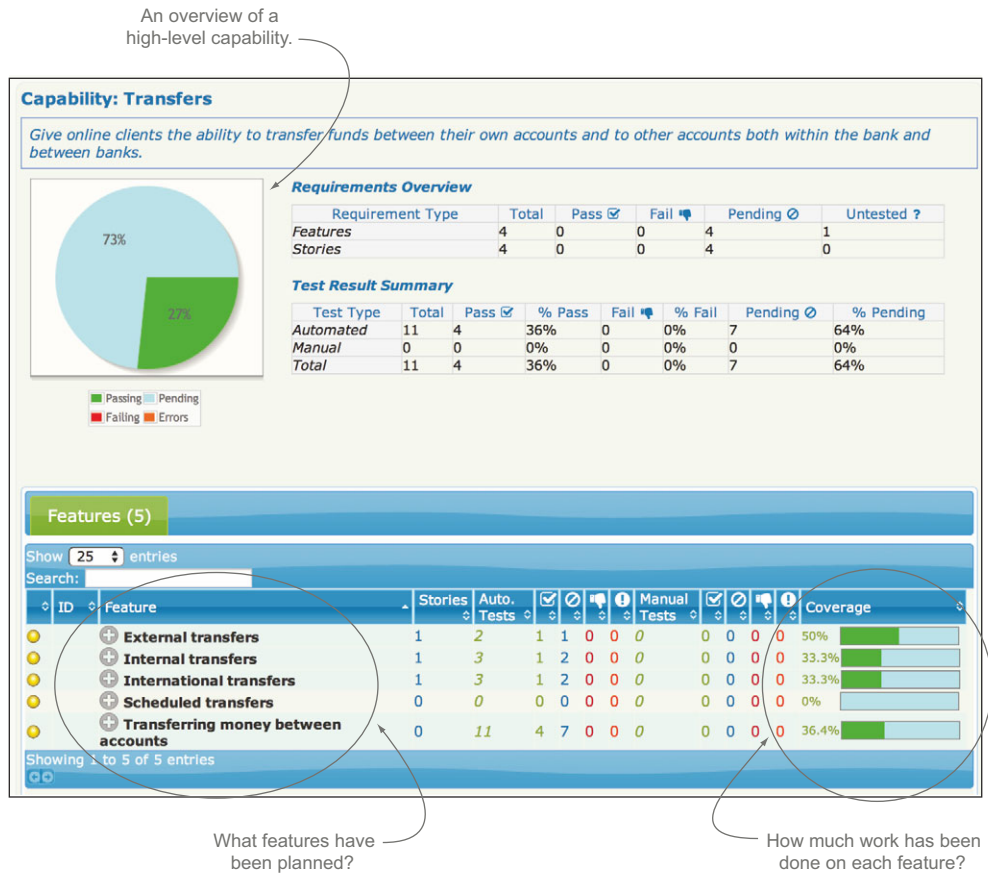


Figure 1.12 Well-organized living documentation can give an overview of the state of a project, as well as describe features in detail.

USE LIVING DOCUMENTATION TO SUPPORT ONGOING MAINTENANCE WORK

The benefits of living documentation and executable specifications don't stop at the end of the project. A project developed using these practices is also significantly easier and less expensive to maintain.

According to Robert L. Glass (quoting other sources), maintenance represents between 40% and 80% of software costs. Although many teams find that the number of defects drops dramatically when they adopt techniques like BDD, defects can still happen. Ongoing enhancements are also a natural part of any software application.¹⁹

In many organizations, when a project goes into production, it's handed over to a different team for maintenance work. The developers involved in this maintenance work have often not been involved in the project's development and need to learn the

¹⁹ Robert L. Glass, *Facts and Fallacies of Software Engineering* (Addison-Wesley Professional, 2002).

code base from scratch. Useful, relevant, and up-to-date functional and technical documentation makes this task a great deal easier.

The automated documentation that comes out of a BDD development process can go a long way toward providing the sort of documentation maintenance teams need in order to be effective. The high-level executable specifications help new developers understand the business goals and flow of the application. Executable specifications at the unit-testing level provide detailed worked examples of how particular features have been implemented.

Maintenance developers working on a BDD project find it easier to know where to start when they need to make a change. Good executable specifications provide a wealth of examples of how to test the application correctly, and maintenance changes will generally involve writing a new executable specification along similar lines or modifying an existing one.

The impact of maintenance changes on existing code is also easier to assess. When a developer makes a change, it may cause existing executable specifications to break, and when this happens, there are usually two possible causes:

- The broken executable specification may no longer reflect the new business requirements. In this case, the executable specification can be updated or (if it's no longer relevant) deleted.
- The code change has broken an existing requirement. This is a bug in the new code that needs to be fixed.

Executable specifications are not a magical solution to the traditional problems of technical documentation. They aren't guaranteed to always be meaningful or relevant—this requires practice and discipline. Other technical, architectural, and functional documentation is often required to complete the picture. But when they're written and organized well, executable specifications provide significant advantages over conventional approaches.

1.4 Benefits of BDD

In the previous sections, we examined what BDD looks like and discussed what it brings to the table. Now let's run through some of the key business benefits that an organization adopting BDD can expect in more detail.

1.4.1 Reduced waste

BDD is all about focusing the development effort on discovering and delivering the features that will provide business value, and avoiding those that don't. When a team builds a feature that's not aligned with the business goals underlying the project, the effort is wasted for the business. Similarly, when a team writes a feature that the business needs, but in a way that's not useful to the business, the team will need to rework the feature to fit the bill, resulting in more waste. BDD helps avoid this sort of wasted effort by helping teams focus on features that are aligned with business goals.

BDD also reduces wasted effort by enabling faster, more useful feedback to users. This helps teams make changes sooner rather than later.

1.4.2 Reduced costs

The direct consequence of this reduced waste is to reduce costs. By focusing on building features with demonstrable business value (building the right software), and not wasting effort on features of little value, you can reduce the cost of delivering a viable product to your users. And by improving the quality of the application code (building the software right), you reduce the number of bugs, and therefore the cost of fixing these bugs, as well as the cost associated with the delays these bugs would cause.

1.4.3 Easier and safer changes

BDD makes it considerably easier to change and extend your applications. Living documentation is generated from the executable specifications using terms that stakeholders are familiar with. This makes it much easier for stakeholders to understand what the application actually does. The low-level executable specifications also act as technical documentation for developers, making it easier for them to understand the existing code base and to make their own changes.

Last, but certainly not least, BDD practices produce a comprehensive set of automated acceptance and unit tests, which reduces the risk of regressions caused by any new changes to the application.

1.4.4 Faster releases

These comprehensive automated tests also speed up the release cycle considerably. Testers are no longer required to carry out long manual testing sessions before each new release. Instead, they can use the automated acceptance tests as a starting point, and spend their time more productively and efficiently on exploratory tests and other nontrivial manual tests.

1.5 Disadvantages and potential challenges of BDD

While its benefits are significant, introducing BDD into an organization isn't always without its difficulties. In this section, we'll look at a few situations where introducing BDD can be more of a challenge.

1.5.1 BDD requires high business engagement and collaboration

BDD practices are based on conversation and feedback. Indeed, these conversations drive and build the team's understanding of the requirements and of how they can deliver business value based on these requirements. If stakeholders are unwilling or unable to engage in conversations and collaboration, or they wait until the end of the project before giving any feedback, it will be hard to draw the full benefits of BDD.

1.5.2 BDD works best in an Agile or iterative context

BDD requirements-analysis practices assume that it's difficult, if not impossible, to define the requirements completely upfront, and that these will evolve as the team (and the stakeholders) learn more about the project. This approach is naturally more in line with an Agile or iterative project methodology.

1.5.3 BDD doesn't work well in a silo

In many larger organizations, a siloed development approach is still the norm. Detailed specifications are written by business analysts and then handed off to development teams that are often offsite or offshore. Similarly, testing is delegated to another, totally separate, QA team. In organizations like this, it's still possible to practice BDD at a coding level, and development teams will still be able to expect significant increases in code quality, better design, more maintainable code, and fewer defects. But the lack of interaction between the business analyst teams and the developers will make it harder to use BDD practices to progressively clarify and understand the real requirements.

Similarly, siloed testing teams can be a challenge. If the QA team waits until the end of the project to intervene, or does so in isolation, they'll miss their chance to contribute to requirements earlier on, which results in wasted effort spent fixing issues that could have been found earlier and fixed more easily. Automating the acceptance criteria is also much more beneficial if the QA team participates in defining, and possibly automating, the scenarios.

1.5.4 Poorly written tests can lead to higher test-maintenance costs

Creating automated acceptance tests, particularly for complex web applications, requires a certain skill, and many teams starting to use BDD find this a significant challenge. Indeed, if the tests aren't carefully designed, with the right levels of abstraction and expressiveness, they run the risk of being fragile. And if there are a large number of poorly written tests, they'll certainly be hard to maintain. Plenty of organizations have successfully implemented automated acceptance tests for complex web applications, but it takes know-how and experience to get it right. We'll look at techniques for doing this later on in the book.

1.6 Summary

In this chapter you were introduced to Behavior-Driven Development. Among other things, you learned the following:

- Successful projects need to build software that's reliable and bug-free and to build features that deliver real value to the business.
- BDD practitioners use conversations about concrete examples to build up a common understanding of what features will deliver real value to the organization.
- These examples form the basis of the acceptance criteria that developers use to determine when a feature is done.

- Acceptance criteria can be automated using tools like Cucumber, JBehave, or SpecFlow to produce both automated regression tests and reports that accurately describe the application features and their implementation.
- BDD practitioners implement features with a top-down approach, using the acceptance criteria as goals, and describing the behavior of each component with unit tests written in the form of executable specifications.
- The main benefits of BDD include focusing efforts on delivering valuable features, reducing wasted effort and costs, making it easier and safer to make changes, and accelerating the release process.

In the next chapter, we'll take a flying tour of what BDD looks like in the flesh, all the way from requirements analysis to automated unit and acceptance tests and functional test coverage reports. So without further ado, let's get started!

BDD—the whirlwind tour



This chapter covers

- An end-to-end walkthrough of BDD practices in action
- Discovering features and describing them through stories and examples
- Using executable specifications to specify features in detail
- Using low-level BDD to implement features
- Using BDD test results as living documentation
- Using living documentation to support ongoing maintenance

In this chapter, we'll look at a concrete example of how BDD might work on a real-world project. As you saw in the previous chapter, BDD involves the development team engaging in conversations with the customer throughout the project, using examples to build up a more concrete and less ambiguous understanding of what the business really needs. You write specifications in an executable form that you can use to define software requirements, drive their implementation, and validate the product you deliver. You can also apply these techniques during more high-level

requirements analysis, helping you focus on the capabilities and features of the application that will genuinely add value to the business.

A key part of this practice involves defining scenarios, or concrete examples of how a particular feature or story works. These scenarios will help you to validate and extend your understanding of the problem, and they're also an excellent communication tool. They act as the foundation of the acceptance criteria, which you then integrate into the build process in the form of automated acceptance tests. In conjunction with the automated acceptance tests, these examples guide the development process, helping designers to prepare effective and functional user-interface designs and assisting developers to discover the underlying behaviors that they'll need to implement to deliver the required features.

In the rest of this chapter, we'll look at a practical example of this process in action. We'll touch on aspects of the whole development cycle, from business analysis to implementing, testing, and maintaining the code.

2.1 Introducing the train timetable application

For this chapter's example, suppose you work for a large, government public-transport department. You've been asked to lead a small team building a service that will provide train timetable data and real-time updates about delays, track work, and so on for various mobile apps used by commuters. Figure 2.1 illustrates the rail network you'll be working with.

The department has just introduced Agile and BDD practices, so you'll start by talking to the key stakeholders to make sure that you and your team have a clear idea of the business goals driving the project. This will help the team deliver a better, more targeted application.

When you've understood and articulated the business goals, you'll need to work with your business analyst and business stakeholders to decide what software features will be able to achieve these goals. These features are high-level requirements, such as "Provide travellers with optimal itineraries between stations" or "Notify commuters if their train is late."

You probably won't be able to deliver features this big in one go, so you'll need to break them down into smaller units, known to Agile practitioners as *stories*. These stories might include things like "Find the optimal itinerary between stations on the same line" and "Find the optimal itinerary between stations on different lines."

When it comes to implementing a story, you get together with your business analyst, developer, and tester to describe the story in terms of concrete examples. Many of these examples will already have been discussed with the business stakeholders. These examples become the *acceptance criteria* for the story, and they're expressed in a formal BDD style that you can later automate:

```
Given Western line trains leave Parramatta at 7:58, 8:02, 8:08, 8:11
When I want to travel from Parramatta to Town Hall at 8:00
Then I should be told to take the 8:02 train
```

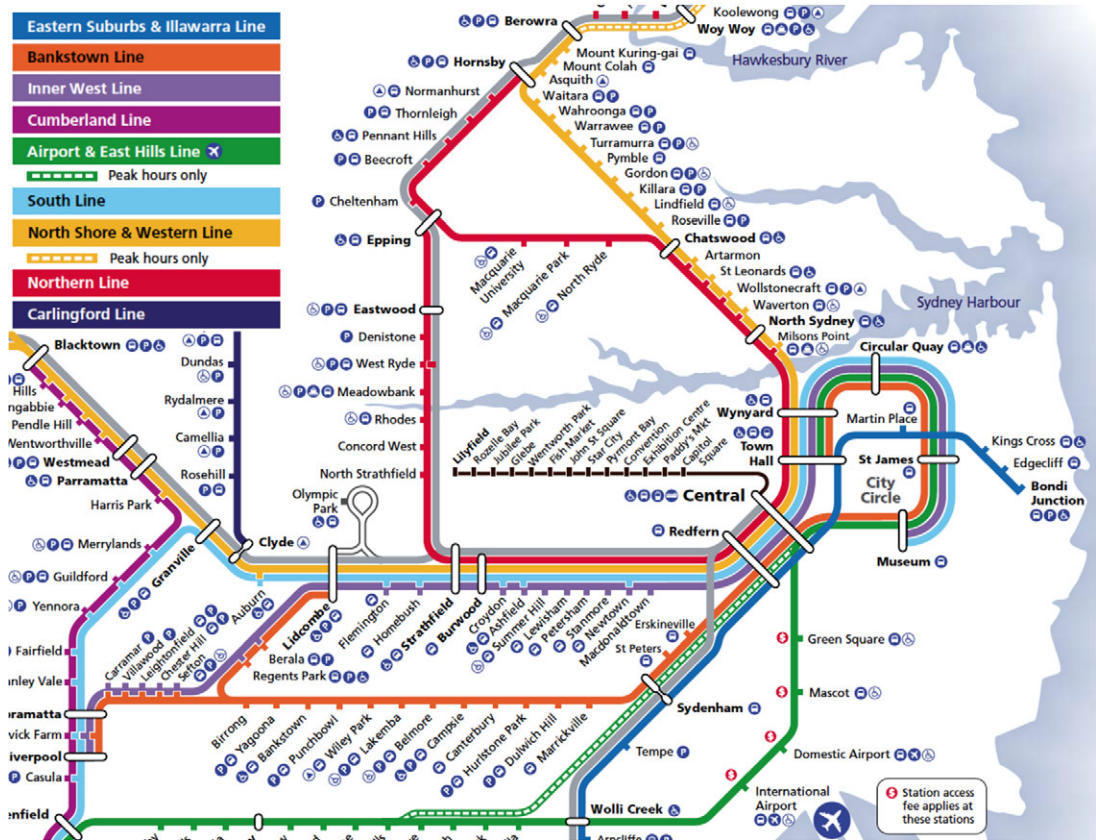


Figure 2.1 Part of the Sydney rail network

These acceptance criteria act as the starting point for development work. Because the department uses Java for their development projects, you'll automate the acceptance criteria using a Java tool called JBehave and write the application code in Java.

As you build the features, you'll use a lower-level BDD unit-testing tool called Spock to help you design, document, and verify your implementation.

You'll also generate test reports and living documentation from the automated acceptance criteria to illustrate what features have been completed and how they work.

The aim of this chapter is to give you an idea of the approach and some of the technologies involved, rather than to provide a full working example of any particular technology stack, but we'll go into enough technical detail for you to follow along. In the chapters that follow, we'll look at each of the topics covered in this chapter, and many others, in much more detail.

2.2 Determining the value proposition of the application

One of the key goals of BDD is to ensure that everyone has a clear understanding of what a project is trying to deliver, and of the underlying business objectives of the project. This, in itself, goes a long way toward ensuring that the application actually meets these objectives.

You can achieve this by working with users and other stakeholders to define or clarify a set of high-level business goals for the application. These goals should provide a concise vision of what you need to build. Business goals are about delivering value, so it's common to see them expressed in terms of increasing or protecting revenue, or of decreasing costs.

In this case, the aim of the application you have to build is to provide train schedules and real-time updates for commuters. You could express the primary business goal behind this application like this:

```
Increase ticket sales revenue by making it easier and more time-efficient to  
travel by train
```

Understanding and defining these goals makes it much easier to determine the relative value of a proposed feature. For example, a feature that notifies commuters if their train is late would contribute to the overall goal, because it would give travellers the opportunity to change their plans accordingly. On the other hand, a feature that lets commuters rate railway stations might not be considered to be of particularly high value.

2.3 Requirements analysis: discovering and understanding features

Once you have a better understanding of the high-level goals of your application, you can work with the stakeholders to determine exactly what they need to achieve these goals. This typically involves defining a set of features that the application will need in order to deliver the value you're after.

For this chapter, assume you've agreed with the stakeholders on the following essential features:

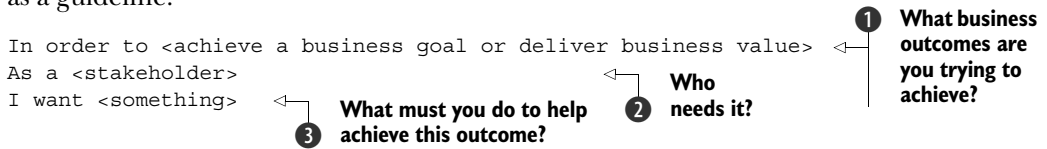
- Provide travellers with optimal itineraries.
- Provide real-time train timetable information about service delays to travellers.
- Allow commuters to record their favorite trips.
- Notify commuters if their train is late.

Let's look at how you might describe some of these features.

2.3.1 Describing features

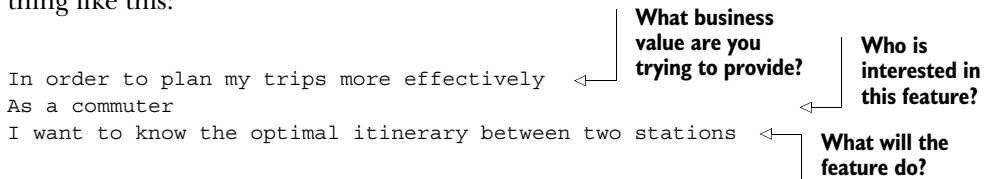
Once you have a general idea of the features you want to deliver, you need to describe them in more detail. There are many ways to describe a requirement.

Agile teams like to write a short outline of the requirement in a format that's small enough to fit on an index card.¹ Teams practicing BDD often use the following format as a guideline:²

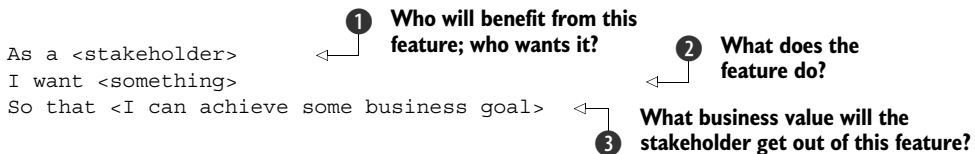


The order here is important. When you plan features and stories, your principal aim should be to deliver business value. Start out with what business value you intend to provide **1**, then who needs the feature you're proposing, **2**, and finally what feature you think will support this outcome **3**.

This helps ensure that each feature actively contributes to achieving a business goal, and so reduces the risk of scope creep. It also acts as a healthy reminder of why you're implementing this feature in the first place. For example, you could say something like this:



This is not your only option. Many teams use a template popular in earlier Agile approaches:



This variation aims to help developers understand the context of the requirement in terms of who will be using a feature and what they expect it to do for them. The stakeholder **1** refers to the person using the feature **2**, or who is interested in its output. The business goal **3** identifies why this feature is needed and what value it's supposed to provide. The equivalent of the feature mentioned earlier might be something like this:

```
As a commuter
I want to know the best way to travel between two stations
So that I can get to my destination quickly
```

Both of these are handy conventions, but there's no obligation to choose one format over another, as long as you remember to express the business benefits clearly. For

¹ These index cards can then be used to plan and visualize your progress.

² This format was originally proposed by Chris Matts, in the context of Feature Injection, which we'll look at in the next chapter.

example, some experienced practitioners are happy to use the “in order to ... as a ... I want” format for higher-level features, where the emphasis is very much on the business value the system should deliver, but they switch to “as a ... I want ... so ... that” for more detailed user stories within a feature, when the stories clearly are about delivering value to particular users in the context of that feature.

2.3.2 Breaking features down into stories

A feature may be detailed enough to work with as is, but often you’ll need to break it up into smaller pieces. In Agile projects, larger features are often broken into user stories, where each story explores a different facet of the problem and is small enough to deliver in a single iteration.

For example, the “Provide travellers with optimal itineraries” feature might be too large to build in one go (the developers think that finding itineraries involving connecting trains will be a complicated piece of work). In addition, you might like to get some feedback on the user interface design before you build the whole feature. You could break this feature into smaller stories, such as the following:

- Find the optimal itinerary between stations on the same line.
- Find out what time the next trains for the destination station leave.
- Find the optimal itinerary between stations on different lines.

You can describe these stories in a little more detail using the same format that’s used for the features:

```
Story: Find the optimal itinerary between stations on the same line
In order to get to my destination on time
As a commuter
I want to know what train I should take
```

```
Story: Find out what time the next trains for my destination station leave
In order to plan my trips more effectively
As a commuter
I want to know the next trains going to my destination
```

```
Story: Find the optimal itinerary between stations on different lines
In order to get to my destination on time
As a commuter
I want to know what train I should take
And details for any connections I need to make
```

Note that this sort of story list is by no means a rigid set of specifications that users and developers need to sign off on. Defining stories is a dynamic, iterative process, designed to facilitate communication and a shared understanding of the problem space. As you implement each story, you can get feedback from the stakeholders that will allow you to refine or drop stories or add new ones that might contribute to the business goals in other ways. Discovering features and stories is an ongoing learning process.

2.3.3 *Illustrating the stories with examples*

Once you have some features and stories of value, you can start to explore them in more detail. One very effective way to do this is to ask the users and other stakeholders for concrete examples.

When you hear a user asking for a feature, you'll often immediately start to build a conceptual model of the problem you'll need to solve. In doing so, it's easy to let implicit and unstated assumptions cloud your understanding, leading to an inaccurate mental model and an incorrect implementation further down the line. Asking the stakeholders to give you concrete examples of what they mean is a great way to test and confirm your understanding of the problem.

For example, you might have the following exchange with Jill, the rail network domain expert:³

You: Can you give me an example of a commuter travelling between two stations?

Jill: Sure, how about going from Parramatta to Town Hall.

You: And what would that look like?

Jill: Well, they'd have to take the Western line. That's a heavily used line, and there are between eight and sixteen trains per hour, depending on the time of day. We'd just need to propose the next scheduled trips on that line.

You: Can you give me an example of a trip where a commuter would have the choice of more than one line?

Jill: Yes, a commuter going from Epping to Central could take the Epping line or the Northern line. The trip time will vary from about 27 minutes to around 43 minutes, and trains from any of these lines would typically be arriving every couple of minutes, so we'd need to give commuters enough information on departure and arrival times for the trains on both lines.

Even in this simple example, you can see that there are some subtleties. It's not always a simple matter of proposing the next train; you have to give the commuter details about departure and arrival times for all the scheduled upcoming trains.

2.4 *Implementation: building and delivering features*

Once you understand the features your application needs, you have to build them. In this section, we'll look at the core BDD lifecycle.

You'll learn how to take the business-focused examples we discussed in the previous section, and rewrite them in the form of executable specifications. You'll see how you can automate these specifications, and how doing so leads to discovering what

³ You can follow along with these examples by referring to the map in figure 2.1.

code you need to write. And you'll see how these executable specifications can be a powerful reporting and living-documentation tool.

2.4.1 Going from examples to acceptance criteria

You can use your examples (such as Jill's commuter trip examples) as the basis for the acceptance criteria. In a nutshell, the acceptance criteria are what will satisfy stakeholders (and QA) that the application does what it's supposed to do.

Conversations like the one with Jill (in section 2.3.3) are a great way to build up your understanding of the problem space, but you can take things a lot further if you use a slightly more structured style. In BDD, the following notation is often used to express examples:⁴

```
Given <a context>
When <something happens>
Then <you expect some outcome>
```

This format helps you think in terms of how users interact with the system and what the outcomes should be. As you'll see in the next section, they're also easy to convert into automated acceptance tests using tools like Cucumber and JBehave. But because you may want to automate these tests later on, their format is a little less flexible. As you'll see, words like *Given*, *When*, and *Then* have special meanings for these tools, so it's best to think of them as special keywords.

Using this notation, you could express the requirement mentioned previously as follows:

```
Given Western line trains leave Parramatta at 7:58, 8:02, 8:08, 8:11
When I want to travel from Parramatta to Town Hall at 8:00
Then I should be told to take the 8:02 train
```

When you talk through this example with Jill, she points out that trains travel in two directions on a line, so the *Given* section ① is incomplete—you also need to give a starting point and a direction. Or, if you give the destination station, you can work out the direction of the trains from this. You could refine this scenario as follows:

```
Given Western line trains from Emu Plains leave Parramatta
for Town Hall at 7:58, 8:00, 8:02, 8:11
When I want to travel from Parramatta to Town Hall at 8:00
Then I should be told to take the 8:02 train
```

When you talk this example through, though, you realize that you only have two minutes to buy a ticket and get to the right platform. You really need to be told about the next few trains:

⁴ The syntax shown here is sometimes referred to as the Gherkin format, but this is not strictly accurate—Gherkin is the syntax used by Cucumber and related tools, whereas these examples use JBehave. We'll look at all this in great detail in chapter 5.

Given Western line trains from Emu Plains leave Parramatta
for Town Hall at 7:58, 8:00, 8:02, 8:11, 8:14, 8:21
When I want to travel from Parramatta to Town Hall at 8:00
Then I should be told about the trains at: 8:02, 8:11, 8:14

Identifies six train times
to make the example
more representative.

Now you expect to
obtain three results.

Before we take this example any further, or move on to more complicated requirements, let's see how you can transform these acceptance criteria into executable specifications, using JBehave, Maven, and Git in this chapter.

2.4.2 *Setting up Maven and Git*

There are many specialized BDD tools that you can use to automate your acceptance criteria. Popular choices include tools like JBehave, Cucumber, SpecFlow, and Behat. While not indispensable, these tools make it easier to express the automated tests in a structured form similar to the “Given ... When ... Then” expressions used in the previous section. This makes it easier for product owners and testers to understand and identify the automated acceptance criteria, which in turn can help increase their confidence in the automated tests and in the automated acceptance-testing approach in general.

Throughout the rest of this book, I'll illustrate examples using several different BDD tools. In this chapter, I'll use examples written with JBehave and Java,⁵ and the project will be built and run using Maven.⁶ The test reports will be generated using Thucydides,⁷ an open source library that makes it easier to organize and report on BDD test results.

The source code for this chapter is available on GitHub⁸ and on the Manning website. If you want to follow along, you'll need a development environment with the following software installed:

- A Java JDK (the sample code was developed using Java 1.7.0, but it should work fine with JDK 1.6.0)
- Maven 3.0.x
- Git

GitHub lets you access a repository in a number of ways. If you have Git installed and a GitHub account set up with SSH access,⁹ you can clone the sample code repository like this:

```
$ git clone git@github.com:bdd-in-action/chapter-2.git
```

If you haven't set up and configured SSH keys for GitHub, you can also use the following command (Git will ask you for your username and password):

```
$ git clone https://github.com:bdd-in-action/chapter-2.git
```

⁵ If Java isn't your cup of tea, don't worry; the code samples are designed to be readable by anyone with some programming background. We'll look at BDD tools in .NET, Ruby, and Python from chapter 5 onwards.

⁶ Maven (<http://maven.apache.org/>) is a widely used build tool in the Java world.

⁷ See the Thucydides site (<http://thucydides.info>) for more details about this library.

⁸ The source for this chapter on GitHub is at <https://github.com/bdd-in-action/chapter-2>.

⁹ See GitHub help (<https://help.github.com/articles/set-up-git>) for a good tutorial on installing Git for your OS.

Once you've cloned the project, it's a good idea to run `mvn verify` in the project directory so that Maven can download the dependencies it and the project needs. It will only need to do this once, but it can take some time. Run the following commands:

```
$ cd chapter-2
$ mvn verify
```

If you want to code each step yourself, go to the `train-timetables` directory and check out the start branch:

```
$ git checkout start
```

When you do this, you'll get a simple project skeleton with a properly configured Maven build script (the `pom.xml` file) and a directory structure that you can use to get started. If at any point you'd like to take a look at the sample solution, run the following command:

```
$ git checkout master
```

The initial project structure looks something like figure 2.2 and follows the standard Maven conventions. Application code will go in the `src/main/java` directory, and test code will go in the `src/test/java` directory. The JBehave stories that you'll write will go in the `src/test/resources/stories` directory. The `AcceptanceTestSuite` class is a simple JUnit-based test runner that will run all the JBehave stories in or under the `src/test/resources` directory.



Figure 2.2 The initial project structure

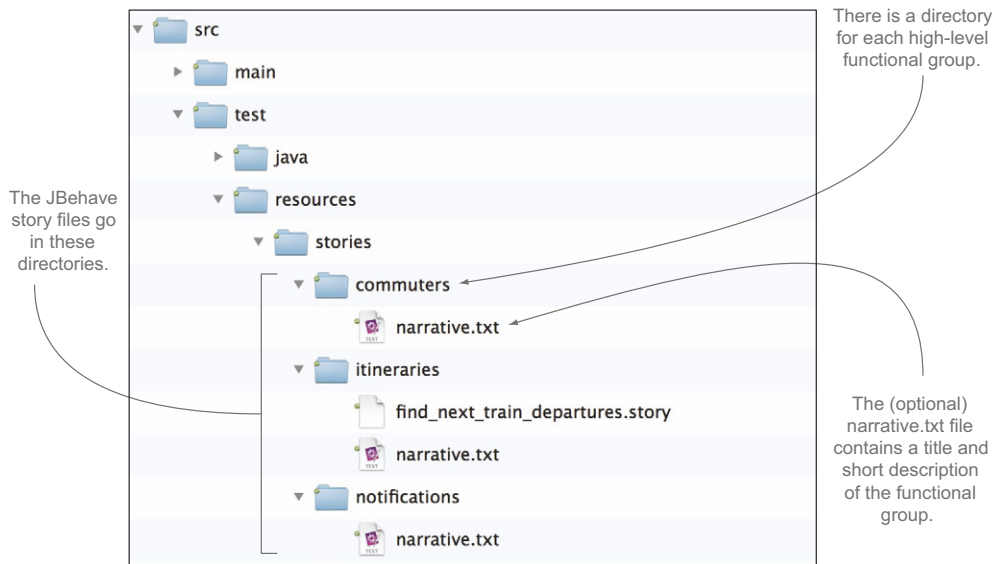


Figure 2.3 The JBehave story files are organized into directories.

For documentation purposes, you can also add a text file called `narrative.txt`¹¹ in each of these directories. The text file will contain the name of the functional group and a short description of what it covers. For example, the `narrative.txt` file for the `itineraries` directory might look like the following listing.

Listing 2.2 The `narrative.txt` file describes high-level functionality

```
Itineraries and timetables
Providing real-time information on timetables and itineraries.
```

A one-line title

A short, free-form description of this high-level functionality

You now have an executable specification. Though there's no code behind this JBehave scenario, you can still execute it. If you want to try this out, go into the `train-timetables` directory and run the following command:

```
$ mvn verify
```

This generates a set of reports in the `target/site/thucydides` directory.¹² If you open the `index.html` file in this directory and click on the only test in the Test table at the bottom of the screen, you should see something like figure 2.4.

¹¹ If provided, the `narrative.txt` file is also used by Thucydides to generate the living documentation you'll see further on.

¹² If you're not a regular Maven user, Maven will first download the libraries it needs to work with—this may take some time, but you'll only need to do it once.

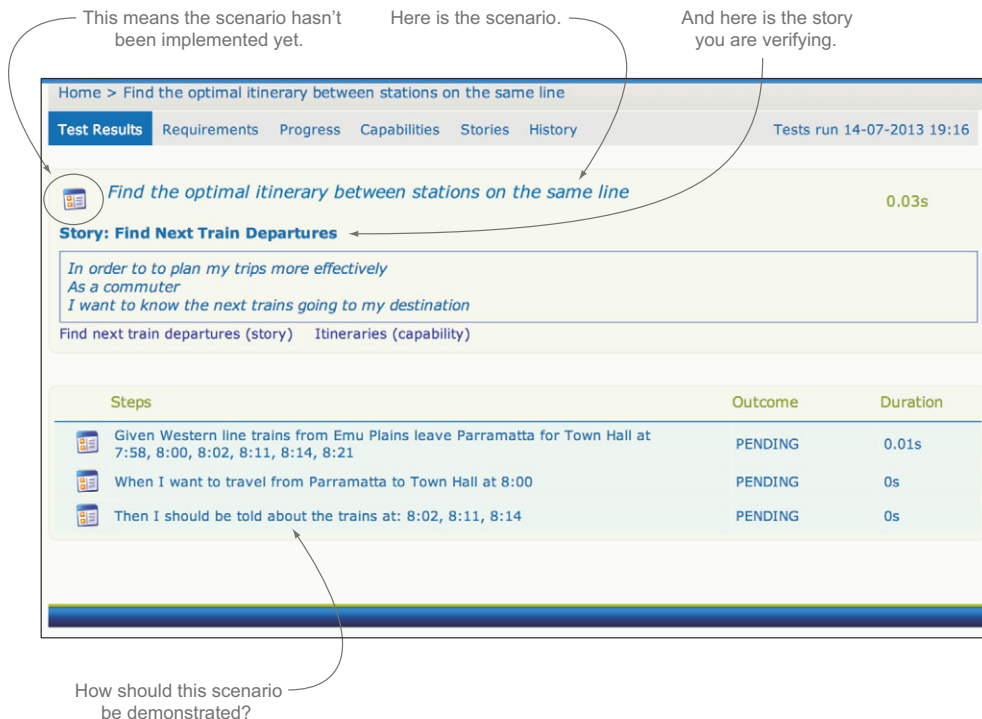


Figure 2.4 The JBehave story in the acceptance test reports

At this point, your scenario is no longer a simple text document; it's now an *executable specification*. It can be run as part of the automated build process to automatically determine whether a particular feature has been completed or not. When tests like this are first executed, they're flagged as “pending,” which means, in BDD terms, that the test has been automated but that the code that implements the supporting features has not yet been written. As the features are implemented and the acceptance tests succeed, they're marked as “passed” to indicate that you've completed work in this area.

But living documentation is more than just test reporting. It should also report on the state of all of your specified requirements, even the ones that don't have any tests yet. This gives a much more complete picture of your project and your product. You can see an example of this type of requirements-level reporting by clicking on the Requirements tab in the reports you just generated (see figure 2.5). Here you should see a list of all the high-level requirements we discussed earlier, along with an idea of how much work has been completed for each of them (which, at this stage, will be precisely none).

Before we look at how to automate scenarios like this in Java, let's look at another variation. An important part of this application is the ability to tell commuters when they'll arrive at their destination if they leave at a certain time. Jill provided some examples that you can use to build up a scenario describing this requirement.

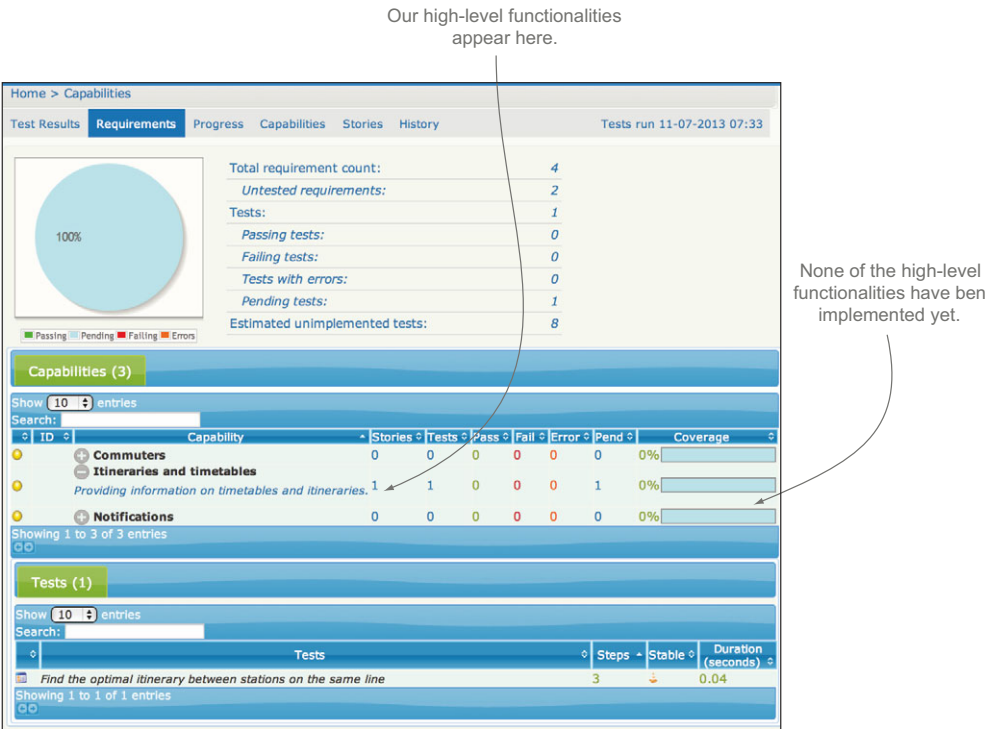


Figure 2.5 Living documentation should also tell you what requirements you have specified, even if there are no tests for them yet.

Listing 2.3 Calculating arrival times

Tell travellers when they will arrive at their destination

Narrative:
In order to plan my voyage more effectively
As a commuter
I want to know what time I will arrive at my destination

Scenario: Calculate arrival times
Given I want to go from <departure> to <destination>
And the next train leaves at <departure-time> on <line> line
When I ask for my arrival time
Then the estimated arrival time should be <arrival-time>
Examples:

departure	destination	departure-time	line	arrival-time
Parramatta	Town Hall	8:02	Western	8:29
Epping	Central	8:03	Northern	8:48
Epping	Central	8:07	Newcastle	8:37
Epping	Central	8:13	Epping	8:51

A one-line title

Scenario title

Scenario details start here

Scenario uses data from the table below

Table headings identify values in test data

Each row represents separate set of test data

This scenario uses variables from the examples table.

0.08s

Calculating arrival times

Story: Calculate Estimated Arrival Times

*In order to plan my voyage more effectively
As a commuter
I want to know what time I will arrive at my destination*

Calculate estimated arrival times (story) Itineraries (capability)

Scenario:

Given I want to go from <departure> to <destination>
And the next train is scheduled to leave at <departure-time> on the <line> line
When I ask for my arrival time
Then the estimated arrival time should be <arrival-time>

Examples:

Show 10 entries

Search:

Departure	Destination	Departure-Time	Line	Arrival-Time
Epping	Central	8:03	Northern	8:48
Epping	Central	8:07	Newcastle	8:37
Epping	Central	8:13	Epping	8:51
Parramatta	Town Hall	8:02	Western	8:29

Showing 1 to 4 of 4 entries

Steps	Outcome	Duration
[1] {departure=Parramatta, destination=Town Hall, departure-time=8:02, line=Western, arrival-time=8:29}	PENDING	0.04s
[2] {departure=Epping, destination=Central, departure-time=8:03, line=Northern, arrival-time=8:48}	PENDING	0.01s
[3] {departure=Epping, destination=Central, departure-time=8:07, line=Newcastle, arrival-time=8:37}	PENDING	0.01s
[4] {departure=Epping, destination=Central, departure-time=8:13, line=Epping, arrival-time=8:51}	PENDING	0.01s

Each row in the table is a separate test.

Test results for each row.

Figure 2.6 Living documentation for a table-based scenario

The main novelty here is the use of a table in text form to represent the test data. The table headings ❶ identify the values in the test data. Each row in the table ❷ represents a separate set of test data to be used with this scenario.

You can put this story into a file called `calculate_estimated_arrival_times.story`, next to the previous story file. This will add it to the set of stories that JBehave runs, so it will become part of your living documentation when you run the tests (see figure 2.6).

The language used in both of these scenarios is very close to that provided by the user. When the scenarios appear in the test reports, the use of this familiar language makes it easier for testers, end users, and other non-developers to understand what features are being tested and how they're being tested.

2.4.4 Automated tests: implementing the acceptance criteria

Now that you've defined and automated some acceptance criteria, the real work begins. Naturally the logic required to verify acceptance criteria doesn't write itself: you'll need to add some code to make the tests actually do their jobs.

You'll start with the first scenario we looked at in listing 2.1:

```
Scenario: Find the optimal itinerary between stations on the same line
Given Western line trains from Emu Plains leave Parramatta for Town Hall at
7:58, 8:00, 8:02, 8:11, 8:14, 8:21
When I want to travel from Parramatta to Town Hall at 8:00
Then I should be told about the trains at: 8:02, 8:11, 8:14
```

Continuing this example using JBehave and Java, you could implement an empty automated test for this scenario in a class called `OptimalItinerarySteps.java` as shown in the next listing.

Listing 2.4 A pending JBehave scenario implementation

```
package com.bddinaction.chapter2.jbehave.steps;

import org.jbehave.core.annotations.Given;
import org.jbehave.core.annotations.Pending;
import org.jbehave.core.annotations.Then;
import org.jbehave.core.annotations.When;
import java.util.List;
import org.joda.time.LocalTime;

public class OptimalItinerarySteps {
    @Given("$line line trains from $lineStart leave $departure for $destination at $departureTimes")
    @Pending
    public void givenArrivingTrains(String line,
                                   String lineStart,
                                   String departure,
                                   String destination,
                                   List<LocalTime> departureTimes) {

    }

    @When("I want to travel from $departure to $destination at $time")
    @Pending
    public void whenIWantToTravel(String departure,
                                   String destination,
                                   LocalTime time) {

    }

    @Then("I should be told about the trains at: $viableTrainTimes")
    @Pending
    public void shouldFindTheseTrains(List<LocalTime> viableTrainTimes) {

    }
}
```

1 A Given step

2 Flag this step as pending

3 A When step

4 A Then step

This class goes in the `steps` package underneath the `jbehave` package in the `src/test/java` directory (see figure 2.7). JBehave tests can be implemented in Java or in other JVM languages such as Groovy or Scala. When it executes a scenario, JBehave will use the text from the `@Given` ❶, `@When` ❸, and `@Then` ❹ annotations to determine what method to call at each step. As the listing shows, you can also pass elements from the scenario text to the test methods in the form of parameters.

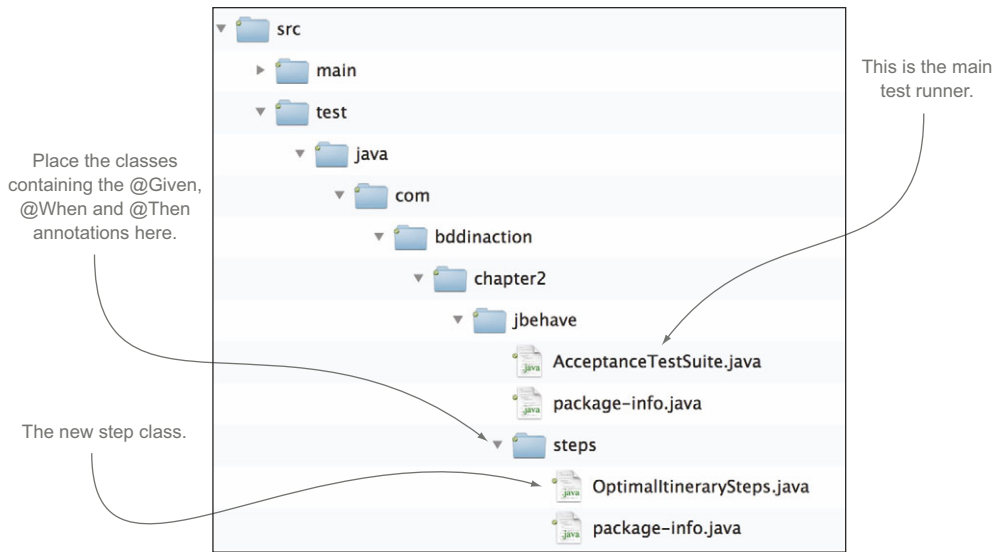


Figure 2.7 The package structure containing the new steps class

As it's written here, the `@Pending` annotation ❷ will ensure that this test will produce exactly the same results shown in figure 2.4. But as soon as possible, you'll want to flesh out the implementation of each method and turn this into a fully operational test.

These steps act as the starting point for your production code: they tell you what you need to build to get this feature out the door. BDD practitioners like to start with the outcome they need to obtain and work backwards, so let's see how that would work here.

Start with the `@Then` step, which expresses the outcome you expect. What you really want is for the service to give you a list of proposed train times that match the train times you expect. One way you could express this is as follows:

```
@Then("I should be told about the trains at: $expectedTrainTimes")
public void shouldBeInformedAbout(List<LocalTime> expectedTrainTimes) {
    assertThat(proposedTrainTimes).isEqualTo(expectedTrainTimes);
}
```

JBehave will match the first line of your scenario, extract the list of expected times (indicated by `$expectedTrainTimes` in the `@Then` annotation), and pass them to the method as a parameter. Note that this example also removes the `@Pending` annotation so that JBehave will know that it's supposed to execute this step. But this won't compile yet—you still need to decide where the proposed train will come from.

BDD helps you discover the technical design you need in order to deliver the business goals. Suppose you decide to implement this application using a number of different web services. For example, you may need an itinerary service that provides information about departure times to a website or a mobile app. This service doesn't exist yet, so you need to discover what this service should do. One of the most effective

ways to do this is simply to write the code you would like to have. This allows you to design a clean, readable, self-documenting API.

For example, to find the next departure times from a given station, one very simple implementation might look something like this:

```
proposedTrainTimes = itineraryService.findNextDepartures(departure,
                                                         destination,
                                                         startTime);
```

Integrating this into the @When step, you'll get something like this:

```
List<LocalTime> proposedTrainTimes;

@When("I want to travel from $departure to $destination at $startTime")
public void whenIWantToTravel(String departure,
                              String destination,
                              LocalTime startTime) {
    ItineraryService itineraryService = new ItineraryService();
    proposedTrainTimes = itineraryService.findNextDepartures(departure,
                                                             destination,
                                                             startTime);
}
```

Finding the next departure times ←

The When step ←

Passing in parameters from the When step ←

Creating a new itinerary service ←

To get this acceptance criteria to pass, you'll implement the `findNextDepartures()` method. But before you do so, you need to change gears from acceptance testing to unit testing. As you'll see, acceptance testing is used to demonstrate the high-level, end-to-end behavior of an application, and unit testing is used to build up the components you'll use to implement this behavior. Acceptance tests typically use a full or near-full application stack, whereas unit tests concentrate on individual components

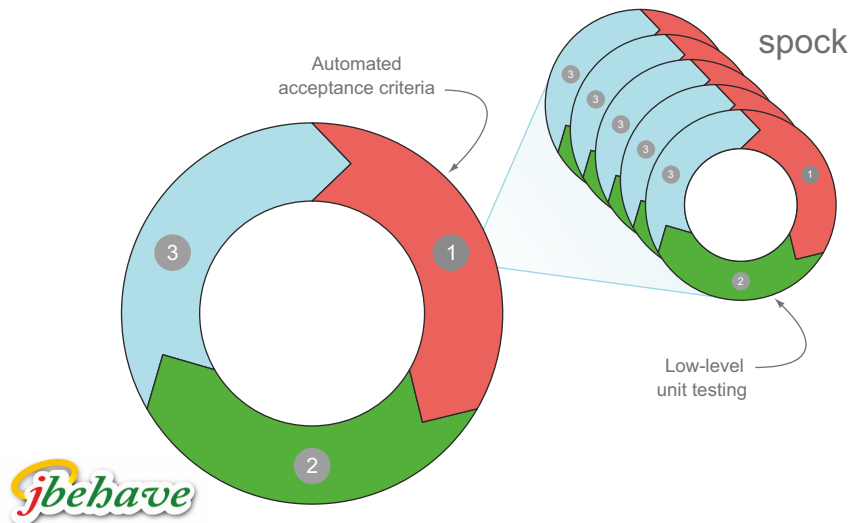


Figure 2.8 You'll typically need to write many low-level, TDD-style unit tests to get an automated acceptance criterion to pass. The example in this chapter uses JBehave for the acceptance criteria and Spock for the unit tests.

in isolation. Unit tests make it easier to focus on getting a particular class working and identifying what other services or components it needs. Unit tests also make it easier to detect and isolate errors or regressions. You'll typically write many small unit tests in order to get an acceptance criterion to pass (see figure 2.8).

You could use a conventional unit-testing framework such as JUnit for these unit tests. But because this book is about BDD, you'll use a more BDD-specific unit-testing tool called Spock. Spock is a lightweight and expressive BDD-style testing library for Java and Groovy applications that you'll use extensively later on in this book.

Groovy is a dynamic language built on top of Java that incorporates many features from languages like Ruby and Python. It's a very expressive language that works well for checking outcomes in a clean and concise manner.

Writing unit tests in Spock

In Spock, you write unit tests in the form of “specifications,” using a very readable “given ... when ... then” structure similar to that used in the JBehave scenarios. For example, if you wanted to test a `Calculator` class in Spock, you could write something like this:

```
def "should calculate the sum of two numbers"() {
    given: "two numbers"
        int a = 1
        int b = 2
    when: "we add the numbers together"
        def calculator = new Calculator()
        int sum = calculator.add(a,b)
    then: "the result should be the sum of the two numbers"
        sum == 3
}
```

Defining a specification

The given clause

The when clause

The then clause

The equivalent of “assert sum == 3”

Many Java developers like to write unit tests using Spock because it lets them write concise and descriptive tests with less boilerplate code than would be needed using Java.

You'll start off by essentially testing the acceptance criteria described in the JBehave scenario in listing 2.1, but at a unit-testing level. In Spock, you could create a Groovy class called `WhenCalculatingArrivalTimes.groovy` in an appropriate package in the `src/test/groovy` directory.¹³ This class might look something like the following:

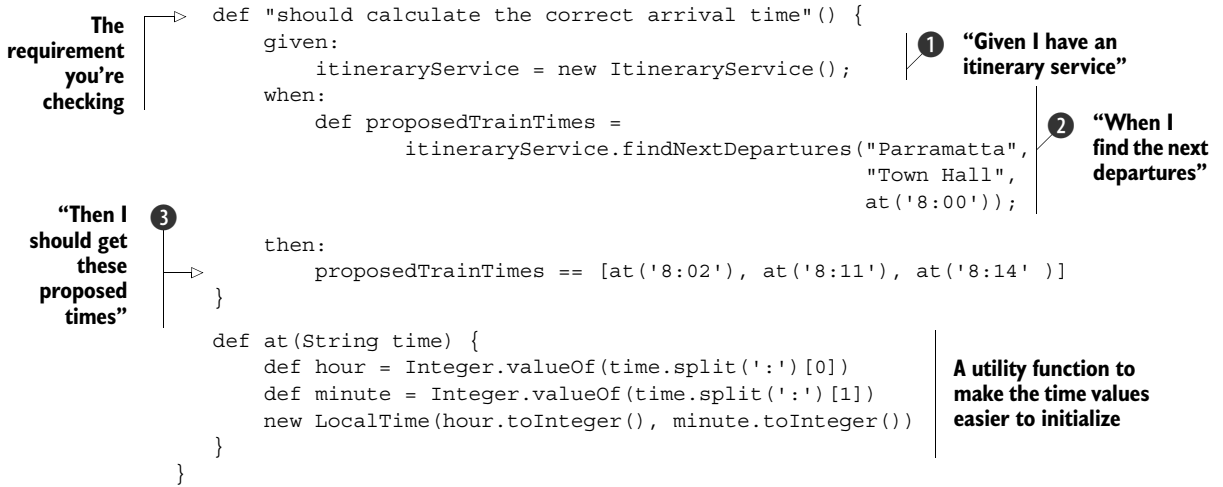
```
package com.bddinaction.chapter2.services

import org.joda.time.LocalDateTime
import spock.lang.Specification

class WhenCalculatingArrivalTimes extends Specification {
    ItineraryService itineraryService;
}
```

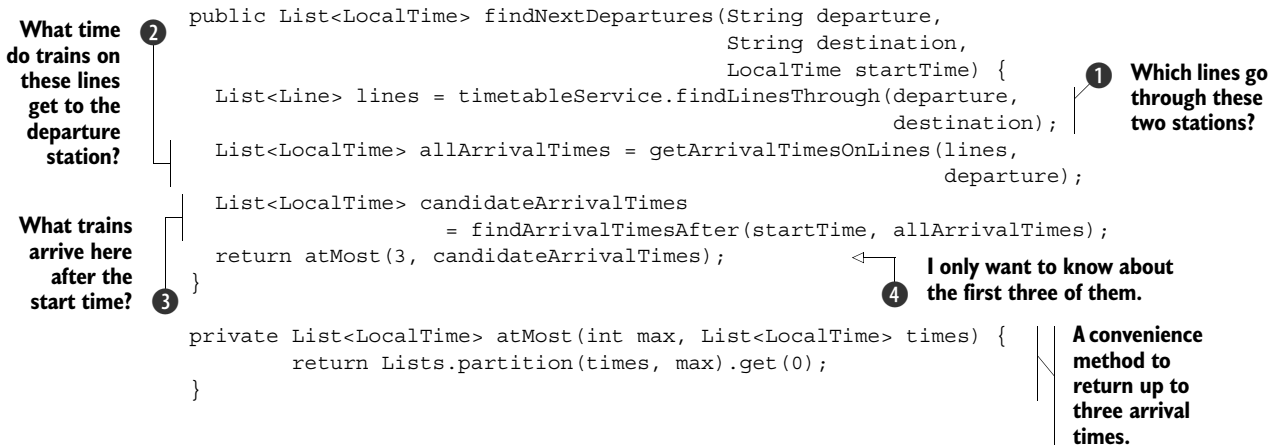
Spock tests must extend the Specification class

¹³ In the sample solution, we put this class in the `com.bddinaction.chapter2.services` package.



This specification does essentially the same thing as the JBehave test. It creates a new itinerary service ①, finds the next departures between Parramatta and Town Hall starting from 8:00 am ②, and checks that it matches the times you expect ③. When this passes, you can be confident that the `ItineraryService` class does what you expect it to do.

Of course, if you run this specification, it will fail, because you haven't written any application code yet. It's now time to write that code. Let's see what the `findNextDepartures()` method might look like:



The job of the itinerary service is to calculate information about departure times and trip details based on the current timetables, and to do this, it will need timetable data. But you haven't written the code to figure out this bit yet: timetables are complicated things and represent a separate area of concern. Following good design practices, you can use a separate service to provide timetable details to the itinerary service. The preceding code introduces a `timetableService` object ① to take care of this.

The first thing this code does is find the train lines that will take the user from the departure station to the destination. This is the domain of the timetable service, so you ask it to provide you with a list of lines going from one station to another ❶. In this context, a *line* represents the path that trains take from one terminus to another, in a given direction. Remember, the timetable service doesn't have any methods yet, so you've just discovered something that you need the timetable service to do.

Once you have a list of the lines, you can find the times of the trains arriving at the departure station on any line. There are a few ways you could do this, but you'll delegate it to another method ❷ and move on with the main business logic.

The last thing this method needs to do is to find the arrival times after the requested starting time ❸ and return the next three departure times ❹.

You can see the complete source code on GitHub,¹⁴ but let's focus on the `getArrivalTimesOnLines()` method, shown here:

```
private List<LocalTime> getArrivalTimesOnLines(List<Line> lines,
                                              String station) {
    TreeSet<LocalTime> allArrivalTimes = Sets.newTreeSet();
    for(Line line : lines) {
        allArrivalTimes.addAll(
            timetableService.findArrivalTimes(line, station));
    }
    return new ArrayList<LocalTime>(allArrivalTimes);
}
```

❶ Collect arrival times

This method is interesting because it points out something else you need the timetable service to do. The logic isn't complicated—it just collects the arrival times of all the lines going through a given station ❶. A *Line* is a simple domain object with a name, a departure station, and a list of stations on that line.¹⁵

But for the `getArrivalTimesOnLines()` method to work, you need to know the times that trains on a particular line are due to arrive at the departure station, and you need to get this information from the timetable service.

This means you need a *TimetableService* object that does the following:

- Finds the lines through any two stations.
- Finds the arrival times for a given line at a particular station.

You can formalize what you need in the *Given* step of your Spock specification by creating a “pretend” timetable service that behaves exactly as you'd like it to, and ensuring that your itinerary service processes the data it gets from the timetable service correctly. You could do this with the following code:

```
TimetableService timetableService = Mock(TimetableService)
def "should calculate the correct arrival time"() {
    given:
```

Create a “pretend” timetable service.

¹⁴ The GitHub page is at <https://github.com/bdd-in-action/chapter-2>.

¹⁵ For simplicity, the full implementation of the *Line* class is provided in the sample code.

The pretend timetable service returns this train line when you ask which lines go through Parramatta and Town Hall.

```

1 def westernLine =
    Line.named("Western").departingFrom("Emu Plains")

    timetableService.findLinesThrough("Parramatta",
                                     "Town Hall") >> [westernLine]

    timetableService.findArrivalTimes(westernLine, "Parramatta") >>
        [at(7,58), at(8,00), at(8,02), at(8,11), at(8,14), at(8,21)]

    when:
    def proposedTrainTimes =
        itineraryService.findNextDepartures("Parramatta",
                                             "Town Hall",
                                             at(8,00));

    then:
    proposedTrainTimes == [at(8,02), at(8,11), at(8,14)]
}

```

Define a train line.

2 The pretend timetable service also gives you the correct arrival times at Parramatta.

This sets up a “pretend” (or “mock”) timetable service to model what you need the real timetable service to do. You know that it needs to tell you what lines go through any two stations ❶ and what time trains on a given line get to a particular station ❷. The >> sign in Spock is shorthand for saying “when I call this method with these parameters, return these values.”

For this to compile, you need a `TimetableService`. In Java, you’d typically define an interface. This lets you put off actually implementing the `TimetableService` until later, after you’ve finished with the `ItineraryService` class. You defined the methods the `TimetableService` needs in ❶ and ❷, so the interface might look like this:

```

package com.bddinaction.chapter2.services;

import com.bddinaction.chapter2.model.Line;
import org.joda.time.LocalDateTime;

import java.util.List;

public interface TimetableService {
    List<LocalTime> findArrivalTimes(Line line, String targetStation);
    List<Line> findLinesThrough(String departure, String destination);
}

```

The final piece in the puzzle is the `findArrivalTimesAfter()` method, which returns a list of departure times after a given time, as shown in this possible implementation:

```

private List<LocalTime> findArrivalTimesAfter(LocalTime startTime,
                                             List<LocalTime> times) {
    List<LocalTime> viableArrivalTimes = Lists.newArrayList();
    for(LocalTime arrivalTime : times) {
        if (arrivalTime.isAfter(startTime)) {
            viableArrivalTimes.add(arrivalTime);
        }
    }
    return viableArrivalTimes;
}

```


Now, when you run this test using `mvn verify`, it should pass, which demonstrates that you now have a working itinerary service. But you aren't done yet. The itinerary service is built on the assumption that the timetable service does its job correctly and uses a “dummy” timetable service (known as a “stub” or “mock” object) to avoid needing a real one. This is a very effective way to build the itinerary service, as you can concentrate exclusively on the business logic of this service. But to get your acceptance criteria, you'll also need to implement this timetable service.

The behavior defined for the mock timetable service provides very precise low-level requirements about how the timetable service should behave. This will be the starting point when you implement this service. You're going to write a new Spock test called `WhenFindingLinesThroughStations.groovy` (once again in the `com.bddinaction.chapter2.services` package) that builds on these requirements and describes what the timetable service should do in more detail:

```
package com.bddinaction.chapter2.services

import com.bddinaction.chapter2.model.Line
import spock.lang.Specification

class WhenFindingLinesThroughStations extends Specification {

    def timetableService = new TimetableService()

    def "should find the correct lines between two stations"() {
        when:
            def lines = timetableService.findLinesThrough(departure,
                                                         destination)
        then:
            def expectedLine = Line.named(lineName).
                                   departingFrom(lineDeparture)
            lines == [expectedLine]
        where:
            departure | destination | lineName | lineDeparture
            "Parramatta" | "Town Hall" | "Western" | "Emu Plains"
            "Town Hall" | "Parramatta" | "Western" | "North Richmond"
            "Strathfield" | "Epping" | "Epping" | "City"
    }
}
```

Use
example
data from
this table.

①

Action
under test.

Timetable service
should return
these lines.

This test builds on the example from the acceptance criteria to explore the requirements. But unit testing should be more thorough than acceptance testing. In this case, you use a table-driven approach ① similar to the JBehave table in listing 2.3. This makes it easy to add a more comprehensive set of examples, which is what you want at this level of detail.

Once you've implemented the `findLinesThrough()` method, you can proceed to the next function you need: finding the arrival times at a given station. Here you can use a similar approach, writing a new Spock specification as your starting point.

EXERCISE 2.1 Write the unit tests for the “Find the arrival times of a given line at a particular station” feature, and implement the corresponding code.

There are many possible ways to implement this service, and we won't delve into the details here. But along the way, you may discover other services or components that you need, which in turn you can mock out and then implement. Once there are no more mocked-out classes to implement, the acceptance criteria will run correctly, and you'll have finished the feature.

2.4.5 Tests as living documentation

Once a feature has been implemented, you should be able to run your tests and see passing acceptance criteria among the pending ones (see figure 2.9). When you're applying practices like BDD, this result does more than simply tell you that your application satisfies the business requirements. A passing acceptance test is also a concrete measure of progress. An implemented test either passes or fails. Ideally, if all of the acceptance criteria for a feature have been automated and run successfully, you can say that this feature is finished and ready for production.

More than just evaluating the quality of your application, the state of the tests gives a clear indication of where it's at in the development progress. The proportion of passing tests compared to the total number of specified acceptance criteria gives a

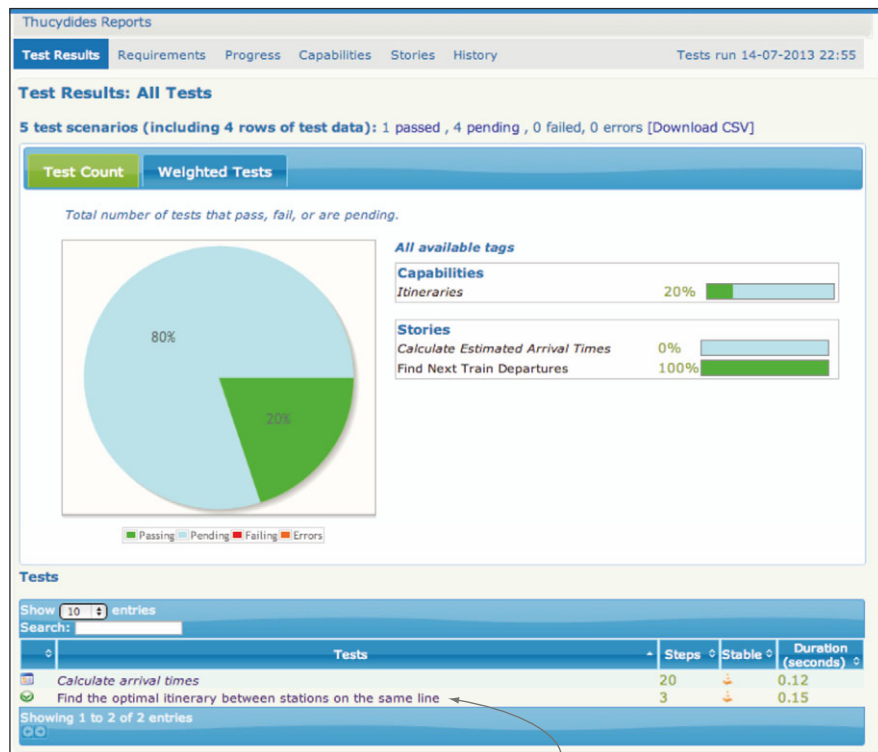


Figure 2.9 The passing test should now appear in the test reports

good picture of how much work has been done so far and how much remains. In addition, by tracking the number of completed automated acceptance tests against the number of pending tests, you can get an idea of the progress you’re making over time.

When you write tests in this narrative style, another benefit emerges. Each automated acceptance test becomes a documented, worked example of how the system can be used to solve a particular business requirement. And when the tests are web tests, the worked examples will even be illustrated with screenshots taken along the way.

But what about the testers? Automated acceptance testing and QA.

Automatically deploying your application into production when the automated acceptance tests pass requires a great deal of discipline and the utmost confidence in the quality and comprehensiveness of your automated tests. This is a worthy goal, and a number of organizations do manage this, but for most, things are not quite that simple.

In typical enterprise environments, the testers will probably still want to do at least some exploratory testing before releasing the application to production. But if the automated test results are clear and visible, they can save the QA team days or weeks of time that would normally be spent on regression or basic mechanical testing, and let them focus on more interesting testing activities. This, in turn, can speed up the release cycle significantly.

2.5 Maintenance

In many organizations, the developers who worked on the initial project don’t maintain the application once it goes into production. Instead, the task is handed over to a maintenance or BAU (Business as Usual) team. In this sort of environment, executable specifications and living documentation are a great way to streamline the hand-over process, as they provide a set of worked examples of the application’s features and illustrations of the code that supports these features.

Executable specifications also make it much easier for maintenance teams to implement changes or bug fixes. Let’s see how this works with a simple example. Suppose that users have requested to be informed about trains that are due to arrive within the next 30 minutes, and not just the next 15, as is currently the case.

The scenario related to this requirement is as follows:

Find out what time the next trains for my destination station leave

Narrative:

In order to plan my trips more effectively

As a commuter

I want to know the next trains going to my destination

Scenario: Find the optimal itinerary between stations on the same line

Given Western line trains from Emu Plains leave Parramatta for Town Hall at 7:58, 8:00, 8:02, 8:11, 8:14, 8:21

When I want to travel from Parramatta to Town Hall at 8:00

Then I should be told about the trains at: 8:02, 8:11, 8:14

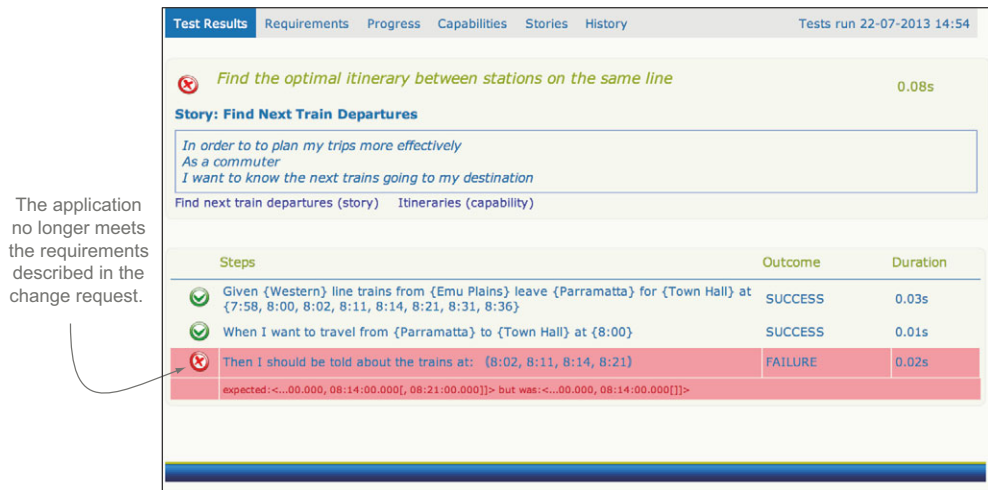


Figure 2.10 A failing acceptance criterion illustrates a difference between what the requirements ask for and what the application currently does.

This scenario expresses your current understanding of the requirement: the application currently behaves like this, and you have automated acceptance criteria and unit tests to prove it.

But the new user request has changed all this. The scenario now should be something like this:

You need more sample departure times.

Scenario: Find the optimal itinerary between stations on the same line
 Given Western line trains from Emu Plains leave Parramatta for Town Hall
 at 7:58, 8:00, 8:02, 8:11, 8:14, 8:21, 8:31, 8:36
 When I want to travel from Parramatta to Town Hall at 8:00
 Then I should be told about the trains at: 8:02, 8:11, 8:14, 8:21

You now want to know about all the trains in the next 30 minutes.

When you run this new scenario, it will fail (see figure 2.10). This is good! It demonstrates that the application doesn't do what the requirements ask of it. Now you have a starting point for implementing this modification.

From here, you can use the unit tests to isolate the code that needs to be changed. You'll update the "should calculate the correct arrival time" Spock specification to reflect the new acceptance criterion:

```
def "should calculate the correct arrival time"() {
  given:
    def westernLineFromEmuPlains =
      Line.named("Western").departingFrom("Emu Plains")

    timetableService.findLinesThrough("Parramatta", "Town Hall") >>
      [westernLineFromEmuPlains]

    timetableService
      .findArrivalTimes(westernLineFromEmuPlains, "Parramatta") >>
      [at(7,58), at(8,00), at(8,02), at(8,11), at(8,14),
       at(8,21), at(8,31), at(8,36)]
```

The pretend service now returns more trips.

```

when:
    def proposedTrainTimes = itineraryService.
        findNextDepartures("Parramatta", "Town Hall", at(8,00));

then:
    proposedTrainTimes == [at(8,02), at(8,11),
                           at(8,14), at(8,21)]
}

```

You now expect the itinerary service to return four times.

This, in turn, helps you isolate the code that needs to change in the `ItineraryService` class. From here, you'll be in a much better position to update the code correctly.

For larger changes, more work will obviously be involved. But the principle remains the same for modifications of any size. If the change request is a modification of an existing feature, you need to update the automated acceptance criteria to reflect the new requirement. If the change is a bug fix that your current acceptance criteria didn't catch, then you need to first write new automated acceptance criteria to reproduce the bug, then fix the bug, and finally use the acceptance criteria to demonstrate that the bug has been resolved. And if the change is big enough to make existing acceptance criteria redundant, you can delete the old acceptance criteria and write new ones.

2.6 Summary

In this chapter, you learned about the overall BDD project lifecycle. In particular, you saw the following:

- Understanding the underlying business objectives of a project lets you discover features and stories that can deliver these business objectives.
- Features describe functionality that will help users and stakeholders achieve their goals.
- Features can be broken down into stories that are easier to build and deliver in one go.
- Concrete examples are an effective way to describe and discuss features.
- Examples, expressed in a semi-structured “Given ... When ... Then” notation, can be automated in the form of automated acceptance criteria.
- Acceptance criteria drive the low-level implementation work and help you design and write only the code you really need.
- You can also use the BDD-style “Given ... When ... Then” structure in your unit tests.
- The automated acceptance criteria also document the features delivered, in the form of living documentation.
- Automated acceptance criteria and BDD-style unit tests make maintenance considerably easier.

In the following chapters, we'll discuss each of these themes in much more detail, and we'll also look at how the approaches we discuss can be put into practice using different tools and technologies.

Part 2

What do I want? Defining requirements using BDD

BDD principles are applicable at all levels of software development, from requirements discovery and definition right through to low-level coding and regression testing. Part 2 of this book focuses on the first aspect of BDD: how BDD is used to discover and describe the features you need to build. Part 2 is written for the whole team.

In chapter 3, we'll take a step back. You'll see how important it is to consider and understand the business motivation and value behind the software you're asked to deliver. You'll learn how to discuss the relative value of proposed features and use these discussions to determine what features to build, and, more importantly, what features not to build. This is at the heart of *building the right software*.

In chapter 4, we'll go into more detail. You'll learn how BDD teams collaborate to discover, describe, and define the features they need to deliver. You'll discover how to explore the scope and detailed requirements of a feature using conversation and concrete examples, and how these conversations can change your understanding of the features you're trying to define.

In chapter 5, you'll learn how to formalize the examples we discussed in chapter 4 in a clear, unambiguous form that we'll refer to as *executable specifications*. This too is a collaborative process; as you'll see, the act of expressing the

examples in a more rigorous form helps eliminate ambiguities and assumptions around the requirements, and it also paves the way for automation.

Finally, in chapter 6, you'll learn about tools that you can use to automate these executable specifications in different environments and languages, including Java, .NET, Python, and JavaScript. Automation is an important part of BDD; it forms the basis of the automated acceptance criteria that verify the features being delivered and of the living documentation that illustrates and documents these features.

At the end of part 2, you should have a solid understanding of why and how BDD teams collaborate to discover and define the features they need to deliver, and you'll be able to express the acceptance criteria around these features in the form of executable specifications.



Understanding the business goals: Feature Injection and related techniques

This chapter covers

- High-level requirements-analysis techniques
- Using Feature Injection to identify business goals and supporting features
- Visualizing high-level requirements with Impact Mapping
- Identifying stakeholders and their roles

Before you can implement a software solution, and before you can even judge what features you should implement, you need to understand the problem you're solving and how you can help. Who will be using the system, and what benefits will they expect from it? How will your system help users do their jobs or provide value to your stakeholders?

How can you know if a particular feature will really benefit the organization as you suppose it should? Are you building software that will have a measurable, positive impact for your client's business? Will your project make a difference? Sometimes a particular feature, or even a particular application, shouldn't be implemented because it will clearly not deliver the business benefits expected of it.

This makes for a lot of unknowns. And as we discussed in chapter 1, you'll rarely understand the requirements completely or correctly at the start of a project, and

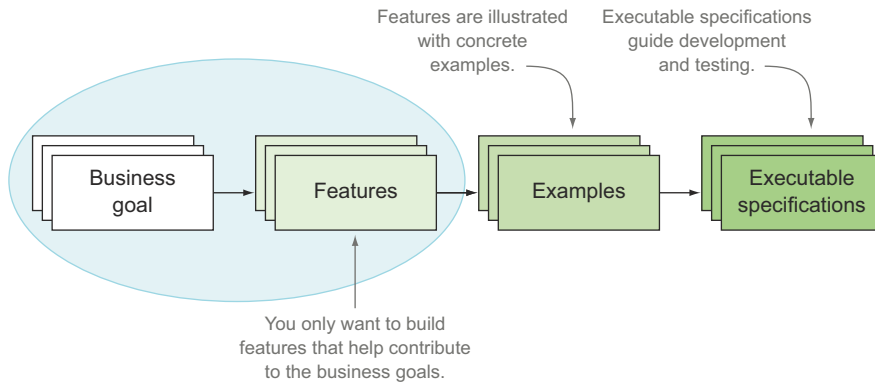


Figure 3.1 In this chapter we'll focus on discovering the features that will add real value to the business.

requirements and assumptions often change during the life of a project. It's unproductive to try to draw up detailed specifications in the initial phases of a project, when you know relatively little about it: this just sets you up for rework later, when the requirements, or your understanding of them, inevitably change. Instead, you should try to build a deep understanding of the business context behind the project so that you can react appropriately to change and remain focused on delivering the value that the business expects from the project. This is what we'll focus on in this chapter (see figure 3.1).

BDD places great emphasis on building “software that matters” and defines several processes for turning client requirements into something that developers can use to code against and that accurately reflects the core values of the software the client wants.

To ensure that the team is focusing on building valuable features, business analysts will find it useful to identify four things:

- 1 Why is the software being built (what is the project's *vision statement*)?
- 2 How will the project deliver value to the organization (what are the project's *business goals*)?
- 3 What *stakeholders* are involved in the project, and how will the project affect them?
- 4 What high-level *capabilities* should the software provide for stakeholders to enable them to achieve their business goals more effectively?

Techniques such as *Feature Injection*, *Impact Mapping*, and the *Purpose-Based Alignment Model* can help identify how the project will benefit the organization and what features can best realize those benefits:

- *Feature Injection* takes BDD beyond scenarios and stories and can help you discover what features you really need to satisfy the underlying business goals of a project.
- *Impact Mapping* helps you identify and visualize the relationships between business goals, stakeholders, and features.

- The *Purpose-Based Alignment Model* can help you judge how much effort you should put into different features.

But before you learn about all this, it's time to meet the (imaginary) client you'll work with throughout the rest of the book: Flying High Airlines.

3.1 Introducing Flying High Airlines

Flying High Airlines is a large commercial airline that runs both international and domestic flights. Flying High has been under pressure due to increasing costs and competition from low-cost carriers, so management has recently launched a new and improved version of their Frequent Flyer program to try to retain existing customers and attract new ones. This new program will offer many compelling reasons to join; like all Frequent Flyer programs, members will accumulate points when they fly, but members will also benefit from many exclusive privileges, such as access to lounges and faster boarding lines, and they'll be able to easily spend their accumulated miles on flights and on other purchases for themselves or their family members.

As part of this initiative, management wants a new website where Frequent Flyer members can see their current status in real time, redeem points, and book flights. The existing system just sends out paper account statements to members each month to tell them how many points they've accumulated. In addition, the Flying High call center is currently overloaded with calls, as Frequent Flyer members can only benefit from their member privileges and use their accumulated points if they book over the phone. Management hopes that being able to book directly online instead of over the phone will encourage Frequent Flyer members to book more often with Flying High.

In this chapter, and throughout the rest of the book, we'll use examples from this project to illustrate the concepts and techniques we discuss.

3.2 Feature Injection

The overarching approach we'll discuss in this chapter is called *Feature Injection*. Feature Injection is an approach that tries to identify the value that a project is meant to deliver and the features that will be able to deliver this value.¹ Initially elaborated by Chris Matts, and since championed by Liz Keogh² and other members of the BDD community, Feature Injection distills many techniques and approaches that BDD practitioners apply and have found useful in the wild. Feature Injection provides teams with a framework that helps focus BDD efforts on those features that will deliver real business value.

The aim of Feature Injection is to flesh out the minimum set of features that will provide the most benefit to stakeholders in terms of achieving their business goals. Feature Injection emphasizes the importance of engaging in an ongoing conversation

¹ See, for example, Chris Matts and Gojko Adzic, "Feature Injection: three steps to success," InfoQ (Dec. 2011), <http://www.infoq.com/articles/feature-injection-success>.

² Liz Keogh, "Pulling Power: A New Software Lifespan" (2009), <http://www.infoq.com/articles/pulling-power>.

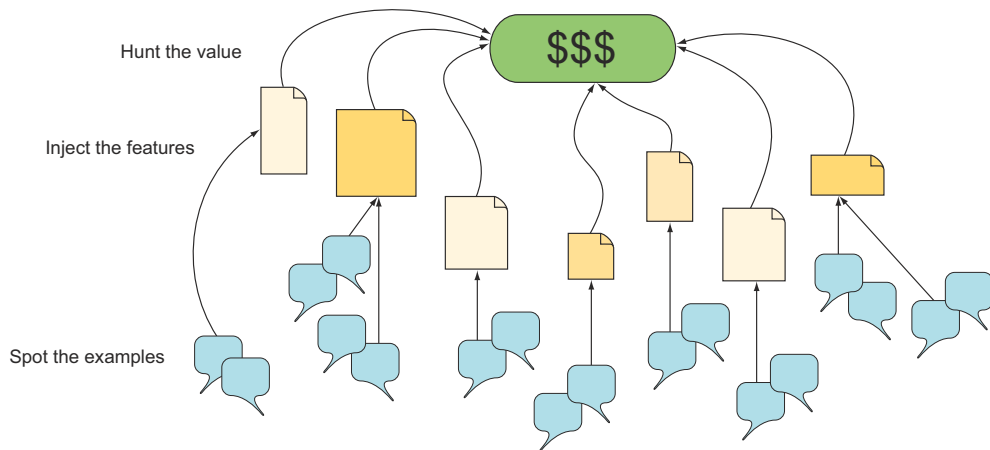


Figure 3.2 Feature Injection involves a three-step process: hunt the value, inject the features, and then spot the examples.

with stakeholders in order to progressively explore, elaborate, and expand on a shared understanding of what needs to be delivered and why. It revolves around a simple three-step process (see figure 3.2):³

- 1 Hunt the value.
- 2 Inject the features.
- 3 Spot the examples.

3.2.1 Hunt the value

Many Agile teams build up a large list of user stories (often referred to as a *backlog*), and then prioritize the stories to decide which features they really need to build in the next iteration. But sifting through stories to find the high-value ones takes time and effort, especially as the list of stories grows longer. Wouldn't it be easier if the stories you don't really need weren't placed in the list in the first place?

Feature Injection does things the other way round. In Feature Injection, you start by trying to clarify exactly how you expect the system to deliver business value, and only then do you identify the features that might best deliver this outcome. This makes it easier to focus your work on the features that will deliver the best return on investment. This is what's called "hunting the value." The aim is to understand the business value that lies behind a feature so that you can objectively decide which features are really worth creating.

Teams that use Feature Injection use a number of techniques that help model the relative value of features in terms of the business value they can provide. Don't be

³ Kent McDonald and Chris Matts, "Feature Injection: A Gentle Introduction," Agile 2009 Conference, <http://agile2009.agilealliance.org/node/185/>.

afraid of the word *model*: the best models are actually pretty simple. In Impact Mapping, for example, you use a simple mind-mapping approach to identify and compare the relative value of features, mapping them back to their underlying business goals. The Purpose-Based Alignment Model is another graphing technique that can help you decide how much effort you should invest in each feature. Models make it easier to discuss the assumptions that underlie the value proposition of each feature, and they prevent low-value features from slipping in unnoticed.

We'll look at several of these models and techniques in more detail later in this chapter.

3.2.2 Inject the features

As you've seen, once you've identified the goals of your project and how it's expected to deliver value to the business, you can objectively decide what features are worth including in the application. But you can do better than that. A team practicing Feature Injection doesn't just decide what features in the product backlog are worth doing; they proactively look for features that will deliver business value.

One of the fundamental tenants of Feature Injection is that the value of a piece of software generally comes from the outputs. For example, the value of an application comes from the information it provides, the sales that result from its use, or the reports it generates that allow users to make decisions. It comes from the financial transactions that it processes or from the sold products it ships. Value is to be found not in the inputs, but in the outputs.

When you want to know what features will deliver the most value, or the most return on investment, you should start with the outputs or outcomes. Take the outputs that will deliver the most value, and work back to find the minimum set of features you need in order to produce these outcomes. For example, you might identify the following features that could contribute to the goal of increasing sales revenue:

- Increase sales by allowing Frequent Flyer members to purchase flights with redeemed miles.
- Increase sales by allowing travellers to book flights online.

Again, this is quite similar to the Impact Mapping approach, where you start with a business goal and work back to discover the features that could deliver this goal.

3.2.3 Spot the examples

When you inject features this way, you only get a partial view of how a feature might work. A high-level feature description such as "Purchase flights with redeemed miles" or "Book a flight online" only gives a superficial idea of how a feature should behave. When it comes to implementing such a feature, you obviously need to know more.

Once you have a high-level set of features that you want to implement, you need to expand your understanding so that you can cater for the variety of inputs or behavior that may affect the outcomes. In other words, you need to expand the scope of the features that you'll build until you're satisfied that they'll deliver the business value you expect of them.

In Feature Injection, you use high-level examples to flesh out this missing scope. As you'll see, examples are a very effective way to communicate ideas about a problem domain and to discover and clarify what you don't know. Examples are intuitive and easy to understand, and they help flush out assumptions and misunderstandings. They can help you discover areas that need further investigation. They also act as the basis for more in-depth analysis, using both traditional business analysis techniques and more collaborative approaches.

We'll look at techniques and practices that can help in each of these stages in this chapter and the next. But before we do so, let's look at some of the terms we'll use.

3.2.4 Putting it all together

In a nutshell, Feature Injection involves discovering exactly how a system is expected to contribute value to the business and using real-world examples to determine the minimum set of features that will allow stakeholders to deliver this value. You can see how these ideas are interconnected in figure 3.3.

At the highest level is the *project vision*, a short statement that provides a high-level guiding direction for the project. This statement helps ensure that all the team members understand the principal aims and assumptions of the project. For the Flying High Frequent Flyer website, the vision could be to create a website that will enable Frequent Flyer members to consult and use their points and membership privileges more quickly and more easily than with the current manual system.

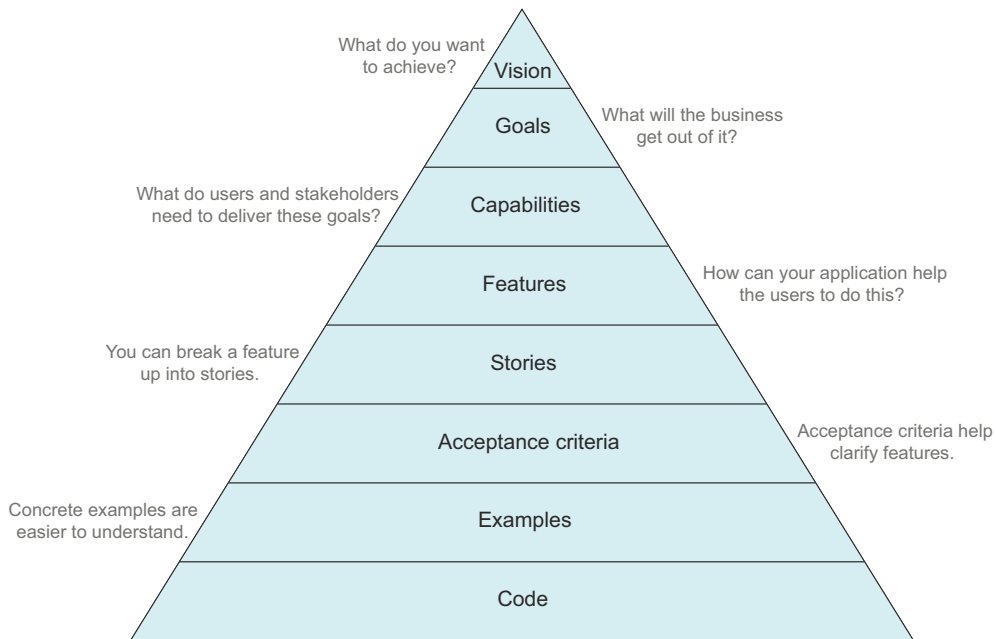


Figure 3.3 All features, and ultimately all code, should map back to business goals and the project vision.

More precisely, a project aims to support a number of *business goals*. Business goals are executive-level concepts that generally involve increasing revenue, protecting revenue, or reducing costs. For the Flying High Frequent Flyer website, one of the primary business goals might be “Earn more ticket sales revenue through repeat business from Frequent Flyer members.”

As a software developer, your job is to design and build *capabilities* that help the business realize these goals. A *capability* gives your users the ability to achieve some goal or fulfill some task, regardless of implementation. Liz Keogh says that a good way to spot a capability is that it can be prefixed with the words “to be able to.” For example, Flying High Frequent Flyer members need the capability *to be able to* book flights and benefit from their member privileges while doing so.

TIP *Capabilities* don’t imply a particular implementation. For example, “the ability to book a flight” could be provided online or over the telephone.

You design and implement *features* to deliver these capabilities. Features are what you actually build, and they’re what deliver the value. One feature that helps deliver the capability to book flights might be “Book flights online using Frequent Flyer points.”

TIP *Features* are pieces of deliverable software functionality, such as an “Online Frequent Flyer account summary.”

To build up your understanding of the features, you can use concrete *examples* of what the system should do in different situations. These examples can form the basis for the *acceptance criteria* of a feature. Once you have a set of features and scenarios to work with, you can start designing a user interface and writing the *code* that goes behind it.

TIP Examples illustrate how a feature works, such as “Account summary should display a list of recently earned points.”

Of course, the process is not as straightforward and linear as it might appear here. Feature Injection is a very iterative process. You try to identify the features that will provide the most business value, represent the most significant risks, or reduce your ignorance. You build and deliver these, and see what you can learn in the process. Then you adjust your assumptions based on this feedback, and repeat the cycle.

3.3 What do you want to achieve? Start with a vision

The first step in Feature Injection is “hunt the value.” But it’s much easier to identify the value that you expect out of a project if you have a clear idea of the overall vision of the project.

Suppose I was to ask you to go and buy me a new power drill for some DIY jobs I need to do around the house. I’m not sure how much a power drill costs these days, but I certainly don’t want to spend more than I have to. A red one would be nice, as it would go with my red toolbox.

Being an obliging sort of person, you drive down to the local hardware store and take a look at the range of power drills on display. But without a bit more background, you'd be hard put to make me happy. If I'm trying to assemble a large bed with a lot of screws, a small and inexpensive cordless electric drill, or maybe even a cordless screwdriver, should suffice, and a heavier model would just be cumbersome. On the other hand, if I'm going to install some new shelves on a brick wall in my garage, I'll need a more powerful hammer drill as well as a set of masonry drill bits and some screw anchors. If the garage has no power outlets, the drill will have to be of the more expensive cordless variety.

To get things done effectively, you need a clear understanding both of what you're trying to achieve and of the ultimate goal or purpose of your work. Studies have demonstrated that team collaboration improves significantly when members share a clearly identifiable goal.⁴ This is equally true for software development. If you are to deliver effective solutions for the problems your clients face, you need to understand the underlying business goals.

EXERCISE 3.1 In the previous chapter, you worked on a timetable application to publish timetables and real-time data feeds for train lines. That client also subsidizes bus companies for the bus trips they run. Currently, the bus companies send in paper forms detailing the bus trips they do each month, and they're paid on this basis. Management now wants you to write an application that keeps track of the real-time bus data and matches it against the subsidy requests that the bus companies send in. Try to articulate the business goals for this project in a few short sentences.

3.3.1 *The vision statement*

One useful way to express what a project is supposed to achieve is to write up a *vision statement*. A vision statement outlines the expected outcomes and objectives of the project in a few concise and compelling phrases. It gives the team an understanding of what they're trying to achieve, helping to motivate them and focus their efforts on the essential value proposition of the product. It provides a beacon for all to see, helping teams align their direction and concentrate their efforts on building features that actively contribute to the project's goals.

TIP The vision statement should be simple, clear, and concise—short enough to read and assimilate in a few minutes. It should also be intimately familiar to the whole team. Indeed, in the midst of an iteration, when requirements are changing quickly and frequently, it's all too easy for teams to get sidetracked by minor details, and the vision statement can remind them about the broader goals of the project.

⁴ Panagiotis Mitkidis et al., "Collective-Goal Ascription Increases Cooperation in Humans," PLOS ONE (May 2013) 8:5, <http://www.plosone.org/article/info%3Adoi%2F10.1371%2Fjournal.pone.0064776>.

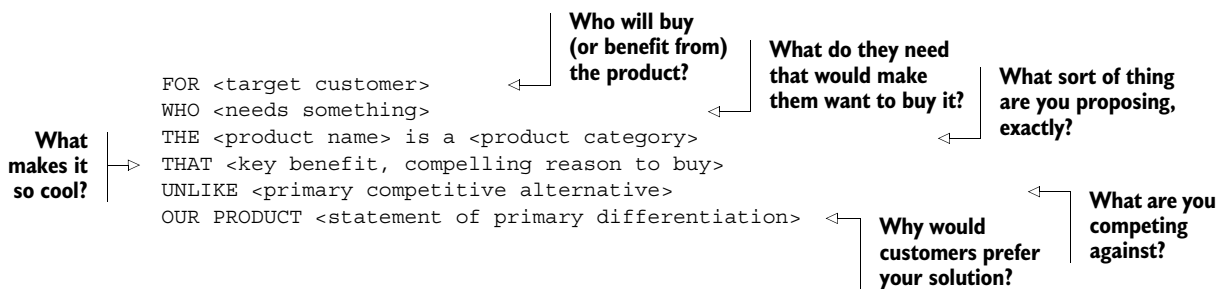
A good vision statement will focus on the project's objectives, and not on how it will deliver these objectives. It won't go into details about what technology should be used, what timeframe the project delivery should respect, or what platform it should run on. Rather, it presents the project objectives in the context of the problem the project is trying to address.

Most traditional projects generally do have a vision of some sort. Executive managers usually know, at least in the back of their minds, what they expect out of a project in terms of business value, or they wouldn't have approved it. By the nature of their jobs, executives usually have a very keen understanding of how a project is expected to contribute to the bottom line. Despite this, they may not have articulated the expected outcomes in a way that can be easily shared with the development team.

Larger organizations typically require some sort of formal documentation that explains the business case and scope of a project, and this document will usually contain a vision statement of sorts. But these are often dry, rigid documents, written to respect the local project management process requirements, and they're relegated to a project management office once the project has been signed-off on. They're the realm of project managers and project management offices (PMOs). In many cases, these documents never make it to the development team, which effectively renders them useless as vision statements in the Agile sense. Indeed, how can a shared vision be useful if not everyone can see it?

3.3.2 Using vision statement templates

A project vision statement need not be a long and wordy document. It can be a succinct paragraph that sums up the focus of the project in one or two sentences. In his book *Crossing the Chasm: Marketing and Selling High-Tech Products to Mainstream Customers* (Harper Business, 2002), Geoffrey A. Moore proposes the following template for a good product vision statement, which I've often heard mentioned in Agile circles:



For example, the vision statement for Flying High's Frequent Flyer program might go something like this:

```
FOR travellers
WHO want to be rewarded for travelling with Flying High Airlines
THE Flying High Frequent Flyer program IS A loyalty program
THAT lets members easily and conveniently view and manage their accumulated
    points in real time, and spend their points for real purchases with
    unequalled ease.
```


UNLIKE other airline Frequent Flyer programs,
OUR PRODUCT lets members use their points easily for any sort of online or
brick-and-mortar purchase.

This short text sums up who your target audience is, what they want, how your product will give it to them, and how your product distinguishes itself from its competitors. Of course, there's no obligation to follow this (or any other) template to the letter, but it does provide a good summary of the sort of things that you should consider including in your vision statement.

Writing a viable vision statement can be a tricky exercise, but it pays off over the life of the project. One effective approach to writing a good vision statement is to think in terms of designing a product flyer. In a nutshell, what does your product do? How will the product benefit your organization? Who is your target audience, and why would they buy your product rather than that of a competitor? What are its three or four principal selling points? Discuss these questions with all of the key project players, including users, stakeholders, and the development team. Then try to formulate a vision statement using Moore's template. If your project team is big enough, you can split up into several cross-functional groups and compare results, refining the statement until everyone agrees.

EXERCISE 3.2 Write a vision statement for the bus subsidies project from the previous exercise using Moore's template. How does it compare with what you wrote at the start of this section?

3.4 *How will it benefit the business?* *Identify the business goals*

Once you have a clear idea of the project vision, you need to define the underlying business goals that drive the project and contribute to realizing this vision. A business goal succinctly defines how the project will benefit the organization or how it will align with the organization's strategies or vocation.

All projects have business goals; otherwise management wouldn't have approved them in the first place. But not all projects have clearly defined and visible business goals. And projects with well-defined and well-communicated business goals have a much better chance of success than those that don't. A recent university study found that teams with a clear common understanding of their goal cooperated better and worked more effectively.⁵

Understanding these goals is even more important when unforeseen issues arise, when technical challenges make implementing a particular solution harder than initially thought, or when the team realizes they've misunderstood the requirements and need to do things differently. If developers are expected to respond appropriately to

⁵ Panagiotis Mitkidis et al., "Collective-Goal Ascription Increases Cooperation in Humans," PLOS ONE (May 2013) 8:5, <http://www.plosone.org/article/info%3Adoi%2F10.1371%2Fjournal.pone.0064776>.

this sort of challenge, they need to have a solid understanding of what business value is expected from the system.

At the end of the day, business people want the software being built to help them achieve their business goals. If the software delivers in this regard, the business will consider it a success, even if the scope and implementation vary considerably from what was originally imagined. But if the software fails to meet the underlying business goals, then it will rightly be considered a failure, even if it meets the requirements provided by the customers down to the letter.

3.4.1 Writing good business goals

Business goals are high-level statements that focus on business value or opportunity, statements that executives would be able to relate to and discuss. For example, one of the more important business goals for the Frequent Flyer program discussed earlier might be the following:

```
Increase ticket sales by 5% over the next year by encouraging travellers to  
fly with Flying High rather than with a rival company.
```

A statement like this has the merit of clarifying the “why” of the project: why exactly are you building this software? But like the project vision, business goals are often very poorly communicated. Many development teams have only a vague notion of the business goals they’re trying to deliver. And yet, understanding these goals is essential to making correct scope, design, and implementation choices during the project.

You can also write goals using the following “In order to ... As a ... I want to ...” format:

```
In order to increase ticket sales by 5% over the next year  
As the Flying High Sales Manager  
I want to encourage travellers to fly with Flying High rather than with a  
rival company
```

This is a variation on the more traditional “As a ... I want ... so that ...” form used in many project teams for story cards. This form puts the goal first (“In order to ...”) as opposed to the more traditional user story format, where the stakeholder is placed in the primary position. Putting the goal first places the emphasis on the expected outcome, which should be of obvious value to the business. When the stakeholder comes first, the last section (“so that ...”) tends to be eclipsed, and you may find yourself proposing high-level capabilities and then struggling to justify them with a solid business goal.

This is a very versatile format, and you’ll use it to describe requirements at various levels of abstraction throughout the book.

Whatever format you prefer, a good business goal should be precise. Some business managers use the SMART acronym: business goals should be

- Specific
- Measurable
- Achievable
- Relevant
- Time-bound

First and foremost, a goal should tell readers specifically what you're trying to do ("encourage travellers to fly with Flying High"). But it should also describe why you want to do this and outline the business benefits you're trying to achieve by doing so ("increase ticket sales").

A goal should also be measurable. A measurable goal will give you a clearer idea of business expectations and also help you determine whether it has been achieved once the work is done. You can make a goal measurable by introducing notions of quantity and time. For example, "Increase ticket sales by 5% over the next year."

Quantifying a goal in this way will also make it much easier to determine how best to approach the problem, or whether the goal is even achievable in the first place. Increasing sales by 200% in 6 months would be quite a different proposition than increasing by 5% over the next year. Knowing the timeframe (6 months) also makes the goal more tangible and focused. In all of these cases, precise figures help manage expectations and ensure that everyone is on the same page.

But possibly the most important attribute of a good goal is its relevance. A goal is relevant if it will make a positive contribution to the organization in the current context within the specified timeframe, and if it is aligned with the overall strategy of the organization. Relevant goals are goals that will make a difference to the business.

3.4.2 Show me the money—business goals and revenue

Let's take a closer look at the business goals behind the Frequent Flyer program. A more complete list of goals might include the following:

- Increase ticket sales revenue by 5% over the next year by encouraging travellers to fly with Flying High rather than with a rival company.
- Increase the customer base by 10% within a year by building a positive image of the Frequent Flyer program.
- Avoid losing existing customers to the new rival Hot Shots Frequent Flyer program.
- Reduce hotline costs by enabling Frequent Flyer members to purchase flights with their points directly online, unlike the current program, where travellers need to call to make a booking.

Notice that the preceding goals all seem to boil down to earning more money, either by increasing revenue or by reducing costs. The goals of most commercial organizations are, by definition, ultimately financial in nature. In fact, almost all business goals can be grouped into one of the four following categories:

- Increasing revenue
- Reducing costs
- Protecting revenue
- Avoiding future costs

For example, "Increasing ticket sales revenue by 5%" (the first of the goals previously listed) is clearly about increasing revenue by selling more tickets. The second, "Increase

the customer base by 10% within a year” is similar, though the connection with generating revenue is slightly more indirect. “Avoid losing existing customers” is an example of protecting revenue, whereas the fourth, “Reduce hotline costs,” is clearly about reducing costs.

An example of avoiding future costs might be implementing certain reports that will be required by new legislation due to take effect next year. In this case, you’re investing effort now in order to avoid fines for noncompliance in the future.

Business goals in a public service organization—Transport for NSW

Transport for NSW, the government agency responsible for public transport in New South Wales, Australia, recently implemented a project to provide Google Maps with real-time scheduling information for trains and buses, as is done in many large cities around the world. Travellers can use Google Maps not only to find public transport options between two locations, but also to find out when the next bus or train is scheduled to arrive, based on real-time data coming from the buses and trains.

There’s no profit motivation here: the aim is to allow public transport passengers to be able to use the public transport network more efficiently and reduce the time passengers need to wait for public transport. The motivation behind this may be increasing customer satisfaction. Happier passengers are more likely to use this service instead of any alternatives. A secondary goal could be to save people time, freeing them up for more productive work, which in turn benefits the local economy. This project will generate no increased revenue for the government agency implementing the project.

This rule still holds true for government or non-profit organizations, but in a slightly modified form. Public services, for example, are not as interested in generating revenue as they are in providing valuable services to the public. This type of organization also typically has an externally defined budget: the service is responsible for how taxpayer dollars are spent, so cost plays a critical role in business goals. In general, the business goals for projects in non-profit organizations tend to fall into the following categories:

- Improving service
- Reducing costs
- Avoiding future costs

EXERCISE 3.3 Identify some business goals that might be applicable to the bus subsidies project in the previous exercise. Propose concrete strategies to measure how well the application discussed in exercise 3.1 achieves these goals.

3.4.3 Popping the “why stack”: digging out the business goals

If you want to build software that delivers real value to your customers, understanding the core business goals of a project is essential. Unfortunately, neither business sponsors nor end users typically express their needs in terms of pure business value. Instead, they talk in terms of concrete features or solutions that they have in mind. In

addition, stakeholders are rarely capable of providing all the requirements up front, even if they do know them ahead of time.

Stakeholders in large organizations have often been trained to believe that they must provide a detailed list of all their requirements at the start of a project. In traditional Waterfall-style projects, once the requirements have been signed-off on, the project management process deliberately hinders any changes, corrections, or additions that users might want to make: this is known as *change control*. This dissuasive change-control process tends to result in very large sets of requirements full of everything the stakeholder can possibly think of, with many features being included just because they “might come in handy” someday.

It would be easy to take the feature requests that come out of such a process at face value. After all, shouldn’t the users know better than anyone else what they want? Why not just note down what they want, and then do it? There are two very good reasons why this would be a bad idea.

First, the people who produce these requirements—the business sponsors, the end users, and so forth—are typically not well versed in the technologies that will be used to deliver the solution. The features they request may not be optimal from a technical viewpoint (there may be better ways of getting the job done), from a usability perspective (they are not UX experts), or from a financial one (you may be able to get the job done faster and cheaper using a different approach). The development team has a professional responsibility to propose the most appropriate solution for a given problem. But if the business goals are not clearly expressed and understood, you’ll be hard put to suggest potentially more appropriate solutions.

But there’s a second reason that’s arguably even more important than the first. If you don’t know why you need to deliver a feature in a particular way, you’ll have no way of knowing whether this feature is still useful or relevant when change happens.

Any software project is an exercise in ongoing learning. Things inevitably change along the way, including assumptions and even the team’s understanding of how a particular feature might benefit the business. A requirement that’s expressed in the form of a detailed technical solution is embedded in a fabric of assumptions and preconceptions. You can be sure that some of these assumptions will turn out to be incorrect, or will change along the way, but it’s hard to predict which ones will be wrong. When changes do happen, you need to be able to reassess the relevance of the features that you’re building, but if you don’t know why the business wants a feature, this will be much more difficult.

You usually need to drill down a little to discover what the business really needs. One of the best ways to do this is to repeatedly ask “why” until you get to a viable business goal. As a rule of thumb, five why-style questions are usually enough to identify the underlying business value (see figure 3.4).

Let’s look at a practical example. Suppose you’re working for a large media corporation. The director of marketing has come to you with a requirement that seems simple enough on the surface: “I want people to be able to post their printed classified ads online.” To learn a little more, you might respond with something along the

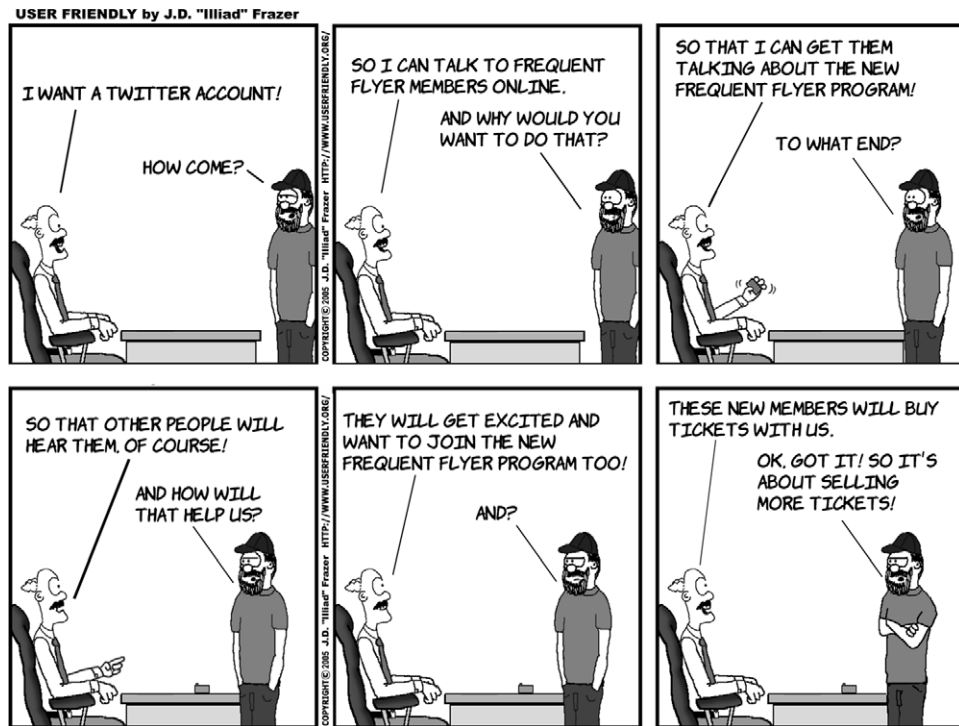


Figure 3.4 It's always important to understand why you're undertaking a project.

following lines: "I am but a humble developer, unversed in the art of newspaper advertising. Why do you need to enter the classified ads online?"⁶ To which, the marketing director replies, in a slightly annoyed tone, "Because people don't read the classifieds anymore; they prefer to go online."

Objective-driven management

Don't tell people how to do things, tell them what to do and let them surprise you with their results.

George S. Patton

General George S. Patton was a very successful American general known for his achievements in the European theater during the Second World War and for his unconventional and dynamic leadership style. Patton was once asked at a press conference how he was able to get such outstanding competence and devoted loyalty from his staff. "I never tell people how to do things," replied Patton. "I tell them what to do but not how. If you give people responsibility, they will surprise you with their ingenuity and reliability."

⁶ OK, so you might skip the first bit.

At this point, you still haven't really identified any value that the requested system might deliver, so you could follow up with another question: "Why is that a problem for us?" To which the director relies, "Because revenue from the classified ads service is falling."

So now at least you're talking about financial impact. Because you're still getting answers, you might press your luck a bit further: "And why will entering the ads online stop our classified ads revenue from falling?" To which the director answers, "Because more people will be able to browse our classified ads online."

"So, to make sure I have the whole picture," you reply, "how will more people seeing our ads online stop our classified ads revenue from falling?"

"Well, if more people can browse the ads online, the products are likely to sell faster. We earn a commission on each sale, so the faster the products sell, the more we earn."

So here we have a clear mapping back to a measurable business goal: you want to place the ads online so that more people will see the ads and hopefully purchase the advertised goods. Because the company earns a commission on each item sold, this will increase the classified ads revenue.

This process, sometimes known as "popping the why stack," is a powerful analysis tool. Discovering and crystalizing the value proposition behind the features that users ask for not only helps you understand why they're asking for these features but also puts you in a position to evaluate their relative value and to adapt them to changing circumstances.

We'll develop this concept further in the next chapter. But first, let's look at how you can discover what capabilities a system will need to realize its business goals.

EXERCISE 3.4 Management has asked for a feature that can send SMS alerts to managers whenever a bus company claims to have run more trips than they did according to the real-time data. Team up with a colleague, with one person playing the role of the business stakeholder. Use the "popping the why stack" approach to identify the underlying business goal. Once you've identified a goal, discuss whether this is the only or best way to deliver this goal.

3.5 *Impact Mapping: a visual approach*

In Feature Injection, the "hunt the value" step involves identifying how your project will deliver value to the organization. In the second step, "inject the feature," you flush out the features that can deliver this value. In his book *Impact Mapping* (Provoking Thoughts, 2012), Gojko Adzic describes an interesting, visually-based approach to the problem of aligning the software features you deliver with the underlying business goals.

Impact Mapping is a way of visualizing the relationship between the business goals behind a project, the actors that will be affected by the project, and the features that will enable the project to deliver the expected results. Impact maps are simple to create and easy to understand, and they can be a very useful tool for documenting business goals and validating assumptions. This makes them a great place to kick off

the requirements-analysis process and to get a high-level view of what you're trying to achieve.

An impact map is a mind-map built during a conversation, or series of conversations, between stakeholders and members of the development team. The conversation centers around four types of questions:

- Why?
- Who?
- How?
- What?

The first question you need to ask is *why* you're building the software in the first place. Or, in other words, what are the underlying business goals you're trying to achieve? This corresponds to the business goals we discussed earlier.

Visually, each impact map starts with a single business goal at its center. For example, using our Frequent Flyer example, you might start with the first goal, "increase ticket sales revenue by 5% over the next year by encouraging travellers to fly with Flying High rather than with a rival company." This would be the first node in your impact map, as illustrated in figure 3.5.

In Impact Mapping, the second question to ask is *who*. Who are the stakeholders impacted by the project? Who are the users of the product? Who will benefit from the outcomes? If you're selling something, who are your customers? Who will be in a position to cause or influence the outcome you're trying to achieve? Who can prevent your project from being a success?

This is an area of requirements analysis that's often overlooked or skimmed over with a few token user roles. But the best-designed feature in the world will be of little value to the business if the users never use it or don't use it in the anticipated way. Stakeholders have their own motivations and agendas, which can be different from those of the project, and which you need to understand in the context of what you're trying to deliver. Oftentimes, to achieve a business goal, it's not enough to write software: you must also succeed in changing people's behavior.

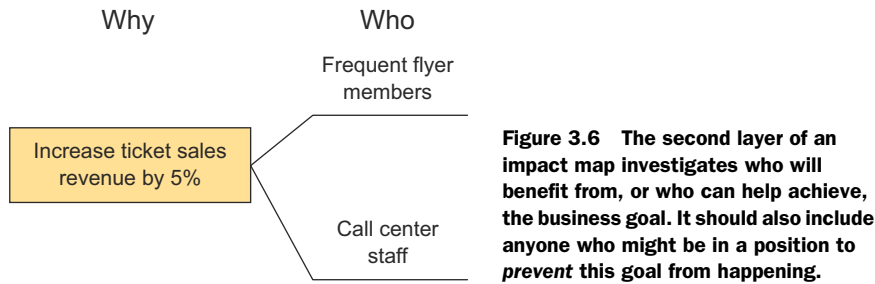
Of course, one of the most important stakeholders in almost any project is the customer or end user. For example, the Frequent Flyer website will only be a success if it can encourage travellers to become Frequent Flyer members and book more flights with Flying High airlines. To do this, you'll need to understand and address the needs of your existing and potential customer base and deliver features that address these needs. If you're building an internal application for internal users, you'll need to understand what features will enable your users to help achieve the higher-level business goals in the most efficient way possible.

But customers and end users are not the only stakeholders that you need to consider. Executives will be interested in how the project helps them meet their business

Why

Increase ticket sales
revenue by 5%

Figure 3.5 An impact map starts with the business goal you want to achieve.



objectives. System administrators may have security and architecture constraints that need to be catered to, and so on.

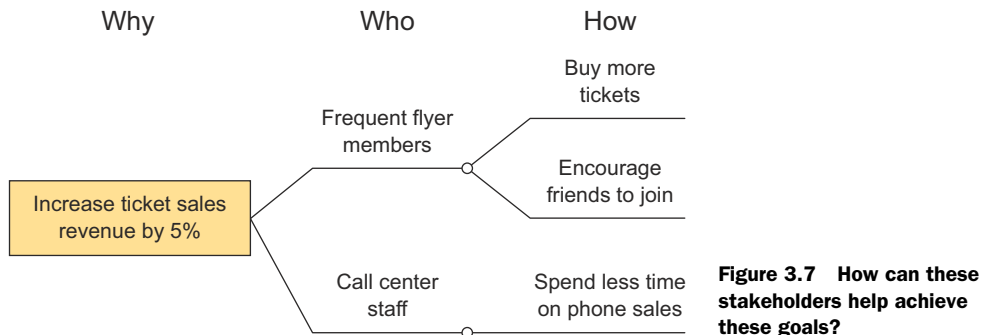
For the Frequent Flyer project, you could identify several significant stakeholders who might have an influence on the “increase sales” business goal. For the sake of simplicity, let’s focus on two: existing Frequent Flyer members and call center staff (see figure 3.6).

The next question to ask is *how*. How can stakeholders and users contribute to achieving the business goals? How could their behavior or activities change to better meet these goals? How could you make it easier for them? How might they or their behavior prevent or hinder these goals from being met, and cause the project to fail?

Here you’re thinking about how the system might make it easier for stakeholders to contribute to the business goals, or how you can influence their behavior and encourage them to do so in other ways. You’re expressing things from the point of view of the stakeholder, which emphasizes that you’re trying to write software that’s going to change the way people do things, preferably in a positive way for the business. For example, to increase ticket sales revenue, existing Frequent Flyer members could buy more tickets, but they might also encourage their friends to join, who would in turn buy more tickets.

Telephone sales are time-consuming and costly. Reducing the time taken to sell a ticket over the phone could also contribute to increasing sales revenue by reducing the number of call center staff required.

You could illustrate these concepts in the impact map as shown in figure 3.7.



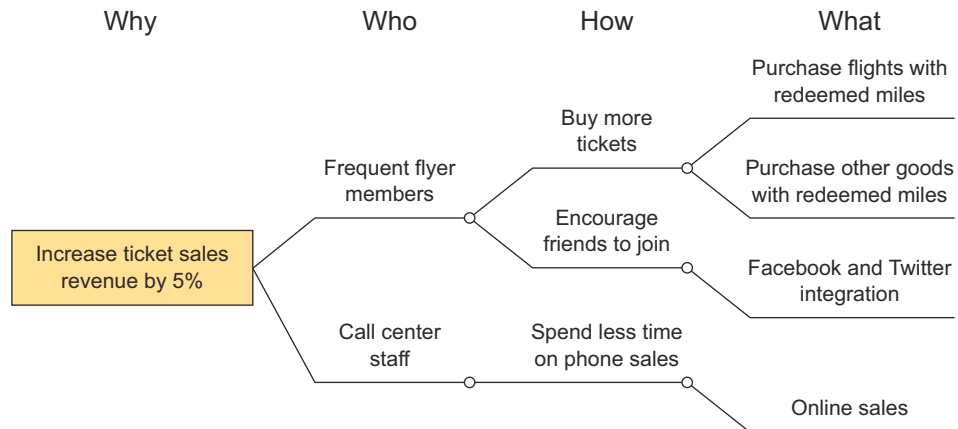


Figure 3.8 What can you do to encourage or allow the stakeholders to modify their behavior to help achieve the business goal?

The last type of question you need to ask is *what*. What can your application do to support the impacts you’ve listed in the previous three stages? Or are there other ways to achieve these results without using software? In the context of building an application, the “what” corresponds to high-level features. You may need to break these high-level features down into more detailed ones as your understanding of the requirements increases, until you get to a manageable size. In an Agile project, these features are good candidates for high-level user stories or epics. If you’re using a Unified Process methodology, they might become use cases.

In our example, you might encourage Frequent Flyer members to buy more tickets by making it easier to redeem miles on flights or on other products (see figure 3.8). You might make it easier to talk about the great deal they just got with their Frequent Flyer discounts on Facebook or Twitter. And you might reduce the number of telephone bookings by determining what sort of bookings are being done over the phone and making online purchases for these tickets easier.

Impact Mapping helps you visualize how features and deliverables contribute to a business goal. It also helps underline any assumptions you may be making. For example, here you’re assuming that Facebook and Twitter integration will be enough for Frequent Flyer members to proudly post about all their trips. Impact Mapping encourages you not only to sketch out these relationships and assumptions, but also to define metrics that you can use when the feature goes live to test whether your assumption was correct. In this case, you might keep track of the number of Facebook and Twitter posts that are generated by Frequent Flyer members in order to learn how effective your new social media integration strategy really is.

It’s important to remember that impact maps are not plans: they’re iterative, living documents that can help you visualize assumptions and relationships between capabilities and business goals. Whenever you deliver something into production,

you should be able to validate certain assumptions, and you may prove others wrong. Based on this feedback, you'll be able to update your impact map and your short-term plans accordingly.

EXERCISE 3.5 Flying High Airlines partners with the major credit card providers. They issue credit cards that members can use to accumulate Frequent Flyer points when they make ordinary day-to-day purchases. Flying High Airlines receives commissions when Frequent Flyer members purchase goods this way. Identify the underlying business goal behind this strategy, and draw up an impact map to discover how the Frequent Flyer website might contribute to this goal.

Impact Mapping is a simple and convenient approach to building up an initial picture of what you're trying to achieve in a project. Impact maps are visual and intuitive, fast to draw, and accessible for both business people and technical folk.

Impact maps can also be used quite effectively in the other direction, though for a slightly different purpose. Suppose, for example, that you already have a set of proposed features: a product backlog in an Agile project, a set of use cases, or even a set of high-level requirements in a more traditional requirements-specification document. Suppose, while you're at it, that the requested features come from different stakeholders, who are each convinced that "their" feature should be done first.

In this case, impact maps make a great conversation starter. I've found the following strategy effective: get your stakeholders together, put the requested features on a whiteboard, and identify who they'll benefit and how they'll benefit them, working back to the underlying business goals. Eventually you'll end up with a graph that shows a number of goals, with a relatively clear illustration of which features map to which goals. This visual representation of all of the goals and the supporting features is an excellent starting point for a discussion of the relative merit of each goal, and of each feature, and makes it much easier to prioritize the different features more objectively.

If you're interested in learning more about Impact Mapping, take a look at the Impact Mapping website (<http://impactmapping.org>) and read the book *Impact Mapping* by Gojko Adzic.

3.6 Who will benefit? Identify stakeholders and their needs

As you've seen, all projects ultimately aim to benefit the organization in some way. But organizations are made up of people, and it will be people within this organization (or people who interact with this organization) who will be affected by the outcomes of your project. These people are the *stakeholders* or *actors*. The impact maps we looked at earlier emphasize the importance of stakeholders in identifying valuable features. Let's look at this in more detail.

Note that even when the overall effect of a project is designed to be beneficial for the organization, the impact of the project on an individual basis may be negative. For

example, adding additional steps in a mortgage application process may be perceived as being cumbersome and annoying for the banker selling the mortgage and for the client applying for a house loan. But the net effect of reducing the risk of bad loans may be considered beneficial enough for the bank as a whole to outweigh these disadvantages. It's important to be aware of these negative impacts, and to try to minimize them where possible.

Many different types of stakeholders will be interested in the outcomes of your work. Future users of your product are probably the most obvious example; their daily work will be impacted by your project, for better or for worse. For the Frequent Flyer website, this category of stakeholder will include current and future Frequent Flyer members. It may also include call center staff, who will have to answer questions from Frequent Flyer members about the new program and be able to view a client's current status and miles. They may also need to perform more complicated tasks that are not yet supported on the Frequent Flyer website, such as transferring Frequent Flyer miles between family members or crediting earned miles manually.

Note that future users may not always be directly interested in the business goals. A Frequent Flyer member has no particular reason to want to see an increase in Flying High ticket sales. Sometimes what the user wants and asks for will have little or no bearing on the original business objectives. When you think about what features you need to deliver to achieve the business goals, you need to think in terms of how you can encourage these stakeholders to behave in a way that helps you achieve the goals. For example, you'll want to provide features that encourage users to book flights with Flying High rather than with a competitor in order to increase ticket sales revenue.

Another category of stakeholder includes people who won't be using the application themselves, but who are directly affected by, or interested in, the outcomes of the project. For example, Bill is the Director of Sales at Flying High Airlines and is therefore responsible for ticket sales. An increase in ticket sales revenue of 5% will help him meet his own goals of increasing sales revenue in general. In addition, his budget is paying for the new Frequent Flyer members' website, so understanding and quantifying this business goal will give Bill a way to justify the cost of the project in terms of ROI (return on investment). With this information, Bill can see how much the project is likely to earn in increased revenue, and therefore have a clearer idea of how much he's willing to spend to achieve these increased earnings.

Other stakeholders will not be directly affected by the project, but will want to have a say in how the project is implemented, and may have the power to block the project if they aren't contented. Regulators, security folk, and system administrators are good examples of this sort of role.

The actual role of the various stakeholders in achieving business goals and delivering business value is often overlooked. If users don't use the application in the expected way, the software may not realize the benefits you expected. Looking at things from another angle, a very effective way to identify the features that are most likely to support the business goals and provide real value is to think in terms of

changing stakeholder behavior. How can you encourage users to behave in a way that supports the business goals?

Once you've identified the principal stakeholders and understood their goals, you can think about the *capabilities* that you need to provide to meet these goals.

3.7 What do you need to build? Identify capabilities

A *capability* is the ability of your application to help stakeholders realize a business goal. Capabilities are high-level concepts that don't commit to a particular implementation. Because they're implementation-neutral, they give you a lot of flexibility as to how you build the underlying features.

I was once called in to help out a software development shop that specialized in the real estate market. They were trying to build a new application to help real estate agents provide property management services for property owners.

One of the aims of the application was to compete with and beat an existing rival product that already had a good hold on the market. After reading the competition's user's manual, the business owner listed the requirements that he wanted us to implement. It went along the following lines:

- Add a new property owner
- Archive a property owner
- Delete a property owner
- Send notifications to property owners
- ...

These requirements were quite possibly necessary for the application to work correctly, but it was less clear precisely what role they played in meeting the application's business goals. Remember, every feature a developer builds needs to be delivering value to the organization in some way (see figure 3.9). For example, why did they need to add a new property owner? How many new property owners would be added

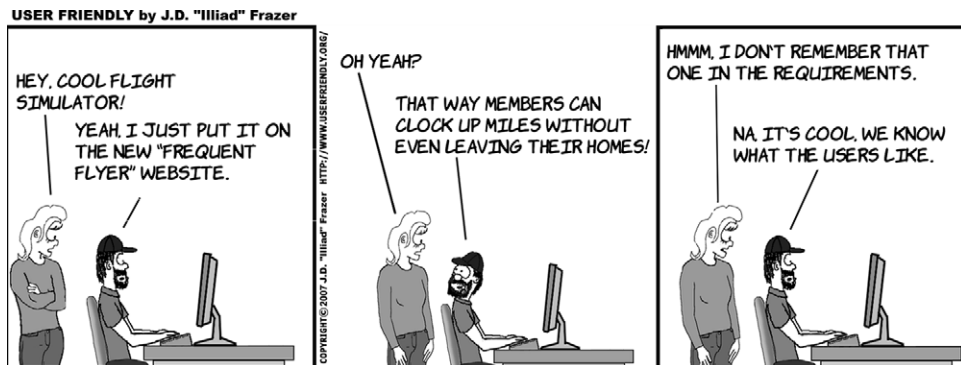


Figure 3.9 The ultimate purpose of any new feature is to deliver value to the organization. If it's not doing this in some way, it is probably waste.

per month? Is this likely to change over time, and, if so, how quickly? Viable implementations of this capability could range from a manual SQL insert performed by a system administrator to a fully automated web interface where property owners sign up and create their own accounts with no manual intervention. Until the business goals were clearly defined, they had no way of knowing. These requirements were also quite low-level, and gave no context or background as to why they might want to add a new property manager or send property owners notifications, for example.

As we discussed earlier, this “shopping-list” style of requirements doesn’t lend itself well to understanding why a particular feature is needed and makes it hard to adapt to change when your understanding of the true business needs evolves, when new requirements emerge, or when technical or nontechnical issues force you to change your tack.

To rectify this, the real estate development team tried to determine what it was that the application was trying to achieve. After popping the why stack a few times, they managed to identify the following core business goal:

```
Goal: Win more management contracts by providing better service to property
      owners
In order to win more management contracts
As a property manager
I want to be able to provide better service to property owners
```

Supporting this goal, they came up with a set of high-level capabilities that included the following:

```
Capability: Notify the property owner at promised times
In order to provide property owner-specific notification schedules
As a property manager
I want the property owner to be able to set their preferred notification
      schedule
```

```
Capability: To have peace of mind regarding managed properties
In order to provide property owners easy access to their statements
As a property owner
I want to be able to access vital statistics about my managed properties at
      any time
```

```
Capability: To ensure that I pay the promised recurring expenses
In order to manage recurring expenses
As a property manager
I want the payments to be done automatically or be reminded about them if I
      need to intervene
```

Not all capabilities are focused on business functionality. Some, such as security and performance, can apply across the whole system. For example, they were able to define the following security capability:

```
Capability: To keep confidential information safe
In order to ensure that my customer information is only available to my
      employees
As a property manager
I want client data to be only transmitted over a secured connection
```

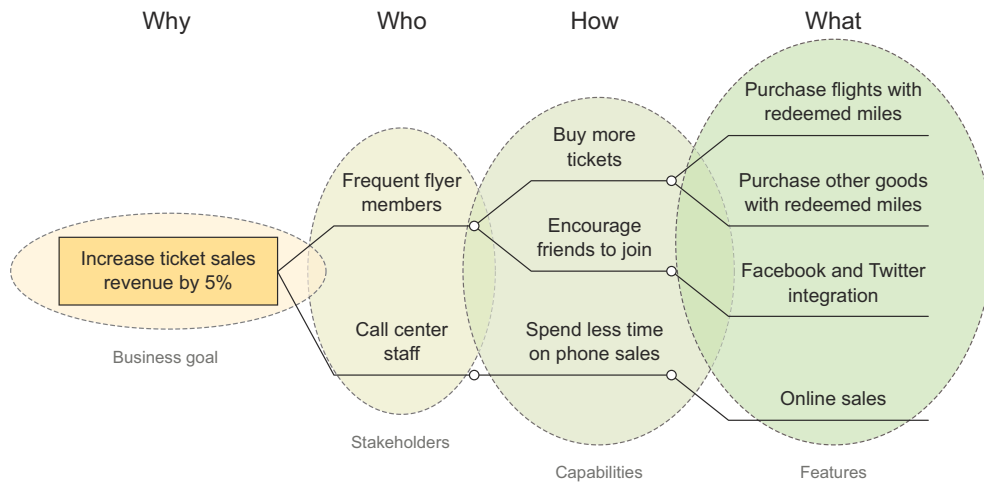


Figure 3.10 Impact maps are a good way to explore goals, capabilities, and features at a high level.

In another situation, they needed to be able to deliver reliable service to property managers in rural areas, where internet connections are notoriously slow:

Capability: To provide a viable service in rural areas

In order to benefit from the service when working in remote rural areas

As a property manager

I want to be able to still use the essential features of the system effectively over a slow internet connection

From here on in, they were able to start having conversations about features that they could build to deliver these capabilities: what sort of statistics would a property owner need to see, what would it look like on the screen, what would the user experience be like for the property owner, and so on.

Features are the implemented components in an application that deliver a capability to the users. For example, several different features might have helped the development team deliver the “provide property owners easy access to their statements” capability mentioned previously, such as a property dashboard page on the client website or an iPhone property-management dashboard app. We’ll discuss the concept of features in much more detail in the next chapter.

Impact maps, as you saw earlier, can help you explore these ideas at a higher level, before discussing each capability or feature in more detail (see figure 3.10).

3.8 **What features will provide the most ROI?** **The Purpose-Based Alignment Model**

Feature Injection can help you decide what features you really must have in order to deliver the business value you need to deliver. But there’s another dimension that’s also worth considering. Not all features are equal. Some features will be areas of innovation, requiring specialized domain knowledge and expertise and adding significant

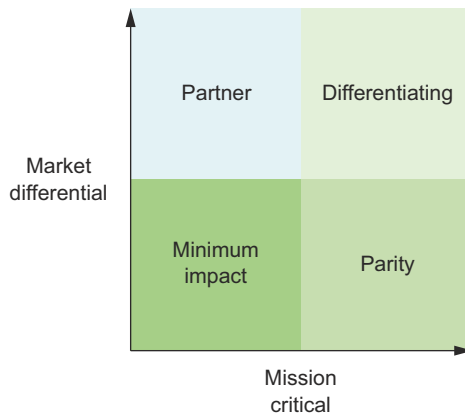


Figure 3.11 The Purpose-Based Alignment Model helps identify where you should focus your efforts when building features.

value. Others, such as online payment with credit cards, might be necessary in a market, but won't distinguish your product from the competition in a meaningful way. Knowing where to invest your time and effort is essential in ensuring that your product not only provides value, but also provides high return on investment.

One convenient way to measure this is to use the Purpose-Based Alignment Model, invented by Niel Nickolaissen.⁷ Using the Purpose-Based Alignment Model, you classify features into four quadrants of a diagram, like the one in figure 3.11, using two simple criteria:

- *Mission critical*—How essential is the feature for your ability to operate as a business?
- *Market differentiation*—Is it significantly different from what your competitors propose? Will it make you stand out in the marketplace?

Depending on where a feature is positioned in this diagram, you can decide how best to focus your efforts on those features that will make a difference. A feature will fall into one of four categories:

- Differentiating
- Parity
- Partner
- Minimum impact

3.8.1 Differentiating features

A *differentiating* feature is both mission critical and marks your product apart from the competition. These are the features that provide high return on investment or that win market share, or both. They give your organization a competitive edge. An example might be a risk analysis tool for an investment bank or an innovative new way to reconcile bank account statements in an online accounting package.

⁷ Pollyanna Pixton, Niel Nickolaissen, Todd Little, and Kent McDonald, *Stand Back and Deliver: Accelerating Business Agility* (Addison-Wesley Professional, 2009).

These features typically rely on specialized domain knowledge and skills, so it makes sense to build them with in-house resources. Features in this category need to be highly innovative in order to gain and maintain a competitive advantage.

3.8.2 Parity features

A *parity* feature is mission critical but doesn't provide much market differentiation. These are features that aren't particularly glamorous but are required for the product to be viable. All your competitors provide these features, and it's expected that you do too. An example of a parity feature might be online payment with credit cards for an e-commerce site, or the ability to view old account statements for an online retail bank.

As the name suggests, you need to implement these features to provide roughly the same level of service as is found in competing products. If you do any less than that, you'll put your organization at a competitive disadvantage. But it would be wasteful to spend too much effort on these features, as the return on investment will be marginal.

3.8.3 Partner features

Partner features are not mission critical but can still differentiate your product from the competition. These features are not usually part of the core expertise of your organization, so it might be inefficient to build up this expertise for one project. In this case, it makes sense to team up with a partner who is more specialized in this area.

3.8.4 Minimum impact

Some features are neither mission critical nor market differentiating. You should try to spend as little time and effort as possible on features in this category. These low-value features might be a good candidate for outsourcing.

3.9 Summary

In this chapter we discussed Feature Injection and various related concepts. You learned about the following:

- Identifying and understanding the fundamental business goals behind the applications you build makes it much easier to design and build valuable features.
- Feature Injection is a technique that tries to identify the business value that an application is expected to deliver, and to identify the essential features that will deliver this value.
- Impact Mapping enables you to visualize the relationships between stakeholders, capabilities, features, and business goals.
- The Purpose-Based Alignment Model helps you decide how much effort to put into the different features you decide to implement.

In the next chapter, we'll take these ideas further and look at how you can discover the features and scenarios most likely to deliver value. We'll also look at how you can better manage your lack of knowledge, and understand and prioritize the features you build and deliver, using concepts such as Deliberate Discovery and Real Options.

4

Defining and illustrating features

This chapter covers

- Describing and organizing features and user stories
- Illustrating features with examples
- Using these examples to build up a common understanding of the requirements

In the previous chapter you learned how important it is to understand why you're building a piece of software and what its ultimate purpose will be in business terms. We looked at how you can clarify *what* you want to achieve and *how* you expect this to benefit the business (the business goals), and also at *who* will benefit or be affected by the project (the stakeholders) and *what* you need to deliver at a high level to achieve the business goals (the capabilities).

Now it's time to describe *how* you can provide these capabilities. In this chapter we'll learn about what BDD practitioners often refer to as features, and about some of the techniques they use to describe and discuss these features (see figure 4.1):

- In BDD terms, a *feature* is a piece of software functionality that helps users or other stakeholders achieve some business goal. A feature is not a user story, but it can be described by one or several user stories.

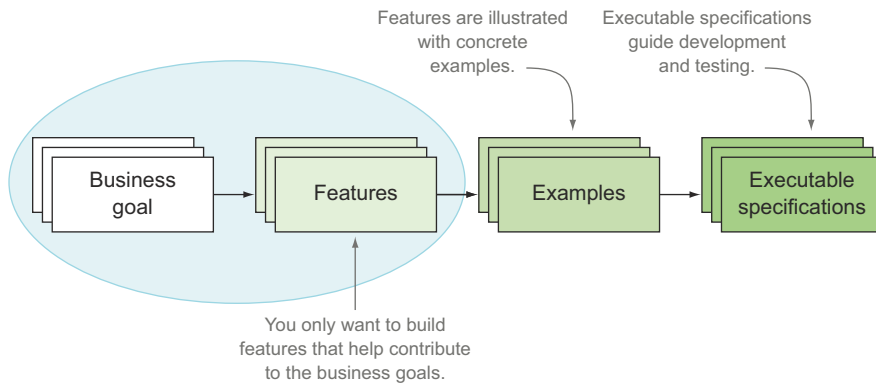


Figure 4.1 In this chapter we'll focus on how to discover and describe the features that will enable your application to achieve its business goals.

- A *user story* is a way of breaking the feature down into more manageable chunks to make them easier to implement and deliver.
- BDD practitioners use *concrete examples* to build up a *shared understanding* of how a feature should behave. These examples also help flush out and clarify *ambiguities and uncertainties* in the requirements.
- *Managing uncertainty* plays a major role in BDD practices. When they've identified areas of uncertainty, experienced BDD practitioners avoid committing to a definitive solution too early, keeping their options open until they know enough to be able to deliver the most appropriate solution for the problem at hand. This approach is known as *Real Options*.
- *Deliberate Discovery* tries to reduce project risk by managing uncertainty and ignorance proactively wherever possible.
- The examples that illustrate the features help drive the development process, becoming *automated acceptance criteria* that the developers use as a guide when they implement the features.

Let's look at these points in more detail.

4.1 What is a “feature”?

In Agile projects, developers use lots of different words to describe what they want to build (see figure 4.2). Epics, capabilities, themes, requirements, features, use cases, user stories, tasks. Confusing? Guilty as charged. The Agile community has done a relatively poor job of defining these concepts in a clear and universally understood way. Different methodologies and different teams use varying—sometimes contradictory—terms. And although the terms “user story,” “epic,” and “theme,” for example, do actually have a long tradition in Agile and Scrum circles, many teams still waste long hours debating what terminology they should use, or whether a particular requirement should be called a story, a feature, an epic, or something else entirely.

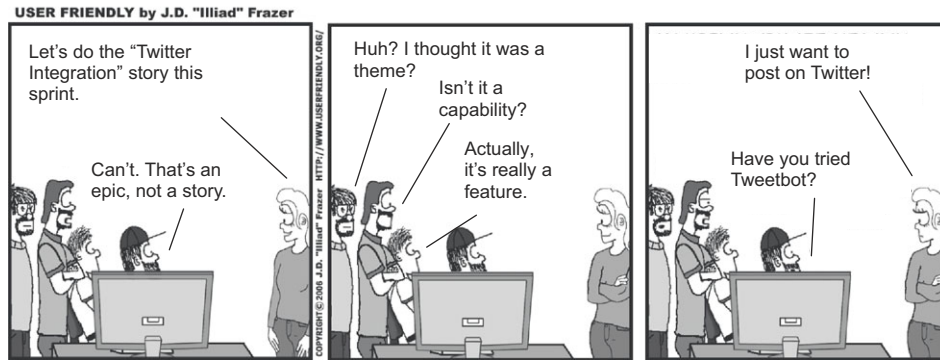


Figure 4.2 The vocabulary around Agile requirements can be a little confusing at times.

How did we get ourselves into this predicament? All we really want to do is describe what we think our users need. We want to express ourselves in a way that business stakeholders can understand, so that they can validate our understanding, contribute, and provide feedback as early as possible. We want to be able to describe what our users need in a way that makes it easier to build and deliver software that meets these needs.

The terms we use are intended to simplify discussion around user requirements. In essence, we're trying to do two things:

- Deliver tangible, visible value to the business at regular intervals
- Get regular feedback so we know if we're going in the right direction

The way we describe and organize the requirements should support these goals. The various ways different teams organize and structure stories, epics, features, and so forth are simply ways to decompose higher-level requirements into manageable sizes, describe them in terms that users can understand, and allow them to provide feedback at each level. There are many perfectly legitimate ways to do this, and what works best for your team will depend on the size and complexity of your project and on the background and culture of your organization and team members.

For the sake of simplicity and consistency, let's step through the vocabulary we'll use throughout the rest of this book. We'll mainly be dealing with four terms: *capabilities*, *features*, *user stories*, and *examples* (see figure 4.3).

I introduced these concepts in the previous chapter, but here's a quick refresher:

- *Capabilities* give users or stakeholders the ability to realize some business goal or perform some useful task. A capability represents the ability to do something; it doesn't depend on a particular implementation. For example, "the ability to book a flight" is a capability.
- *Features* represent software functionality that you build to support these capabilities. "Book a flight online" would be a feature.
- When you build and deliver these features, you can use *user stories* to break down the work into more manageable chunks and to plan and organize your work.

- You can use *examples* to understand how the features will help your users and to guide your work on the user stories. You can use examples to understand both features and individual user stories.

4.1.1 **Features deliver capabilities**

As developers, we build features so that end users, and stakeholders in general, can achieve their goals. Users need our software to give them the *capabilities* that will help them contribute to these business goals.

Features are what we deliver to users to support these capabilities. A feature is a tangible piece of functionality that will be valuable for users, that relates closely to what the users actually ask for, and that may or may not be deliverable in a single iteration. It can be delivered relatively independently of other features and be tested by end users in isolation. Features are often used to plan and document releases.

Features are expressed in business terms and in a language that management can understand. If you were writing a user manual, a feature would probably have its own section or subsection. Back in the day when off-the-shelf software was packaged in boxes, features would be what appeared on the side of the package.

Let's look at an example. Flying High Airlines is very proud of its recently introduced Frequent Flyer program. Belonging to the Flying High Frequent Flyer club lets members earn points that they can spend on flights or upgrades and so forth, and it's designed to encourage travellers to book with Flying High rather than a competitor. But management has noticed a high rate of lapsed memberships, which they suspect is due to the cumbersome renewal process. Currently, members need to call Flying High or return a renewal form by non-electronic mail to renew.

Suppose you have identified the capability, “to enable members to renew their membership more easily.” Some useful features supporting this capability on the Flying High customer site might be “allow members to renew their membership online” and “notify members when their membership is due for renewal.” To support these capabilities, you could define features such as “renew membership online” or “notify members that their membership is due for renewal by email” (see figure 4.4).

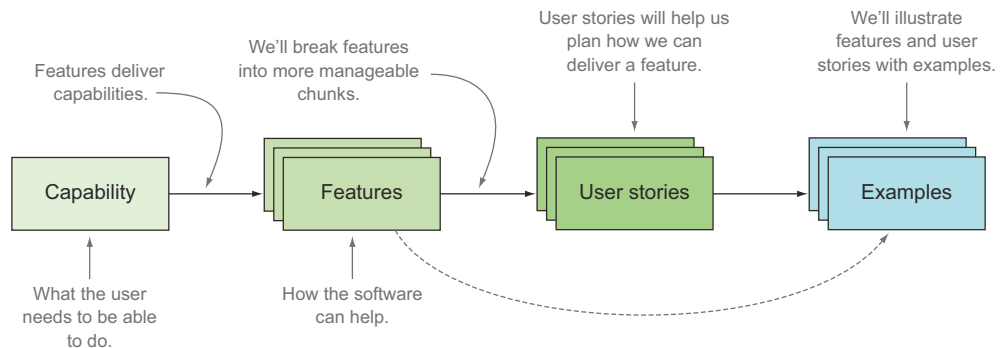


Figure 4.3 Features deliver capabilities to stakeholders. We'll use user stories to plan how we'll deliver a feature. We'll use examples to illustrate features and user stories.

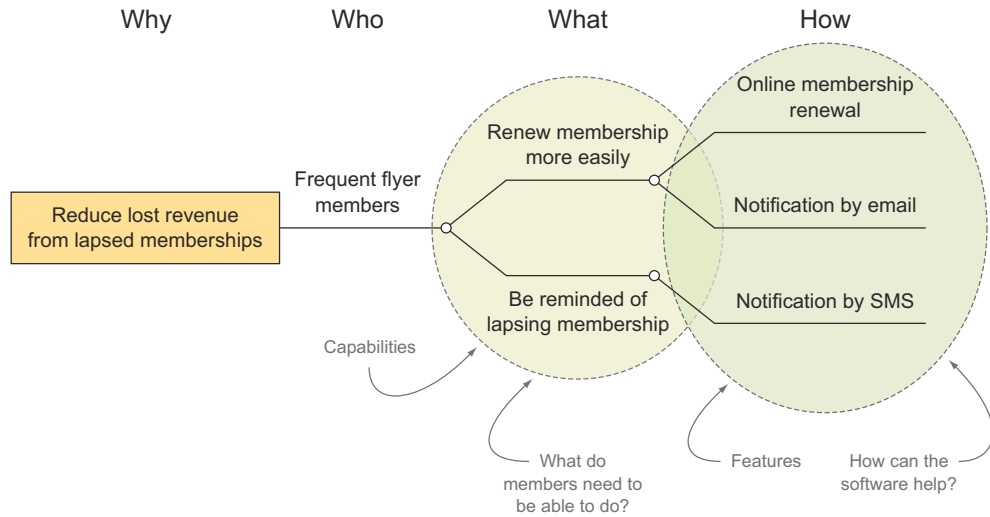
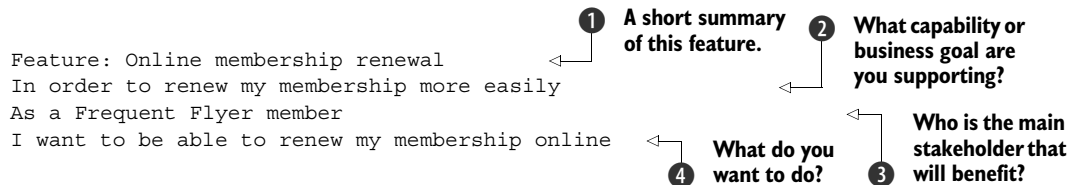


Figure 4.4 Features provide users or stakeholders with the capability to do something useful.

Let’s focus on the online renewal feature. You could describe the “online membership renewal” feature using the “in order to ... as a ... I want” format, like this:



You used this format in the previous chapter to describe capabilities, but it works equally well for requirements at any level. You start off with a short summary or title of the feature to give some context **1**. This is the text proposed in the impact map in figure 4.4, in the “How” section. In fact, until you come to actually scheduling, designing, and implementing this feature, this short overview is usually enough to work with. You only really need to flesh out the details when you’re fairly sure you want to (and are ready to) start working on the feature.

At that point, you can try to formulate what the feature is about in more detail. First you outline what business goal you think this feature will support **2**. This helps remind you why you’re building this feature in the first place, and ties back nicely to the capability in the impact graph. It also allows you to do a sanity check on your requirement. You can ask yourself questions like, “Will the feature I’m proposing really help us achieve this business goal? If not, what business goal is it supporting? Based on what I know now, is it still worth building this feature?” Both your understanding of the requirements and the business context behind the project may have changed since you first envisaged the feature (see figure 4.5), and you might need to reevaluate how important the feature really is.

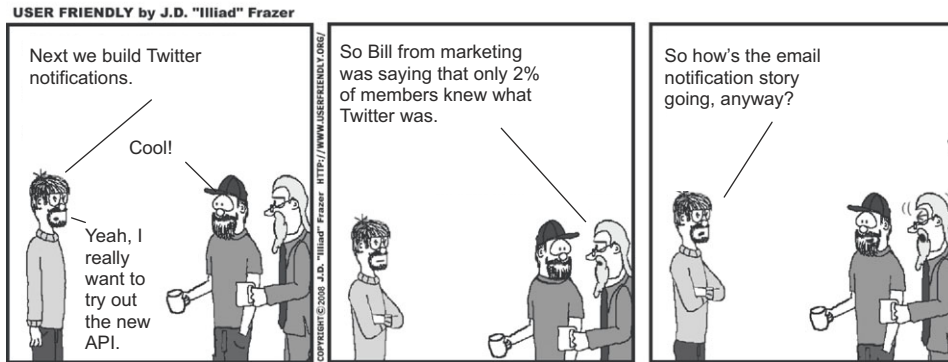
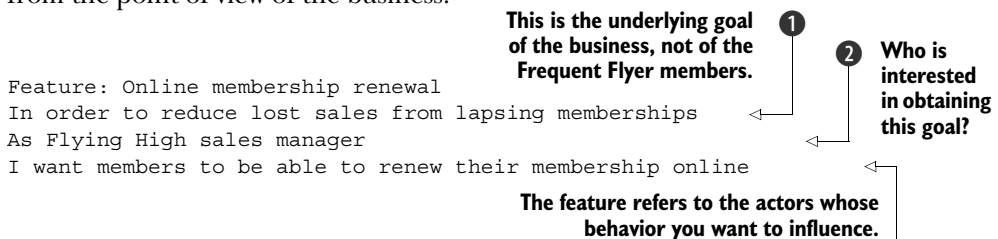


Figure 4.5 Sometimes features become less important as we learn more about them.

Next, you identify which users you think this feature will affect, or which stakeholders will benefit **3**. This helps you look at things from the point of view of the people who will be using the feature or who expect to benefit from its outcomes.

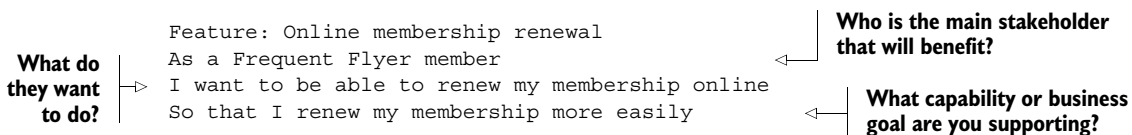
Finally, you describe the feature itself and what it's meant to do **4**. Here you focus on describing what the software does in business terms; you don't want to get too hung up on the technical details or commit yourself to a particular implementation just yet.

Sometimes it's more useful to consider a requirement not from the point of view of the end user, but from the perspective of the stakeholder who's ultimately interested in this business outcome. For example, you might find it more useful to look at things from the point of view of the business:



This highlights an interesting point. It's really the Flying High sales manager **2** who wants members to renew so they'll continue to book flights on Flying High planes **1**. So in this case, the real stakeholder that you need to satisfy is the sales manager, not the members. Maybe the members aren't always that motivated to renew. Maybe you need to find ways to entice them into renewing their memberships.

The format we've been using here is popular among BDD practitioners because it focuses on the business value or capability that the feature is meant to deliver. But many teams also use the more traditional "as a ... I want ... so that" format:



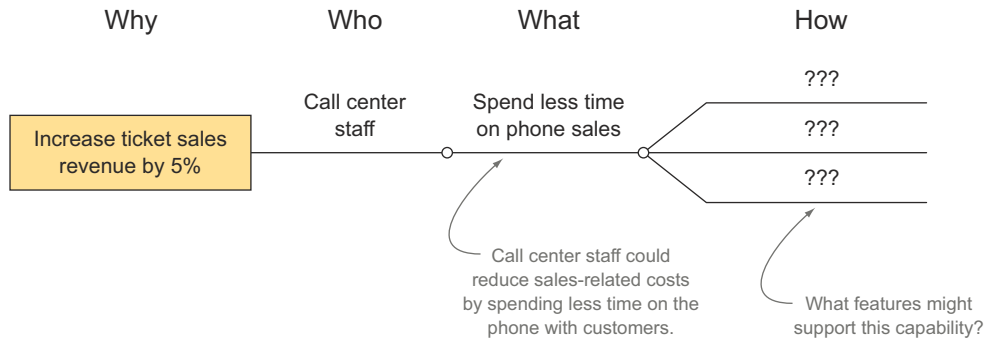


Figure 4.6 What features would support this capability?

If you’re new to all this, the “in order to ... as a ... I want” format will help you stay focused on the business goals, but both forms are valid, and they convey essentially the same information. Experienced practitioners will be able to produce high-quality and meaningful definitions in both formats.

Ultimately, there’s no right or wrong way to describe a feature, and no standard canonical format that you must use, though it’s nice to agree on a consistent format within a team or project.

EXERCISE 4.1 Look at the impact map in figure 4.6. You want to give call center staff the capability to sell tickets more quickly over the phone. Define some features that would help support this capability.

4.1.2 Features can be broken down into more manageable chunks

When you describe a feature, you need to think in terms of functionality that delivers some useful capability to the end user. When you come to building and delivering a feature, you’ll often need to break the feature down into smaller, more manageable pieces. You may or may not be able to deliver the whole feature in one iteration.

You can break features down further as you explore the best way to deliver a particular capability, using what’s effectively a form of functional decomposition. In an Agile project, when you’ve broken the features down into chunks small enough to build within a single iteration, you can call the chunks “user stories.” As you can see in figure 4.7, it’s common to need more than one level of decomposition to get from a real-world feature to a reasonable-sized user story.

Knowing when the decomposed chunks are no longer features is a little subjective and varies from project to project. As we discussed earlier, a feature is something that users can test and use in isolation. A feature can deliver business value in itself; once a feature is completed, you could theoretically deploy it into production immediately, without having to wait for any other features to be finished first. Let’s look at some examples:

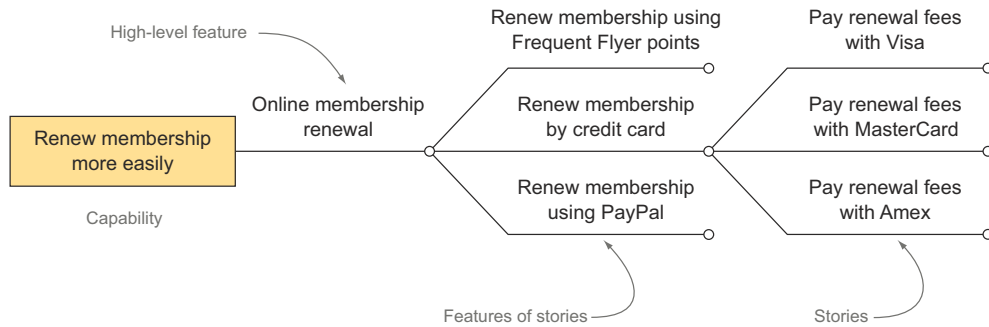


Figure 4.7 Breaking down features into smaller features or user stories makes them easier to organize and deliver.

- “Online membership renewal” would certainly qualify as a feature. If you were to deploy this feature into production by itself, it would still be of significant business value. But could you provide business value faster by incrementally delivering smaller parts of this feature, rather than waiting for it to be completely finished? This is generally a question for the business stakeholders.
- “Renew membership by credit card” would not be a feature in itself unless it included the entire renewal process. Even if it did, you’d still have to ask the project sponsor if they would be happy to deploy this feature into production without the other payment methods.
- “Pay renewal fees with Visa” and so forth would usually be considered too low-level to be delivered in isolation. Paying by Visa, for example, is just one aspect of “Renew membership by credit card,” and would be of little business value in isolation. So you’d represent these in the form of user stories rather than features.

There are two main strategies when it comes to decomposing features. The one used here involves decomposing a feature into a number of smaller business processes or tasks (“Renew by credit card,” “Pay with MasterCard,” and so forth). You express tasks in terms of business goals and try to avoid committing to a particular implementation solution until you know more about what solution would be most appropriate. When you use this strategy, visual approaches such as Impact Mapping also make it easier to keep the larger business goals in perspective. This is generally the approach that works best when practicing BDD (and Agile in general, for that matter).

The other strategy that teams sometimes use is deciding what needs to be built early on, and coming up with user stories to deliver whatever technical solution is envisaged. This approach is risky and involves much more upfront work, with the dangers that that entails. For example, figure 4.8 shows a different decomposition of the “Membership renewal online” feature into a number of user stories.

In this decomposition, you’ve already imagined or designed a particular sequence of screens to implement this feature and have created user stories based on these screens. The problem is that you can lose focus on the real business goals when you

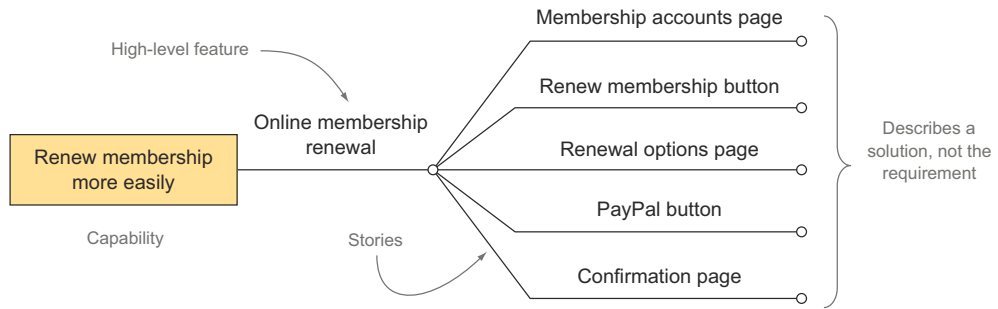


Figure 4.8 It’s dangerous to decompose features with a particular solution in mind.

commit early to a given solution, and you can miss the opportunity to provide a more appropriate solution. In figure 4.8, for example, the ability to renew memberships using Frequent Flyer points has been forgotten in all the excitement around implementing a PayPal-based solution.

4.1.3 A feature can be described by one or more user stories

User stories are the bread and butter of Agile projects, and they’ve been around, in slightly differing forms, since the origins of Agile. A user story is a short description of something a user or stakeholder would like to achieve, expressed in language that the business can understand. For example, the following user story describes a requirement around forcing users to enter at least a moderately complex password when they register to be a Frequent Flyer member. For this story, you can use a format very similar to the ones used for features earlier on:

```
Story: Providing a secure password when registering
In order to avoid hackers compromising member accounts
As the systems administrator
I want new members to provide a secure password when they register
```

**This is the
same format
that you used
for features.**

Agile practitioners are fond of emphasizing that a user story is not actually a requirement, but more a promise to have a conversation with the stakeholders about a requirement. Stories are a little like entries in a to-do list, giving you a general picture of what needs to be done, and reminding you to go ask about the details when you come to implement the story.

User stories are traditionally represented on story cards like the one in figure 4.9, which also includes other details, such as a priority and a rough estimate of size in some agreed metric (estimates are often in hours, or they may use the more abstract notion of “story points”). You can also use similar cards to represent features.

On the flip side of the card, you can put an initial list of acceptance criteria in simple bullet points (see figure 4.10). These acceptance criteria clarify the scope and boundaries of the story or feature. They help remove ambiguities, clarify assumptions, and build up the team’s common understanding of the story or feature. They also act

#120 Providing a secure password when registering
In order to avoid hackers compromising member
accounts
As the systems administrator
I want new members to provide a secure password
when they register
Priority: HIGH
ESTIMATE: 4

Figure 4.9 A typical user story card format

as a starting point for the tests. But the aim of these acceptance criteria isn't to be definitive or exhaustive. It's unreasonable to expect the product owner or stakeholders to think of a definitive list of the acceptance criteria when the stories are being discovered. You just want enough information to be able to move forward. You'll have plenty of time to refine, expand, and complete them, and to add any additional requirements documentation that the team might need, when it comes to implementing the story, and even later on when you learn more about the requirements.

As you can see in figure 4.10, these user stories look a lot like features, but they tend to be a little lower-level. A user story doesn't have to be deliverable in isolation but can focus on one particular aspect of a feature. User stories can help you plan and organize how you'll build a feature. Although you may not deliver a user story into production by itself, you can and should show implemented stories to end users and other stakeholders, to make sure that you're on the right track and to learn more about the best way to implement the subsequent stories.

You can use user stories to break down the features we discussed in the previous section. For example, figure 4.11 builds on figure 4.4, continuing the investigation of what features might help you reduce lost revenue from lapsed Frequent Flyer memberships.

- password should be at least 8 characters
- password should contain at least 1 digit
- password should contain at least 1 PUNCTUATION
MARK
- I should get an error message telling me what I
did wrong if I enter an insecure password

Figure 4.10 You can put an initial list of acceptance criteria on the back of the story card.

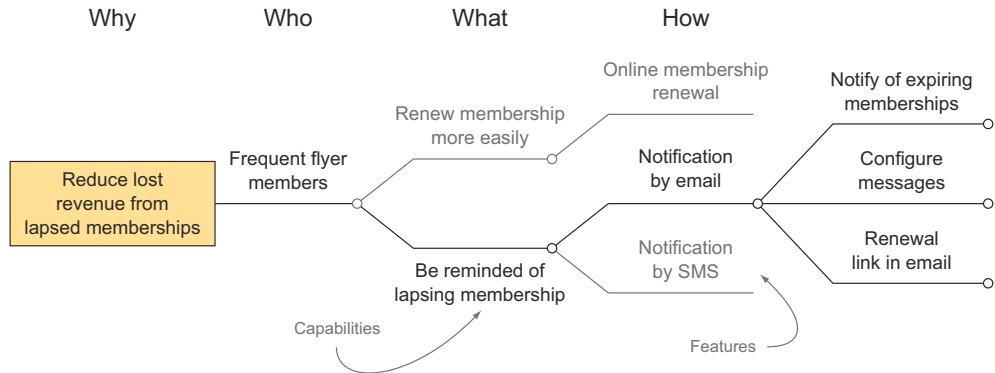


Figure 4.11 You can break large features down into smaller, more manageable ones.

One of the features you discovered for this requirement was “Email notification of lapsing membership.” This is a fairly large piece of work, so you could break it down into stories like the following:

- Send notification emails to members whose membership will finish within a month.
- Configure notification message texts.
- Open renewal page from the notification email.

Although each of these stories adds business value in its own way, these stories aren’t designed to be deployed into production independently. But they do provide great opportunities for getting useful feedback from stakeholders.

For example, suppose you’re working on the following story:

Story: Send notification emails to members whose membership will finish within a month
 In order to increase retention rates for our Frequent Flyer program
 As a sales manager
 I want members to be notified a month before their memberships finish

When you show the implementation of this story to the Flying High sales manager, the conversation goes something like this:

You: And this is how the email notification works. When their membership is about to expire, they receive an email that looks like this.

John the Sales Manager: Looks good. And what about the follow-up email?

You: Is there a follow-up email?

John: Of course. Bill from marketing wants a follow-up email that will include a discount offer of some kind to encourage ex-members to come back.

You: And is the discount always the same?

John: No, Bill needs to be able to change it depending on his latest marketing strategy. We talked about configuring the messages last time.

The value of this sort of feedback is huge. You’ve just discovered a new story for a requirement that had been overlooked or initially misunderstood: “Send follow-up notification emails to ex-members whose membership has just lapsed.”

In addition, you now have a clearer understanding of what’s expected regarding the “Configure notification message texts” story, which we’ll look at next. Originally this was conceived of as a configurable template that the development team could change when required and deliver in the next release. But based on this conversation, you now know that the marketing people want to be able to configure the message at any time. So you can now describe this next story as follows:

```
Story: Include a configurable incentive in the follow-up notifications
In order to increase retention rates for our Frequent Flyer program
As a sales manager
I want to be able to include a configurable text describing incentives to
rejoin such as discount offers or bonus points in the notification message
```

This example illustrates another point. User stories allow you to put off defining detailed requirements until as late as possible. As time goes on, you’ll learn more and more about the system you’re delivering. This is what the ongoing conversations promoted by BDD are designed to facilitate. If you specify the details of a user story too early, you may miss some important fact that you’ll learn later on. If you went ahead and specified the details of the “Configure notification message texts” story first, you’d implement a piece of functionality with little business value, and it wouldn’t correspond to the stakeholders’ expectations at all.

But you can’t procrastinate forever. If you leave it too late, you won’t have time to talk to stakeholders and understand the detailed requirements before the feature is due. In lean software development, this is known as the last responsible moment. This concept is also heavily used in an approach called Real Options, which we’ll look at in more detail in section 4.3.

EXERCISE 4.2 Develop the features you defined in the previous exercise, and break them down into stories of different sizes until you get to stories that you think are of a manageable size. Describe some of them in more detail using the “in order to ... as a ... I want” format.

4.1.4 **A feature is not a user story**

In many projects, the features we’ve been discussing would be represented as high-level user stories, and some teams don’t find it necessary to break the features down into smaller stories. This is fine and will work well on smaller projects.

But there are some advantages to keeping a distinction between the two. Remember,

- A *feature* is a piece of functionality that you deliver to the end users or to other stakeholders to support a capability that they need in order to achieve their business goals.
- A *user story* is a planning tool that helps you flesh out the details of what you need to deliver for a particular feature.

You can define features quite a bit ahead of time, but you only want to start creating stories for a feature when you get closer to actually implementing the feature.

It’s important to remember that user stories are essentially planning artifacts. They’re a great way to organize the work you need to do to deliver a feature, but the end user doesn’t really care how you organize things to get the feature out the door, as long as it gets delivered. Future developers are more interested in what the application currently does than how you went about building it.

Once the feature has been implemented, the user stories can be discarded. The description of the features (see section 4.1.1) is generally more effective at describing what the application does. The examples you use to illustrate the features and stories (see section 4.2) do a great job of illustrating how the software actually works, as do the automated acceptance criteria that you’ll write later on in this book.

4.1.5 **Epics are really big user stories**

Many teams, especially ones that use Scrum, use the term “epic” to refer to a very large user story that will eventually be broken down into smaller stories. There’s no magic definition for what makes a story *epic*, but if a story turns out to be so big that it needs several sprints to complete, it would typically be broken up into a number of smaller, more manageable user stories.

There are a few obvious similarities between the definition of *features* in section 4.1.1 and the way I’m describing epics. Both may need to be broken down into a number of smaller stories, and both can span several iterations. Epics, like features and stories, are focused on delivering business value to the users in some form. But epics, like stories, are primarily useful for project planning.

We won’t use epics much in the rest of this book, but let’s see how an epic would fit in with the requirements organization we’ve been looking at so far. If your idea of an epic is simply a very large user story that can be broken down into smaller user stories, then epics would fit under features, or possibly be synonymous with features (see figure 4.12).

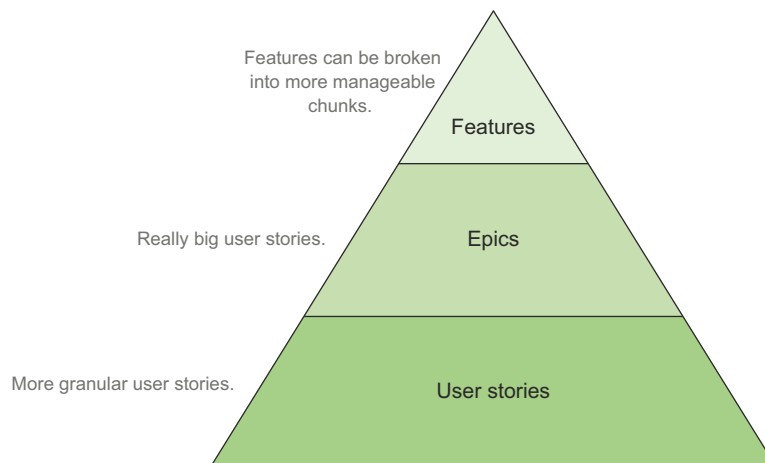


Figure 4.12
Some teams simply define epics as very large user stories.

Some teams see epics as much more, well, epic, and would place them above features. In that case, they're probably using a different, more granular, definition of the word "feature."

In any case, as you saw earlier, it's perfectly reasonable to have several levels of decomposition before you get to the user stories that you can implement in a sprint.

4.1.6 Not everything fits into a hierarchy

In real-world projects, not all requirements fit into the sort of neat hierarchical structures we've been talking about. Although this will work for many user stories, you'll sometimes come across a story that supports several features. For example, the "providing a secure password when registering" story we discussed in section 4.1.3 might relate to two features:

- Join the Frequent Flyer program online.
- Keep client data safe.

In that situation, you might say that the "join the Frequent Flyer program online" feature is a logical parent for this user story, but it's clearly also related to the cross-functional feature, "keep client data safe." This often happens when multiple stakeholders are involved. In this case, the business stakeholder wants travellers to be able to join the Frequent Flyer program, and the security stakeholder wants the feature to be delivered safely. There may be other stakeholders as well, such as compliance, legal, operations, and so forth.

Tags are a good way to handle this sort of situation. Many requirements-management and reporting tools let you use tags to organize your requirements, in addition to enabling a more conventional parent-child relationship. This way, you can present a relatively structured view of the main requirements hierarchy, but also keep track of any looser relationships. We'll look at using tags as part of the living documentation in part 3 of this book.

Once you have a better idea of which features you want to build, you need to flesh out your understanding of them. One of the best ways to do this is to talk through some concrete examples.

4.2 Illustrating features with examples

Examples are at the heart of BDD. In conversations with users and stakeholders, BDD practitioners use concrete examples to develop their understanding of features and user stories of all sizes, but also to flush out and clarify areas of uncertainty (see figure 4.13). These examples, expressed in language that business can understand, illustrate how the software should behave in very precise and unambiguous terms.

According to David Kolb's Experimental Learning theories, effective learning is a four-stage process.¹ In Kolb's model, we all start learning from concrete experiences

¹ David A. Kolb, *Experiential Learning: Experience as a Source of Learning and Development* (Prentice Hall, 1984).

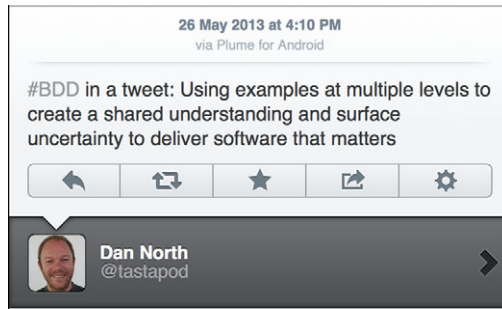


Figure 4.13 The essence of BDD, according to its inventor Dan North

of some real-world situations or events (*experience*). When we observe and think about an experience (*reflection*), we analyze and generalize that example, forming a mental model that represents our current understanding of the problem space (*conceptualize*). Finally we can test this mental model against other real-world experiences to verify or invalidate all or part of our understanding (*test*).

BDD uses a very similar approach (see figure 4.14), where examples and conversation with users, stakeholders, and domain experts drive the learning process. You discuss concrete examples of how an application should behave ❶ and reflect on these examples ❷ to build up a shared understanding of the requirements ❸. Then you look for additional examples to confirm or extend your understanding ❹.

Let's see how this works in practice. The story card and the initial acceptance criteria jotted down on the back (figures 4.9 and 4.10) make a great place to start a conversation that will discover these examples. To see what such a conversation might look like, let's revisit the “secure password” user story we discussed in section 4.1.3.

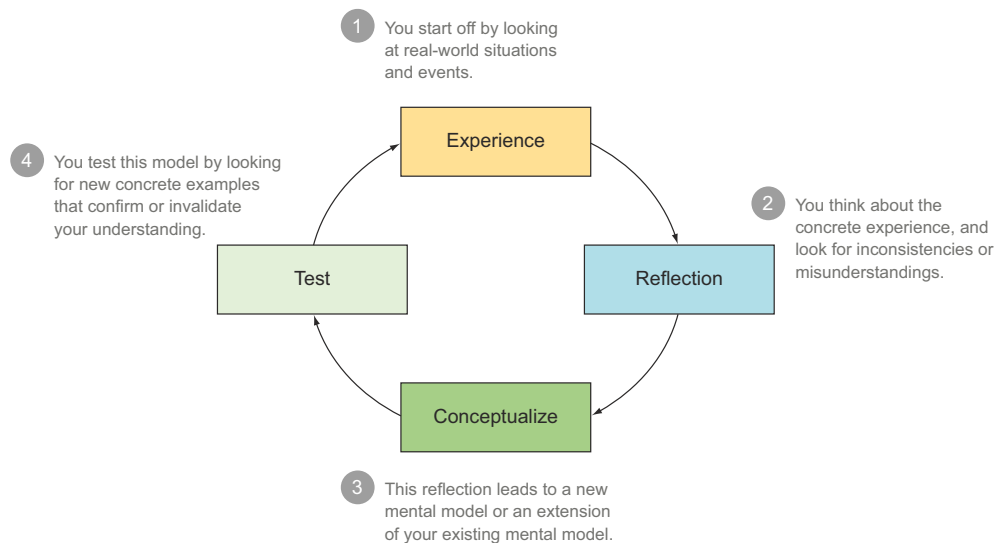


Figure 4.14 David Kolb's Experimental Learning theories apply well to BDD.

The “secure password” story you defined earlier went like this:

```
Story: Providing a secure password when registering
In order to avoid hackers compromising member accounts
As the systems administrator
I want new members to provide a secure password when they register
```

You also had an initial set of acceptance criteria:

- The password should be at least 8 characters.
- The password should contain at least 1 digit.
- The password should contain at least 1 punctuation mark.
- I should get an error message telling me what I did wrong if I enter an insecure password.

These acceptance criteria are a good start, but there are still some potential ambiguities. Can you have all lowercase characters, or do you need a mixture of uppercase and lowercase? Does the position of the number in the password matter? How detailed should the error message be?

When you talk to the systems administrator about this story, you could use examples to try to clarify these questions and others (because you certainly haven’t thought of everything). You can use a few key examples as the basis for your formal acceptance criteria (we’ll discuss how to express acceptance criteria in a more structured way in the next chapter). Not all of the examples that you’ll discover in these conversations will make it into the scenarios—many will just be useful to guide the conversation and expand your understanding of the problem space.

This sort of conversation is more productive if you use some simple strategies. Remember, the aim of this exercise is to build a mental model of the requirements and to illustrate this mental model with a number of key examples. Think of the problem space for the story as a set of jigsaw puzzle pieces. When you ask for an example, you’re really asking for clarification of your understanding of the requirements. This is like picking up a piece of the jigsaw and placing it where you think it should go. If it fits, you’ve confirmed your understanding and expanded your mental model. If it doesn’t, then you’ve flushed out an incorrect assumption and can move forward on a more solid basis.

Back to our password scenario. Raj the System Administrator is an expert in system security and knows a great deal about what makes a secure password. Raj is very concerned about this problem, as in his experience most users naturally use passwords that are very easy to hack. To clarify the exact requirements, you take Susan, a tester, and Joe, a developer, along to see Raj to learn more about what he needs. The conversation with Raj goes along the following lines:

You: I’d like to make sure I’ve understood what you need for the “secure password” story. The first acceptance criteria we defined is about password length. So a password should be rejected if it has less than 8 characters?

Raj: Yes, that's right. Passwords need to be at least 8 characters to make it harder for hacking algorithms to guess them.

You: So "secret" would be rejected because it has only 6 characters?

Raj: Correct.

You: So what about "password". Would that be acceptable?

Raj: No, we also said that we need at least 1 digit.

You: So we did. So "password1" would be OK?

Raj: No, actually that's still really easy to hack. It's a word from a dictionary: the digit at the end wouldn't slow down a hacking algorithm for very long. Random letters and punctuation marks make it a bit harder.

You: OK, so would "password1!" be OK? It has a number and an exclamation mark, and it has more than 8 characters.

Raj: No, like I said, using words from a dictionary like "password" is really bad. Even with numbers and punctuation, a hacking algorithm would solve that pretty much instantaneously.

Notice what has just happened here. You are testing your assumption that the initial acceptance criteria represent all of the constraints that make a secure password. At each step, you used a different example to verify your understanding of the various rules. Now you seem to have found another requirement that you need to represent: dictionary words should be avoided. You decide to push this further:

You: How about "SeagullHedgehog"?

Raj: That would be better.

You: But there are no numbers or punctuation marks in it.

Raj: Sure, that would make it better. But it's still a random sequence of words, which would be pretty hard to crack.

You: How about "SeagullHedgehogCatapult"?

Raj: Pretty much uncrackable.

To keep track of these cases, Susan the tester has been noting a simple table of examples to use for her tests. Here's what she has so far:

Password	Secure
secret	No
password	No
password1	No
SeagullHedgehog	Yes
SeagullHedgehogCatapult	Yes

You decide to check another of your assumptions:

You: OK, how about “aBcdEfg1”?

Raj: That one would actually be pretty easy for a machine to crack—it’s just a sequence of alphabetically ordered letters and a number. Sequences are easy to crack, and just adding a single number at the end doesn’t add much complexity.

You: What about qwerty12

Raj: That’s just a sequence of keys on the keyboard. Most hacking algorithms know about that trick, so it would be very easy to guess.

You: Oh. OK, how about “dJeZDip1”?

Raj: That would be a bit short, but OK.

So now you have another requirement: alphabetical sequences of letters or spatial sequences of keys on the keyboard are both a no-no. But you’ve noticed something interesting:

You: Raj, rather than just saying if a password I give you is secure or not, you seem to be grading them by how hard they are to hack—is that intentional?

Raj: Well, I wasn’t thinking of it like that, but yes, of course: the whole point of a secure password is so that it doesn’t get hacked, and the passwords most people use are pretty easy to hack.² There are a lot of studies and a lot of algorithms out there that measure password strength.³ And many sites provide feedback on the strength of the passwords you enter. In those terms, we need passwords to be of at least medium strength.

Raj brings up a screen similar to the one in figure 4.15.

You: Raj, I think we’ve been focusing on the detailed rules for password validation too much. Working through these examples seems to indicate that the rules are less clear-cut than we initially thought. And the rules we’re describing focus on one particular solution to the problem we’re trying to solve: the real value in this story comes from ensuring that users have a strong password, not enforcing a particular set of rules. If we reason in terms of password strength rather than specific rules, maybe we could rephrase the acceptance criteria like this:

- The password should be at least of medium strength to be accepted.
- I should be informed of the strength of my proposed password.
- If the password is too weak, I should be informed why.

Raj: Yes, that sounds fine. But how do we know what qualifies as a medium-strength password? (See figure 4.16.)

² See, for example, Dan Goodin, “Anatomy of a hack: even your ‘complicated’ password is easy to crack,” <http://www.wired.co.uk/news/archive/2013-05/28/password-cracking>.

³ For anyone interested in this field, there’s an interesting article on password strength by Dan Wheeler, “zxcvbn: realistic password strength estimation,” at <https://tech.dropbox.com/2012/04/zxcvbn-realistic-password-strength-estimation>.

change your password:

Enter a new password for [redacted]. We highly recommend you create a unique password - one that you don't use for any other websites.

Note: You can't reuse your old password once you change it.

[Learn more about choosing a smart password.](#)

Password strength: Too short

Use at least 8 characters. Don't use a password from another site or something too obvious like your pet's name. [Why?](#)

Current password

.....

[Don't know your password?](#)

New password

.....

Confirm new password

.....

[Change Password](#) [Cancel](#)

Figure 4.15 A password security meter helps users provide more secure passwords by measuring how easy a password would be to crack.

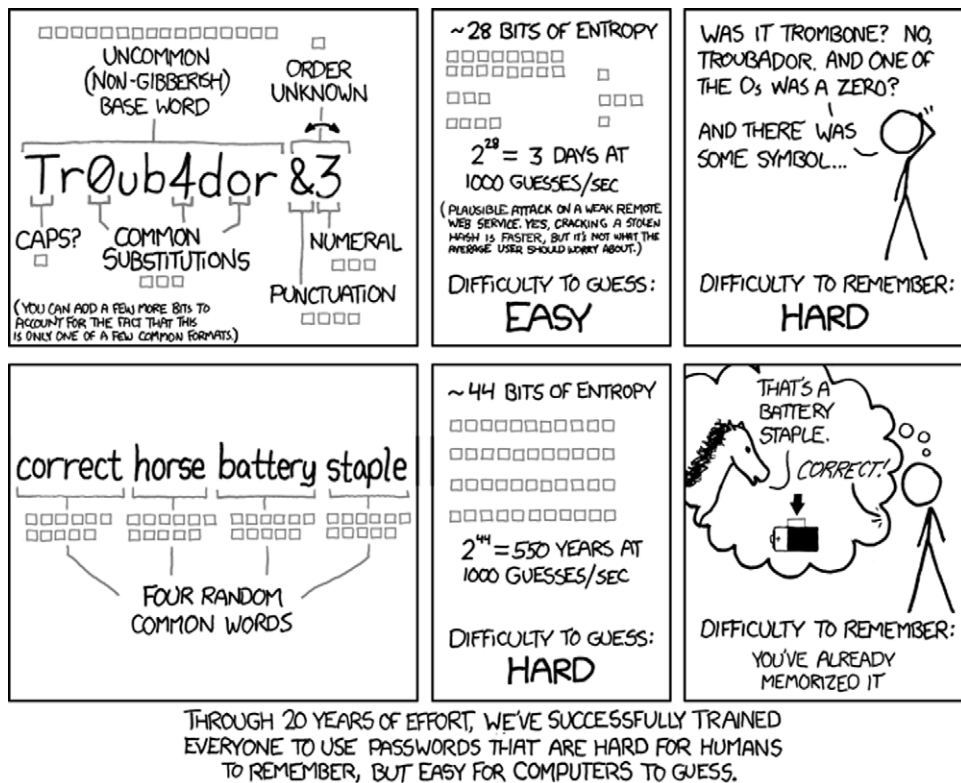


Figure 4.16 Password strength is not as simple as it seems (courtesy of xkcd.com).

Joe: It looks to me like we have a few options there. We can either write our own password-strength algorithm or use an existing one. There are pros and cons to each approach, but using an existing library would probably be faster to implement.

You: Let's keep our options open: we don't know enough about what will suit us best to commit to a particular solution just yet, so let's do what we can to learn more. We'll see if we can find a good existing library and experiment with it, but we should integrate it in a way that we can easily switch to another library or our own custom solution later on if we aren't happy with the one we find.

Joe: Raj, I should be able to build a version of this using a couple of possible libraries by Thursday that you can play around with. Based on your feedback, we can fine-tune the solution we pick, or try out another one.

Susan: We can use this table of sample passwords as a starting point for the acceptance criteria. We may refine it or add new examples later on as we learn more about what we can do.

Password	Strength	Acceptable
secret	Weak	No
password	Weak	No
password1	Weak	No
aBcdEfg1	Weak	No
qwerty12	Weak	No
dJeZDip1	Medium	Yes
SeagullHedgehog	Strong	Yes
SeagullHedgehogCatapult	Very Strong	Yes

You've now gone from having what appeared to be a clear and simple set of requirements to discovering that the real requirements are not quite so obvious. What initially appeared to be business rules requested by the user turned out to be just one possible solution to the underlying business problem of ensuring that members have secure passwords. The team identified several possible approaches, but deferred choosing a specific option until they knew more about what solution they would use. And they identified a strategy that the team could use to get useful feedback from the business; this will help them select the most appropriate solution.

This sort of situation occurs often in software development; it's important to know what you don't know, and to cater for it in your decisions. In fact, many of the design decisions the team made in this example are founded on two important BDD concepts: Real Options and Deliberate Discovery.

4.3 Real Options: don't make commitments before you have to

In the mid-2000s, Chris Matts identified a fundamental principle underlying many Agile practices: putting off decisions until the “last responsible moment,” an idea that comes from lean software development. He called this principle *Real Options*. Understanding this principle changes the way you think about many Agile practices and opens the door to a few new ones.

In finance, an *option* gives you the possibility, but not the obligation, to purchase a product sometime in the future at today's price. For example, imagine that there's a high probability that you'll need to purchase a large quantity of steel in the next three months, and that the price of steel is currently on the rise. You don't want to buy the steel now, because you aren't completely sure that you'll need it; you expect to know for sure sometime in the next two months. But if you wait another few months, the price of steel might have gone up, which means that you'll lose money. To get out of this conundrum, you can buy an option to purchase the steel sometime within the next three months, at today's price. If the price of steel goes up, you can still buy at today's price. And if the price goes down, or if you don't need the steel, you can choose not to use the option. You need to pay for this option, but it only costs a fraction of the total price of the steel: it's worthwhile because it allows you to not commit yourself to buying the steel until you're sure you need it.

This principle also applies in day-to-day life. When you buy a plane ticket, you're actually buying an option to travel: the ticket places you under no obligation to travel. But the price you pay for this option varies. Imagine your favorite airline is offering tickets for only \$600 to go from Sydney to Wellington, but these cheaper tickets are nonrefundable if you decide not to travel. You're not sure that you'll be able to make the trip, so you opt for a more expensive \$800 ticket, which has a \$25 cancellation fee.

Let's look at the math here. The option to cancel the flight costs you an extra \$200. If you're fairly likely to travel, this might be a lot to pay for an option you're unlikely to use, so you might prefer the cheaper ticket. But if you think that there's a 50% chance that you won't be able to fly, you may well be happy to pay the extra \$200. If you cancel, you'll only lose \$225 (the extra \$200, plus the \$25 cancellation fee), whereas if you cancel after opting for the cheaper flight, you'll lose \$600.

Real Options is an application of these principles to software development invented by Chris Matts (see figure 4.17).⁴ Chris summarizes the principles of Real Options in three simple points:

- Options have value.
- Options expire.
- Never commit early unless you know why.

⁴ Chris Matts and Olav Maassen, “Real Options’ Underlie Agile Practices,” InfoQ (2007), <http://www.infoq.com/articles/real-options-enhance-agility>.

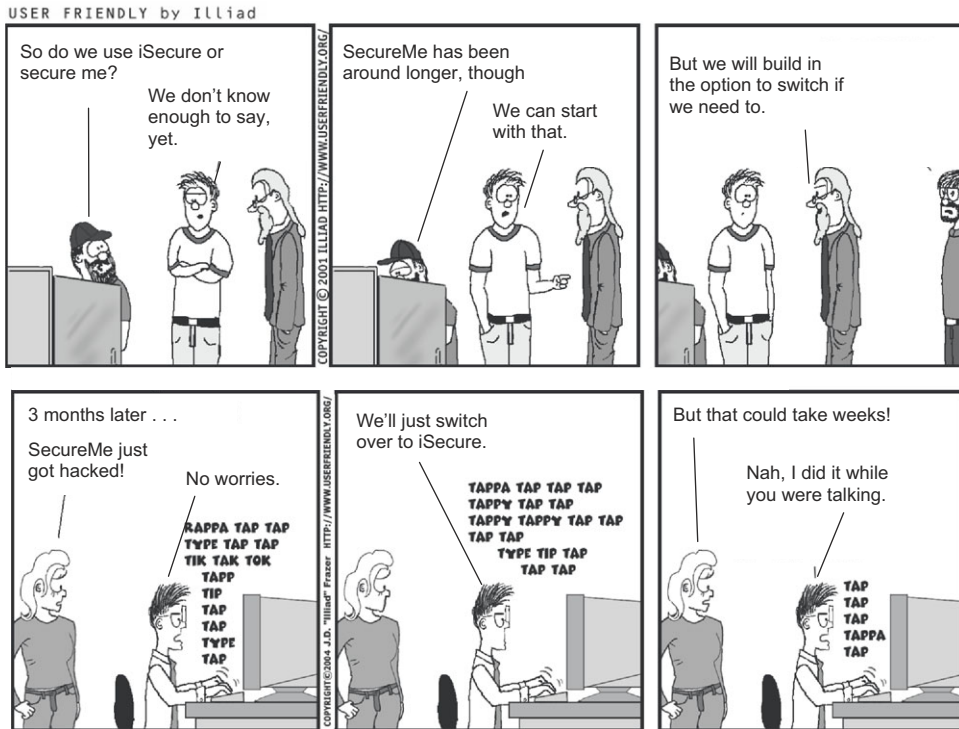


Figure 4.17 Real Options lets you reduce risk by leaving your options open.

Let's look at each of these principles in a little more detail.

4.3.1 Options have value

Options have value because they allow you to put off committing to a particular solution before you have enough knowledge to determine what solution would be best. In the finance industry, the value of an option can be calculated precisely. This isn't the case in software development, but, in general, the less you know about the optimal solution for a particular problem, the more value there is in being able to keep your options open.

Options also have a price. The price in software development is the effort involved in incorporating this flexibility. This price might involve discussing the possible options upfront, adding layers of abstraction to allow a different implementation to be switched in more easily, making certain parts of the application configurable, and so forth.

For example, suppose that you're building a new website for a dynamic young startup. The founders have no clear idea of the volume of users they expect; they know it will start small, but they're very ambitious and expect millions by the end of the year.

You have three options here. You could build the application with no particular regard to scalability, and make it more scalable if and when the need arises, using the

YAGNI (“You Ain’t Gonna Need It”) principle. This is fine if the application never scales. But if it does, the refactoring work will be extensive.

Alternatively, you could invest in a highly scalable architecture from the word go. This would avoid rework, but it would be wasted effort if volume remains low.

A third possibility would be to buy an option to scale up later. You wouldn’t implement a fully scalable architecture immediately, but you could spend a little time upfront to see what would be needed to make the initial implementation easily scalable in the future if required. If you don’t need to scale, you’ve only invested a little upfront design time, and if you do, you’ll be able to do so at reduced costs.

4.3.2 Options expire

You can’t keep an option open forever. In software development, an option expires (that is, you can no longer use it) when you no longer have time to implement it before the related feature is due to be delivered. For example, in figure 4.18, you have the choice between two implementations (solution A and solution B). At this point, you don’t know which solution would be best, so you add a layer of code to make it possible to switch to either solution A or B at a later date.

If you decide for solution A, it will take 10 days to integrate. Implementing solution B, on the other hand, would only take 5 days. In practical terms, this means that if you decide to implement solution A, you must do so at least 10 days before the delivery date, which is when your option on solution A expires. If you delay any further, you won’t be able to exercise this option. You have a bit more time to opt for solution B, as this option only expires 5 days before the delivery date.

Unlike financial options, you sometimes have the power to push back expiry dates. For example, if you can find a way to integrate solution A more quickly, you can leave that option open longer.

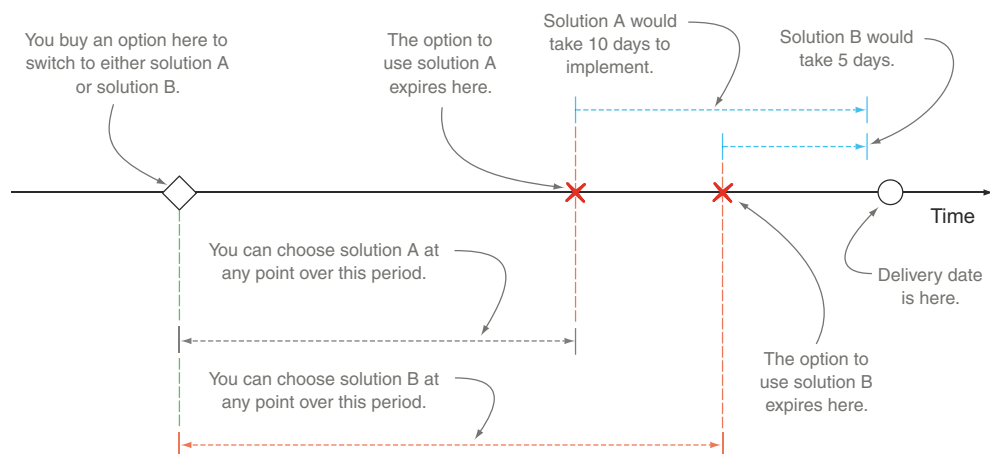


Figure 4.18 Real options expire. Once you pass an option’s expiry date, you can no longer exercise this option.

4.3.3 *Never commit early unless you know why*

The third principle of Real Options is simply to defer committing to a particular solution (“exercising the option”) until you know enough about why you’re choosing that solution.

Real Options gives you the possibility to put off making a decision, but the aim is not to systematically delay until the last possible (or “responsible”) moment. Using Real Options, you only delay until you have enough information to act. When you have enough information, you implement your chosen solution as quickly as possible. For example, you may be delaying your decision because solution A is being developed by another team, and you want to wait to see if this library will work well in your situation. In this case, you can wait until the option for solution A expires, but no longer. If the library takes longer than this to develop, you’ll be forced to exclude solution A from your list of options.

But you may choose to act sooner if you can. For example, if your team could build either solution A or solution B, but you don’t know which is the most appropriate, you might choose to build experimental versions of both solutions concurrently. If you obtain enough information to act before the options expire, it makes sense to act sooner rather than later.

To help you learn enough to make sensible design and implementation decisions, you can use an approach called Deliberate Discovery.

4.4 *Deliberate Discovery*

Deliberate Discovery is the flip side of Real Options, and the two principles go hand in hand. Deliberate Discovery was originally proposed by Dan North⁵ and was developed by Liz Keogh and other members of the London BDD community.

In software development, ignorance is the constraint. You know a lot more about the best way to build a particular solution after you’ve finished building it, but by then it’s too late to take advantage of your knowledge. You can use the principles of Real Options to put off choosing a particular implementation, or implementing a particular feature or story, until you know enough to make a reasonable decision. But if you’re aware that you don’t know what the best solution is, you can proactively investigate your options in order to make a reasonable decision sooner rather than later. Uncertainty represents risk, and where possible you should hunt out and reduce uncertainty. This is where Deliberate Discovery steps in.

Deliberate Discovery starts with the assumption that there are things you don’t know. This might be something bad that you couldn’t possibly have anticipated and that will pop up and cause you problems at some point during the project. Or it might be an opportunity to innovate: “If only we’d known about that technology earlier, we could have built this feature in half the time.”

⁵ Dan North, “Introducing Deliberate Discovery” (2010), <http://dannorth.net/2010/08/30/introducing-deliberate-discovery>.

Real Options help you keep your options open until you have enough information to act; Deliberate Discovery helps you get this information. If you actively try to increase your knowledge in this area, you can both reduce the risk of uncertainty and make decisions faster; remember, as soon as you know enough to commit to a particular solution, you can choose to exercise your option or not.

But Deliberate Discovery also has broader applications. For example, suppose you've decided to implement a particular feature and have broken it down into a number of stories. Some of these stories may seem straightforward, and others may not be so simple.

The natural tendency is to implement the simplest stories first, and there are good reasons why you might do this. But reducing your ignorance should be high on your priority list. Wherever possible, identify the stories that involve the most uncertainty, and tackle these ones first. Then review the remaining stories, keeping in mind what you've learned and considering the feedback the stakeholders give you. This simple approach can go a long way in helping you increase your knowledge and understanding in areas that matter.

EXERCISE 4.3 Raj, Joe, and Susan used both Real Options and Deliberate Discovery to decide on the best way to implement the password-strength feature. Discuss how.

4.5 From examples to working software: the bigger picture

In BDD, conversations around an example mark the first step in actually building and delivering a feature or story. You don't have this conversation until you're committed to implementing the corresponding functionality. But once you do, you kick off a process that will hopefully lead to putting a useful new feature in front of your users (see figure 4.19).

The process starts when you choose a story to work on and select one of the acceptance criteria to implement ❶. Next you discuss this acceptance criterion with the relevant stakeholders, using examples to explore the problem space, as we discussed in the previous section ❷. These conversations should produce a better understanding of what the user needs, and a set of examples, or scenarios, to illustrate the acceptance criterion.

Different types of conversations

In practice, conversations about features and examples can take many forms. For example, teams that are new to BDD often benefit from workshops early on in the iteration, where the whole team, including business stakeholders, is involved. These workshops are a great way to get communication happening sooner rather than later, to give the team a shared understanding of the features they're building, and to produce a set of high-quality examples. On the downside, they can be hard to organize and are expensive in terms of people-hours.

A more lightweight approach, which can work well when teams start to become more experienced in BDD, is known as the "Three Amigos." Three team-members—a developer, a tester, and a business analyst or product owner—get together to discuss a feature and draw up the examples. For this to work well, all three need to be reasonably familiar with the problem space, but the dynamic interaction of each role is often very

(continued)

productive. The tester, with great attention to detail and a focus on validation, will propose obscure edge cases and often point out scenarios that the other team members have missed. The developer will point out technical considerations, and the business analyst or product owner will be able to judge the relevance and relative value of the different scenarios. In addition, the developer will gradually obtain a much deeper understanding of the business requirements than would normally happen in a more traditional project, and this understanding becomes more and more useful as the project progresses. Often, in this approach, the three will sit around a computer and write up an initial draft of the automated scenarios together. This helps reduce the risk of information loss later on down the track.

In some teams, the business analyst prefers to do the bulk of the scenario writing, referring to stakeholders if they have any questions. This approach doesn't build a shared understanding as effectively as the previous strategies, but it can be made to work. The scenarios should be prepared by the business analyst but then reviewed with a developer and a tester. Developers can provide useful feedback on how best to express the scenarios to take advantage of the language features of the automation tooling. And testers can provide valuable input about additional scenarios that might need exploring.

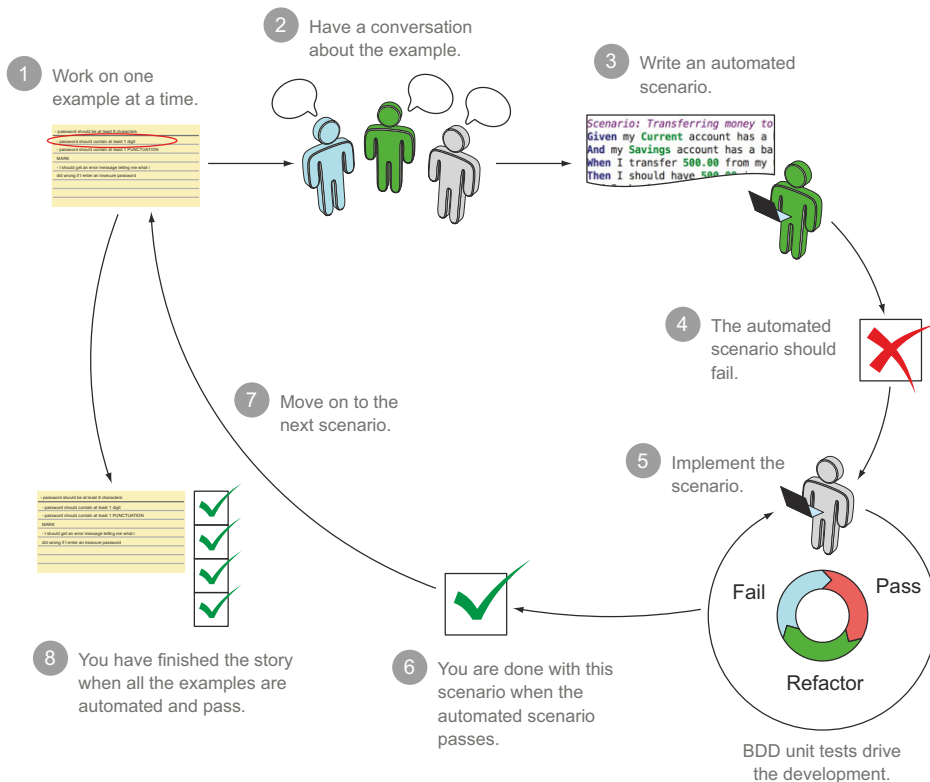


Figure 4.19 From examples to working code

You convert these examples into a slightly more structured format ③, known as scenarios, so that you can automate them in the form of automated acceptance tests. Naturally, this test should fail at first ④, because you haven't implemented anything yet. The developers now proceed to write the code required to make this acceptance criterion pass ⑤. The developer uses TDD, or more precisely, BDD at a unit-testing level, to implement the various layers needed to deliver the requested functionality. We'll discuss how to write and automate these scenarios in the next chapter.

If the developer has worked well, the acceptance criterion should now pass ⑥, which indicates that you're done with this scenario and can move on to the next one ⑦. When all of the scenarios pass, the developers are done with this story ⑧. QA can now use the passing automated acceptance tests as a basis for their exploratory testing, and stakeholders can review the new feature to see how well it matches their expectations.

4.6 Summary

In this chapter, you learned how to describe and organize features and how to illustrate them with examples. Along the way, you learned about the following:

- A *capability* enables some business goal, regardless of implementation.
- A *feature* is a piece of deliverable software functionality that provides users with a capability.
- Large features can be broken down into smaller features to make them easier to organize and deliver.
- Agile projects use *user stories* to plan and deliver features.
- Concrete examples help you build up a shared understanding about a feature.
- The principle of *Real Options* recommends that you shouldn't commit to a particular solution until you have enough information to be confident that it's the most appropriate one.
- *Deliberate Discovery* points out that one of the biggest risks in any software project is your own ignorance, and that you should actively aim to identify and reduce uncertainty wherever you can.

A surprising number of the benefits of BDD come from simply having a conversation with the business, using examples to challenge assumptions and build a common understanding of the problem space. One of the principle benefits of BDD is to encourage and structure this kind of conversation. But there's also a great deal to gain by automating these examples, in the form of automated acceptance criteria. In the next chapter, you'll learn how to express clear, precise examples in a structured format, and how to turn these examples into executable specifications that can be read by tools like JBehave, Cucumber, and SpecFlow.

5

From examples to executable specifications

This chapter covers

- Turning concrete examples into executable scenarios
- Writing basic scenarios
- Using data tables to drive scenarios
- Writing more advanced scenarios using more JBehave/Gherkin keywords
- Organizing scenarios in feature files

In the last chapter, you saw a number of techniques to identify and describe valuable features. You also saw how conversations with the stakeholders around concrete examples are a very effective way to build up a common understanding of a problem space. In this chapter, you'll learn how to express these examples clearly and precisely, in a way that will allow you to transform them into executable specifications and living documentation (see figure 5.1).

The aim of this chapter is to help developers, business analysts, testers, and other interested team members get a solid shared understanding of how to read

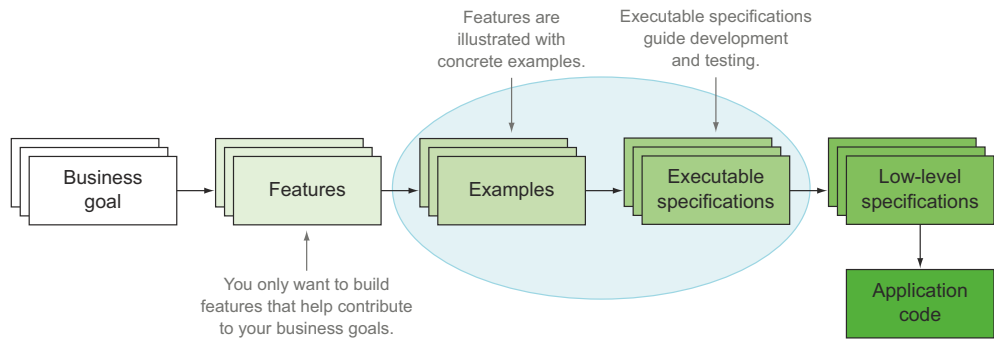


Figure 5.1 In this chapter we’ll take examples we used to discuss and illustrate features in previous chapters and turn them into executable specifications.

and write executable specifications in a way that makes it easy to automate them. BDD has a number of well-defined practices to achieve this shared understanding:

- BDD practitioners express concrete examples as *executable scenarios*, using a semi-structured “given ... when ... then” format that’s easy for both stakeholders and team members to read.
- This format can be automated using BDD tools such as JBehave, Cucumber, and SpecFlow.
- *Tables* can be used to combine several similar examples more concisely in a single scenario, or to express test data or expected results in a more succinct way.
- Experienced BDD practitioners take care to write their scenario steps well, providing enough detail for the scenario to be meaningful, but not so much that the essential business goals of the scenario are hard to find.
- Scenarios are organized in feature files and can be annotated with *tags* to indicate cross-functional concerns and to coordinate test execution.

It’s important for everyone to be comfortable with the notation and structures used for these scenarios; that way, team members can focus on discussing the requirements and not be distracted by the form that you use to express them. As in any language, there are common patterns and structures that recur (idioms, so to speak), and those can help you express your ideas more fluently.

In the next chapter, you’ll see how to automate these examples using popular BDD tools: JBehave (Java), Cucumber (Java and Ruby), SpecFlow (.NET), and Behave (Python). But if you want to experiment with the examples we discuss in this chapter, you can download the JBehave and Cucumber versions from GitHub (<https://github.com/bdd-in-action/chapter-5>) or the Manning website.

5.1 Turning concrete examples into executable scenarios

Imagine you’re working on the Flying High Frequent Flyer application we discussed in chapters 3 and 4. Your job is to implement a feature that will allow Frequent Flyer

members to earn points when they fly. A traditional requirements specification document might include something like this:

```
Members will earn Frequent Flyer points from Flying High flights and from  
partner flights.
```

This may capture the essence of what you need to build, but it's a little vague. How many points should a member earn per flight? Will members earn the same number of points on Flying High flights as on partner flights? Do the flights on partner companies have to be booked through Flying High, or is any flight on a partner airline applicable? Will members earn more points if they fly in Premium Economy or Business? And so on.

If you leave these questions unaddressed now, the development team will have to make decisions and judgment calls about the most appropriate solutions. They may have to ask additional questions during development, which will slow down the project as they wait for answers. Or they may incorrectly assume that they've understood what's needed, and implement a solution that doesn't correspond to what the business really needs. In both cases, time and effort is wasted.

As we saw in the previous chapter, discussing concrete examples with the users and stakeholders is a great way to flush out and eliminate this sort of ambiguity, making sure everyone is on the same page. You use the language and vocabulary of your stakeholders to clarify aspects that you're not clear about, and you'll often discover things that the stakeholders hadn't originally thought about, had assumed you knew, or had forgotten to mention. For example, Sarah, a business analyst, and Paul from the Flying High marketing team might have a conversation like this one to drill deeper into how Frequent Flyer points are earned:

Sarah: Can you give me an example? How many Frequent Flyer points would I earn if I flew from Sydney to Melbourne in Economy?

Paul: Well, the distance from Sydney to Melbourne is 878 km, and base points are calculated at half a point per kilometer, so if you fly with Flying High from Sydney to Melbourne in Economy, you'd earn 439 points.

Sarah: OK. Is there a way I might earn more or less points flying this trip?

Paul: Well, if you were a Silver Frequent Flyer, you'd get a 50% Status Bonus as well, so you'd earn 659 points.

Sarah: And what if I was a Gold Frequent Flyer?

Paul: In that case, you'd earn a 75% Status Bonus, but you'd also be entitled to the Guaranteed Minimum Point Earnings, which for Economy is 1000 points, so you'd earn 1000 points.

As with many real-world requirements, and as you can also see in figure 5.2, things get more complicated when you start to look into the details. We'll come back to this example several times during this chapter. But already in this conversation you've learned about three business rules:



Figure 5.2 Business requirements are often not as simple as they appear.

- Frequent Flyer members normally earn half a point per kilometer flown.
- Silver and Gold Frequent Flyer members earn extra points.
- Gold Frequent Flyer members are guaranteed a minimum number of points per trip.

There are still some areas you need to investigate further. Do members earn more if they fly in Premium Economy or Business class? And what is the Guaranteed Minimum Point Earnings if you fly in Premium Economy or Business?

But let's leave these questions for now and see how you can express the examples you have so far as executable requirements.

One of the core concepts behind BDD is the idea that you can express significant concrete examples in a form that's both readable for stakeholders and executable as part of your automated test suite. You'll write executable specifications in the native language of your users, and produce test results that report success or failure not in terms of classes and methods, but in terms of the features that the stakeholders requested. Stakeholders will be able to see their own words appear in the living documentation, which does wonders in increasing their confidence that you've understood their problems. This is what BDD tools like Cucumber, JBehave, and SpecFlow bring to the table.

When you automate your acceptance criteria using this sort of BDD tool, you express your examples in a slightly more structured form, often referred to as *scenarios*. Dan North defined a canonical form for these scenarios in the mid-2000s, built around a simple "Given ... When ... Then" structure, and this format has been widely adopted by BDD practitioners ever since.

You could write the first example we discussed earlier like this:

```
Scenario: Earning standard points from an Economy flight
Given the flying distance between Sydney and Melbourne is 878 km
And I am a standard Frequent Flyer member
When I fly from Sydney to Melbourne
Then I should earn 439 points
```

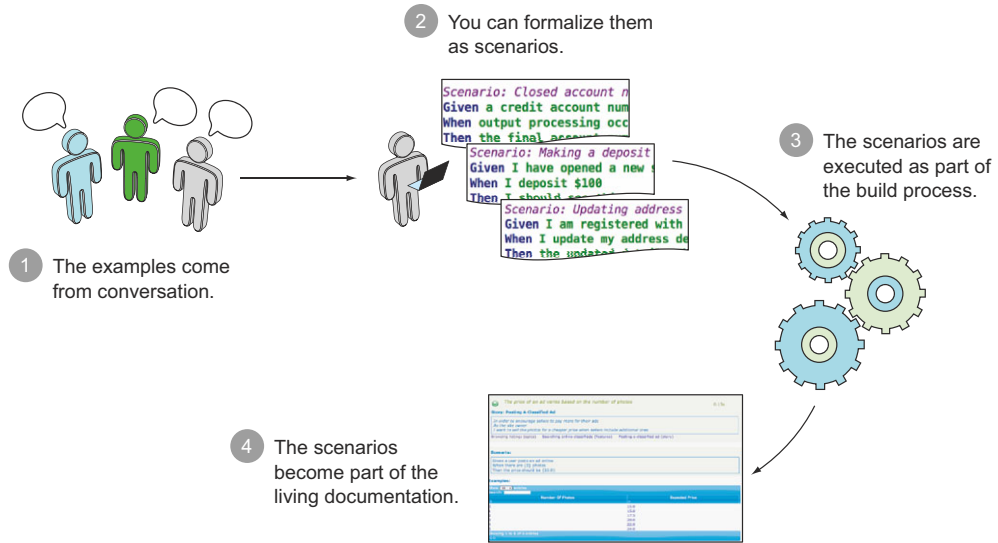



Figure 5.3 When examples are expressed as scenarios, they can be automated and used to generate living documentation.

Although the example is a little more stylized than a free-text paragraph, you're still speaking the language of the stakeholders. With a little practice, stakeholders quickly become comfortable enough with the format to be able to propose and discuss examples like this one.

This approach works equally well when the native language of the stakeholders is not English: you can write scenarios like this in any language. For example, here's the equivalent of this scenario in French:¹

```
Scénario: Gagner des points standards sur un vol en Economy
Etant donné que la distance entre Sydney et Melbourne est de 878 km
Et que je suis un member Frequent Flyer standard
Quand je voyage de Sydney jusqu'à Melbourne
Alors je devrait gagner 439 points
```

This sort of scenario is not only quite readable, it's also executable: BDD tools like JBehave and Cucumber can read and execute these scenarios to verify your application's behavior and generate meaningful test reports. These test reports are a central part of the living documentation that will help you understand and maintain the application (see figure 5.3).

We'll look at how you can take advantage of this sort of automation in chapter 6. But first, you need to learn how to write effective scenarios in this format using tools like Cucumber and JBehave.

¹ This is the French version for JBehave; the Cucumber version is slightly different.

5.2 Writing executable scenarios

Scenarios written in this format will make up the core of your executable specifications. But to make them truly executable, you need to integrate them into your projects.

In this section, you'll see how to do this in both JBehave and Gherkin. Gherkin is the language used by Cucumber and the vast majority of Cucumber-based BDD tools, including SpecFlow (for .NET), Behave (for Python), and many others. JBehave uses a very similar format: there are slight variations between the tools, but I'll point them out along the way.

Scenarios are stored in simple text files and grouped by feature. These files are called, logically enough, *feature files*.

5.2.1 A feature file has a title and a description

At the top of a feature file is a section where you can include the description of the corresponding feature. For example, in JBehave, you could write something like this:

Narrative keyword introduces an optional description. ➔ 2	← 1	The feature title.
<pre>Earning Frequent Flyer points from flights</pre>		
Narrative: In order to encourage travellers to book with Flying High Airlines more frequently As the Flying High sales manager I want travellers to earn Frequent Flyer points when they fly with us		
3 A short description of the feature.		
<pre>Scenario: Earning standard points from an Economy flight Given the flying distance between Sydney and Melbourne is 878 km And I am a standard Frequent Flyer member When I fly from Sydney to Melbourne Then I should earn 439 points</pre>		
One or more scenarios follow.		

The first line **1** is used as a title for the feature. Dan North suggests that the title should describe an activity that a user or stakeholder would like to perform.² This makes the work easier to contain and the scope easier to nail down. For example, “Earning Frequent Flyer points from flights” is a relatively well-defined user activity: when it’s implemented, Frequent Flyer members will be able to earn points when they fly. On the other hand, “Frequent Flyer Point Management” might also include rewarding Frequent Flyer members points when they make purchases with partner companies, letting members view their current point status, and so forth.

In addition to the title, it’s a good idea to include a short description of your feature so that readers can understand the underlying business objectives and background behind the scenarios that the file contains.

In JBehave, the Narrative keyword **2** is used to mark the start of an optional, free-form description **3**. As illustrated here, it’s common to use the Feature Injection format you saw in chapters 3 and 4 (“In order to ... As a ... I want”) to describe a feature.

² See Dan North’s article, “What’s in a story,” for some interesting tips on writing well-pitched stories and scenarios: <http://dannorth.net/whats-in-a-story/>.

The Gherkin version of this feature file would be very similar:

In Gherkin
use the
Feature
keyword
to indicate
a feature
title.

Feature: Earning Frequent Flyer points from flights
In order to encourage travellers to book with Flying High Airlines
more frequently
As the Flying High sales manager
I want travellers to earn Frequent Flyer points when they fly with us

Scenario: Earning standard points from an Economy flight
Given the flying distance between Sydney and Melbourne is 878 km
And I am a standard Frequent Flyer member
When I fly from Sydney to Melbourne
Then I should earn 439 points

Scenario: Earning extra points in Business class
Given the flying distance between Sydney and Melbourne is 878 km
And I am a standard Frequent Flyer member
When I fly from Sydney to Melbourne in Business class
Then I should earn 878 points

A short
description of
the feature
follows the title.

One or more
scenarios
follow.

In Gherkin, you use the Feature **1** keyword to mark the feature's title. Any text between this title and the first scenario is treated as a feature description **2**.

If you store feature descriptions electronically, using Agile software management tools or even an issue tracking system such as JIRA, you can configure reporting tools such as Thucydides to fetch this information from these systems and display it in the test reports (you'll see how to do this in chapter 11).

5.2.2 Describing the scenarios

In both Gherkin and JBehave, a scenario starts with the Scenario keyword and a descriptive title:

```
Scenario: <a title>
```

The title is important. As with most things in BDD, good communication is essential. The scenario title should summarize what is special about this example in a short, declarative sentence, a bit like a subtitle for a book. It should emphasize how it differs from the other scenarios. For example you would say, "Earning standard points from an Economy flight" or "Earning extra points in Business class", rather than "A frequent flyer member earns standard points when flying in Economy class".

Scenario titles play a key role in reporting, making the living documentation reports easier to read and navigate. Having a succinct list of scenario titles makes it easier to understand what a particular feature is supposed to do, without having to study the details of the "Given ... When ... Then" text. It also makes it easier to isolate issues when tests break.

Figure 5.4 shows an example of a Cucumber report displaying features and scenario headings in this way.

Another good practice suggested by Matt Wynne³ is to summarize the *Given* and *When* sections of the scenario in the title, and avoid including any expected outcomes.

³ See Matt Wynne and Aslak Hellesøy, *The Cucumber Book* (Pragmatic Bookshelf, 2012).

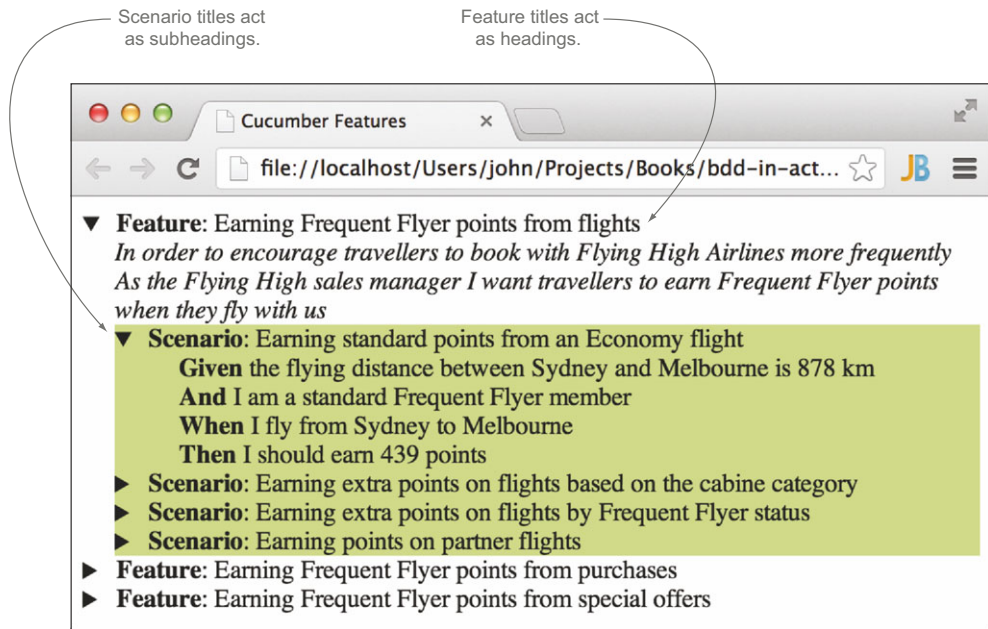


Figure 5.4 A very basic Cucumber report, showing how scenario titles act as subheadings in the living documentation reports

Because scenarios are based on real business examples, the context and events are usually relatively stable, but the expected outcomes may change as the organization changes and evolves the way it does business.

Gherkin also lets you complement the scenario title with a description,⁴ as shown in this example:

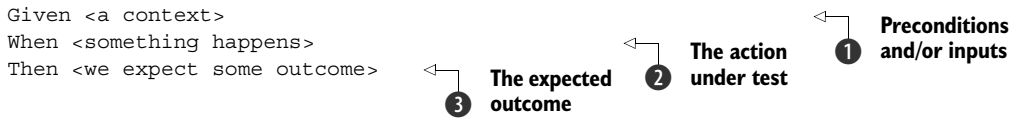
<pre>Scenario: Earning standard points from an Economy flight Normal flights earn 1 point every 2 kilometers Given the flying distance between Sydney and Melbourne is 878 km And I am a standard Frequent Flyer member When I fly from Sydney to Melbourne Then I should earn 439 points</pre>	Anything after the scenario title and before the first Given is considered to be a description.
---	---

This is a great way to add extra details about business rules or calculations, as the additional text will appear as part of the living documentation.

5.2.3 The “Given ... When ... Then” structure

The meat of each scenario is made up of three parts: an initial state or context, an action or event, and an expected result. As you saw in chapters 1 and 2, these are expressed using the following structure:

⁴ Thucydides also honors this convention in JBehave tests.



This is a simple yet surprisingly versatile format. It helps you cleanly define the context of a test **1**, what action is being tested **2**, and what the expected outcome should be **3**. It also helps you focus on what the requirement aims to achieve, rather than on how it will do so.

Let's look at each of these steps in more detail.

GIVEN SETS THE STAGE

The `Given` step describes the preconditions for your test. It sets up any test data your test needs and generally puts the application in the correct pretest state. Typically, this includes things such as creating any required test data, or, for a web application, logging on and navigating to the right page. Sometimes a `Given` step may be purely informative, to provide some context or background, even if no action is required in the test implementation.

You should be careful to only include the preconditions that are directly related to the scenario. Additional `Givens` make it harder for a reader to know precisely what's required for the scenario to work. In the same way, preconditions that should be present in the `Given` steps, but that aren't, are effectively assumptions that can lead to misunderstandings later on.

WHEN CONTAINS THE ACTION UNDER TEST

The `When` step describes the principal action or event that you want to test. This could be a user performing some action on a website, or some other non-UI event, such as processing a transaction or handling an event message. This action will generate some observed outcome, which you'll verify in the `Then` step.

THEN DESCRIBES THE EXPECTED OUTCOMES

The `Then` step compares the observed outcome or state of the system with what you expect. The outcome should tie back to the business value you expect to get out of the story or feature this scenario belongs to.

5.2.4 Ands and buts

In both Gherkin and JBehave, any of the previous steps can be extended using `and`. Gherkin also allows you to use the synonym `but`. You've seen this before:

```

Scenario: Earning standard points from an Economy flight
Given the flying distance between Sydney and Melbourne is 878 km
And I am a standard Frequent Flyer member
When I fly from Sydney to Melbourne
Then I should earn 439 points

```

The diagram shows an arrow pointing from the text "And is equivalent to Given" to the `And I am a standard Frequent Flyer member` line in the scenario above.

Technically, the BDD tools consider any step with `And` or `But` to be the same as the previous step that wasn't `And` or `But`. The main goal is to make the scenarios read more easily.

It's often a good habit to keep “Given ... When ... Then” clauses concise and focused. If you're tempted to place two conditions in the same step, consider splitting them out into two separate steps. This will make the scenario easier to read and give developers more freedom to reuse steps between scenarios.

For example, suppose travellers can earn bonus points if they fly during special bonus-flyer periods. One way to express this might be the following:

```
Scenario: Earning extra points in a bonus flyer period
Given I am a standard Frequent Flyer member and I am flying in 'Bonus
Flyer' period
When I fly from Sydney to Melbourne
Then I should earn 439 points and a special bonus of 400 points
```

A composite Given step

A composite Then step

Alternatively, you could split the Given and Then steps into smaller ones:

```
Scenario: Earning extra points in a bonus flyer period
Given I am a standard Frequent Flyer member
And I am flying in 'Bonus Flyer' period
When I fly from Sydney to Melbourne
Then I should earn 439 points
And I should earn a special bonus of 400 points
```

1 This step can be reused.

2 So can this one.

Although it's slightly longer, this second version has several advantages. Each step is focused on a particular aspect of the problem—if something breaks, or the requirements change, it will be easier to see what needs to be changed. In addition, reuse is easier. Steps **1** and **2** are also used in some of the other scenarios you've seen, so you can simplify writing and maintaining the tests by reusing them.

5.2.5 Comments

You may also occasionally want to place comments in your feature files, such as to note some technical detail about how the scenario should be implemented.

In Gherkin, you can insert a comment, or comment out a line, by placing the hash character (#) at the start of a line:

```
# This feature is really important for the Marketing team
Feature: Earning Frequent Flyer points from flights
In order to encourage travellers to book with Flying High Airlines
more frequently
As the Flying High sales manager
I want travellers to earn Frequent Flyer points when they fly with us
# I don't know how to access the distances service yet
Scenario: Earning standard points from an Economy flight
# Given the flying distance between Sydney and Melbourne is 878 km
And I am a standard Frequent Flyer member
When I fly from Sydney to Melbourne
Then I should earn 439 points
```

1 It can be used to comment out any line.

A comment line in Gherkin starts with the # character.

2 It's often used to comment out steps.

In Gherkin, comments can appear anywhere in the scenario, though they're often used to leave a technical note for other developers **1** or to temporarily comment out a step **2**.

In JBehave, a comment line starts with `!--`. Unlike Gherkin, JBehave comments can only be used to comment out steps, as shown here:

```
Scenario: Earning standard points from an Economy flight
Normal flights earn 1 point every 2 kilometers
!-- Given the flying distance between Sydney and Melbourne is 878 km
And I am a standard Frequent Flyer member
When I fly from Sydney to Melbourne
Then I should earn 439 points
```

**JBehave comments
start with `!--`** ←

TIP Don't forget the space between the `!--` and the rest of the step; otherwise JBehave won't treat it as a comment.

Unlike the narrative and descriptive texts you saw earlier, comments are not part of the living documentation.⁵ They don't appear in the reports and are therefore of limited communication value for the stakeholders. For this reason, other than for temporarily commenting out a step, comments should be used with moderation.

EXERCISE 5.1 Sarah and her friends haven't quite finished their job. They still need to come up with scenarios for each of the acceptance criteria outlined in section 4.2:

- The password should be at least 8 characters.
- The password should contain at least 1 digit.
- The password should contain at least 1 punctuation mark.
- You should get an error message telling you what you did wrong if you enter an insecure password.

Using the notation we've been discussing, write scenarios for these acceptance criteria.

5.3 *Using tables in scenarios*

Scenarios are like application code—you should write with the intention of making readability and maintenance easy. If you're using your scenarios as living documentation, they'll outlast the development project by quite a bit, and it's important to make sure that they're easy to understand and update in the future.

One way *not* to do this is to include a lot of duplicated text in your scenarios. In programming, duplication is one of the worst enemies of maintainable code, and the same applies to scenarios. But if you only use the “Given ... When ... Then” notation we've discussed so far, it's sometimes hard to avoid overly wordy scenarios peppered with duplicated text.

In this section, we'll look at how you can use tables in different ways to avoid duplication, improve readability, and make maintenance easier.

⁵ There's one exception: commented-out JBehave steps will appear as “skipped” steps in the test reports.

5.3.1 Using tables in individual steps

Suppose you're working on a feature that allows Frequent Flyer members to transfer points to other members. For example, suppose Danielle and Martin are Frequent Flyer members. Both Danielle and Martin have accumulated a lot of points over the year. They want to go on holidays together using their points, but neither of them has enough points to buy the tickets outright. Martin needs to be able to transfer some of his points to Danielle so that she can purchase the flights for both of them with her points.

You could express this scenario as follows:

```
Scenario: Transfer points between members
Given Danielle's account has 100000 points and 800 status points
And Martin's account has 50000 points and 50 status points
When Martin transfers 40000 points to Danielle
Then Martin should have 10000 points and 50 status points
And Danielle should have 140000 points and 800 status points
```

A lot of duplication here

Here too

The problem is that there's a lot of repetition and clutter in this scenario, and the meaning gets lost in all the words. A much better way to write this scenario would be to express the data in a more concise tabular format, like this:

```
Scenario: Transfer points between existing members
Given the following accounts:
  | owner   | points | status-points |
  | Danielle | 100000 | 800           |
  | Martin  | 50000  | 50            |
When Martin transfers 40000 points to Danielle
Then the accounts should be the following:
  | owner   | points | status-points |
  | Danielle | 140000 | 800           |
  | Martin  | 10000  | 50            |
```

1 Provide table of data for the Given step

2 Data for the Then step

The tables start directly after the Given **1** and Then **2** steps, with the values being separated by pipes (|). The headers at the top of each column are useful in this case, but are optional. For example, in the following step, you could provide a list of values:

```
Then I should be able to upgrade to one of the following cabin classes:
  | Premium Economy |
  | Business        |
```

Embedding tabular data is a great way to express preconditions and expected outcomes in a clear and concise way. But there's another equally useful way to use tabular data in your scenarios: tables of examples.

5.3.2 Using tables of examples

Suppose you need to implement the feature that calculates the bonus points that travellers earn based on their Frequent Flyer status—the feature that Sarah and Paul discussed in section 5.1. The basic business rules they discussed were the following:

- A standard Frequent Flyer member only earns the base point value of a trip.
- A Silver Frequent Flyer will earn a 50% status bonus.
- A Gold Frequent Flyer will earn a 75% status bonus.
- A Gold Frequent Flyer also benefits from a guaranteed minimum of 1,000 points per trip.

You could write examples illustrating these rules as separate scenarios, as shown in the following listing.

Listing 5.1 Scenarios for calculating bonus points based on Frequent Flyer status

Feature: Earning extra points from Frequent Flyer status

Scenario: A standard Frequent Flyer earns the base point value of a trip
 Given I am a Standard Frequent Flyer member
 When I fly on a flight that is worth 439 base points
 Then I should earn a total of 439 points

Scenario: A Silver Frequent Flyer will earn a 50% status bonus
 Given I am a Silver Frequent Flyer member
 When I fly on a flight that is worth 439 base points
 Then I should earn a status bonus of 220 points
 And I should earn a total of 659 points

Scenario: A Gold Frequent Flyer will earn a 75% status bonus
 Given I am a Gold Frequent Flyer member
 When I fly on a flight that is worth 2040 base points
 Then I should earn a status bonus of 1530 points
 And I should earn a total of 3570 points

Scenario: A Gold Frequent Flyer benefits from a guaranteed minimum of 1000 points per trip
 Given I am a Gold Frequent Flyer member
 When I fly on a flight that is worth 439 base points
 Then I should have a guaranteed minimum of 1000 earned points per trip
 And I should earn a total of 1000 points

This is starting to get quite wordy. Having a lot of similar scenarios to describe a set of related business rules is a poor practice; the duplication makes the scenarios harder to maintain. In addition, after the first couple of almost-identical scenarios, readers are likely to just skim over the subsequent ones and miss important details, making them a poor communication tool.

When you're writing scenarios in BDD, a good rule of thumb is "less is more." You can often describe behavior more concisely and more effectively using a single scenario and a table of examples that summarizes the different cases. For example, you could express these examples more succinctly in JBehave like this:

Scenario: Earning extra points on flights by Frequent Flyer status
 Given I am a <status> Frequent Flyer member
 When I fly on a flight that is worth <base> base points
 Then I should earn a status bonus of <bonus>
 And I should have guaranteed minimum earned points per trip of <minimum>
 And I should earn <total> points in all

Express the rule in more generic terms. Data from the example table is passed into the steps.

Examples:

status	base	bonus	minimum	total
Standard	439	0	0	439
Silver	439	220	0	659
Gold	439	329	1000	1000
Gold	2040	1530	1000	3570

The scenario will be checked for each row of data in the table.

Summarize the different cases using a few well-chosen examples.

This is effectively four scenarios wrapped into one—the scenario will be run four times, each time with the values from one row in the example table. Data from the table is passed into each step via the field names in angle brackets: <status>, <base>, and so forth. You’ve reduced four wordy scenarios into a single concise scenario and a table of examples!

Another advantage of a concise format like this is that it’s easier to spot missing examples that might need clarification and to add extra examples. In the examples we’ve discussed so far, you’ve included the guaranteed minimum points for Gold Frequent Flyers but not for Silver Frequent Flyers. Paul tells you that Silver Frequent Flyers are guaranteed to earn at least 500 points per trip. To make this clear, you can add another example to your table. You can also add a notes column, which isn’t used in the scenario but lets you annotate the examples with a little extra detail:

Scenario: Earning extra points on flights by Frequent Flyer status

Given I am a <status> Frequent Flyer member

When I fly on a flight that is worth <base> base points

Then I should earn a status bonus of <bonus>

And I should have guaranteed minimum earned points per trip of <minimum>

And I should earn <total> points in all

Examples:

status	base	bonus	minimum	total	notes
Standard	439	0	0	439	
Silver	148	74	500	500	minimum points
Silver	439	220	500	659	50% bonus
Gold	439	329	1000	1000	minimum points
Gold	2041	1531	1000	3572	75% bonus

In Gherkin, the format is similar, but you use the Scenario Outline keyword:

Scenario Outline: Earning points on flights by Frequent Flyer status

Given I am a <status> Frequent Flyer member

When I fly on a flight that is worth <base> base points

Then I should earn a status bonus of <bonus>

And I should have guaranteed minimum earned points of <minimum>

And I should earn <total> points in all

Examples:

status	base	bonus	minimum	total
Standard	439	0	0	439
Silver	148	74	500	500
Silver	439	220	500	659
Gold	439	329	1000	1000
Gold	2041	1531	1000	3572

Test data used for this scenario.

Scenario Outline keyword marks this as a table-driven scenario.

When you use a table of sample data like this, the scenario will be checked once for each row in the table, making this the equivalent of five separate scenarios. But presenting

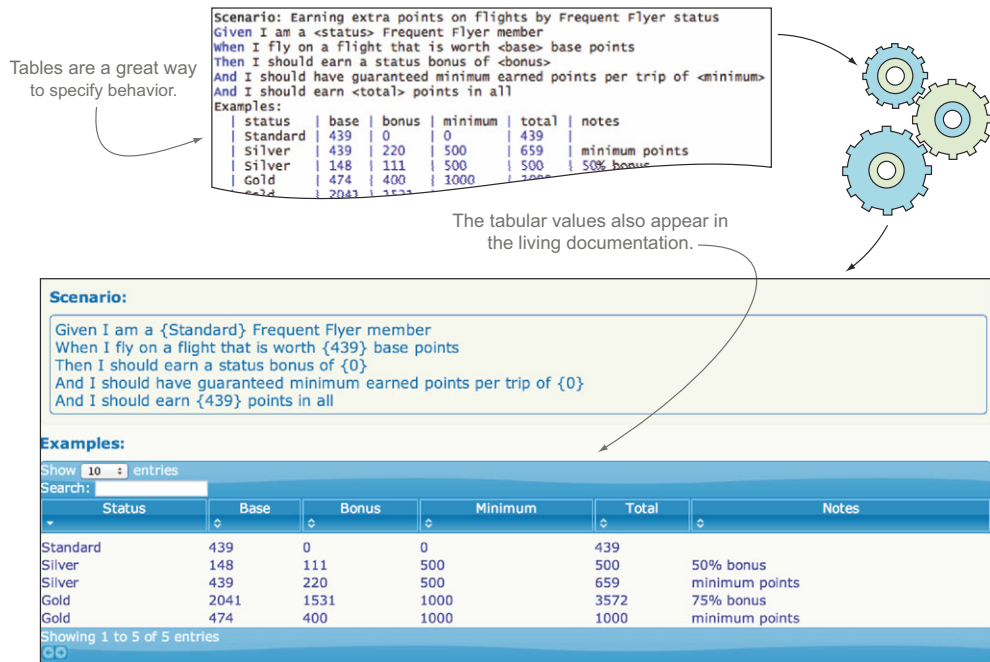


Figure 5.5 Tabular data also produces great living documentation.

data in tabular form can make it easier to spot patterns and get a more holistic view of the problem. It also makes it easy to describe and explore boundary conditions and edge cases (which is what you've done in the preceding examples), and it produces excellent living documentation (see figure 5.5).

EXERCISE 5.2 Frequent Flyer members also get a Cabin bonus if they fly in one of the premium cabins. The Cabin bonus rates are

- 25% for Premium Economy
- 50% for Business
- 100% for First

Based on the examples used in this section, come up with a concrete example of each rate, and write a table-driven scenario illustrating how the Cabin bonus works.

5.4 Expressive scenarios: patterns and anti-patterns

Now that you've seen the mechanics of writing scenarios in Gherkin and JBehave, it's time to take things to the next level. In this section we'll go beyond just seeing how scenarios are structured and written and look at what goes into a good scenario.

5.4.1 Writing expressive Given steps

Let's start with the Given step. This step should aim at getting the application in the appropriate state as quickly as possible. In this regard, it's fine to take a few shortcuts. For example, if you're testing a web application, you should try to imitate the user experience and actions, and so do as much as possible via the web interface. But if you need to set up test data in the database, it's fine to bypass the web interface and use a backend service to update the database.

The Given step, like the others, should reflect the business intent, or *what*, not the technical implementation, or *how*. For example, consider the following:

```
Given that an admin account is set up in the database
And I am logged in as admin
...
```

← **1** This sort of technical detail should not appear in the scenario.

← **2** You're only interested in the business intent expressed here.

In line **1**, too much detail is exposed about how the test is set up. This should be done silently, behind the scenes, both to avoid cluttering the scenario with unnecessary detail and to allow you to set it up as you see fit. All you're interested in at this level is the business context, which in this case is that the user is logged in with the administrator role **2**. The following version focuses the attention on the precondition in business terms:

```
Given I am logged in as an administrator
...
```

The Given step should also contain all the preconditions or steps that must have occurred *before* the action you're testing: no more, no less. For example, the following scenario is a little unclear:

```
Given that Bill registers for online banking
And that Bill opens the following accounts:
account | type | balance
123456 | savings | 1000
123457 | current | 100
When Bill logs in
And Bill goes to the home page
And Bill views his accounts
Then Bill should see a list of his accounts:
account | type | balance
123456 | savings | 1000
123457 | current | 100
```

The preconditions

The action under test (?)

The expected outcome

In this scenario, it's not clear what you're testing. Are you checking that Bill can log in successfully, or are you more interested in the accounts that he can view? In the latter case, you could rewrite this scenario as follows:

```
Given that Bill is registered for online banking
And that Bill has opened the following accounts:
account | type | balance
123456 | savings | 1000
123457 | current | 100
```

1 Putting these in the past tense makes it clear that they're preconditions.

When Bill views his account summary
 Then Bill should see all of his accounts



The preconditions here are that Bill is registered for online banking and that he has two accounts ①. If authentication has been specified elsewhere, you can probably safely assume that Bill needs to log in to see his accounts. The preconditions are also phrased in the past tense to make it more obvious that these actions are assumed to have already occurred. How Bill gets to the accounts summary page is also not the main focus of this test, so this is hidden inside step ②. The action under test is now a one-liner that more accurately represents the business action you're specifying.

The Then clause has also been simplified here ③ by assuming that you can reuse the accounts list provided in the preconditions. This reduces clutter and focuses on the essence of the expected outcome.

5.4.2 Writing expressive When steps

The aim of the When step is to execute the event or action you're testing. Like the Given step, the When step should describe the action in terms of *what*, not *how*.

The following scenario, for example, is too long and far too focused on the detailed steps involved in interacting with the user interface. This makes it tightly coupled to the implementation, which in turn makes it fragile and costly to maintain. The length of the scenario also makes it harder to read, which reduces its use as a communication medium:

```
Scenario: Register for online banking
Given that Bill wants to register for online banking
When he goes to the registration page
And he enters 'Bill' in the first name field
And he enters 'Smith' in the surname field
And he enters 'bill@smith.com' in the email field
And he enters '01/01/1980' in the date of birth field
And he enters '1 George street' in the street field
And he enters 'Sydney' in the city field
And he enters '2000' in the post code field
And he clicks on submit
Then his application should be created in a pending state
And he should be sent a PDF contract to sign by email
```

① A detailed account of the application process

The expected outcomes

This is very dense, and it contains a lot of detail about what values should go in each field on the registration page ①. But do you really need this much detail? Although it's using the user interface, this scenario is actually testing the online registration process, not the user interface. So you could hide these details in a single higher-level step, as shown here:

```
Scenario: Register for online banking
Given that Bill wants to register for online banking with the following
  details:
first-name|surname|email          |dob      |street|city | postcode
Bill      |Smith  |bill@smith.com|01/01/1980|...   |Sydney| 2000
```

① Field values are now stored in a table.

The
expected
outcomes.

3

When he submits his application online
Then his application should be created in a pending state
And he should be sent a PDF contract to sign by email

2

A higher-level view
of the submission
process.

This version doesn't mention the user interface at all and focuses primarily on the submission process ② and the expected outcomes ③. You summarize the fields you need to enter as a table ①.

But this leads us to another question: do you really need this level of detail? For this scenario, the values entered have no importance, so you can skip them entirely and simplify the scenario even further:

Scenario: Register for online banking
Given that Bill wants to register for online banking
When he submits his application online
Then his application should be created in a pending state
And he should be sent a PDF contract to sign by email

What Bill
wants

What he
does to
achieve it

The expected
outcomes

This scenario does a much better job of focusing on the business actions that you're describing. A scenario should only contain information that contributes to communicating the core business actions described by the scenario: anything else is waste.

5.4.3 Writing expressive Then steps

The Then step is where the testing takes place—this is where you describe what outcome you expect.

A common anti-pattern among new BDD practitioners is to mix the When and Then steps: a well-written scenario should only test a single action. Many teams new to BDD write scenarios that look like this:

Scenario: Updating my account address
Given I am logged in as a bank client
Then I should see the home page
When I select the 'accounts' menu
Then I should see the 'Accounts' page
And I should see a list of accounts
When I click on 'Edit'
Then I should see 'Account Details'
When I type '100 Main Street' into the street
And I type 'Armidale' into the city
And I type 'Update'
Then I should see the updated account details on the account summary page

This is more of a test script than a description of a behavior, which is very much a BDD anti-pattern. Scenarios written this way communicate poorly and are hard to maintain because they're very tightly coupled to the implementation.

TIP Good scenarios are *declarative*, not imperative. They describe the requirements in terms of *what* a feature should do, not *how* it should do it.

Once again, you can simplify this scenario by removing UI-specific implementation details and data that the business doesn't really care about. Doing this, you could write a much cleaner version of the scenario along the following lines:

```

Given I am registered with online banking
When I update my address details
Then the updated details should be visible in the account summary

```

5.4.4 *Providing background and context*

Another common way that duplication slips into scenarios is when several scenarios start off with the same steps. For example, Frequent Flyer members can log on to the Frequent Flyer website to consult their points and status, book flights, and so forth. The following Gherkin scenarios relate to logging on to this site:

```

Feature: Logging on to the 'My Flying High' website
  Frequent Flyer members can register on the 'My Flying High' website
  using their Frequent Flyer number and a password that they provide

  Scenario: Logging on successfully
    Given Martin is a Frequent Flyer member
    And Martin has registered online with a password of 'secret'
    When Martin logs on with password 'secret'
    Then he should be given access to the site

  Scenario: Logging on with an incorrect password
    Given Martin is a Frequent Flyer member
    And Martin has registered online with a password of 'secret'
    When Martin logs on with password 'wrong'
    Then he should be informed that his password was incorrect

  Scenario: Logging on with an expired account
    Given Martin is a Frequent Flyer member
    And Martin has registered online with a password of 'secret'
    But the account has expired
    When Martin logs on with password 'secret'
    Then he should be informed that his account has expired
    And he should be invited to renew his account

```

Note the duplicated steps

These scenarios contain a lot of repetition, which makes them harder to read and to maintain. In addition, if you ever need to change one of the duplicated steps, you'll need to do so in several places.

In Gherkin, you can avoid having to repeat the first two steps by using the Background keyword, as shown here:

```

Feature: Logging on to the 'My Flying High' website
  Frequent Flyer members can register on the 'My Flying High' website
  using their Frequent Flyer number and a password that they provide

  Background: Martin is registered on the site
    Given Martin is a Frequent Flyer member
    And Martin has registered online with a password of 'secret'

  Scenario: Logging on successfully
    When Martin logs on with password 'secret'
    Then he should be given access to the site

  Scenario: Logging on with an incorrect password
    When Martin logs on with password 'wrong'
    Then he should be informed that his password was incorrect

```

These steps will be run before each scenario.

The scenarios are more focused.

```

Scenario: Logging on with an expired account
  Given the account has expired
  When Martin logs on with password 'secret'
  Then he should be informed that his account has expired
  And he should be invited to renew his account

```

The Background keyword lets you specify steps that will be run before each scenario in the feature. You can use this to avoid duplicating steps in each scenario, which also helps focus attention on the important bits of each scenario.

In JBehave, you can do something similar with the `GivenStories` keyword, though it's a bit more technical. This keyword lets you run the scenarios in arbitrary .story files either before each scenario or once before any of the scenarios are executed. For example, you could place the background steps in a file called `Martin_has_registered.story`:

```

Scenario: Registering for the site
Given Martin is a Frequent Flyer member
And Martin has registered online with a password of 'secret'

```

**This goes in
a separate
.story file.**

Then you would modify your main .story file to refer to the preceding file using the `GivenStories` keyword, as follows.

Listing 5.2 A JBehave story using the `GivenStories` keyword

```

Logging on to the 'My Flying High' website
Scenario: Logging on successfully
GivenStories: Martin_has_registered.story
When Martin logs on with password 'secret'
Then he should be given access to the site

Scenario: Logging on with an incorrect password
GivenStories: Martin_has_registered.story
When Martin logs on with password 'wrong'
Then he should be informed that his password was incorrect

Scenario: Logging on with an expired account
GivenStories: Martin_has_registered.story
Given the account has expired
When Martin logs on with password 'secret'
Then he should be informed that his account has expired
And he should be invited to renew his account

```

5.4.5 Avoid dependencies between scenarios

It's important to remember that even if the scenarios live together in the same feature file, each scenario should be able to work in isolation. A scenario should not depend on a previous one running to set up data or to put the system into a particular state.

For example, suppose you're writing scenarios for an e-commerce site. You might write something like this:

```

Scenario: Adding an item to the cart
  Given I select 'BDD in Action'
  When I add it to the cart
  Then my cart should contain one copy of 'BDD in Action'

```

**1 Adding
something
to the cart**


```
Scenario: Purchasing items in my cart
  Given I am ready to order
  When I pay for the items in my cart
  Then I should be sent a copy of 'BDD in Action'
```

2 Checking out

This is wrong on several levels. The second scenario ② will not work unless the first one ① was run directly before it, and the system remained in the same state. But the testing framework will create a fresh environment for each scenario to run in, so the cart will be empty at the start of the second scenario.

In addition, you can't guarantee the order in which the scenarios will be run, or even which scenarios will be run (you may only run a subset of the scenarios for a particular test run).

On another level, these tests are a little too granular. They focus on small chunks of the business flow, at the risk of losing view of the overall objectives. In fact, a good way to make your tests more robust is to make them very high-level. A good acceptance criterion captures the business intent of a requirement, not the mechanics of how it gets done. For example, suppose your website prides itself on being environmentally friendly and is famed for its innovative delivery methods. Your real requirement might be more accurately captured in a high-level scenario like this one:

```
Scenario: Buying an item
  Given I am looking for a good book to read
  When I purchase a copy of 'BDD in Action'
  Then I should be sent my copy by specially trained environmentally
  friendly courier pigeon
```

High-level scenarios like this give a much better picture of why you need to implement a feature, and what sort of things you'll value in it. Scenarios like this are highly maintainable, because the business scenario you're describing is much less likely to change than, for example, the user interface you'd build to deliver this feature.

Another way to avoid dependencies between scenarios is to focus on specific business rules. Not all scenarios will be high-level, end-to-end scenarios like the preceding "Buying an Item" example. Others will capture more specific business requirements such as the Frequent Flyer point calculations that you saw earlier in this chapter. Once again, the scenarios should focus on the business requirements and expected outcomes, rather than on the implementation details of how this happens.

Of course, sometimes you'll want to see the details. Don't worry: in the following chapters you'll see how to write high-level tests at this level that are both very maintainable and that still provide all the nitty-gritty low-level details for anyone who needs them.

5.5 *Organizing your scenarios using feature files and tags*

In real-world projects, scenarios can become quite numerous, and it's important to keep them well organized. You may also need to be able to identify and group scenarios in different ways; for example, you might want to distinguish UI-related scenarios from batch-processing scenarios, or identify the scenarios related to cross-functional

concerns. In this section, you'll learn how to organize and group your scenarios to make them easier to understand and maintain.

5.5.1 The scenarios go in a feature file

The role of a scenario is to illustrate a feature, and you place all the scenarios that describe a particular feature in a single file, usually with a name that summarizes the feature (for example, `earning_points_from_flights.feature`). In JBehave, these files conventionally use the `.story` suffix (`earning_points_from_flights.story`), whereas the Gherkin-based tools use the `.feature` suffix. These files can be read and edited in a simple text editor, though plugins also exist for most modern IDEs. During the rest of the chapter, we'll refer to these files as *feature files*, regardless of the file suffix used.

As you saw in chapter 2, these scenarios are part of the project's source code, and they'll be placed under version control. Many teams write the feature files during the “Three Amigos” sessions (see the “Different types of conversations” sidebar in section 4.5) and store them in the source code repository at the end of the meetings.

The exact file structure used to store the feature files varies from tool to tool. For example, figure 5.6 illustrates a typical Cucumber/Java project, whereas figure 5.7 shows the equivalent story files in a JBehave project.

The `.story` suffix used by JBehave has historical origins and is a little misleading. It's generally a bad idea to have a `.story` file for each user story.⁶ Remember, stories are transitory planning artifacts that can be disregarded at the end of an iteration, but features are valuable units of functionality that stakeholders can understand and relate to.

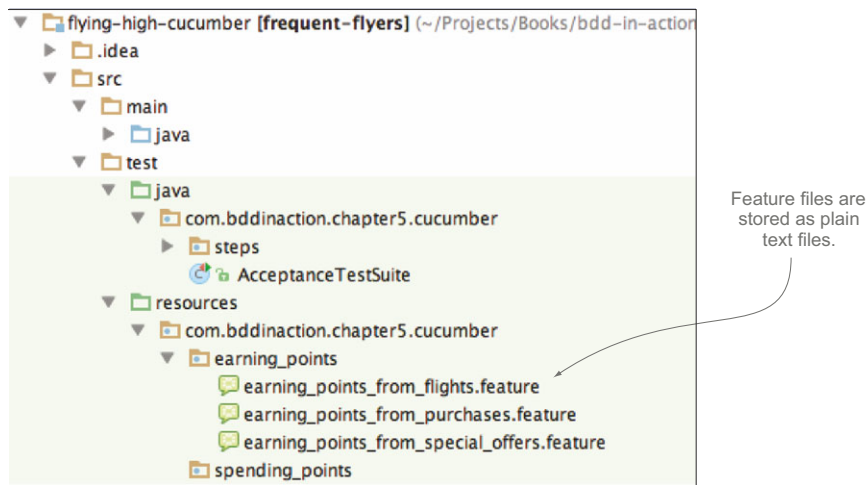


Figure 5.6 Feature files are stored in plain text files as part of the project source code (this example is from a Cucumber project in Java).

⁶ There's nothing wrong with using tags or some other metadata to relate a scenario to a story for planning and reporting purposes. You'll see how to use tags as an additional way to organize your requirements in section 5.5.4.

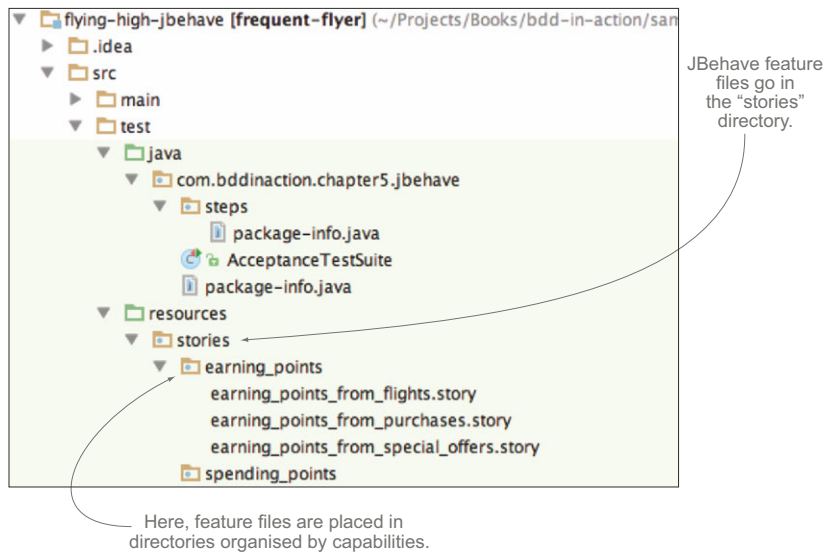


Figure 5.7 Organizing feature files into folders makes them easier to manage and navigate. In this JBehave project, the features are organized by high-level capability.

Organizing scenarios in terms of features rather than stories makes it easier to generate meaningful living documentation. It can be useful to associate scenarios with the corresponding stories for planning and reporting purposes during an iteration, but the primary association should be between a scenario and the feature it illustrates.

5.5.2 A feature file can contain one or more scenarios

As discussed in chapter 4, you use concrete examples to illustrate each feature. Any example you decide to automate will be represented by a single scenario in the corresponding feature file. As a result, the feature file will contain all of the examples that illustrate its expected behavior. These examples illustrate not only the simple cases but also alternative paths and edge cases that flesh out how the feature behaves in different situations.

5.5.3 Organizing the feature files

When you start to get a large number of feature files in your project, it's important to keep them organized in a way that makes them easy to find and browse. A good way to do this is to place them into subdirectories, as illustrated in figure 5.7. Of course, you'll need to decide what directory structure makes the most sense for your team. In my experience, if you've organized your features in terms of capabilities, as described in chapters 3 and 4, it's very natural to create a directory for each capability.

As you'll see later on, the Thucydides reporting tool (<http://www.thucydides.info>) can be configured to use this directory structure to group executable requirements by features, capabilities, tags, and so forth.

5.5.4 Annotating your scenarios with tags

You’ve seen (in section 5.5.1) how you can organize your feature files into subdirectories that mirror your requirements’ structure. But often it’s nice to have other ways to categorize your scenarios. Fortunately, this is quite easy to do. Both JBehave and Gherkin let you add *tags* to scenarios.

For example, many projects store details about features or user stories in issue-tracking software such as Atlassian’s JIRA. In this case, it’s useful to relate a feature or an individual scenario back to the corresponding issue, both for information and so that reporting tools can use this data to create a link back to the corresponding issue. In JBehave, you can do this using the *Meta* keyword:

```
Earning extra points from Frequent Flyer status
```

```
Meta:
```

```
@issue FF-123
```

```
Scenario: Earning extra points on flights by Frequent Flyer status
```

```
...
```

Tags need to be introduced
by the *Meta* keyword.

Tags start with @ and
can be any text value.

TIP When you’re defining tags with the *Meta* keyword in JBehave, it’s especially important not to leave out the title in the first line. If you forget to include this text, JBehave will treat the *Meta* statement as a title, which may result in errors in your living documentation.

In Gherkin, things are even simpler:

```
Feature: Earning extra points from Frequent Flyer status
```

```
@issue FF-123
```

```
Scenario Outline: Earning points on flights by Frequent Flyer status
```

```
...
```

Gherkin doesn’t need
the *Meta* keyword.

Tags are also a great way to categorize scenarios by other cross-functional concerns, to identify related parts of the system, and to help organize test execution. For example, you might want to flag all of the web tests, or mark certain tests as being slow, so that they can be grouped together during the automated build process:

```
Feature: Earning extra points from Frequent Flyer status
```

```
@issue FF-123
```

```
@web @slow
```

```
Scenario Outline: Earning points on flights by Frequent Flyer status
```

```
...
```

Tags can be used to
identify related cross-
functional concerns.

This way, when executing the tests, you can configure a filter to only run the tests with a particular tag (or *without* a particular tag).⁷

Some BDD tools (Cucumber, in particular) also let you write *hooks*—methods that will be executed before or after a scenario with a specific tag is executed. This is a

⁷ You’ll see how to do this in chapter 6.

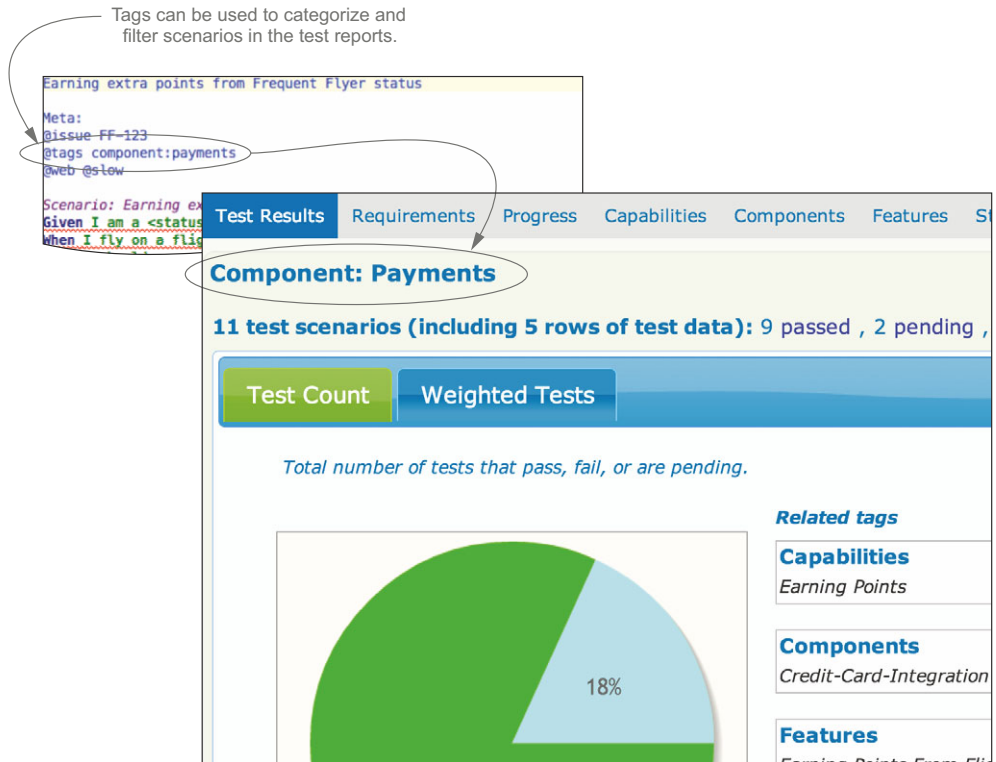


Figure 5.8 Using tags to organize the test reports

great way to set up a test environment for certain types of tests, or to clean up afterwards. You'll learn more about how to do this in chapter 6.

You can also use tags as a powerful reporting tool (see figure 5.8). For example, Thucydides lets you define tags that will appear in the test reports using `@tag` or `@tags`:

```
Logging on to the 'My Flying High' website
```

```
Meta:
@tag component:SSO
```

```
Scenario: Logging on successfully
...
```

Scenarios in this feature file all involve the SSO component.

These tags can take any name/value combination and so can be used to report on whatever cross-functional aspects you need in the living documentation.

5.6 Summary

In this chapter you learned about creating executable scenarios:

- Examples can be turned into executable specifications by writing them as scenarios using variations on the “Given ... When ... Then” format.

- The executed scenarios are organized in feature files. The exact format varies slightly between JBehave and the Gherkin-based tools (Cucumber and others).
- Scenarios can be made more concise and more expressive by using embedded data tables and table-driven scenarios.
- Scenarios can be completed with background information and tags.

You've now hopefully learned enough to be able to write your own feature definitions, illustrated with concise, expressive scenarios. In the next chapter, you'll see how to automate and implement this using several different languages and environments.

Automating the scenarios

This chapter covers

- The basic principles of automating your scenario steps
- The responsibilities of a step definition method
- Implementing step definitions in Java using JBehave and Cucumber-JVM
- Implementing step definitions in Python using Behave
- Implementing step definitions in .NET using SpecFlow
- Implementing step definitions in JavaScript using Cucumber-JS

So far, you've seen how you can describe and discuss your requirements very effectively using concrete examples. You also learned how you can express these examples in a loosely structured format built around the "Given ... When ... Then" structure.

A lot of the value in BDD comes from the conversations around these scenarios. This is why collaborating to write these scenarios is so important. Not all scenarios need to be automated; some may be too tricky to automate cost-effectively, and can

be left to manual testing. Others may only be of marginal interest to the business, and might be better off implemented as unit or integration tests. Still others may be experimental, and might not be understood well enough to define clear scenarios; in this case, it could be worthwhile to do some initial prototyping to get a better feel for what's really needed.¹

But when a scenario can be automated, when it makes sense to do so, and when it's done well, automating the scenario brings its own set of undeniable benefits:

- *Testers spend less time on repetitive regression testing.* When acceptance criteria and the corresponding scenarios are written in close collaboration with testers, automated versions of these scenarios give testers more confidence in new releases. The testers can understand and relate more easily to what the automated tests are verifying, because they took part in defining them. In addition, the application that the testers receive for testing will have already passed a broad range of simpler test cases, letting the testers focus on more complex or exploratory testing.
- *New versions can be released faster and more reliably.* Because less manual testing is required, new releases can be pushed out more efficiently. New versions are less likely to introduce regressions. Comprehensive automated testing is essential if you're trying to implement continuous integration, continuous delivery, or continuous deployment (see chapter 12).
- *The automated scenarios give a more accurate vision of the current state of the project.* You can use them to build a progress dashboard that describes which features have been delivered and how they've been tested, based on the results of the automated scenarios.

Continuous integration, continuous delivery, and continuous deployment

Continuous integration is a practice that involves automatically building and testing a project whenever a new code change is committed to the source code repository. Continuous integration is a valuable feedback mechanism, alerting developers to potential integration issues or regressions as early as possible. But to be really effective, continuous integration relies strongly on a robust and comprehensive set of automated tests.

Continuous delivery is an extension of continuous integration, where every build is a potential release. Whenever a developer puts new code into the source code repository, a build server compiles a new release candidate version. If this release candidate passes a series of automated quality checks (unit tests, automated acceptance tests, automated performance tests, code quality metrics, and so on), it can be pushed into production as soon as business stakeholders give their go-ahead.

¹ Highly experimental startup applications might fall into this category. If the business needs to get market feedback to discover what features they really need, it will be hard to formalize scenarios with very much detail.

(continued)

Continuous deployment is similar to continuous delivery, but there's no manual approval stage. Any release candidate that passes the automated quality checks will automatically be deployed into production. The deployment process itself is often automated using tools like Chef, Puppet, or Octopus Deploy.

Both continuous delivery and continuous deployment encourage a much more streamlined, efficient deployment process. And both require a very high degree of confidence in the application's automated test suites.

In chapter 5 you saw how to express scenarios using the “Given ... When ... Then” notation. In this chapter, you'll learn how to write the test code that automates these scenarios (see figure 6.1).

In this chapter, you'll learn how to automate these scenarios:

- You can automate a scenario by writing step definitions that interpret each step in a scenario and execute the corresponding test code.
- By using different BDD tools, step definitions can be implemented in different languages, including Java (JBehave and Cucumber), JavaScript (Cucumber-JS), Python (Behave), and .NET (SpecFlow).

BDD automation libraries are all quite similar, and what's applicable to one tool can often be directly transferred to others, with one minor caveat. The “Given ... When ... Then” format you learned in chapter 5 is often referred to as *Gherkin syntax*. For the purists, this isn't strictly accurate: Gherkin is the syntax used by Cucumber and by the ports of Cucumber in different languages. JBehave has its own syntax, which was developed independently and has a few minor differences.

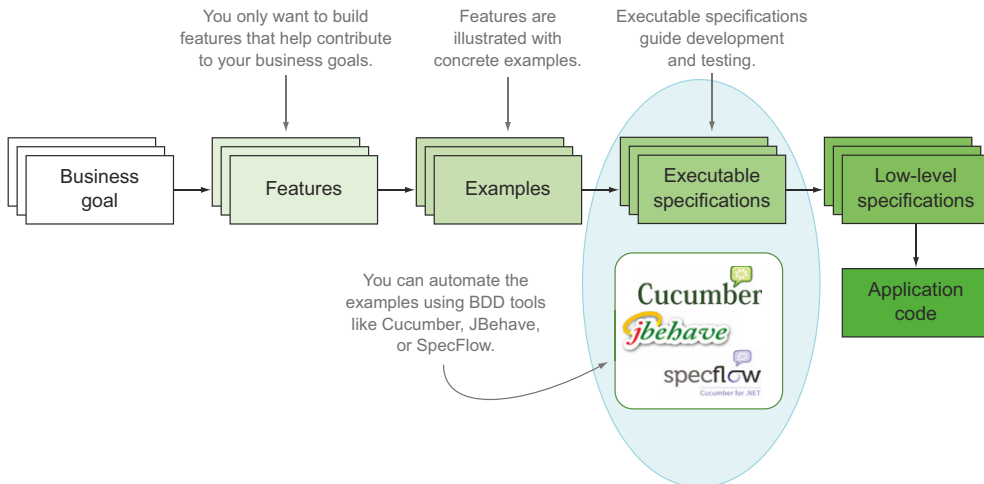
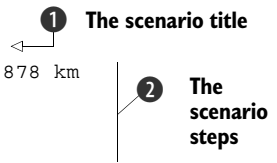


Figure 6.1 In this chapter we'll concentrate on how you can automate the executable specifications using BDD tools like JBehave, Cucumber, and SpecFlow.

For the rest of this chapter, we'll take a closer look at what BDD-style automation looks like on a variety of platforms, pointing out the specificities of each platform. But before we do this, we need to discuss some general principles that will apply no matter what tool you choose.

6.1 Introduction to automating scenarios

Before we look at specific tools, let's go through the basics. In chapter 5, you saw how to describe requirements in terms of plain-text scenarios like the following:



```
Scenario: Earning standard points from an Economy flight
Given the flying distance between Sydney and Melbourne is 878 km
And I am a standard Frequent Flyer member
When I fly from Sydney to Melbourne
Then I should earn 439 points
```

This scenario is just loosely structured text. It explains what requirement you're trying to illustrate ① and how you intend to demonstrate that your application fulfills this requirement ②. But how you actually perform each step will depend on your application and on how you decide to interact with it.

For example, the preceding scenario is made up of four steps ②. Each of these steps needs to interact with the application to prepare the test environment, perform the action under test, and check the results. For example, consider the first Given step:

```
Given the flying distance between Sydney and Melbourne is 878 km
```

Here you need to configure a test database to provide the correct distance between Sydney and Melbourne. You could do this in many ways: you may want to inject data directly into a test database, call a web service, or manipulate a user interface. The text describes *what* you intend to do, but *how* you do this will depend on the nature of the application and on your technical choices.

The other steps are similar. For example, the first When step describes the action you're testing:

```
When I fly from Sydney to Melbourne
```

Again, this step describes *what* you want to do. You want to record a flight from Sydney to Melbourne so that you can check how many points the member earns. But *how* you do this requires more knowledge about your application and its architecture.

Tools like JBehave and Cucumber can't turn a text scenario into an automated test by themselves; they need your help. You need a way to tell your testing framework what each of these steps means in terms of your application, and how it must manipulate or query your application to perform its task. This is where *step definitions* come into play.

6.1.1 Step definitions interpret the steps

Step definitions are essentially bits of code that interpret the text in feature files and know what to do for each step (see figure 6.2).

In Ruby it would look like this:

```
Given /^the flying distance between (.*) and (.*) is (\d+) km$/ do
  |departure, destination, distance|
  ...
end
```

The test automation library will read the feature files and figure out what method it should call for each step. You can also tell the library how to extract important data out of the text and pass that data to the step definition method. But it's the step definition's job to do whatever needs to be done to perform this step.

Because the feature files and annotations consist of free text, there's always a risk that when the text in a feature file is modified, you may forget to update the text in the annotation, or vice versa. In this case, the affected scenario or scenarios will be flagged as pending once again. To help developers manage this sort of issue, many modern IDEs have plugins for BDD tools like JBehave, Cucumber, and SpecFlow, which highlight steps in scenarios that don't have matching methods.

6.1.2 Keep the step definition methods simple

A step definition method is like the conductor of an orchestra: it knows at a high level how to perform a task, and it coordinates calls to other more application-specific libraries that do the more detailed work.

Step definitions should be clean, simple, and descriptive: they should describe what you need to do to the application, or what you need to ask it. For any but the most trivial of applications, you should regroup the code that manipulates the application into a layer that's separate from the step definitions themselves. A typical architecture involves a minimum of three layers:

- *Scenarios* describe the high-level requirement:

```
Scenario: Earning standard points from an Economy flight
Given the flying distance between Sydney and Melbourne is 878 km
And I am a standard Frequent Flyer member
When I fly from Sydney to Melbourne
Then I should earn 439 points
```

- *Step definitions* interpret the scenario texts and call the test automation layer to perform the actual tasks:

```
@Given("the flying distance between $a and $b is $distance km")
public void defineTheFlyingDistanceForATrip(...) {...}
```

- *The test automation layer* interacts with the application under test:

```
inTheTestDatabase.theDistanceBetween(departure)
    .and(destination)
    .is(distance);
```

This architecture is illustrated in figure 6.3.

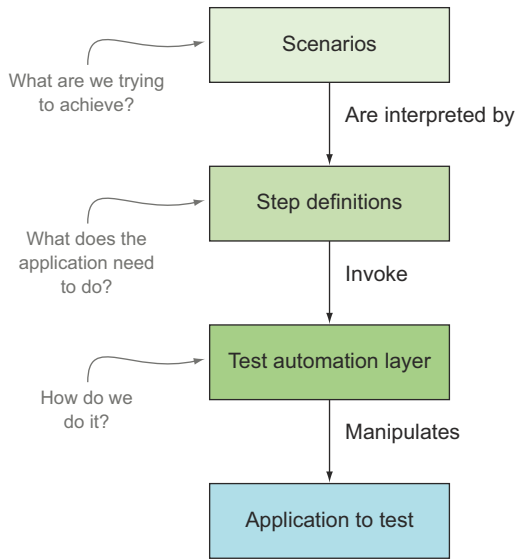


Figure 6.3 A well-designed test automation suite has a number of layers.

For example, you could implement the step that defines the flying distance by directly injecting data into a test database, by calling a web service, or via the user interface. But the step definition shouldn't need to worry about such details; its job is to set up the test data described in the scenario. You might do this as shown here:

Configure the test database through a dedicated API.

1

```

TestDatabaseAPI inTheTestDatabase;

@Given("the flying distance between $departure and $destination is
      $distance km")
public void defineTheFlyingDistanceForATrip(String departure,
      String destination,
      int distance) {
    inTheTestDatabase.theDistanceBetween(departure)
        .and(destination)
        .is(distance);
}
  
```

Use this API to set up the test data.

2

Inject the test data into the database.

3

In this example, the step definition method interprets the text in the feature file 2 and uses a dedicated class 1 to configure the test database 3. The step definition method doesn't care how it's done; it just describes the action it needs to achieve. The application-specific library worries about the rest.

Using a layered architecture like this is a key part of writing maintainable, readable, automated acceptance tests. You'll see much more of this approach in the coming chapters, particularly in chapter 7.

6.2 Implementing step definitions: general principles

There are many BDD tools you can use to automate scenarios like the ones we've been looking at, using different languages and often targeting different environments. The choice of which tool is right for your team will depend on how comfortable the team is

with a particular language and environment (and how willing they are to learn a new one!), what technology stack your project is using, and the goals and target audience of your BDD activities. Throughout the rest of this chapter, we'll compare and contrast several of the major BDD tools. The list of tools we'll study is far from exhaustive, but the techniques we'll discuss should be generally applicable no matter which tool you choose.

To explore the features of each tool, we'll look at how you might automate scenarios involving the Flying High airline's Frequent Flyer application, introduced in chapter 3. The Frequent Flyer program is designed to encourage flyers to fly with Flying High by allowing them to accumulate points that they can spend on flights or on other purchases. In this chapter's examples, we'll study some of the requirements around accumulating and managing Frequent Flyer points.

But first, let's take a general look at some of these tools' principles and uses.

6.2.1 Installing BDD tools

When it comes to installation, BDD tools fall generally into one of two categories. Some, like the original Ruby implementation of Cucumber, and most of the Cucumber family, are essentially command-line tools. These are easy to install and run from the command line, though they can be executed from a build script as well. For example, to run Cucumber scenarios written in a scripting language like Ruby, you simply place your feature files in a directory called features and run Cucumber from the command line.

Others are primarily run from within a build scripting environment (such as Maven, Ant, or Gradle in the JVM world), or from within an IDE. This includes JBehave (as you'll see in this section), Cucumber-JVM, and SpecFlow. These tools require a little more work and project infrastructure to set up.

6.2.2 Implementing step definitions

Implementing step definitions is at the heart of any BDD automation library. Step definitions tell the BDD tool exactly what code it should execute for each step in the feature file, and they extract any variables that are embedded in the text so that they can be used in the test code.

The way this is done is generally constrained by the language you're using to implement the steps. Many BDD tools use annotations, as illustrated here for JBehave:

```
@When("I register for the Frequent Flyer program")
public void whenIRegisterForTheFrequentFlyerProgram() {
    ...
}
```

Scripting languages such as Ruby and Groovy can take advantage of language features to make the step definitions more concise. In Ruby,² for example, a Cucumber step implementation might look like this:

² Although Ruby is a major actor in the BDD universe, I won't cover Cucumber in Ruby in as much detail as the other tools. This is mainly because of space—there are other great references for implementing BDD automation in Ruby, in particular *The Cucumber Book* by Matt Wynne and Aslak Hellestøy (Pragmatic Bookshelf, 2012).

```
When /^I register for the Frequent Flyer program$/ do
  ...
end
```

If you prefer sticking to the JVM ecosystem, the following would be the equivalent of the previous step implementation using Cucumber-JVM and Groovy:³

```
@When(~"I register for the Frequent Flyer program") {
  ...
}
```

No matter which tool you're using, step definitions are usually global—you can't have the same text referring to different methods in different scenarios, although at times this might be useful.

6.2.3 Passing parameters to step implementations

Simple step definitions like the preceding ones won't get you very far. To automate scenarios effectively, you need to be able to extract data from the step definitions so that you can use them in the implementation code. Fortunately, BDD tools make it very easy to do this.

Suppose you need to automate the following three steps from different scenarios:

```
Given I am a Bronze Frequent Flyer member
Given I am a Silver Frequent Flyer member
Given I am a Gold Frequent Flyer member
```

These all represent effectively the same action (setting up a Frequent Flyer member with a given status). It would clearly be inefficient if you had to write a separate step implementation for each one of these variations. Fortunately, there's an alternative. All of the BDD tools let you pass variables into your step implementations—both individual field values and entire tables of data. For example, in Ruby, you could parameterize the previous steps like this:

```
Given /^I am a (.*) Frequent Flyer member$/ do |status|
  ...
end
```

This would pass the Frequent Flyer status value in to the `status` parameter, so that it could be used in the step implementation.

BDD tools that use typed languages can also often take advantage of automatic type conversions when the parameters are passed in to the step implementations. For example, here's the same step implementation, implemented using Cucumber-JVM in Java, which uses an enumeration to define the different possible status values:

³ Groovy is a dynamic JVM language with a syntax similar to Java but with many features usually found in more flexible languages such as Ruby and Python.

```
public enum Status {Gold, Silver, Bronze};

@Given("I am a (.*) Frequent Flyer member")
public void useAMemberWithAGivenStatus(Status status) {
    ...
}
```

This use of regular expressions is typical of most of the Cucumber-based tools. Regular expressions are extremely powerful and allow a great deal of flexibility in how you express your step definitions. But they can be intimidating for developers who aren't used to them. Some tools, such as JBehave and SpecFlow, propose simpler (and often more readable) alternatives to regular expressions. For example, in JBehave you can use variable names instead of regular expressions, as shown here:

```
@Given("I am a $status Frequent Flyer member")
public void useAMemberWithAGivenStatus(Status status) {
    ...
}
```

This style gains in readability, at the cost of a certain loss of flexibility.

6.2.4 Maintaining state between steps

A scenario contains several steps, and sometimes you'll need to pass information from one step to another. The simplest way to do this is to use member variables within the class or script containing the step definitions. A new instance of the step definitions will be created for each new scenario execution, so there's no chance of the values being overwritten.

The following code illustrates this principle. Suppose you're implementing the following scenario:

```
Scenario: Earning extra points on flights based on frequent flyer status
  Given I am a Silver Frequent Flyer member
  When I fly from Sydney to Melbourne on 01/01/2013 at 09:00
  Then I should earn 439 points
```

To do this, you need to create or retrieve a Frequent Flyer member object in the first step and an object corresponding to the requested trip in the second, and then use both of these objects to calculate how many points this member would earn for this trip. The corresponding step definition class using JBehave might look like this:

```
public class EarningPointsSteps {
    TripSteps trips;
    Members members;

    FrequentFlyerMember member;
    Trip trip;

    @Given("I am a $status Frequent Flyer member")
    public void defineAMemberWithStatus(String status) {
        member = members.getMember().withStatus(status);
    }
}
```

Application-specific libraries.

Domain objects used to store state between steps.

Find or create a member with a given status.


```

@When("I fly from $departure to $destination on $date at $time")
public void whenIFlyFrom(String departure, String destination,
                          DateTime date, LocalTime time) {
    trip = trips.lookupTrip(departure, destination, time, date);
}

@Then("I should earn $points points")
public void thenIShouldEarn(int expectedPoints) {
    int earnedPoints = member.getPointsFor(trip);
    assertThat(earnedPoints).isEqualTo(expectedPoints);
}

```

Look up the corresponding trip.

Determine how many points are earned for this trip.

Because a new instance of the class containing the step definitions will be created for each scenario, it's safe to use member variables this way. Even if these steps are used in other scenarios, each scenario will have its own member variable.

Although this practice is safe and simple, it can lead to code that's hard to maintain if there are variables used by different scenarios in the same step definition class. This is why many practitioners prefer to put related step definitions (for example, the step definitions for a given feature file or even for a given scenario) in the same class to make the flow of the steps easier to understand.

6.2.5 Using table data from step definitions

In chapter 5 you saw that you can embed tables within the steps to represent more complicated structures. The following scenario uses embedded tabular data to both set up the initial test database and verify the outcomes:

Scenario: Transfer points between existing members

Given the following accounts:

owner	points	statusPoints
Jill	100,000	800
Joe	50,000	50

When Joe transfers 40000 points to Jill

Then the accounts should be the following:

owner	points	statusPoints
Jill	140,000	800
Joe	10,000	50

Using tabular data like this is an important part of step implementation, and it's something you're likely to do often. In the preceding scenario, for example, you'd use the data in the first step to create these Frequent Flyer accounts, and then read the resulting accounts and compare them with the figures provided in the second table.

Most tools provide APIs to help you do this sort of thing. The capabilities of these APIs vary from one tool to another, and some languages are easier to work with in this regard than others. One very common requirement is to iterate over the table rows and extract the field values, and then use them to set up test data. In JBehave, for example, you could write a step implementation to do this in the following way:

Iterate over each row in example table	<pre>@Given("the following accounts: \$accounts") public void givenTheFollowingAccounts(ExamplesTable accounts) { for(Parameters account : accounts.getRowsAsParameters()) String owner = account.valueAs("owner", String.class); int points = account.valueAs("points", Integer.class); int statusPoints = account.valueAs("statusPoints", Integer.class); inTheTestDatabase.addAccount (Account.forMember(owner) .withPointBalance(points) .withStatusPoints(status)); } }</pre>	Extract field values for row
Create new account record using these values		

An equivalent Cucumber implementation written in Ruby, on the other hand, could use some of Ruby's dynamic language features to write a more concise step implementation:

```
Given /^the following accounts$/ do |accounts|
  Account.create! (accounts.hashes)
end
```

Pass in the accounts table.

Convert the table data to a list of hash maps, and use this to create corresponding account records.

Most tools also let you use more advanced API or language-related features to convert example tables directly into domain objects, which can simplify the automation code considerably. We'll look at some examples of these more advanced features in the sections on each specific tool, later in this chapter.

The other main use of tabular data is to compare actual results to expected results. For example, you might want to check that records in the database have been correctly added or updated, or that you obtain the search results you expect from a multi-criteria search. But writing custom logic to compare the results row by row is time-consuming and error-prone.

Fortunately all the tools make it easy to compare sets of tabular data, either with other tables that you build or by comparing the data tables with lists of hash maps or domain objects. For example, using Cucumber-JVM in Java, you can directly compare a table with a list of domain objects:

Load the relevant accounts from the database.	<pre>@Then("^the accounts should be the following:\$") public void the_accounts_should_be_the_following(DataTable expected) throws Throwable { List<Account> actualAccounts = loadCurrentAccountsFor(fromMember, toMember); expectedAccounts.diff(actualAccounts); }</pre>	Compare their field values with the values in the expected table.
--	---	--

When you use the provided table APIs, error reporting is better and troubleshooting easier. For example, when you run this scenario, if the accounts aren't what you expect, the tools will report what data you were expecting and what it actually found.

6.2.6 Implementing example-based scenarios

You saw in chapter 5 how you can summarize a number of related examples into a single scenario using tables of examples, like the following:

Scenario Outline: Earning extra points on flights by Frequent Flyer status
 Given I am a <status> Frequent Flyer member
 When I fly on a flight that is worth <base> base points
 Then I should earn a status bonus of <bonus>
 And I should have guaranteed minimum earned points per trip of <minimum>
 And I should earn <total> points in all

Examples:

status	base	bonus	minimum	total	notes
Standard	439	0	0	439	
Silver	439	220	500	659	50% bonus
Silver	148	74	500	500	minimum points
Gold	474	400	1000	1000	minimum points
Gold	2041	1531	1000	3572	75% bonus

Create new
Frequent
Flyer
member
with a given
status

1

Although this has a similar format to the embedded data tables in the previous section, it's a very different beast and generally easier to use. For example, in JBehave, for line ❶ you could simply reuse the step definition you saw earlier:

```
@Given("I am a $status Frequent Flyer member")
public void defineAMemberWithStatus(String status) {
    member = members.getMember().withStatus(status);
}
```

This is useful, as steps are often used interchangeably between the two types of scenarios.

6.2.7 Understanding scenario outcomes

Traditional unit tests generally *pass* (green) or *fail* (red). But in BDD tests, several extra outcomes are possible. For example, scenarios can also be *pending* (when they haven't been started yet, or haven't yet been completely implemented).

When you run a scenario, each step will have its own outcome. The overall result of the scenario will depend on the outcomes of the different steps. If all of the steps succeed, then the scenario will succeed. If one of the steps fails, then the scenario will fail. And if any of the steps are incomplete or undefined, then the scenario as a whole will be reported as pending. Furthermore, when a step fails or is pending, there's no point running the subsequent steps because the test outcome is already compromised. In this case, any steps following the failed or pending step are marked as *skipped*.

The outcome of each step depends on a number of factors (see figure 6.4), including whether the step succeeded or not and the outcomes of previous steps.

If a BDD tool finds no matching definition for a step in a scenario, it will report this step as pending. It will also write a skeleton step definition to the console, which you can use as a starting point for your own step definition.

Any step that's implemented and doesn't fail in some way is considered successful. Because of this, if you have only partially implemented a step, it may be reported as successful even if the corresponding application code isn't complete. To avoid this, you sometimes need to mark a step as *work-in-progress*. Each tool provides its own way of doing this. In Ruby, for example, you can mark a partially implemented test as pending like this:

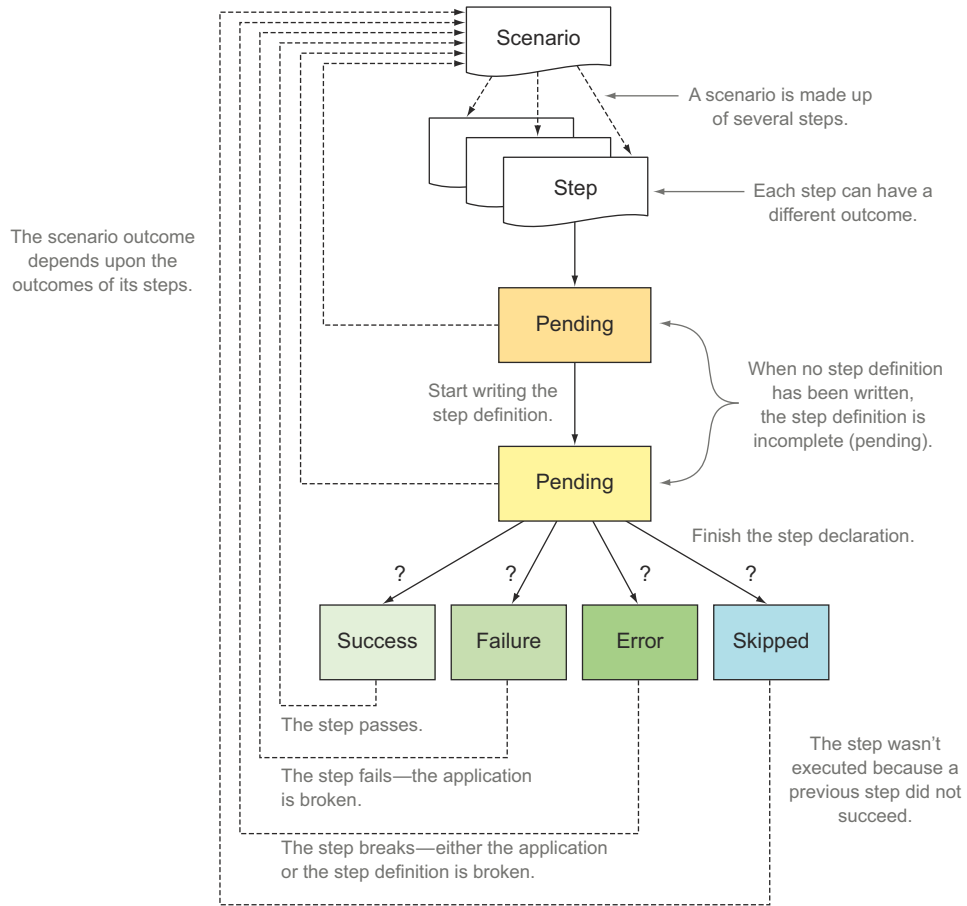


Figure 6.4 Individual steps can have several possible outcomes.

```
When /^Joe transfers (\d+) points to Jill$/ do |pointCount|
  pending
end
```

In Cucumber-JVM you throw a `PendingException`:

```
@Given("I am a (.*?) Frequent Flyer member")
public void useAMemberWithAGivenStatus(String status) {
    throw new PendingException();
}
```

Before we look at how you can implement the step definitions in various BDD tools like JBehave, Cucumber, and SpecFlow, we need to discuss another tool that we'll use at various stages: *Thucydides*.

6.3 **Implementing BDD more effectively with Thucydides**

Thucydides (<http://thucydides.info>) is an open source library that adds better-integrated and more-comprehensive reporting capabilities to conventional BDD tools such as JBehave and Cucumber.

You’ve already briefly seen Thucydides in action in chapter 2, where it helped generate the test reports. When you use it with a fully supported BDD tool, it helps you break down scenario implementations into smaller reusable steps and report on these steps in the living documentation. At the time of writing, Thucydides works well with JVM-based testing libraries such as JBehave and JUnit, but you can also import test results generated by other BDD tools such as Cucumber-JVM,⁴ SpecFlow, Cucumber-JS, or Behave.

For the fully integrated tools, Thucydides also makes writing automated acceptance tests with WebDriver, a popular open source browser automation library, easier and more productive.

But the specialty of Thucydides is taking the test results produced by BDD tools like JBehave and turning them into rich, well-integrated living documentation, such as by relating the test outcomes to the requirements they test. This gives stakeholders a clear picture of not only what tests have been executed, but what requirements have been tested, and how well they’ve been tested.

Even if you’re not using one of the fully supported BDD tools, you can still benefit from many of the reporting and requirements integration features, and use Thucydides as a generator of living documentation. We’ll look at Thucydides in much more detail when we discuss Living Documentation in chapter 11.

In the remainder of this chapter, we’ll discuss specific BDD tools so you can get started using a BDD tool in your environment and trying out these techniques for yourself. The first tool we’ll look at is JBehave.

6.4 **Automating scenarios in Java with JBehave**

JBehave (<http://jbehave.org>) is a popular Java-based BDD framework that was originally written by Dan North. In JBehave, you write step definition methods in Java or in other JVM languages such as Groovy or Scala.

We’ll look at how to use JBehave in conjunction with Thucydides, as Thucydides supports a number of conventions that make setting up and configuring a JBehave project simpler.

6.4.1 **Installing and setting up JBehave**

The easiest way to build and run a JBehave/Thucydides test suite is to use Maven. Maven (<http://maven.apache.org>) is a widely used build tool in the Java world. Maven provides a number of features that simplify and standardize the build project for Java-based projects, including a standardized directory structure and build lifecycle, and powerful dependency management capabilities.

⁴ At the time of writing, full Cucumber integration was being actively developed.

You can incorporate JBehave and Thucydides into an existing Maven project⁵ by adding the corresponding dependencies to the Maven pom.xml file:

```
<dependency>
  <groupId>net.thucydides</groupId>
  <artifactId>thucydides-core</artifactId>
  <version>0.9.239</version>
</dependency>
<dependency>
  <groupId>net.thucydides</groupId>
  <artifactId>thucydides-jbehave-plugin</artifactId>
  <version>0.9.236</version>
</dependency>
```

By convention, Thucydides relies on a few simple directory conventions, illustrated in figure 6.5. In particular, Thucydides will configure JBehave to look for feature

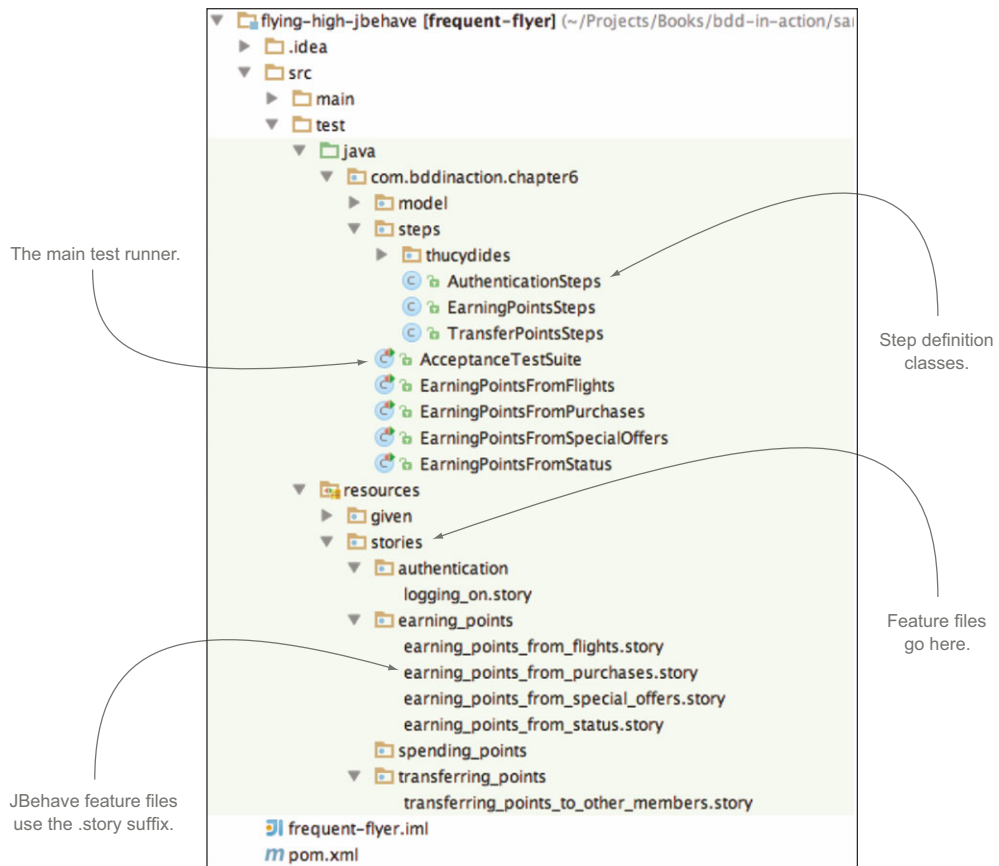


Figure 6.5 The default project directory structure for a JBehave project using Thucydides

⁵ For a detailed description of how to create a brand new Thucydides/Maven project, refer to http://thucydides.info/docs/thucydides-one-page/thucydides.html#_getting_started_with_thucydides.

files in the `src/test/resources` directory of your project (or in subdirectories of this directory).

The `AcceptanceTestSuite` class is a test runner that will execute all of the scenarios in the feature files. This is simply an empty class that extends the `ThucydidesJUnitStories` class:

```
public class AcceptanceTestSuite extends ThucydidesJUnitStories {}
```

Alternatively, some teams prefer to write a separate test runner for each story. This makes it easier to run individual stories from within the IDE and to run the stories in parallel. By default, Thucydides uses the name of the class to know what story to run. For example, the following class will run the `earning_points_from_flights.story` story:

```
public class EarningPointsFromFlights extends ThucydidesJUnitStory {}
```

Step definition classes can be placed in the same directory (or package, in Java terminology) as the test runner, or in any package underneath this package.

6.4.2 JBehave step definitions

Like many Java-based tools, JBehave uses annotations to bind the text in the feature files to the Java methods that implement them. Suppose you wanted to automate the following step in one of your scenarios:

Given I am a Gold Frequent Flyer member

You could write a definition for this step like this:

Step definitions are marked by an annotation.

```
2 FrequentFlyerMember member;

public class EarningPointsSteps {
    @Given("I am a $status Frequent Flyer member")
    public void defineAMemberWithStatus(String status) {
        member = Members.getMember().withStatus(status);
    }
}
```

1 You'll need to use the member in later steps.

Step definitions can go in any class.

Here you interact with the application, by manipulating the UI or via API calls, to perform the corresponding application logic.

3 The method name has no importance.

JBehave step definitions are just annotated Java⁶ methods that live in ordinary Java classes 1. JBehave uses an `@Given`, `@When`, or `@Then` annotation 2 to match a step definition method with a line in the feature file. The method name itself 3 isn't important.

Unlike some of the other tools, `Given`, `When`, and `Then` steps can't be used interchangeably; JBehave will treat them as different steps. For example, "When I fly from Sydney to Melbourne" (an action under test) will run a different step definition than "Given I fly from Sydney to Melbourne" (a precondition).

JBehave lets you pass parameters to your step definitions by putting placeholder variables at the appropriate spots in the annotation text. In the preceding step

⁶ Or Groovy or Scala, depending on the language you choose.

definition, you pass the Frequent Flyer status parameter to the step definition. The actual name of the variable you use makes no difference; JBehave will pass any parameter values it finds to the annotated method in their order of appearance.

Steps often contain more than one piece of useful information, and you can pass any number of parameters into a step definition. For example, let's revisit the following step, which sets up the database with a flight for your test:

```
Given the flying distance between Sydney and Melbourne is 878 km
```

Here, there are three key fields you might want to pass to your step definition: the departure city, the destination city, and the distance. You could do this with a step definition like the following:

The step definition text, complete with variables

1

```
@Given("the flying distance between $departure and $destination is
      $distance km")
public void defineTheFlyingDistanceForATrip(String departure,
      String destination,
      int distance) {
    // TODO: Set up the test database with this trip
}
```

The step definition method with matching parameters

2

Here JBehave isolates the three fields you need and passes them to the step method. As mentioned earlier, the parameters extracted from the annotation 1 are passed in to the method 2 in their natural order; the parameter names are not important.

Notice how JBehave even converts the type for you, so you get the distance field directly in the form of an integer. JBehave will automatically convert numerical values, and with a little extra configuration, you can set it up to convert many other parameter types as well.

6.4.3 Sharing data between steps

When implementing step definitions, you'll frequently need to share data across multiple steps. For example, you might fetch a member account in one step, and then use this account in a subsequent step. You can do this with member variables, but this requires all of the step implementations to be in the same class or inherit from a common base class containing the shared member variables.

If you're using JBehave with Thucydides, a better alternative is to use the Thucydides session. Thucydides maintains a hash map for the life of a scenario that you can use to share data between steps, no matter what class they're implemented in. For example, you might retrieve a Frequent Flyer member from the database in a step like this:

```
@Given("a frequent flyer member called $name")
public void givenAFrequentFlyerMember(String name) {
    Member member = Members.findByName(name);
    Thucydides.getCurrentSession().put("member", member);
}
```

Fetch this member from the database.

Store it for later reference.

You could then reuse this data in subsequent steps, like this:

```
@When("$name books a flight")
public void booksFlight(String name) {
    Member member = Thucydides.getCurrentSession().get("member")
    ...
}
```

← Reuse the member you stored earlier.

6.4.4 Passing tables to steps

As you've seen, JBehave provides the `ExamplesTable` class to handle embedded tables in your scenarios. In JBehave, these parameters are known as *tabular parameters*. Suppose you need to automate the following scenario step:

Given I have travelled on the following flights:

flight	from	to	date
FH-603	Sydney	Cairns	01-05-2012
FH-604	Cairns	Sydney	05-05-2012
FH-603	Sydney	Melbourne	01-07-2012
FH-604	Melbourne	Sydney	02-07-2012
FH-603	Sydney	Brisbane	28-07-2012
FH-604	Brisbane	Sydney	02-08-2012

You could implement a step in JBehave using an `ExamplesTable` parameter like this:

```
SimpleDateFormat formatter = new SimpleDateFormat("dd-MM-yyyy");

@Given("I have travelled on the following flights: $flights")
public void travelled_on_flights(ExamplesTable flights)
    throws ParseException {
    for(Map<String,String> flightDetails : flights.getRows()) {
        Flight flight = Flight.number(flightDetails.get("flight"))
            .from(flightDetails.get("from"))
            .to(flightDetails.get("to"));

        Date date = formatter.parse(flightDetails.get("date"));
        member.flewOnFlight(flight).on(date);
    }
}
```

A date formatter used to parse the date column.

Pass the table as an `ExamplesTable`.

Iterate over the rows in the table.

Extract field values from the row.

Update the test database with the new data.

This is a very basic use of the `ExamplesTable` class, but it does reflect typical usage of tabular parameters.

6.4.5 Step definitions for tables of examples

Another very powerful use of tables, which you saw in section 5.3.2, is to summarize a number of similar scenarios into a single scenario with a table of examples, as shown here:

```
Scenario: Earning extra points on flights by Frequent Flyer status
    Given I am a <status> Frequent Flyer member
    When I fly on a flight that is worth <base> base points
    Then I should earn a status bonus of <bonus>
    And I should have guaranteed minimum earned points per trip of <minimum>
    And I should earn <total> points in all
```

Examples:

status	base	bonus	minimum	total	notes
Standard	439	0	0	439	
Silver	439	220	500	659	50% bonus
Silver	148	74	500	500	minimum points
Gold	474	356	1000	1000	minimum points
Gold	2041	1531	1000	3572	75% bonus

When JBehave runs this scenario, it'll replace the placeholder variables (<status>, <base>, and so on) with values from the example table and call the corresponding step definitions. When you write a step definition method for these steps, JBehave practitioners often quote the step text exactly as it appears in the scenario, including the angle brackets:

```
@Given("I am a <status> Frequent Flyer member")
public void defineAMemberWithStatus(String status) {
    member = Members.getMember().withStatus(status);
}
```

This makes it more obvious that the data is coming from a table of examples, and it was required in older versions of JBehave for proper reporting. Current versions⁷ of JBehave are more flexible on this point, and will match step definitions that use the conventional \$ notation as well:

```
@Given("I am a $status Frequent Flyer member")
public void defineAMemberWithStatus(String status) {
    member = Members.getMember().withStatus(status);
}
```

Using the \$ notation also makes it easier to reuse step definitions in both example-based and standard scenarios.

6.4.6 Pattern variants

Sometimes it's useful to have several similar expressions in different scenarios match the same step definition. For example, you might want to be able to write both “When I fly from Sydney to Melbourne” and “When I travel from Sydney to Melbourne.”

In JBehave, you can use the @Alias annotation, as shown here:

```
@When("I fly from $departure to $destination")
@Alias("I travel from $departure to $destination")
public void whenIFlyFrom(String departure,
    String destination) {
    // TODO: add this flight to the member's account
}
```

Aliases are also useful when you need to reuse the same step definition for an example-based scenario and a normal scenario, because the step definition syntax is slightly different:

⁷ From JBehave 3.8 onwards.

```
@When("I fly from $departure to $destination")
@Alias("I travel from <departure> to <destination>")
public void whenIFlyFrom(String departure,
                        String destination) {
    // TODO: add this flight to the member's account
}
```

If the variations are very close, you can also use the more compact format shown here:

```
@When("I {fly|travel} from $departure to $destination on $date at $time")
public void whenIFlyFrom(String departure,
                        String destination,
                        DateTime date,
                        LocalTime time) {
    // TODO: add this flight to the member's account
}
```

These approaches give more flexibility in writing the scenarios, and streamline implementing the step definitions, because there's no need to write duplicate methods or normalize the scenario texts unnecessarily.

6.4.7 Failures and errors in the scenario outcomes

As with all BDD tools, JBehave reports on successful, failing, and pending steps. But when used with Thucydides, JBehave also distinguishes between failures (the application doesn't produce the expected outcome) and errors (the application, or possibly the scenario itself, throws an unexpected exception). A failure generally indicates an application bug, whereas an error may indicate a scenario that's broken or that no longer reflects the correct behavior of the application. For example, you saw this scenario earlier:

```
Scenario: Earning standard points from an Economy flight
Given the flying distance between Sydney and Melbourne is 878 km
And I am a standard Frequent Flyer member
When I fly from Sydney to Melbourne on 01/01/2013 at 09:00
Then I should earn 439 points
```

You could implement the last step of this scenario using the following step definition:

```
@Then("I should earn $points points")
public void thenIShouldEarn(int points) {
    assertThat(points).isEqualTo(earnedPoints);
}
```

Did you earn the right number of points?

1 The earnedPoints variable is assigned in a previous step.

Here you're checking that the calculated number of points is what you expected ❶. If the number of points earned isn't 439, this step will fail, which will cause the scenario as a whole to fail.

But failures are not always this simple. In figure 6.6, the third step has failed with the error message, "Frequent Flyer account not activated." This isn't an expected exception, and it could come either from an application bug or incorrect test logic.



Figure 6.6 Each step in a scenario has its own outcome.

For example, perhaps the “And I am a standard Frequent Flyer member” line failed to create a correctly configured account. When you’re writing and maintaining automated acceptance criteria, it’s important to be able to distinguish between these two cases.

To make this sort of issue easier to spot and troubleshoot, Thucydides flags any exception that’s not an `AssertionError`⁸ as an *error* and not as a *failure*.

6.5 Automating scenarios in Java using Cucumber-JVM

The next tool we’ll look at is Cucumber (<http://cukes.info>). Cucumber is a very popular BDD tool from the Ruby world. Using Cucumber in Ruby is well documented on the Cucumber website and in *The Cucumber Book*,⁹ so in this section we’ll focus on writing scenarios using Cucumber-JVM. Cucumber-JVM is a more recent Java implementation of Cucumber, which allows you to write step definitions in Java and other JVM languages. Teams familiar with the Ruby version of Cucumber may be more comfortable with the Cucumber flavor of “Given ... When ... Then” scenarios than with the JBehave variation. But at the time of writing, Thucydides doesn’t yet provide full Cucumber integration.

⁸ `AssertionErrors` are caused by logical tests such as the one in the “I should earn \$points points” step definition.

⁹ Matt Wynne and Aslak Hellesøy, *The Cucumber Book* (Pragmatic Bookshelf, 2012).

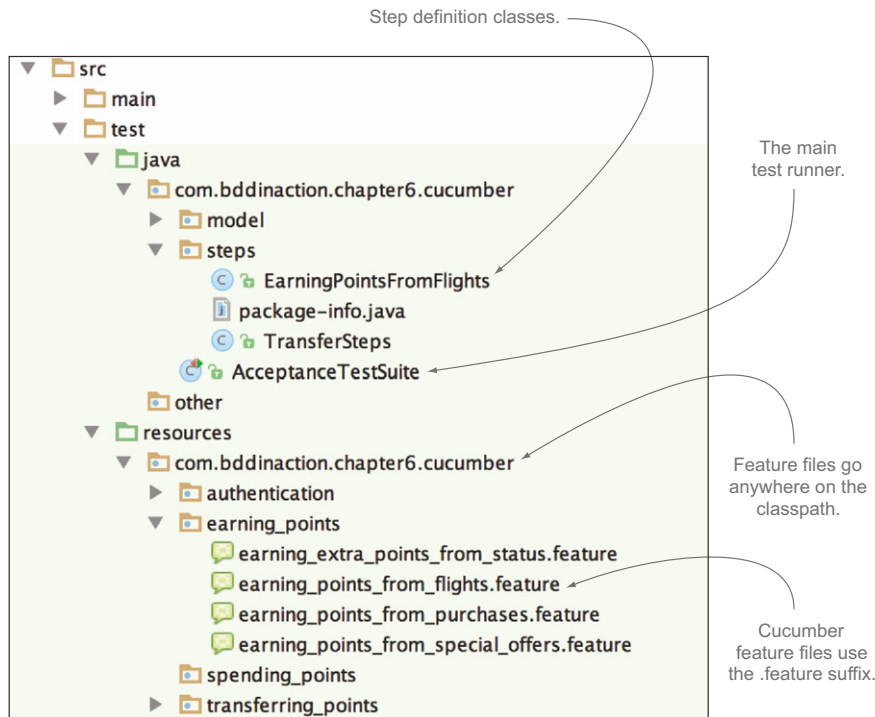


Figure 6.7 The Cucumber-JVM project directory structure

6.5.1 Cucumber-JVM project setup and structure

Setting up Cucumber-JVM in a Maven project is just a matter of adding the Cucumber dependencies to the Maven pom.xml file.¹⁰ Cucumber then runs on top of JUnit, like any other unit test. You can find a full Cucumber-JVM sample project in the sample code for this chapter.

The Cucumber project structure for Java projects is similar to the structure for JBehave/Thucydides and is illustrated in figure 6.7. The main difference relates to the feature files: in Cucumber, they use the .feature suffix instead of .story, and they go in a package directory structure in src/test/resources matching the one found in src/test/java, rather than directly underneath the src/test/resources directory. In figure 6.7, the Cucumber test classes are in the com.bddinaction.chapter6.cucumber package in src/test/java, so the Cucumber feature files need to be in the com/bddinaction/chapter6/cucumber subdirectory of the src/test/resources directory.

The features are executed by an annotated test runner class like this one:

```
@RunWith(Cucumber.class)
@Cucumber.Options(format = {"html:target/cucumber-html-report"})
public class AcceptanceTestSuite {}
```

¹⁰ You can find an example of the exact dependencies in the sample code for this chapter.

As with JBehave, Cucumber expects the step definition classes to be in a package below the test runner class.

6.5.2 Cucumber-JVM step definitions

Let's see how you could use Cucumber-JVM to automate the following scenario:

```
Given the flying distance between Sydney and Melbourne is 878 km
And I am a Gold Frequent Flyer member
When I fly from Sydney to Melbourne
Then I should earn 878 travel points
```

Cucumber-JVM step definitions are annotated Java methods, similar to the ones used by JBehave:¹¹

```
import cucumber.api.java.en.Given;
...
@Given("I am a Gold Frequent Flyer member")
public void useAMemberWithGoldStatus() {
    // TODO: use a member with a Gold status
}
```

Note that the Cucumber annotations come from a different package than the JBehave ones.

Unlike JBehave, Cucumber doesn't mind which annotation you use: @Given, @When, and @Then are treated as synonyms.

As you've seen, the Gherkin language used by Cucumber relies on regular expressions to identify variables in the steps. For example, if you want to pass the status to the step definition just discussed, you could write something like this:

The **(*)** regular expression will be passed in to the status parameter.

```
@Given("^I am a (.*?) Frequent Flyer member$")
public void useAMemberWithAGivenStatus(String status) {
    // TODO: use a member with a Gold status
}
```

Regular expressions appear in parentheses. The **^** symbol indicates the start of a line, and the **\$** symbol indicates the end of a line.

Cucumber will match the regular expressions in parentheses and assign them to the method parameters. In the following step definition, you pass in three parameters:

Here you have three regular expressions.

```
@When("^I fly from (.*?) to (.*?) on (.*?)$")
public void I_fly_from(String departure,
    String destination,
    Date date){
    // TODO
}
```

Matches the first **(*)**.

Matches the third **(*)**.

Matches the second **(*)**.

Cucumber will convert many parameter types for you. In this step definition, for example, the third parameter will be converted to a Date.

6.5.3 Pattern variants

The use of regular expressions adds a great deal of power to step definitions, and it's easy to define several variations of the same step. But there are a few tricks to know.

¹¹ In fact, the only difference here is that the @Given annotation comes from a different package.

The following scenario, for example, won't work, because the (fly|travel) expression will be considered a regular expression, and Cucumber will try to pass it to one of the parameters. As a result, there will be too many parameters, and the step definition will break:

```
@When("^I (fly|travel) from (.*?) to (.*?) on (.*?)$")
public void I_fly_from(String departure, String destination, Date date) {
    // TODO
}
```

The first regular expression will match either "fly" or "travel".

Now there aren't enough parameters for all the regular expressions.

To get around this, you need to tell Cucumber that the first regular expression shouldn't be passed as a parameter. In regular expression terms, this is known as a *non-capturing* expression, and you can write it as follows:

```
@When("^I (?:fly|travel) from (.*?) to (.*?) on (.*?)$")
public void I_fly_from(String departure, String destination, Date date) {
    // TODO
}
```

Now the first regular expression will be matched but not captured.

This will allow both variations, but will only pass the parameters you want to the step definition method.

6.5.4 Passing tables to steps

In Cucumber, you can use the DataTable class to pass tabular data to a step definition, which provides a rich API to extract, manipulate, and compare values in the table. But Cucumber also provides powerful built-in converters that can convert an embedded table to a list of domain objects.

Suppose you're writing Cucumber step definitions for the following scenario:

```
Scenario: Transfer points between existing members
Given the following accounts:
    | owner | points | statusPoints |
    | Jill  | 100000 | 800          |
    | Joe   | 50000  | 50           |
When Joe transfers 40000 points to Jill
Then the accounts should be the following:
    | owner | points | statusPoints |
    | Jill  | 140000 | 800          |
    | John  | 10000  | 50           |
```

Now suppose you have an Account class that represents these fields:

```
public class Account {
    private final String owner;
    private final int points;
    private final int statusPoints;

    public Account(String owner, int points, int statusPoints) {
        this.owner = owner;
        this.points = points;
        this.statusPoints = statusPoints;
    }
}
```

You have a field for each table column.

This class represents rows in the scenario tables.

Cucumber knows how to instantiate an Account object for each row of the table using the constructor parameters.

```

    public String getOwner() { return owner; }
    public int getPoints() { return points; }
    public int getStatusPoints() { return statusPoints; }
}

```

The table is read-only, so you only need getters, not setters, to access the field values.

Cucumber lets you pass a list of Accounts directly to the step definition, as shown here:

```

@Given("^the following accounts:$")
public void the_following_accounts(List<Account> accounts) {
    InTestDatabase.createAccounts(accounts);
}

```

This will automatically use the table header values to create a list of matching Account objects.

6.5.5 Step definitions for tables of examples

Step definitions for example tables are easy to implement in Cucumber-JVM. In fact, because the step definitions are based on regular expressions, there's no difference between a step definition written for a normal scenario step and one written for an example-based step.

Suppose you're automating the following scenario in Cucumber-JVM:

```

Scenario Outline: Earning extra points on flights by Frequent Flyer status
  Given I am a <status> Frequent Flyer member
  When I fly on a flight that is worth <base> base points
  Then I should earn a status bonus of <bonus>
  And I should have guaranteed minimum earned points per trip of <minimum>
  And I should earn <total> points in all

```

Examples:

status	base	bonus	minimum	total	notes
Standard	439	0	0	439	
Silver	439	220	500	659	50% bonus
Silver	148	111	500	500	minimum points
Gold	474	400	1000	1000	minimum points
Gold	2041	1531	1000	3572	75% bonus

You could use the following step implementation to match the first line of this scenario:

```

@Given("I am a (.*) Frequent Flyer member")
public void useAMemberWithAGivenStatus(String status) {
    member = members.getMember().withStatus(status);
}

```

The regular expression will match steps in a table or in a conventional scenario.

This makes it easy to reuse step definitions between table-based and conventional scenarios.

6.5.6 Sharing data between steps

Cucumber provides a simple and convenient way to share data between steps using dependency injection. You can create a helper class to store information you want to share between steps, and then inject it into the constructor of each step definition class that uses this information.

For example, suppose you want to store the Frequent Flyer member details between steps. You could do this as follows:

```
public class FrequentFlyerHelper {
    private FrequentFlyer frequentFlyer;
    public void setFrequentFlyer(FrequentFlyer frequentFlyer) {...}
    public FrequentFlyer getFrequentFlyer() {...}
}
```

You could then inject this class into your step definitions, as shown here:

```
public class TransferSteps {
    private final FrequentFlyerHelper frequentFlyerHelper;

    public TransferSteps(FrequentFlyerHelper frequentFlyerHelper) {
        this.frequentFlyerHelper = frequentFlyerHelper;
    }

    @Given("^(.*) is a Frequent Flyer member$")
    public void a_Frequent_Flyer_member(String name) {
        FrequentFlyer member = FrequentFlyer.called(name);
        frequentFlyerHelper.setFrequentFlyer(member);
    }
}
```

Cucumber will inject the helper class.

You can then use this class to share data across steps.

6.5.7 Pending steps and step outcomes

In Cucumber, the recommended way to say that a step implementation is still a work-in-progress is to throw a `PendingException`:

```
import cucumber.api.PendingException;
...
@Given("^the (flying|travelling) distance between (.*?) and (.*?) is (\\d+) km$")
public void define_flying_distance(String flightMode,
                                   String departure,
                                   String destination,
                                   int distance) {
    throw new PendingException("Not finished yet");
}
```

This will cause this step and the scenario as a whole to be flagged as pending.

Cucumber-JVM can do a great deal more than what we've touched on here. But this should give you enough to get you started with Cucumber-JVM, or to understand a set of existing automated acceptance criteria implemented this way. You can learn more about Cucumber-JVM at <http://cukes.info/>.

6.6 Automating scenarios in Python with Behave

Python is a popular, general-purpose, open source dynamic language, and there are several Gherkin-based BDD tools available for the Python language. Cucumber itself can be used to run scenarios written in Python, though it uses a Python interpreter embedded inside a Ruby process, which can be brittle. For those who prefer

a pure Python solution, there are currently three tools available: Lettuce (<http://pythonhosted.org/lettuce>), Freshen (<https://github.com/rilisagor/freshen>), and Behave (<http://pythonhosted.org/behave>).

These tools are similar, but at the time of writing, Behave is the most stable, best documented, and most feature-rich of the three. Despite the name, Behave has no relationship with JBehave. There's also a Thucydides plugin that allows you to generate Thucydides reports from the Behave test results.

Let's take a closer look at BDD in Python using Behave.

6.6.1 Installing Behave

Unlike the JVM BDD tools we've looked at so far, Behave is primarily a command-line tool. Installing Behave is straightforward and can be done using Pip, the standard Python package installation tool. Pip is the Python equivalent of Gem for Ruby, NuGet for .NET, or npm for Node.js. You use it like this:

```
$ pip install behave
```

This will install the behave command-line tool and make the Behave package available to your Python scripts. When this is done, you should be able to run Behave from the command line:

```
$ behave --version
behave 1.2.3
```

6.6.2 The Behave project structure

Behave projects use a simple directory structure, illustrated in figure 6.8. Feature files go in a directory called features, and the step definitions go in a subdirectory of this directory called steps.

6.6.3 Behave step definitions

Behave uses the Gherkin format for its scenarios. In fact, Behave is essentially a Python port of Cucumber, so the feature files are very similar to those used by Cucumber.

Step definitions are written in Python, using `@given`, `@when`, and `@then` decorators, as shown here:

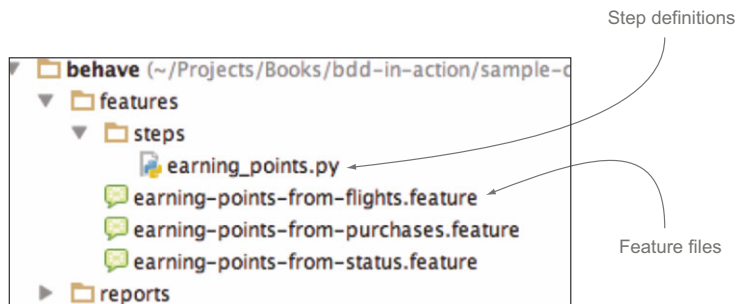


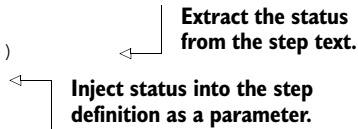
Figure 6.8 The Behave project directory structure

```
from behave import given, when, then

@given('I am a Gold Frequent Flyer member')
def step_impl(context):
    assert True
```

Behave uses a different notation when passing parameters in to step definitions. Although Behave can be configured to use regular expressions, by default it uses named variables, a little like JBehave:

```
@given('I am a {status} Frequent Flyer member')
def step_impl(context, status):
    assert True
```



Extract the status from the step text.

Inject status into the step definition as a parameter.

Naturally, you can pass in more than one parameter:

```
@given('the flying distance between {departure} and {destination} is {distance} km')
def step_impl(context, departure, destination, distance):
    assert True
```

Behave will automatically convert basic parameter types such as numbers, percentages, dates, and times. You can also, with a little extra effort, configure automatic type conversions for the parameters that you pass in to the step definitions.

6.6.4 Combining steps

It's often useful to be able to combine several steps into a single step in order to avoid repetition and to simplify the scenarios. For example, suppose you already implemented the following two scenario steps:

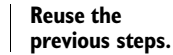
```
Given Joe is a Frequent Flyer member
Given Joe logs in
```

In some scenarios, it would be useful to combine these two steps into a single one:

```
Given a Frequent Flyer member Joe has logged in
```

You could implement this step in Behave by reusing the previous two step definitions as shown here:

```
@given('a Frequent Flyer member {name} has logged in')
def step_impl(context, name):
    context.execute_steps(u"""
        Given {name} is a Frequent Flyer member
        And {name} logs in
    """)
```



Reuse the previous steps.

This is a useful trick, as it encourages step reuse, limits duplication, and makes it easier to write shorter, more concise scenarios.

6.6.5 Step definitions using embedded tables

Like most of the other tools discussed here, Behave lets you pass tabular parameters to your step definitions. Suppose you're implementing the following scenario using Behave:

Feature: Transfer points to other members

@transfers

Scenario: Transfer points between existing members

Given the following accounts:

owner	points	statusPoints
Jill	100000	800
Joe	50000	50

When Joe transfers 40000 points to Jill

Then the accounts should be the following:

owner	points	statusPoints
Jill	140000	800
John	10000	50

Behave will pass the tabular data to each step in the context object, as illustrated here:

```
@given('the following accounts')
def step_impl(context):
    for row in context.table:
        owner = row["owner"]
        points = row["points"]
        statusPoints = row["statusPoints"]
```

Tabular data is passed in
the `context.table` variable.

Extract values
from each row.

6.6.6 Step definitions for tables of examples

Like Cucumber, Behave makes example tables simple: steps for example tables are no different from any other kind of step.

6.6.7 Running scenarios in Behave

To run your features, just call `behave` from the command line. There are quite a few command-line options (run `behave --help` to see them all), but one of the most useful is the `--tags` option, which lets you limit the scenarios you execute to those with a given tag or set of tags. For example, the following scenario is annotated with the `@transfers` tag:

```
@transfers
Scenario: Transfer points between existing members
...
```

You could execute this scenario, and any other scenarios with this tag, by using the `--tags` option:

```
$ behave --tags=transfers
```

Another very useful option is the `--junit` option, which generates JUnit-compatible test reports in the `reports` directory. This makes it easy to incorporate Behave tests into your build process using a continuous integration server, and also allows for more sophisticated reporting with Thucydides.

Again, we've just scratched the surface of what you can do with Behave. For more information, take a look at the Behave website (<http://pythonhosted.org/behave/>).

6.7 Automating scenarios in .NET with SpecFlow

If you're in a .NET environment, your best option for BDD is SpecFlow (<http://specflow.org>). SpecFlow is an open source Visual Studio extension that provides support for Gherkin scenarios in the .NET and Windows development ecosystem. The rest of this section will assume that you're reasonably familiar with the Visual Studio development environment.

6.7.1 Setting up SpecFlow

SpecFlow is a Visual Studio extension, so you can install it directly from the Visual Studio Gallery (see figure 6.9). Once it's installed, you'll need to create a new Unit Test project in Visual Studio and add the SpecFlow package to your project using NuGet (the .NET package manager found at <http://www.nuget.org/>).

SpecFlow works with several .NET unit testing frameworks, including NUnit, xUnit, and a more specialized commercial tool called SpecRun. In this example, we'll be using NUnit.

You can install SpecFlow with NUnit through the Manage NuGet Packages screen or directly from the NuGet Package Management Console using the following command:

```
PM> Install-Package SpecFlow.NUnit
```

6.7.2 Adding feature files

SpecFlow uses Gherkin feature files like the ones used by Cucumber and Behave. Once SpecFlow is installed, you can add a new item to your project and choose SpecFlow Feature File. This will open a new feature file in the SpecFlow editor (see figure 6.10).

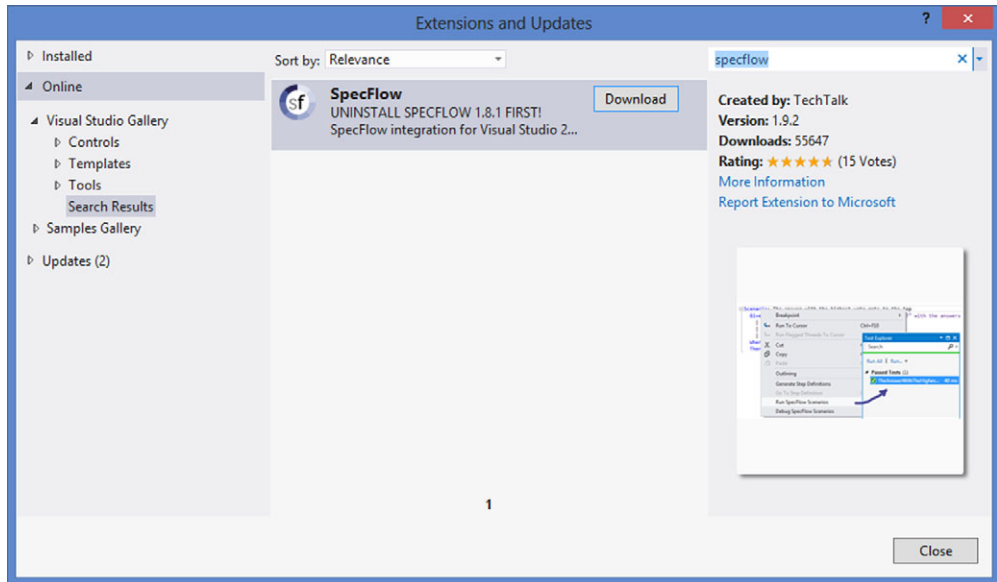


Figure 6.9 You can install SpecFlow in the Visual Studio Gallery.

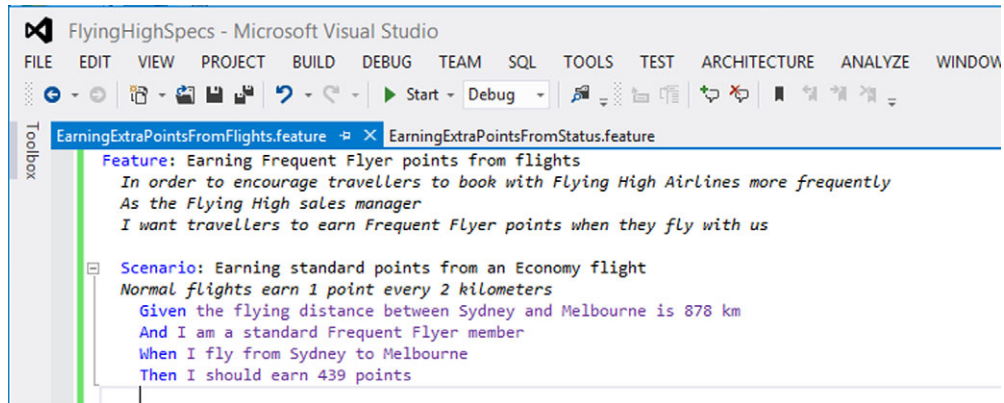


Figure 6.10 Editing a SpecFlow feature file

One of the areas where SpecFlow shines is its tight integration with Visual Studio. In addition to syntax highlighting and auto-complete, this editor has a number of nice integrated features. In particular, you can run or debug the scenarios directly from within the editor, or even perform code-coverage analysis, just like an ordinary unit test. Another handy feature allows you to generate step-definition skeleton methods by right-clicking in the feature file (see figure 6.11).

6.7.3 Running scenarios

The scenario execution will depend on the unit test provider that SpecFlow uses. You can configure this in the app.config file of your test project. If you configure the MsTest provider as shown next, you'll be able to run scenarios from within the Visual Studio Test Explorer window:

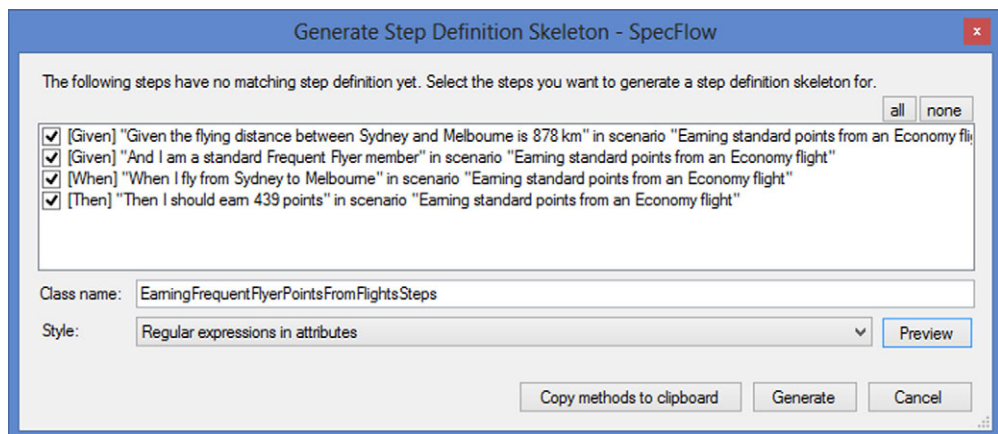


Figure 6.11 SpecFlow lets you generate step definitions from within the feature file editor.

```
<specFlow>
  <unitTestProvider name="MsTest" />
  ...
</specFlow>
```

Alternatively, you can run the tests using NUnit, in which case you'll be running the scenarios via the NUnit test runner:

```
<specFlow>
  <unitTestProvider name="NUnit" />
  ...
</specFlow>
```

6.7.4 SpecFlow step definitions

SpecFlow step definitions are implemented as .NET methods annotated with the [Given], [When], or [Then] attributes. SpecFlow expects these methods to be public, and to be placed in a public class that's marked with the [Binding] attribute.

SpecFlow step definitions use regular expressions in a similar way to the Cucumber-JVM methods you saw in section 6.3. For example, one of the steps in the Flying High scenarios looks like this:

Given the flying distance between Sydney and Melbourne is 878 km

The step definition for this step might look like this:

```
[Given(@"the flying distance between (.*?) and (.*?) is (.*?) km")]
public void GivenTheFlyingDistance(string departure,
                                   string destination,
                                   int distance)
{
    ...
}
```

◀ '@' symbol
disables
escape
sequences
1

In SpecFlow, the step definition texts are typically prefixed by an @ symbol ❶: this disables escape sequences starting with a backslash (\), making it easier to write some of the more complex Cucumber regular expressions.

A step definition method can also have several attributes, so that the same method can be used for several different variations in the scenario texts:

```
[When(@"I go from (.*?) to (.*?)")]
[When(@"I travel from (.*?) to (.*?)")]
public void GivenTheFlyingDistance(string departure, string destination)
{
    ...
}
```

If you aren't fond of regular expressions, SpecFlow also supports alternative step-definition matching rules based on the method name. You name the step definition method after the step text you want to match, using either underscores or camel case. You can also include parameter placeholders in uppercase. The parameter placeholders

will be matched with the corresponding method parameters. For example, you could use this approach for the following step:

Given I am a <status> Frequent Flyer member

Using this convention, the corresponding step definition could be written as follows:

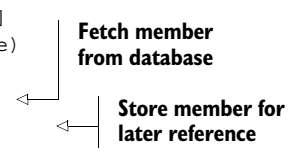
```
[Given]
public void I_am_a_STATUS_Frequent_Flyer_member(string status)
{
    ...
}
```

6.7.5 Sharing data between steps

As you saw previously, it's often useful to be able to share data between steps during a scenario execution, even if the step definitions aren't in the same class. The SpecFlow scenario context provides a clean and convenient API that lets you do just that.

For example, you could store a Frequent Flyer member that you obtained from the database like this:

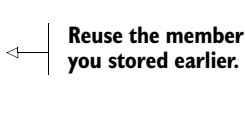
```
[Given(@"a frequent flyer member called {name}")]
public void GivenAFrequentFlyerMember(string name)
{
    var member = MemberHelper.FindByName(name);
    ScenarioContext.Current[name] = member;
}
```



The diagram shows two annotations with arrows pointing to the code. The first annotation, "Fetch member from database", has an arrow pointing to the `MemberHelper.FindByName(name)` line. The second annotation, "Store member for later reference", has an arrow pointing to the `ScenarioContext.Current[name] = member;` line.

This field is now available for future use in subsequent steps executed during this scenario. Here, for example, you fetch it in order to let the member make a flight booking:

```
[When(@"{name} books a flight")]
public void BooksAFlight(string name)
{
    var member = ScenarioContext.Current[name];
    ...
}
```



The diagram shows an annotation "Reuse the member you stored earlier." with an arrow pointing to the `ScenarioContext.Current[name]` line in the `BooksAFlight` method.

6.7.6 Step definitions using example tables

If you need to pass a table of data to a step method, you can add a parameter of type `Table` to the step definition method. This class encapsulates the contents of the tabular parameter with a convenient API that lets you both extract data and compare tables.

Suppose you're writing a step definition for the following step:

```
Given the following accounts:
  | owner | points | statusPoints |
  | Jill  | 100000 | 800          |
  | Joe   | 50000  | 50           |
```

You could do this in SpecFlow as follows:

```
[Given(@"the following accounts:")]
public void givenTheFollowingAccounts(Table accounts)
```



```

{
    foreach (var row in accounts.Rows)
    {
        var owner = row["owner"]
        var points = row["points"]
        var statusPoints = row["statusPoints "]
        ...
    }
}

```

You can also convert tabular parameters to domain objects or collections of domain objects. For example, you could convert the previous table into a list of Account objects like this:

```

[Given(@"the following accounts:")]
public void givenTheFollowingAccounts(Table accounts)
{
    var accounts = table.CreateSet<Account>();
}

```

Convert each row
of the table into
an Account object.

In a similar way, you could use the table in the Then step to compare expected results with actual results:

Then the accounts should be the following:

owner	points	statusPoints
Jill	140000	800
John	10000	50

You could implement this step in the following way:

```

[Then(@"the accounts should be the following ")]
public void givenTheFollowingAccounts(Table expectedAccounts)
{
    actualAccounts = AccountDatabaseHelper.GetRelevantAccounts();
    expectedAccounts.CompareToSet<Account>(actualAccounts);
}

```

Retrieve
the actual
accounts from
the database.

Compare the expected accounts
to what you actually obtained.

What you've seen here is just a small overview of some of SpecFlow's more interesting features. SpecFlow is a very complete Gherkin implementation that's tightly integrated into the .NET development environment, which makes it a logical choice for teams using BDD in a .NET context.

6.8 Automating scenarios in JavaScript with Cucumber-JS

Users are increasingly expecting richer and more interactive websites, and as a result, JavaScript is becoming increasingly important in modern web development. If you're writing rich client-side applications with complex business logic, it makes good sense to write BDD scenarios related to this business logic natively in JavaScript.

Unit testing is well supported in JavaScript, and low-level BDD unit-testing libraries like Jasmine and Mocha are widely used. We'll look at these libraries in chapter 10, when we discuss BDD-style unit and integration testing. But in this chapter we'll

focus on options available for higher-level BDD in JavaScript, using Gherkin-style scenario definitions.

There are a number of tools available to JavaScript developers that support Gherkin scenario descriptions, including Cucumber-JS (<https://github.com/cucumber/cucumber-js>), which is probably the best known of the JavaScript BDD libraries, and Yadda (<https://github.com/acuminous/yadda>), an alternative to Cucumber-JS that allows more flexibility in the scenario wording.

Let's take a closer look at what BDD in JavaScript using Cucumber-JS involves.

6.8.1 Setting up Cucumber-JS

Cucumber-JS is intuitive and easy to set up, and it's straightforward to use. Cucumber-JS relies on Node.js (a JavaScript application platform widely used in modern JavaScript development, found at <http://nodejs.org>) and npm (the Node.js package manager) to work, so you will need to install Node.js if you want to follow along with this example. Once it's installed, you can install Cucumber-JS by running the following:

```
$ npm install -g cucumber
```

This will install Cucumber-JS globally, so you can call it from any project. To check that it worked, run `cucumber-js -help` from the command line.

6.8.2 Writing feature files in Cucumber-JS

By default, Cucumber will expect the feature files and step definitions to be in specific directories, as illustrated in figure 6.12. Of course, you can override this structure if you want to, but for this example you'll go with the flow and use the defaults.

Much like Cucumber-JVM, Cucumber-JS expects the feature files to be in a directory called `features`. Cucumber-JS also understands the same Gherkin syntax that you've seen for the other Cucumber-derived BDD tools.

Imagine you're writing a client JavaScript application for the Frequent Flyer website that needs to be able to calculate the points earned for a given trip in real time. You might reuse the following scenario for your client-side business logic:

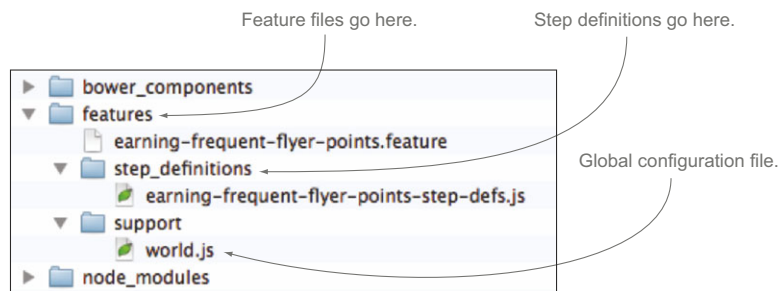


Figure 6.12 The default Cucumber-JS directory conventions

Feature: Calculating earned Frequent Flyer points

Scenario: Earning standard domestic Frequent Flyer points

Given the flying distance between Sydney and Melbourne is 878 km

And I am a standard Frequent Flyer member

When I fly from Sydney to Melbourne

Then I should earn 439 travel points

And I should earn 40 status points

The code might call a web service on the server or do the calculations locally. From the point of view of the requirements, this doesn't matter. What's important here is the application's behavior from the point of view of the user.

If you place this scenario in a feature file called `earning-frequent-flyer-points.feature` in the feature directory, Cucumber-JS will be able to find it. You can run Cucumber-JS from the command line by typing `cucumber-js`. Like Cucumber-JVM, if there are no matching step implementations, Cucumber-JS will propose some skeleton steps to get you started:

```
$ cucumber-js
1 scenario (1 undefined)
5 steps (5 undefined)
```

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

```
this.Given(/^the flying distance between Sydney and Melbourne is (\d+) km$/,
  function(arg1, callback) {
    // express the regexp above with the code you wish you had
    callback.pending();
  });

this.Given(/^I am a standard Frequent Flyer member$/, function(callback) {
  // express the regexp above with the code you wish you had
  callback.pending();
});
...
```

This shows that you've specified pending scenarios. Now you can add some test logic.

6.8.3 Implementing the steps

Let's look at how to implement steps in Cucumber-JS. By default, Cucumber-JS will look for the step definition file in the `step_definitions` directory. A very simple step implementation file might look like this:

```
var stepDefinitionWrapper = function () {
  var should = require('chai').should()

  var distance_travelled, traveller_status;

  var calculate_travel_points = function(status, distance) {
    return distance / 2;
  }

  this.Given(/^the distance between (.*) and (.*) is (\d+) km$/,
    function(start, end, distance, callback) {
```

Chai is a BDD assertion library for JavaScript.

Variables that will be shared between steps.

In a real application, you'd call the application code here.

```

    distance_travelled = distance;
    callback();
  });

  this.Given(/^I am a (.*) Frequent Flyer member$/,
    function(status, callback) {
      traveller_status = status;
      callback();
    });

  this.When(/^I fly from (.*) to (.*)$/,
    function(start, end, callback) {
      callback();
    });

  this.Then(/^I should earn (\d+) travel points$/,
    function(expected_travel_points, callback) {
      var calculatedTravelPoints = calculate_points(traveller_status,
        distance_travelled);
      calculatedTravelPoints.should.equal(parseInt(expected_travel_points))
      callback();
    });
};

module.exports = stepDefinitionWrapper;

```

Call `callback()` to tell Cucumber-JS that it can proceed to the next step.

Use the Chai assertion library to express your expectations.

This is a simple example to give you a feel for what BDD in JavaScript looks like. We'll look at some more advanced features of Cucumber-JS in the next chapter.

6.8.4 Running the scenarios

When you run Cucumber-JS feature files that have step implementations, Cucumber-JS will produce colored output on the console (see figure 6.13). You can also produce output in JSON format that can be used to import the results into other reporting tools such as Thucydides for more comprehensive reporting, or display them on a continuous integration server.

The terminal shows the command `cucumber-js -f pretty` being run in a directory named `flying-high-cucumber-js`. The output displays a feature and a scenario, both of which passed. Annotations point to specific parts of the output: 'Run the scenarios' points to the command, 'Overall results' points to the summary at the bottom, and 'What steps were run' points to the list of steps in the scenario.

```

➔ flying-high-cucumber-js git:(master) ✖ cucumber-js -f pretty
Feature: Calculating earned Frequent Flyer points

Scenario: Earning standard domestic Frequent Flyer points # features/earning-frequent-flyer-points.feature:3
  Given the distance between Sydney and Melbourne is 878 km # features/earning-frequent-flyer-points.feature:4
  And I am a standard Frequent Flyer member # features/earning-frequent-flyer-points.feature:5
  When I fly from Sydney to Melbourne # features/earning-frequent-flyer-points.feature:6
  Then I should earn 439 travel points # features/earning-frequent-flyer-points.feature:7

1 scenario (1 passed)
4 steps (4 passed)

```

Run the scenarios

Overall results

What steps were run

Figure 6.13 Running Cucumber-JS on the command line

Cucumber-JS enjoys a large developer community, and it's easy to set up and use. It also integrates well with other tools in the JavaScript ecosystem. For example, if you're writing an AngularJS application using the Karma test runner, you can use the Cucumber/Karma integration package (<https://npmjs.org/package/karma-cucumberjs>) to test your AngularJS code directly using Cucumber. At the time of writing, this tool is still under active development, and not all of its features are mature or available, but it's an active project with frequent updates.¹²

6.9 Summary

In this chapter, you learned about automating scenarios:

- To automate a scenario, you write *step definitions* using the BDD tool of your choice.
- Step definition methods interpret the step text and call the appropriate methods to manipulate the application under test.
- JBehave is a well-documented Java-based BDD tool with tight Thucydides integration.
- Cucumber-JVM provides a Gherkin implementation for JVM-based languages.
- Behave is a Gherkin implementation for Python.
- SpecFlow is a Gherkin implementation for .NET with tight Visual Studio integration.
- Cucumber-JS is one of several BDD implementations in JavaScript.

You should now have a good idea of how to automate your acceptance criteria using the language and framework of your choice. But automating scenarios is only the first step—you also need to get these automated scenarios to actually test your application. In the next chapter, you'll learn how to turn these automated scenarios into effective, expressive, and maintainable automated acceptance tests.

¹² The Cucumber-JS web page (<https://github.com/cucumber/cucumber-js>) has a table summarizing the current status of feature implementation.

Part 3

How do I build it? Coding the BDD way

In part 2 you learned about the importance of conversation and collaboration in the BDD process, and that you need to work together to define acceptance criteria in a clear and unambiguous format that can be automated using tools such as Cucumber, JBehave, or SpecFlow. In part 3, we'll look under the hood and see how you can do this automation. Whereas part 2 should be understood by the whole team, part 3 gets a bit more technical and will be of more interest to testers and developers.

As with any code base, the maintenance of automated acceptance tests can be costly if they're not written well from the onset. In chapter 7, you'll learn how to structure and organize your automated acceptance criteria to make them easier to understand and to maintain.

In chapter 8, you'll learn about writing automated acceptance criteria for web applications, including why and when you should implement your acceptance criteria as web tests. This is an area where many teams struggle, so we'll look in some detail at how to write high-quality automated web tests using Selenium WebDriver.

Automated acceptance tests aren't just for web testing. In chapter 9, we'll look at techniques for implementing automated acceptance tests for pure business rules and other requirements that don't need to be verified through the UI.

Finally, it's important to remember that BDD is not just about requirements analysis and automated acceptance testing. In chapter 10, you'll learn how to

apply BDD principles to coding. You'll see how BDD principles, techniques, and tools make traditional test-driven development easier and more effective. You'll see how BDD unit tests flow naturally from BDD acceptance criteria. You'll learn how to use unit tests to design, document, and verify your application code, and indeed how to think in terms of writing low-level specifications rather than unit tests.



From executable specifications to rock-solid automated acceptance tests

This chapter covers

- The importance of writing high-quality automated tests
- Preparing test data for the tests
- Implementing reliable and sustainable tests

In the next few chapters, we'll look at turning the automated scenarios we discussed in chapter 6 into fully automated acceptance tests for different types of applications and technologies.

When automated acceptance tests are poorly designed, they can add to the maintenance overhead, costing more to update and fix when new features are added than they contribute in value to the project. For this reason, it's important to design your acceptance tests well. In this chapter, we'll look at a number of techniques and patterns that can help you write automated acceptance tests that are meaningful, reliable, and maintainable (see figure 7.1).

Over the previous few chapters, you've seen how the BDD lifecycle takes acceptance criteria and turns them into executable specifications (see figure 7.2). Acceptance criteria are those brief notes you write on the back of your story cards that help define when a story or feature is complete. As you've seen, you can write more complete versions of these acceptance criteria in the form of scenarios, which

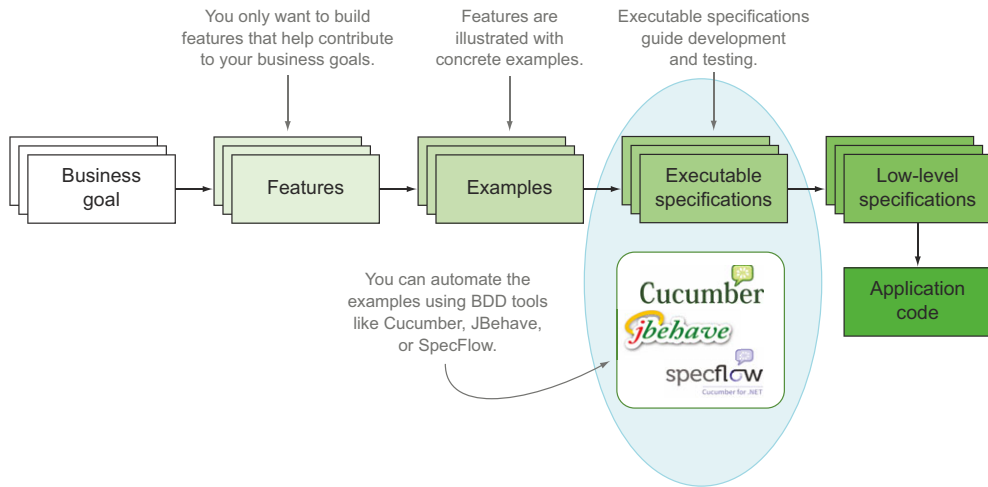


Figure 7.1 In this chapter we'll look at how to make the automated executable specifications robust and maintainable.

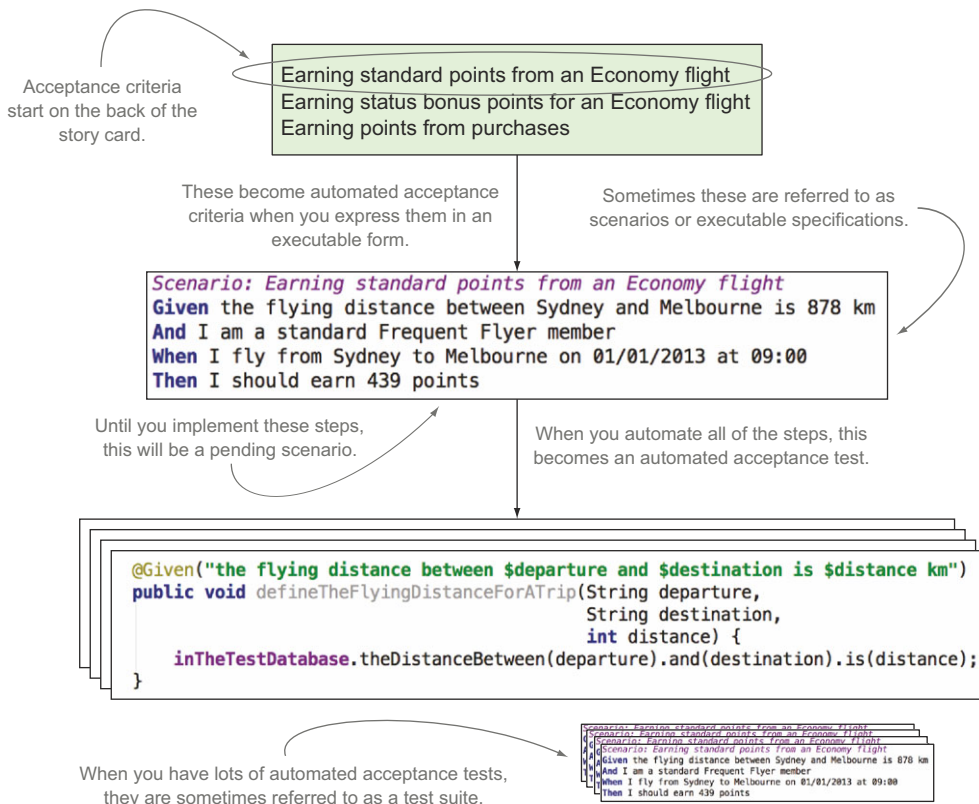


Figure 7.2 Automated data setup is a key part of automated acceptance criteria.

you can automate using BDD tools such as JBehave, Cucumber, and SpecFlow. When you automate these scenarios, they become executable specifications, but these will be reported as pending until you write the underlying code that actually exercises the application and verifies the outcomes. Once this happens, you can talk of automated acceptance tests.

Throughout the rest of this chapter, I'll refer to automated acceptance tests as *automated tests* or simply as *tests*. And for this chapter, *test suite* will refer to all of the automated acceptance tests for a project.

You're now ready to learn how to turn your pending executable specifications into full-fledged automated acceptance tests. But before we can discuss specific test automation approaches and technologies, we need to lay the groundwork and discuss how to ensure that these tests are implemented in a way that will be easy to extend and maintain as the test suite grows:

- A good acceptance test should communicate its intent clearly and provide meaningful feedback about the current state of the application. It should also be reliable and easy to maintain, so that the value provided by the test outweighs the cost of maintaining it.
- Techniques such as *initialization hooks* and *personas* can be used to ensure that the system is in a known state at the start of each test.
- Layers of abstraction can help make the tests clearer and easier to maintain by separating the *what* from the *how*.

Let's start off by discussing what makes an industrial-strength acceptance test.

7.1 Writing industrial-strength acceptance tests

It's not hard to write a simple automated test. There are plenty of books and resources on the internet that can show you how to write automated tests for all sorts of technologies, and we'll look at a few of these later on in this book. But what's harder is ensuring that your tests will stand the test of time. Will your tests continue to provide useful feedback several months down the track? Will they become harder to maintain as the code base grows? Will they become brittle and fragile, discouraging the team from fixing them when they break? Many teams spend a great deal of effort in building large and complex test suites, only to abandon them because they find that the cost of maintaining and updating them outweighs the value of the feedback they provide.

The tests in a well-written automated acceptance test suite should be either passing or pending. A failing automated acceptance test should be a red flag for the development team that demands immediate attention. But when tests fail too often due to sporadic technical issues, the team will lose confidence in them as a feedback mechanism and be less motivated to fix them when they break (see figure 7.3). This leads to a vicious circle, where there are always a few broken tests in the build. When this happens, the automated acceptance criteria no longer perform their primary role of providing feedback about the project's current health. This is what you want to avoid.



Figure 7.3 Acceptance tests that fail too often can lead to complacency.

It's an important issue, and it's worth putting in some effort up front to ensure that the time you invest automating your acceptance criteria yields its full fruits over the duration of the project and beyond. Automated tests are like any other kind of code—if you want them to be robust and easy to maintain, you need to put a little effort into their design and write them in a way that's clear and easy to understand. Writing good automated acceptance tests requires the same software engineering skills and disciplines, and the same level of craftsmanship, as well-written production code. The best automated acceptance tests tend to be the result of a collaborative effort between testers and developers.

Unfortunately, automated tests often don't receive the attention they need. Part of this problem is historical—for many years, test automation has been synonymous with “test scripts.” Teams have been reluctant to put too much effort into writing scripts when they could be writing production code. Many of the more traditional testing tools do indeed use scripting languages that have poor support for, or simply don't encourage, standard software engineering practices, such as refactoring to avoid code duplication, writing reusable components, and writing code in a clean and readable manner.

Many of these issues can be avoided if teams apply a few simple principles. A good automated acceptance test needs to respect a few key rules:

- *It should communicate clearly.* Automated acceptance tests are first and foremost communication tools. Wording and semantics are important: the tests need to clearly explain the business behavior they're demonstrating. They need to explain to stakeholders, business analysts, testers, and other team members what business tasks the application is intended to support, and illustrate how it does so. Ideally, these tests will outlive the development phase of the project and go on to help the maintenance or BAU¹ team understand the project requirements and how they've been implemented.

¹ Business as Usual—the team that takes care of applications once they're deployed into production. In many organizations, this is a different team than the one developing the project.

- *It should provide meaningful feedback.* If a test fails, a developer should be able to understand what the underlying requirement is trying to achieve, and also how it's interacting with the application to do so.
- *It should be reliable.* To get value out of the tests, the team needs to be able to trust the test results. A test should pass if (and only if) the application satisfies the underlying business requirement, and should fail if it doesn't. If a test does break for technical reasons, the issue should be easy to isolate and simple to fix. Although this sounds obvious, it can require care and discipline to ensure that the more complex tests respect this rule.
- *It should be easy to maintain.* A test that breaks too often when the application is updated or modified, or that requires constant maintenance to keep up to date, rapidly becomes a liability for the development team. If this happens too frequently, the developers will often simply cease updating the tests and leave them in a broken state. When this happens, any useful feedback from the test results is lost, and the time invested in building up the test suite is wasted.

In the rest of this chapter, we'll look at how you can write high-quality tests that respect these rules, and that provide valuable feedback and are robust and easy to maintain.

7.2 Automating your test setup process

Before you can automate any acceptance criteria, you need to make sure the system is in a correct and well-known initial state. Also, many automated acceptance criteria—particularly the end-to-end variety—will need to refer to or update data in a database. Some tests may need other services or resources, such as file systems or remote web services, to be initialized and configured. In order to have effective automated acceptance criteria, you need to be able to set up all of these things quickly, precisely, and automatically (see figure 7.4).

If your automated acceptance criteria use a database at all, you should reserve a dedicated database instance for them. Try to ensure that the database configuration is as close to the production one as possible. It's also a bad idea to use a database that's



Figure 7.4 Automated data setup is a key part of automated acceptance criteria.

shared by developers or testers; it should be reserved for the exclusive use of your automated acceptance criteria.

Many teams today try to automate this provisioning process using tools like Puppet and Chef to create a clean test environment from scratch on a freshly created virtual machine before each test run.

Any test that uses a database will typically need the database to be in a known, predictable state at the start of the test. When it comes to setting up your test database, there are several possible approaches; the one that works best for you will largely depend on the nature of your project and technical environment. In the following sections, we'll discuss a few possible strategies.

7.2.1 *Initializing the database before each test*

The most reliable way to set up your test database is to automatically reinitialize the database schema before each test, possibly populating it with a sensible predefined set of reference data. This way, no test can inadvertently break another test by adding unexpected data into the database. It also ensures that each scenario is independent of the others and doesn't rely on another scenario to have been executed beforehand.

7.2.2 *Initializing the database at the start of the test suite*

Unfortunately, in practice your database or technology stack may make initializing the database schema before each test too slow to be a viable option. I've seen several organizations use this approach very successfully, even with large relational databases, but fast feedback is important, and you may have to make trade-offs in this area.

The next-best way to prepare test data is to automatically reinitialize the database schema every time you run the test suite. This is faster than reinitializing the database before each scenario, but it means that each scenario is responsible for deleting any test data that it creates, which isn't without risk. This approach can also introduce subtle dependencies between the scenarios, which can lead to hard-to-reproduce errors.

7.2.3 *Using initialization hooks*

No matter which option you chose, most BDD tools provide "hooks" that allow you to perform actions before and after each scenario and at other strategic points in the test suite lifecycle.

As you've seen, BDD tools like JBehave express requirements in the form of scenarios. When the tests are executed, a scenario will generally correspond to a single automated test, although for table-driven scenarios a test will be executed for each row in the table. The scenarios are placed in feature files or story files. I'll refer to all of these story and feature files as the test suite.

Different tools let you intervene at slightly different stages of test execution, but the main steps in test execution are similar for all of the BDD tools we'll discuss, and they lead to the intervention points illustrated in figure 7.5.

Table 7.1 illustrates some of the methods available in the various tools for intervening at each of these points.

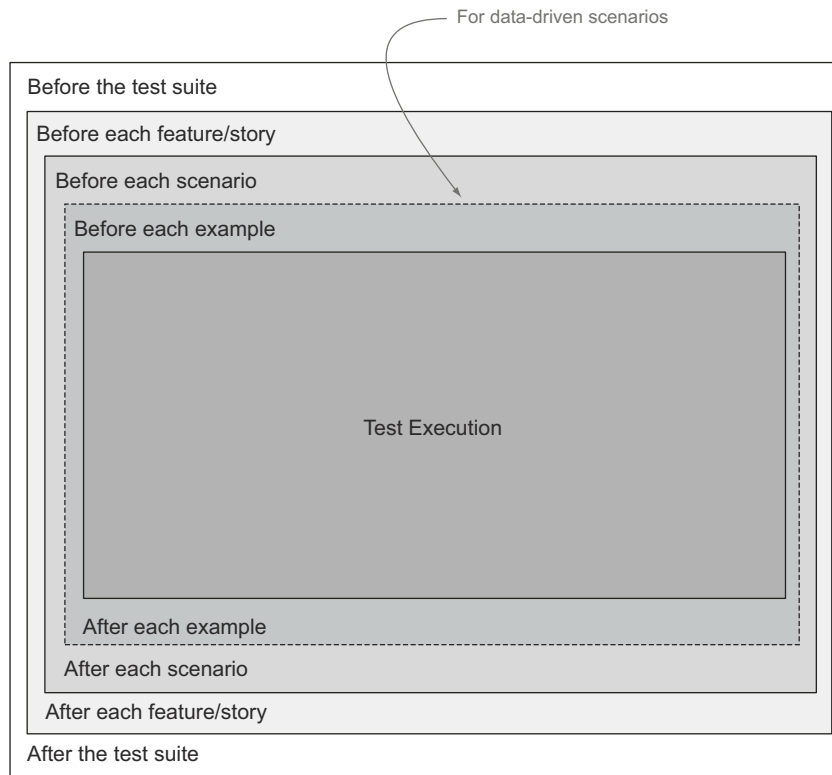


Figure 7.5 Hooks let you intervene at different stages in the test execution.

Table 7.1 Initialization and teardown hooks used by different BDD tools

Stage	JBehave	Cucumber/ JVM	Lettuce	SpecFlow
Before the test suite	@BeforeStories	-	before_all	[BeforeTestRun]
Before each feature or story	@BeforeStory	-	before_feature	[BeforeFeature]
Before each scenario	@BeforeScenario	@Before	before_scenario	[BeforeScenario]
Before each example	@BeforeScenario	-	-	
After each example	@AfterScenario	-	-	
After each scenario	@AfterScenario	@After	after_scenario	[AfterScenario]

Table 7.1 Initialization and teardown hooks used by different BDD tools (*continued*)

Stage	JBehave	Cucumber/ JVM	Lettuce	SpecFlow
After each feature or story	@AfterStory	-	after_feature	[AfterFeature]
After the test suite	@AfterStories	-	after_all	[AfterTestRun]
Before a given tag	-	@Before	before_tag	[BeforeScenario]
After a given tag	-	@After	after_tag	[AfterScenario]

Let's look at how these hooks work in each of these tools.

INITIALIZATION HOOKS IN JBEHAVE

In JBehave you can use the @BeforeScenario or @BeforeStory annotation to reset the database before each scenario or story, whereas @BeforeStories lets you reset the database once at the start of the test suite.

For example, suppose you've written a Database class that's responsible for setting up your test database. To reset the database before each scenario, you could write something like this:

```
@BeforeScenario
public void initializeDatabase() {
    Database.initialize();
}
```

← JBehave will call this method before each scenario.

← A custom class you write to initialize the test database.

The @BeforeStory annotation can also be used to intervene before the execution of a row in a table-driven test, as shown here:

```
@BeforeScenario(uponType = ScenarioType.EXAMPLE)
public void prepareDatabaseForANewExample() {...}
```

← This method will be executed before each row in a table-driven scenario.

There are also symmetrical @AfterScenario, @AfterStory, and @AfterStories annotations that can be used to tidy up the database afterwards.

INITIALIZATION HOOKS IN CUCUMBER-JVM

Other BDD frameworks support similar features. For example, Cucumber-JVM has a more limited set of annotations, providing the @Before and @After annotations to mark methods that will be called before and after each scenario:

```
@Before
public void initializeDatabase() {
    TestDatabase.initialize();
}
```

← Cucumber calls this method before each scenario.

Although it has no equivalent to JBehave's `@BeforeStory` and `@BeforeStories` annotations, Cucumber does provide a feature that native JBehave lacks.² The preceding method will be called before each and every scenario in the test suite, whether it uses the database or not. This is clearly not ideal, especially if only a small part of your acceptance criteria are end-to-end acceptance criteria. Cucumber provides for a more selective approach, where you specify an annotation to indicate which scenarios this hook should apply to:

```
Feature: Joining the Frequent Flyer Program
@end-to-end
Scenario: Registering online for a new Frequent Flyer account
...
```

Flag this scenario as an end-to-end scenario.

You can then refer to this tag in the `@Before` annotation, causing Cucumber to only call this method for scenarios with the `@end-to-end` tag:

```
@Before("@end-to-end")
public void initializeDatabase() {
    TestDatabase.initialize();
}
```

Only call this method before scenarios flagged with the @end-to-end tag.

In this way, Cucumber-JVM provides an elegant way of resetting your test database only when you really need to do so.

INITIALIZATION HOOKS IN BEHAVE

Behave, the Python BDD tool introduced in chapter 6, also provides a rich set of hook methods that you can define in the `environment.py` module. These methods are intuitively named:

- `before_scenario` and `after_scenario`
- `before_feature` and `after_feature`
- `before_tag` and `after_tag`
- `before_all` and `after_all`

To reset the test database before each scenario, you might do something like this:

```
def before_scenario(context, scenario):
    context.database.reset()
```

If you needed to reset the database uniquely for end-to-end scenarios, you could check for the presence of the `@end-to-end` tag, as shown here:

```
def before_scenario(context, scenario):
    if 'end-to-end' in scenario.tags:
        context.database.reset()
```

INITIALIZATION HOOKS IN SPECFLOW

SpecFlow has a particularly broad set of hooks, including `BeforeTestRun`, `BeforeFeature`, and `BeforeScenario`. Like Cucumber-JVM, it allows you to optionally specify

² The Thucydides extension for JBehave does provide support for this feature.

a tag that indicates the scenarios where you would like a method executed. For example, the code to reset the database before each end-to-end scenario might look something like this:

```
[BeforeScenario("end-to-end")]
public static void BeforeEndToEndScenario()
{
    InitializeTestDatabase();
}
```

You’ve now seen how you could place the test database into a clean state before each scenario. Now let’s consider how to inject specific data for each scenario.

7.2.4 Setting up scenario-specific data

When you create a new database, you’ll usually want to populate it with (at least) reference data and possibly some sample production-like data. But even with a fresh database for each new scenario, it’s not a good idea to rely too heavily on prepopulated data. Scenarios should be self-explanatory, and if you rely on predefined test data, it can make it harder to know what the test is actually doing. For example, the following scenario uses a prepopulated set of timetables:

```
Scenario: Tell users about the next departure times from a given station
Given I want to travel from Parramatta to Town Hall at 8:00
When I ask about the next trains to depart
Then I should be told about the trains at: 8:02, 8:11, 8:14
```

This scenario assumes not only that project team members and stakeholders know about the geographical locations of Parramatta and Town Hall stations (this isn’t an unreasonable expectation for people working in the Sydney public transport network—see figure 7.6), but also that they know the times of trains that are scheduled to arrive at Town Hall. This is less obvious, and as a result, the intent of the scenario is unclear. Why these trains and not others? What trains should *not* be included?

Some data, such as the names of the stations and the lines that connect them, would be well known to stakeholders. You can safely populate the initial test database with this sort of reference data; putting it into the scenario would just add clutter and distract from the more essential information. But details that are likely to change often, or that are central to the business rule you’re trying to illustrate, should be included in the scenario.

A better way to express this scenario might be as follows:

```
Scenario: Tell users about the next departure times from a given station
We should show the next three departing trains
Given Western Line trains get to Parramatta at 7:58,8:02,8:08,8:11,8.14
When I want to travel from Parramatta to Town Hall at 8:00
Then I should be told about the trains at: 8:02, 8:08, 8:11
```

Specify the business rule being described. 1

3 **Specify which trains should be included in the results.**

The Western Line is well known to business stakeholders and users. 2

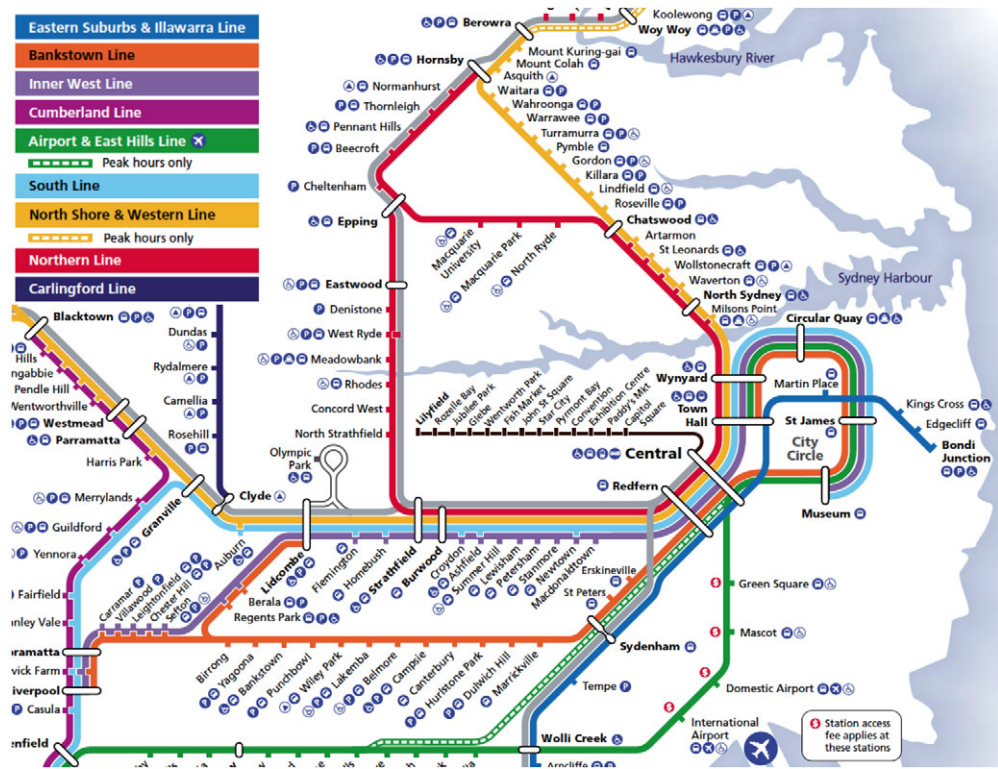


Figure 7.6 Part of the Sydney rail network used in the scenarios for this chapter

In this version, you assume that the stakeholder knows where the Western Line runs, but you spell out the timetable data you're using in the example ②. In this context, the results ③ illustrate not only which trains should be returned, but which trains should be excluded from the mix. For more clarity, you also include a short comment to explain the business rule that you're illustrating ①.

Notice how examples like this can help raise questions about counterexamples and corner cases. What if a train arrives at 8:00? What if there are no more trains until the following day—do you still display the next three?

Embedding tables inside the scenario steps is another popular way to set up test data for a scenario. For example, in the following scenario, you need to tell a traveller how to get to a particular station. You can inject a number of sample timetable entries into the database before using this timetable data to illustrate what scheduled trains you'd like the traveller to see at a given time and station:

Scenario: Tell users about the next departure times from a given station
Given the following train timetable entries

Line	Destination	Departs Central	Platform	The test data used to prepare the scenario
Northern	Epping	8:02	10	
Western	Emu Plains	8:04	12	

North Shore	Berowa	8:10	20	↑ The test data used to prepare the scenario
Bankstown	Liverpool	8:12	16	
When I want to travel from Central to Chatswood at 8:00				
Then I should be told about the following trains:				
Destination	Departure Time	Platform	The expected results	
Epping	8:02	10		
Berowa	8:10	20		

Of course, this scenario would only test one test case, and the feature might be better illustrated by several different examples. Alternatively, you could use a more example-driven approach, as shown here:

Scenario: Tell users about the next departure times from a given station
Given the following train timetable entries

Line	Destination	Departs Central	Platform
Northern	Epping	8:02	10
Western	Emu Plains	8:04	12
North Shore	Berowa	8:10	20
Bankstown	Liverpool	8:12	16

When I want to travel from <departure> to <destination>
Then I should be told about trains on the lines: <expectedLines>

Examples:

departure	destination	expectedLines
Central	Epping	Northern, Epping
Martin Place	Central	Inner City
Redfern	Strathfield	Western, Epping, Inner City

1 The test data used to prepare the scenario.

2 Illustrate possible outcomes with several examples.

Here you set up the test data as in the previous example ① and then illustrate the scenario with examples of several different search criteria ②. Note that you express the expected results as a list of line names, which will be checked against the actual records found by the search.

7.2.5 Using personas and known entities

Another useful technique when setting up data for a scenario is to use *personas* or *known entities*. In the domain of user experience (UX), personas are fictional characters that are meant to represent the different types of people who will be using the system. A persona usually comes with a very detailed description, including everything from the fictional person's name and email address to their interests, hobbies, and work habits. Figure 7.7 shows a persona for a Flying High customer.

When it comes to setting up test data for your scenarios, a persona assembles a set of precise data under a well-known name. So when you refer to Jane in the following scenario, everyone on the team will know who you're talking about. Here's an example of one of Jane's scenarios:

Scenario: Registering online for a new Frequent Flyer account	
Given Jane is not a Frequent Flyer member	
When Jane registers for a new account	
Then Jane should be sent a confirmation email	
And Jane should receive 500 bonus points	

← ① "Jane" refers to the persona with this name.

Casual Flying High Customer: Jane

First name:	Jane
Last name:	Smith
Email:	jane@acme.com
DOB:	29/08/1981
Phone:	0123456789
Address:	10 Partridge Street, Dandenong



Background: Jane is an account manager for a pharmaceutical company in Melbourne.

Family: Married with 2 children, a 5 year old boy and a 2 year old girl

Preferred communication: email and phone

Seating preference: Aisle

Flying profile: Jane flies 2-3 times per month for her job. She takes whatever airline proposes the most attractive deal.

Figure 7.7 A persona is a fictional character meant to represent a category of user of the system.

In the implementation of the first step, you can use the name *Jane* ❶ as an indicator to inject all of the relevant data about this persona into the test database. In subsequent steps, you can use the persona name to identify the data set you need to use for the rest of the scenario.

When you initially configure this test data, you might do something like this:

```

Persona persona;

@Given("$name is not a Frequent Flyer member")
public void givenANonFrequentFlyerMember(String name) {
    persona = Persona.withName(name);
    persona.setStatus(NonMember);
    persona.save();
}

```

Fetch this persona data by name. ❷

❶ A special test-specific domain object to store persona data.

❸ Strip this persona of their membership.

❹ Save the updated persona.

The `Persona` class ❶ is created for these tests to define the known personas and inject them into the test database. In the method body, you retrieve the corresponding persona data ❷, ensure that this persona isn't a Frequent Flyer member ❸, and then save the updated persona to the test database ❹. Note that not all persona objects would need to go into the test database—some might simply be configured in the initial steps and then used in subsequent steps. In either case, the primary goal is to make the scenario more readable and easier to maintain by removing any unnecessary clutter.

You can also use the persona data as a kind of template, changing other fields as required. For example, suppose you needed to implement the following step:

```
Given Jane is 60 years old
```

You could implement this step as shown here:

```

Persona persona;
@Given("name is $age years old")
public void givenAFrequentFlyerMemberOfAGivenAge(String name, int age) {
    persona = Persona.withName(name);
    persona.setAge(age);
}

```

Persona to be used for this scenario

Look up persona by name

Override persona's age

Here you're simply starting off with the default field values for this persona and overriding the fields you want to change. This is very useful for scenarios where there are only slight variations in the data to be used.

A similar approach can be used with other types of test data. The team defines *known entities*—domain objects in a well-known state. For example, a banking application working on transferring bank statement files might define a few standard file types, and then only specify the fields that have different values for a given scenario.

7.3 Separating the what from the how

Automated acceptance tests need to be stable and reliable. When small changes happen in the application, you shouldn't need to update hundreds of acceptance tests just to keep the test suite running. Maintaining your automated acceptance tests should never be a hindrance to your ability to embrace change.

Yet when many teams start automating their acceptance criteria, this is exactly what happens. Often a small change in a single commonly used web page will break a large swath of tests, and developers will need to fix each test individually before the test suite can work again. This is thankless, unproductive work that does little to encourage the team to automate more acceptance tests.

It doesn't have to be this way. The trick is to apply a basic principle of software engineering, known as *layers of abstraction*. Layers of abstraction hide implementation details inside a function or object. This makes the high-level layers cleaner and easier to understand and also isolates them from changes in the implementation details hidden inside the lower layers.

When you write automated acceptance criteria, using layers can help you isolate the more volatile, low-level implementation details of your tests from the higher level, more stable business rules. High-level business rules tend to be relatively stable, and changes to them will be driven by the business rather than by technical constraints. Lower-level implementation details, such as screen layouts, field names, and how a low-level library is called, tend to change more frequently. When changes do happen at the lower implementation levels, the impact on the automated acceptance criteria should be minimal.

Experienced BDD practitioners³ typically use at least three layers of abstraction in their tests, divided into three categories, like the ones illustrated in figure 7.8.

³ See, for example, Gojko Adzik, "How to implement UI testing without shooting yourself in the foot," <http://gojko.net/2010/04/13/how-to-implement-ui-testing-without-shooting-yourself-in-the-foot-2/>.

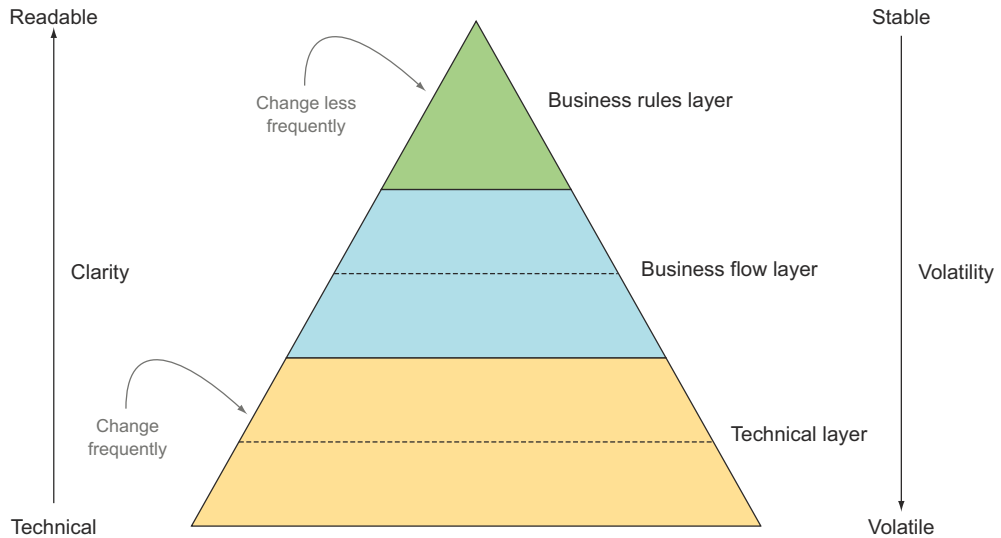
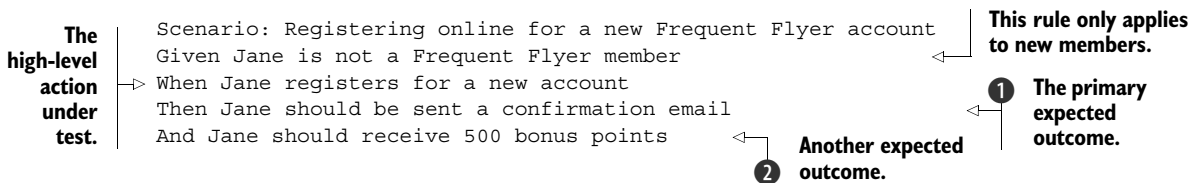


Figure 7.8 Well-written automated acceptance criteria are organized in three main layers.

Let's take a look at each of these three main layers.

7.3.1 The Business Rules layer describes the expected outcomes

The *Business Rules* layer describes the requirement under test in high-level business terms. If you're using a BDD tool such as JBehave, Cucumber, or SpecFlow, the business rule will typically take the form of a scenario in a feature file using either a table or a narrative structure, like the following:



As you saw in the previous chapter, this sort of scenario focuses on the business outcomes of a requirement ① and ②, and isn't too worried about how the system delivers these outcomes. Requirements expressed in this way should only need to change if the business changes its mind (or its understanding evolves) about the expected outcomes.

Figure 7.9 illustrates the way the Business Rules layer works together with the other two layers to implement this scenario. Don't worry about the details of the code just yet; we'll discuss the technologies used in this example later on in the chapter and in the following chapters.

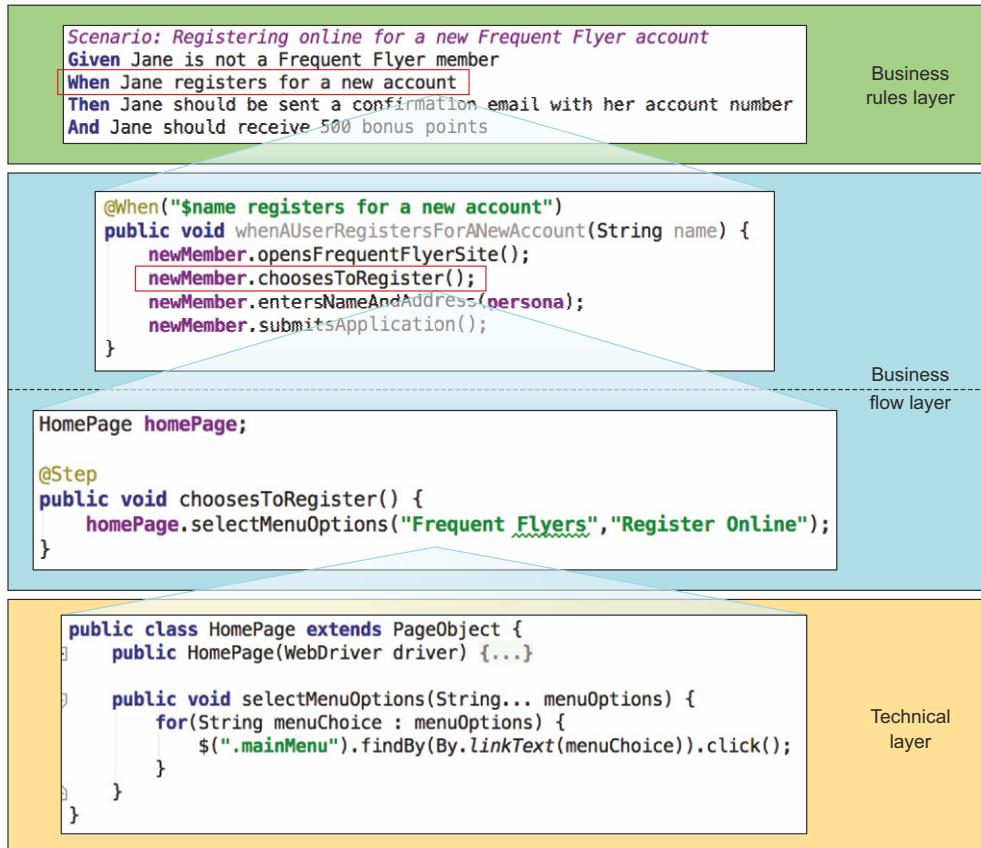


Figure 7.9 Each automation layer uses services from the layer beneath to implement a scenario.

7.3.2 The Business Flow layer describes the user's journey

You implement the scenarios defined in the Business Rules layer using libraries and functions defined in the *Business Flow* layer. This layer represents the user's journey through the system to achieve a particular business goal. What high-level actions does Jane need to perform to register for a new account? How will you know that she has received 500 bonus points? How would you step through the application to demonstrate this scenario?

For example, you might break down the "When Jane registers for a new account" part of the high-level business requirement into the following steps:

- Jane goes to the Frequent Flyer website.
- Jane chooses to register for a new account.
- Jane enters her name and address.
- Jane submits her application.

You'll try to express these steps in a very readable, business-friendly way. For example, the scenario corresponding to this example might look like this:

Scenario: Registering online for a new Frequent Flyer account
 Given Jane is not a Frequent Flyer member
 When Jane registers for a new account
 Then Jane should be sent a confirmation email with her account number
 And Jane should receive 500 bonus points

Using Java and JBehave, you might write step definitions for the first two lines of this scenario like this:⁴

```

@Steps
NewMemberSteps newMember;

Persona user;

@Given("$user is not a Frequent Flyer member")
public void givenANonFrequentFlyerMember(String name) {
    user = Persona.withName(name);
    user.setStatus(NonMember);
    user.save();
}

@When("$name registers for a new account")
public void whenAUserRegistersForANewAccount(String name) {
    newMember.opensFrequentFlyerSite();
    newMember.choosesToRegister();
    newMember.entersNameAndAddress(user);
    newMember.submitsApplication();
}
  
```

1 The `@Steps` annotation indicates a “step library” in Thucydides.

2 Details about each step are hidden inside this step library.

The user that will be used throughout this scenario.

Create the user from a given `Persona`.

The user’s journey through the system.

A business stakeholder or Scrum product owner should have no trouble reading and providing feedback on this. In fact, code like this is often defined and implemented collaboratively by a combination of developers and testers, UX experts, and business analysts. In some teams, the testers even take a very active role in implementing this layer, using the technical components provided by the developers.

These steps are more likely to change than the core business requirements, but they’ll only need to be changed if some aspect of the application workflow changes. For example, maybe you’ll need to add a step where Jane also needs to agree to a set of terms and conditions.

The Business Flow layer may contain several sublayers, depending on the complexity of the scenarios and application being tested. For example, in the `whenAUserRegistersForANewAccount()` method in the preceding code, you don’t interact directly with the technical UI components. Instead, you delegate to a *step library* that describes the details of each step of the business flow. When you use Thucydides with JBehave, the `@Steps` annotation **1** injects a step library object **2** into the step definition class. The `choosesToRegister()` method in the `NewMemberSteps` step library, for example, could be implemented like this:

⁴ This code follows the `@Given` step you saw in section 7.1.5.

A page object from the Technical layer that models an HTML page.

```
HomePage homePage;
```

```
@Step
```

```
public void choosesToRegister() {
```

```
    homePage.selectMenuOptions("Frequent Flyers", "Register Online");
```

```
}
```

This annotation tells Thucydides that this step should appear in the living documentation.

Use the page object to interact with the user interface.

Other steps might be more involved, but they'll still use descriptive method names provided by components in the Technical layer:

```
@Step
```

```
public void entersNameAndAddress(Persona user) {
```

```
    homePage.enterFirstName(user.getFirstName());
```

```
    homePage.enterLastName(user.getLastName());
```

```
    homePage.enterAddress(user.getAddress());
```

```
}
```

Still other steps might interact with other technical components or services. For example, the third step of the scenario might be implemented using a call to an email service component:

```
EmailService email;
```

A technical service that talks to the email server.

Retrieve the email messages for this user.

```
@Then("$name should be sent a confirmation email")
```

```
public void shouldReceiveConfirmationEmailWithAccountNumber(String name) {
```

```
    List<EmailMessage> emails = email.getEmailsForUser(name);
```

```
    assertThat(emails,
```

```
        containMessageWithTitle("Welcome to Flying High!"));
```

```
}
```

Check that one of them is the expected confirmation message.

Methods in the Business Flow layer don't need to be concerned about how these actions are implemented. This highlights one of the advantages of using layers of abstraction: they isolate the *what* from the *how*. The Business Flow steps don't need to worry about *how* you talk to the email server. At this level, all you're interested in is describing *what* email message you expect the user to receive.

The more technical actions are implemented using reusable components from the Technical layer, which we'll look at in the next section.

7.3.3 The Technical layer interacts with the system

The *Technical* layer represents how the user interacts with the system at a detailed level—how they navigate to the registration page, what they enter when they get there, how you identify these fields on the HTML page, and so forth.

These technical steps are generally implemented by developers. The steps interact with the application code either through the user interface or by accessing some other service layer, or by accessing the application code directly. Writing these steps requires knowledge of both the implementation of the application (how the web pages are structured, what web services exist, and so forth) and the test automation software.

The developers effectively provide a library of reusable components and methods to be used by steps in the Business Flow layer. For example, if you're working with a web application, you'll design *page objects*—classes that hide the technical details about HTML fields and CSS classes behind descriptively named methods, such as `enterName()` or `submitApplicationRequest()`.

Following on from the previous example, suppose you were writing a Technical layer using Thucydides and WebDriver, a popular open source web automation tool that we'll discuss in detail in chapter 8. Using this technology stack, the `selectMenuOptions()` method might look like this:

```
public class HomePage extends PageObject {
    public void selectMenuOptions(String... menuOptions) {
        for(String menuChoice : menuOptions) {
            $(".mainMenu").findBy(By.linkText(menuChoice)).click();
        }
    }
}
```

A nice readable API method.

The nitty-gritty CSS and WebDriver logic are hidden away.

A well-written Technical layer does wonders to isolate the other layers from the impact of low-level changes. For example, suppose the design of the registration page changes, involving changes to the HTML structure and field names. Such a change would modify neither the business rule nor the workflow for this requirement, and those levels wouldn't be affected. The only code you'd need to update is within the page object that encapsulates the registration page. This update would work for any scenario that uses this page.

In addition, you can implement the Business Rules and the Business Flow layers *before* the user interface has been implemented. The first layers are written in collaboration with the testers and business analysts and act as guidelines for development work. Only when the user interface is reasonably stable do you implement the technical components.

7.3.4 How many layers?

The precise number of layers you need to make your automated tests readable and maintainable will vary depending on the scope and complexity of the acceptance criteria. There are many different types of acceptance criteria, and some will be more complicated and involved than others.

Some will describe how a user interacts with the application to achieve some business goal, narrating the user's journey through the system and the expected business outcomes, like the "Registering online for a new Frequent Flyer account" scenario you saw earlier. Others will focus on more specific business rules, as with the Frequent Flyer point calculation examples you saw in chapter 6:

```
Scenario: Calculating travel and status points for a domestic flight
Given the flying distance between Sydney and Melbourne is 878 km
And I am a standard Frequent Flyer member
```

```
When I fly from Sydney to Melbourne  
Then I should earn 439 travel points  
And I should earn 40 status points
```

The layered approach we've discussed here is fairly common among experienced practitioners for web-based acceptance testing, but it's equally applicable for other types of acceptance criteria. In the next few chapters, you'll see it applied to a number of different technologies.

7.4 **Summary**

In this chapter you learned how to structure your automated acceptance criteria:

- The general principles of designing meaningful, reliable, and maintainable automated acceptance tests
- How to prepare your test database before executing your automated acceptance criteria
- How using layers of abstraction makes the automated acceptance criteria more robust by separating the *what* from the *how* at multiple levels

In the next chapter, you'll learn how to write automated acceptance tests that exercise a web interface.

Automating acceptance criteria for the UI layer

This chapter covers

- Why and when you should write automated UI tests
- Using Selenium WebDriver for web tests
- Finding and interacting with page elements in your tests
- Using the Page Objects pattern to make your tests cleaner
- Libraries that extend Selenium WebDriver

In the previous chapter, you learned how using a layered approach to automated acceptance testing helps make your tests clearer, more robust, and more maintainable. We discussed the three broad layers used in well-designed automated acceptance tests: the *Business Rules* layer, the *Business Flow* layer, and the *Technical* layer. In the following few chapters, we'll focus on approaches and tools that can be used to implement the Technical layer, starting with the user interface.

In this chapter we'll discuss techniques to automate UI tests for web-based applications (see figure 8.1). Users interact with an application through its user interface, and in modern web applications, the UI implementation plays a major role in

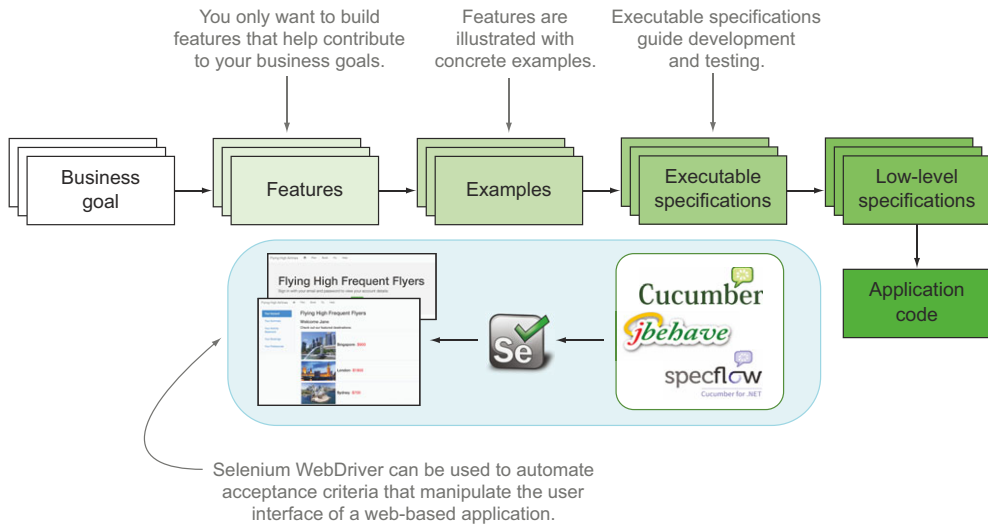


Figure 8.1 In this chapter you'll learn how to automate acceptance criteria in order to exercise the UI of web-based applications.

the overall user experience. The screenshots from automated web tests can be a valuable aid for testers, and they're also a great way to provide illustrated documentation describing how the application behaves.

We'll look at automated web testing from several perspectives:

- *Automated web tests* are very effective at testing user interactions with the UI, and for illustrating end-to-end user interactions with the application. But if they're badly designed or used to test things that would be better tested by non-UI tests, web tests can become a maintenance liability.
- *Selenium WebDriver* is a popular open source library that can be used to write automated web tests in a number of languages, such as Java, Groovy, Ruby, C#, and Python.
- Selenium WebDriver provides good support for *Page Objects*. The Page Objects design pattern can help make automated web tests more readable and easier to maintain.
- Many open source libraries build upon and extend Selenium WebDriver to make writing automated web tests easier and more convenient.

Although we'll focus on web testing in this chapter, many of the tools and approaches discussed here also apply to other types of user interfaces. For example, mobile apps can be tested effectively using the toolset we'll discuss here by using Appium (<http://appium.io/>), a WebDriver-based automation library for mobile apps, and the Page Objects pattern is applicable for any type of GUI.

To write effective automated web tests, you need to know not only how to automate web tests well, but also when you should and shouldn't automate scenarios with web tests.

8.1 When and how should you test the UI?

Web tests have some significant advantages over other types of testing:

- The visual results, when reported well, are an effective way to describe how the user interacts with the application to achieve particular tasks.
- They reproduce end-user behavior much more closely than under-the-hood tests.
- They're a great way to demo features to stakeholders.
- They can significantly reduce the need for manual UI testing, which represents a significant overhead for testers.

A web test, by definition, is designed to verify UI behavior. But web tests, as end-to-end tests, can also be an effective way to illustrate and check how all of the components in the system work together. Used as living documentation, a web test also often does a great job of documenting how a user will use the system to achieve a particular goal. Web tests can also help give business analysts, testers, and stakeholders more confidence in the automated acceptance tests.

8.1.1 The risks of too many web tests

But web tests aren't without a degree of risk. Sometimes when teams start out with BDD, they try to automate all of their requirements exhaustively, almost exclusively using detailed, fine-grained, script-like automated web tests. This is a natural tendency for many testers who come from a background of automated functional testing using commercial tools such as HP's QuickTest Professional. But down this path danger lies. It's very easy to build up a large suite of brittle automated web tests that are costly to maintain and hard to keep up to date. Poorly designed script-like automated web tests often contain a large amount of duplicated code. When a web page is modified, each test that manipulates the page needs to be updated individually, which is time-consuming and error-prone.

Automated web tests generally run much more slowly than non-web tests, and they tend to be more fickle. Some tests may not behave in the same manner from one browser to another, and they may require browser-specific tweaks. Tests can also fail for reasons beyond the control of your code: an incorrect version of Firefox running on the build server, a page timeout because of network latency, and so on.

Record-Replay style scripting tools also present their own particular category of problems. These tools, which include products like Selenium IDE and QuickTest Professional, allow users to record test scripts through a visual tool and replay the tests afterwards. This approach sounds simple and intuitive, but it's deeply flawed. One problem is that these scripts are extremely brittle, hard to read, and unclear about their intent, and are, as a result, virtually unmaintainable. The other problem is that these scripts can't be written until the application development has been completed, which means that they tend to be written as test scripts to verify the implemented behavior, rather than as automated acceptance criteria that contribute to and help guide development efforts from the early stages of the project. For these



Figure 8.2 Automated web tests have a somewhat justified reputation of being slow (courtesy of <http://xkcd.com>).

reasons, it's virtually impossible to do good BDD-style acceptance testing with Record-Replay tools.

Fortunately, web browser automation using open source tools like Selenium WebDriver has improved with age, and today it's very possible to write reliable, robust, and maintainable automated web tests, particularly if you apply the principles of layers and reusable steps that we'll discuss in this chapter. Isolating the code that interacts with the web page in a single class or method (using the Page Objects pattern, for example) goes a long way toward making these tests easier to maintain.

But no matter what tooling you use, the problem of speed still remains (see figure 8.2). An automated web test needs to open a browser and reproduce the actions of the user through the browser, which takes time. Each page load slows down the test. For modern web applications using AJAX-based libraries like AngularJS and Backbone.js, the page updates tend to be much faster, so speed is less of an issue. But, in general, automated web testing will always be significantly slower than tests that exercise the application code directly.

8.1.2 Web testing with headless browsers

One strategy for addressing the issue of slow web tests is to use a headless browser. Headless browser libraries such as HtmlUnit for Java (<http://htmlunit.sourceforge.net>), Webrat for Ruby (<https://github.com/brynary/webrat>), and Twill for Python (<http://twill.idyll.org>) send HTTP queries directly to the server, without having to start up an actual web browser like Firefox or Chrome. HtmlUnit, for example, works with WebDriver and a number of other Java-based web-testing libraries, providing APIs to analyze the structure of the HTML document. PhantomJS (<http://phantomjs.org>) provides a more accurate browser simulation, because it renders the HTML like a real browser would, but does so internally. Headless tests often run more quickly than they would using a real browser, and they also make it easier to run a number of tests in parallel.

Some of these libraries, such as HtmlUnit, have limited support for AJAX and JavaScript, so your mileage with a modern JavaScript-based website may vary. This can

sometimes make them less useful for more complex user interfaces, such as modern JavaScript one-page applications.

If your application does rely heavily on AJAX and JavaScript, then PhantomJS provides significantly more reliable browser emulation than HtmlUnit, including the features that you'd expect of a real browser, such as good support for dynamic asynchronous behavior and the ability to capture screenshots. PhantomJS is used by a number of web-testing libraries, including WebDriver. Although it's not a great deal faster than a real browser for individual tests, it tends to be more tolerant of parallel testing.

It's important to note that headless browsers don't have exactly the same behavior, or render in exactly the same way, as real browsers. HtmlUnit uses the Rhino JavaScript implementation, which isn't used by a real browser. PhantomJS uses WebKit, which may have different behavior than Firefox or Internet Explorer. And if your application contains important business logic in the client, you may want to verify that it also works correctly across different browsers, including a range of real ones.

8.1.3 How much web testing do you really need?

Web tests clearly have their uses. But you rarely need to test every aspect of a system using web tests, and doing so is generally not a good idea. In fact, in a typical BDD project, a significant proportion of automated acceptance tests will be implemented as non-web tests (see figure 8.3). These non-UI tests can take many forms, as you'll see in chapter 9, including what would traditionally be classed as integration or unit tests. Many automated acceptance criteria, particularly those related to business rules or calculations, are more effectively done directly using the application code rather than via the user interface, as non-web tests can test specific business rules more quickly and more precisely than an end-to-end web test.

Of course, it can be tricky to know whether to implement an acceptance test as a web test or a non-web test. You only need a web test for two things:

- Illustrating the user's journey through the system
- Illustrating how a business rule is represented in the user interface

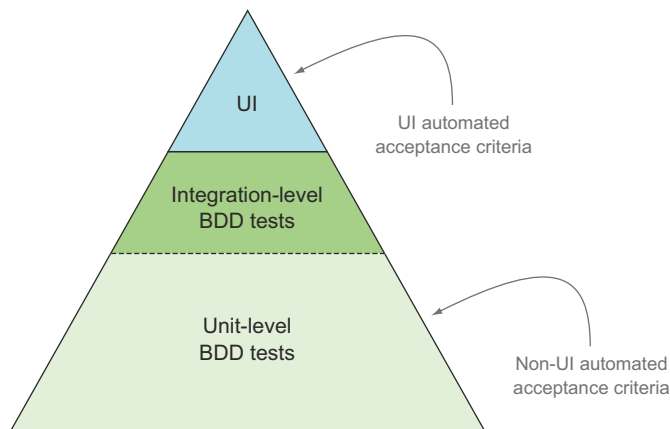


Figure 8.3 A typical BDD project will have many more non-UI automated acceptance tests than UI ones.

Web tests do an excellent job of illustrating how a user interacts with the system via the user interface to achieve a particular business goal. But they don't need to show every possible path through the system—just the more significant ones. More exhaustive testing can be left to faster-running unit tests.

Web tests can also illustrate how business rules are reflected in the user interface. For example, suppose that when they book a flight, Frequent Flyer members should be given the option to choose their seat, a privilege not offered to other customers. This would be a good candidate for an automated web test.

A good rule of thumb is to ask yourself whether you're illustrating how the user interacts with the application or underlying business logic that's independent of the user interface. For example, suppose you were testing a user authentication feature. The acceptance criteria might include the following:

- The user should receive feedback indicating the strength of the password entered.
- Only strong passwords should be accepted.

The first acceptance criterion relates to the user's interaction with the web page, and would need to illustrate how this feedback is provided on the login page. This would be a good candidate for an automated web test.

The second criterion, on the other hand, is about determining what makes a strong password, and what passwords users should be allowed to enter. While this could be done through the user interface by repeatedly submitting different passwords, this would be wasteful. What you're really checking here is the password-strength algorithm, so an application-code-level test would be more appropriate.

8.2 Automating web-based acceptance criteria using Selenium WebDriver

In this section, we'll look at automating web tests using Selenium WebDriver. Selenium WebDriver is a popular open source web browser automation library that can be used to write effective, automated web tests. It also forms the basis for many higher-level web-testing tools. The examples will focus on working with WebDriver in Java, but the principles and techniques we'll discuss will be generally applicable to any WebDriver-based testing.

WebDriver is a browser-automation tool. It lets you write tests that launch and interact with a real browser. This interaction can include simple clicks on buttons or links, or more sophisticated mouse operations such as hovering or dragging and dropping.

WebDriver lets you check the test outcomes by inspecting the state of the page in the browser. WebDriver also gives you the ability to take screenshots along the way—screenshots that can be used later as part of the test reports or living documentation.

Figure 8.4 shows a high-level view of WebDriver. WebDriver supports a large number of web browsers, including Firefox, Chrome, and Internet Explorer. This allows you to test your application in different environments and with different browsers.

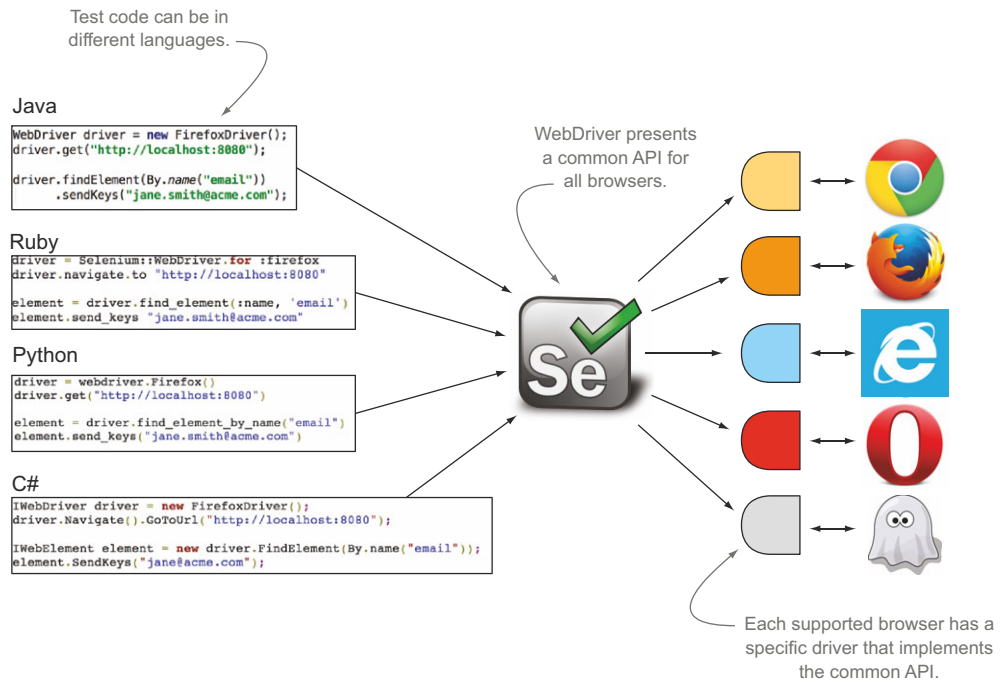


Figure 8.4 Overview of the WebDriver architecture

You can also run web tests in *headless* mode, using a special JavaScript-based browser called PhantomJS.

WebDriver tests can be written in a number of languages, including Java, Groovy, Ruby, C#, and Python. The WebDriver API varies little from one language to another, so you can generally use the language you're most comfortable with, or the one that provides the most value for you in terms of ease of writing, maintainability, and living documentation.

The WebDriver API is powerful and flexible, but there are several open source libraries for different platforms that can help you build on WebDriver to write web tests more efficiently and more expressively, including Thucydides, Watir, WatiN, and Geb. We'll look at some of these libraries in action in section 8.3.2. For most of this chapter, you'll use the WebDriver API with Java.

8.2.1 Getting started with WebDriver in Java

Let's start with a very simple example of web browser automation with WebDriver. You'll illustrate WebDriver's features using a simple version of the Flying High Frequent Flyer website that we've discussed in previous chapters.

If you want to follow along, you can download both the website and the sample code from either the GitHub repository (<https://github.com/bdd-in-action/chapter-8>) or from the Manning website. The sample code repository contains two directories:

- The flying-high directory contains the sample website (see sidebar).
- The flying-high-tests directory contains the sample WebDriver code we'll discuss.

Running the sample website

The Frequent Flyer website you'll use is a simple standalone website that will run on any web server. It's a simple, single-page, JavaScript-based web application. You can either deploy it to your own web server or run it as a standalone website. One way to do this is to use Node.js, which is a lightweight JavaScript platform used to build and run JavaScript-based server-side applications. You don't need to know anything about Node.js to run the sample site; just follow the instructions.

First, you'll need to install Node.js, which you can download from the Node.js website (<http://nodejs.org/>). Once this is done, install the Node.js http-server tool from the command line as follows:

```
npm install -g http-server
```

Now go into the flying-high directory of the chapter 8 sample code, and start up the website with the following command:

```
http-server app
```

To view the running application, open up a web browser and go to <http://localhost:8080> (unless you already have an application running on port 8080, in which case it will run on port 8081). You should see a page similar to the one in figure 8.5.

Suppose you're testing the sign-in feature of your Frequent Flyer website. Registered members need to enter their email address and password to access their account details. The login screen looks something like the one in figure 8.5.

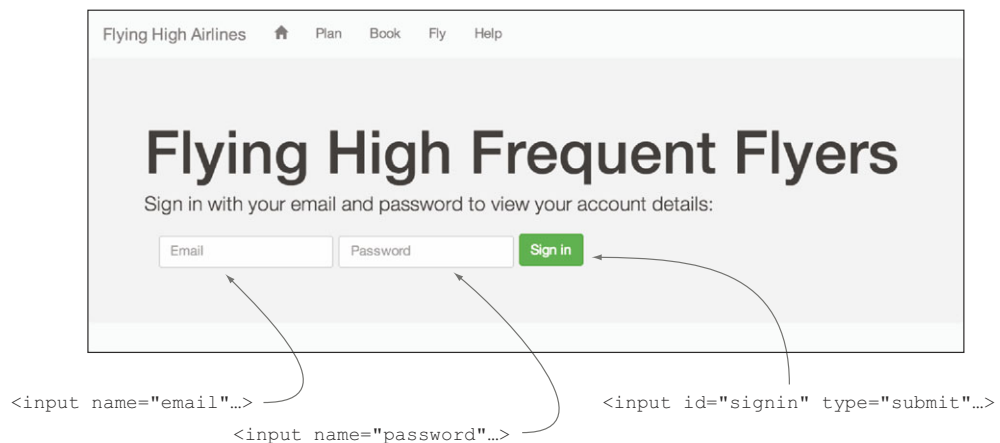


Figure 8.5 Flying High Frequent Flyer members identify themselves using their email address and a password.

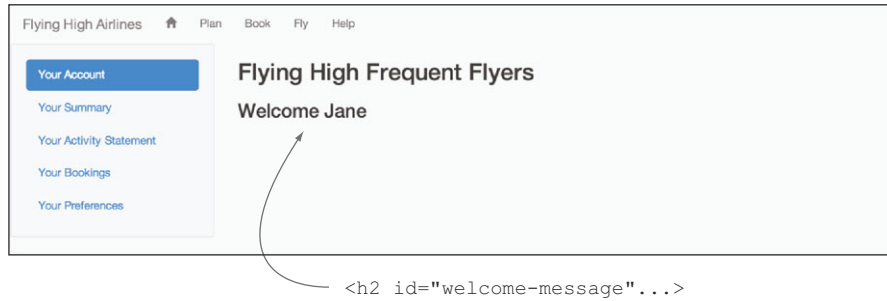


Figure 8.6 Authenticated members are welcomed with a friendly message.

Once a member has entered a matching email and password, they'll be welcomed to the member's area with a friendly message (see figure 8.6).

A scenario describing this requirement might look like this:

What does Jane need to do to authenticate?	Scenario: Successful authentication	Jane is a defined persona known to team members.
	Given Jane is a registered Frequent Flyer	
	When Jane authenticates with a valid email address and password	What is the expected business outcome?
	Then Jane should be given access to her account	

Using the WebDriver Java API, your test might contain the following code:

Enter the member's email and password.	<pre>WebDriver driver = new FirefoxDriver(); driver.get("http://localhost:8080/#/welcome"); driver.findElement(By.name("email")).sendKeys("jane.smith@acme.com"); driver.findElement(By.name("password")).sendKeys("s3cr3t"); driver.findElement(By.id("signin")).click();</pre>	1 Create a new WebDriver instance using Firefox.
		Open Firefox using this URL.
		Click on the Sign In button.
Check the current page title.	<pre>WebElement welcomeMsg = driver.findElement(By.id("welcome-message")); assertThat(welcomeMsg.getText()).isEqualTo("Welcome Jane");</pre>	Check the current page welcome message.

The first thing you do is create a new WebDriver driver instance ①. This driver instance, which implements the WebDriver interface, is the starting point for all of your interactions with the application.

The WebDriver interface has a number of implementations, one for each supported browser, and you need to specify which implementation you want to use. Here, you're creating a new instance of the `FirefoxDriver` class, which you'll use to run the tests in a Firefox browser.¹ You can run the tests in a different browser by using a different implementation (see table 8.1).

¹ Of course, you'll need to have Firefox installed for this to work.

Table 8.1 The main WebDriver implementations

Browser	WebDriver implementation	Notes
Firefox	FirefoxDriver	Normally works out of the box if Firefox is installed.
Chrome	ChromeDriver	Download the chromedriver executable from http://chromedriver.storage.googleapis.com/index.html and put it on the system path (see https://code.google.com/p/selenium/wiki/ChromeDriver).
Internet Explorer	InternetExplorerDriver	Download the standalone server from https://code.google.com/p/selenium/downloads/list and put it on the system path (see https://code.google.com/p/selenium/wiki/InternetExplorerDriver).
PhantomJS	PhantomJSDriver	A headless JavaScript browser that should work out of the box.
HtmlUnit	HtmlUnitDriver	Another headless JavaScript browser, faster than the others but less reliable with modern AJAX-based applications.
Opera	OperaDriver	A vendor-supported driver for the Opera browser written in Java. You just need to add a dependency on the <code>operadriver</code> library in your project. ²
Safari	SafariDriver	Implemented as a Safari browser extension.

Creating the `FirefoxDriver` instance will open a new Firefox window. You can open a specific page by using the `get()` method, as shown here:

```
driver.get("http://localhost:8080");
```

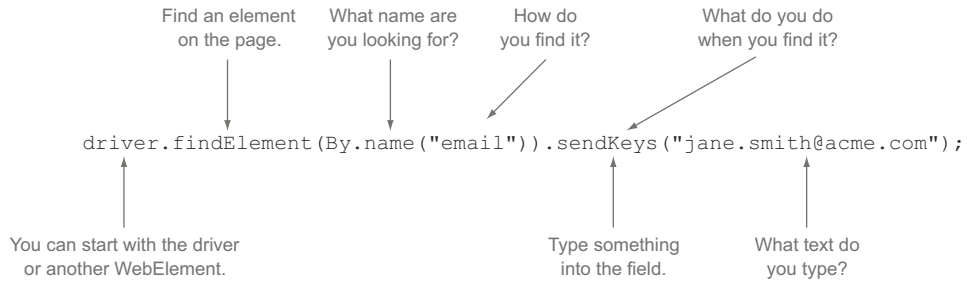
This will open the page shown in figure 8.5. Once the page is open, you can start to see how to identify and manipulate elements on the page.

8.2.2 Identifying web elements

The next step in the test involves entering the user's email address and password into the appropriate fields. In WebDriver, any object you'd like to inspect or manipulate in some way is represented by the `WebElement` class.

You can find a web element on a page using the `findElement()` method. This method uses a fluent API to identify objects in a very readable manner. For example, in the following code, you find the email field by looking for an HTML element with the `name` attribute set to `email`:

² Refer to <https://code.google.com/p/selenium/wiki/OperaDriver> for details about what versions of Opera are supported.



Once you have the element, you can query or manipulate it as required. In the preceding code line, you use the `sendKeys()` method to simulate a user typing something into the field. Later in the test, you click on the login button (identified by its `id` attribute) using the `click()` method:

```
driver.findElement(By.id("signin")).click();
```

Finally, at the end of the test, you check the text contents of the welcome message, conveniently identified by an `id` attribute:

```
WebElement welcomeMsg = driver.findElement(By.id("welcome-message"));
assertThat(welcomeMsg.getText()).isEqualTo("Welcome Jane");
```

One of the nice things about this API is that it's not only very readable, but it's very easy to use. In modern IDEs, the auto-complete feature can be used to list the available methods for the various objects and classes used in the WebDriver API (see figure 8.7). This makes the API both easy for new developers to learn and very productive for more experienced developers.

Identifying fields or objects with a given name or `id` attribute is the easiest and most robust way to obtain a web element; these attributes are less likely to change when the structure or style of the page changes. In fact, it's a good idea to make sure that all semantically significant elements in a page have a unique ID or name.

This example is relatively straightforward, with the fields and buttons being easy to find. In real-world applications, this isn't always the case, and there are some situations

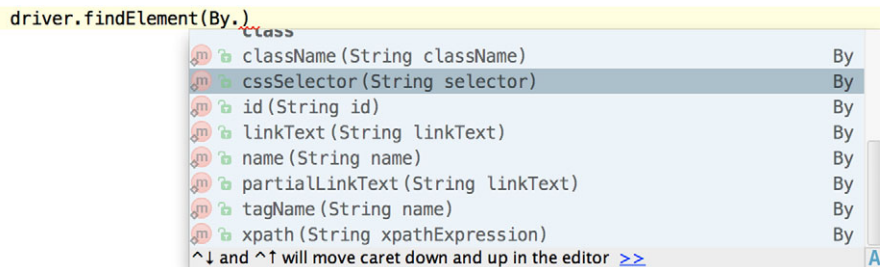


Figure 8.7 Modern IDE features, such as auto-completion, make the WebDriver API easy to work with.

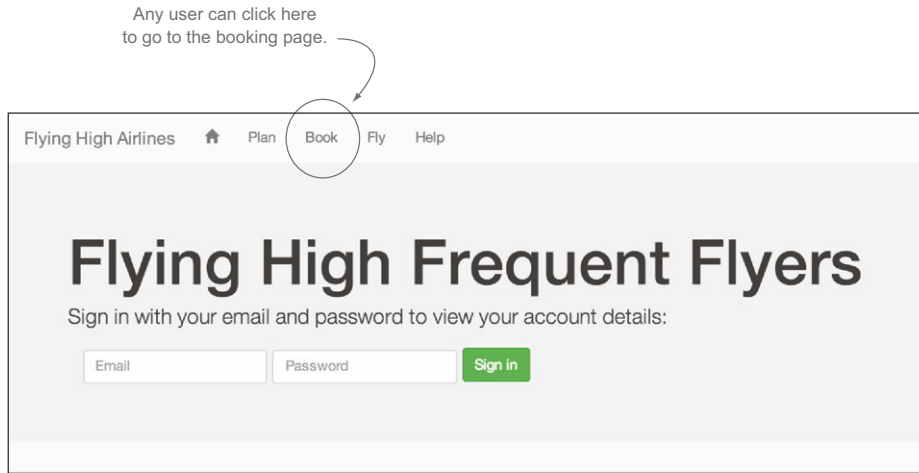


Figure 8.8 Identifying a hyperlink by its text

where other strategies are more convenient. Fortunately, WebDriver provides a number of other ways to identify web elements.

IDENTIFYING ELEMENTS BY LINK TEXT

When you write automated web tests, you often need to click on links, either to navigate to another page or to trigger some action. For example, on the Frequent Flyer site, a user can click on the Book link at the top of the page at any time to go to the booking page (see figure 8.8).

Links like this rarely have a name or id attribute that you can use to identify them. But you can use the next best thing: the text of the link itself. To click on the Book link, you could write the following:

```
driver.findElement(By.linkText("Book")).click();
```

You can also search the link texts for a partial match. To click on the Flying High Airlines link in the top-left corner of the page, the following call would work:

```
driver.findElement(By.linkText("Flying High")).click();
```

Identifying links by their text content is simple, intuitive, and relatively robust, though the test will obviously break if the displayed text is modified.

IDENTIFYING ELEMENTS USING CSS

A more flexible way of identifying elements is to use *CSS selectors*. CSS selectors are patterns designed to identify different parts of a web page for formatting and styling, but they're also a great general-purpose way to identify elements on the page.

Let's see how CSS selectors can be used for automated web testing. Suppose marketing has asked you to display a list of featured destinations on the home page, as shown in figure 8.9.

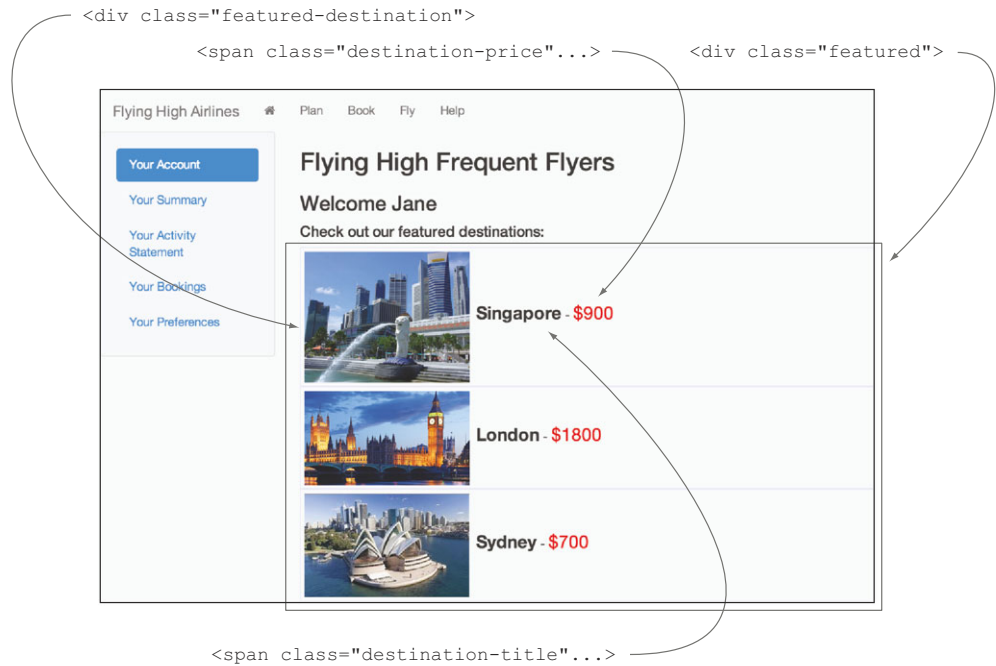


Figure 8.9 CSS selectors can come in handy when you're working with lists like this.

You can find web elements with CSS selectors by using the `By.cssSelector()` method. In CSS, the hash (#) symbol is used to find an object by its ID. To find the welcome message, you could do something like this:

```
driver.findElement(By.cssSelector("#welcome-message"));
```

Of course, this could be done more simply using `By.id()`. But CSS selectors become more valuable when you need to find web elements without clean id or name attributes. Some of the more useful CSS selectors are listed in table 8.2.

Table 8.2 Useful CSS selectors

Selector	Example	Notes
.class	.navbar	Matches all elements with the class navbar
#id	#welcome-message	Matches the element with an id of welcome-message
tag	img	Matches all the elements
element element	.navbar a	Matches all the <a> elements inside an element with the class navbar
element > element	.navbar-header > a	Matches <a> elements directly under an element with the class navbar-header

Table 8.2 Useful CSS selectors (continued)

Selector	Example	Notes
[attribute=value]	a[href="#/book"]	Matches <a> elements with an href value of #/book
[attribute^=value]	a[href^="#"]	Matches <a> elements with an href value that starts with #
[attribute\$=value]	a[href\$="book"]	Matches <a> elements with an href value that ends in book
[attribute*=value]	a[href*="book"]	Matches <a> elements with an href value that contains book
:nth-child(n)	.navbar li:nth-child(3)	Matches the third inside an element of class navbar

Let's look at a more practical example. One of the requirements you've defined with the marketing folk goes along the following lines:

Scenario: Displaying featured destinations
 Given Jane has logged on
 When Jane views the home page
 Then she should see 3 featured destinations
 And the featured destinations should include Singapore

On the Frequent Flyer home page, each featured destination appears inside a <div> element with the featured class. The destination title is nested inside a element with the featured-destination class. The rendered HTML code looks something like this:

```

<div id="featured">
  <div class="featured-destination">...</div>
  <div class="featured-destination">...</div>
  <div class="featured-destination">...</div>
</div>

```

Set the featured destination's title. (points to Singapore)

All the featured destinations appear within this <div>. (points to <div id="featured">)

Each featured destination has its own <div>. (points to <div class="featured-destination">...</div>)

Set the featured destination's image. (points to)

Other featured destinations like this follow. (points to <div class="featured-destination">...</div>)

In CSS, you can match elements with a given class by using the period (.) prefix. Using a CSS selector, you could find all of the <div> elements that represent the featured destinations like this:

```

List<WebElement> destinations
    = driver.findElements(By.cssSelector(".featured-destination"));
assertThat(destinations).hasSize(3);

```

1 Find all the featured destination elements. (points to By.cssSelector(".featured-destination"))

2 Check the number of matching featured destination elements. (points to assertThat(destinations).hasSize(3))

Note that you're using the `findElements()` method ❶ rather than the `findElement()` method you saw previously. As the name suggests, the `findElements()` method returns a list of matching web elements, rather than just a single one. You then check the size of the returned list, using the FEST-Assert library (<https://github.com/alexruiz/fest-assert-2.x>) to make the test more readable ❷.

This would be enough if you just wanted to count the number of featured destinations, but if you need to check the destination titles, you'll need to drill further. Fortunately, CSS selectors are flexible. You could retrieve the titles directly by finding all the web elements with the `destination-title` class:

```
driver.findElements(By.cssSelector(".destination-title"));
```

This would work, but it may not be robust. If destination titles were used elsewhere on the page, you'd retrieve too many titles. A safer approach would be to limit your search to the elements nested within the `<div>` that contains all of the featured destinations:

```
driver.findElements(By.cssSelector("#featured .destination-title"));
```

Once you have a list of matching web elements, you need to convert it to a list of strings that you can verify. You could write something like this:

Find all
featured
destination
title
elements.

❶

```
List<WebElement> destinations
    = driver.findElements(By.cssSelector("#featured .destination-title"));

List<String> destinationTitles = new ArrayList<String>();
for(WebElement destinationElement : destinations) {
    destinationTitles.add(destinationElement.getText());
}
assertThat(destinationTitles).contains("Singapore");
```

Convert these
web elements
into a list of
strings.

❷ Use `getText()` to extract
the string contents of
each element.

Check the contents
of the list.

First, retrieve the list of matching web elements ❶. To get the text content of a `WebElement`, you use the `getText()` method, so loop through the web elements and extract the text contents of each one ❷. Finally, check that the destination titles do indeed contain "Singapore."

We've just scratched the surface of CSS selectors here, but they're very useful for working with modern jQuery-based UI frameworks. You can find more details on the W3 web site (<http://www.w3.org/TR/CSS21/selector.html>). Most modern browsers have excellent native support for CSS selectors, which means that tests using CSS selectors will generally be very fast.

IDENTIFYING ELEMENTS USING XPATH

CSS selectors are flexible and elegant, but they do run into limits from time to time. A more powerful alternative is to use *XPath*. XPath is a query language designed to select elements in an XML document.

XPath expressions are path-like structures that describe elements within a page based on their relative position, attribute values, and content. In the context of WebDriver,

you can use XPath expressions to select arbitrary elements within the HTML page structure. A list of useful XPath expressions can be found in table 8.3.

Table 8.3 Useful XPath expressions

XPath expression	Example	Notes
node	a	Matches all of the <a> elements
//node	//button	Matches all of the <button> elements somewhere under the document root
//node/node	//button/span	Matches elements that are situated directly under a <button> element
[@attribute=value]	//a[@class='navbar-brand']	Matches <a> elements whose class attribute is exactly equal to navbar-brand
[contains(@attribute, value)]	//div[contains(@class, 'navbar-header')]	Matches <div> elements whose class attribute contains the expression navbar-header
node[n]	//div[@id='main-navbar']//li[3]	Matches the third inside the <div> with an id of main-navbar
[.=value]	//h2[.='Flying High Frequent Flyers']	Matches the <h2> element with text contents equal to Flying High Frequent Flyers

You can find an element via an XPath expression by using the `By.xpath()` method. You could find the welcome message heading by using the following expression:

```
driver.findElement(By.xpath("//h3[@id='welcome-message']"));
```

XPath requires more knowledge of the document structure than CSS, and it doesn't benefit from the intimate understanding of HTML that's built into CSS selectors. This makes simple selectors more verbose than their equivalents in CSS. For example, in the previous section you saw how you could find the list of featured destination titles using the following CSS selector:

```
driver.findElements(By.cssSelector(".destination-title"));
```

The equivalent in XPath might be something like this:

```
driver.findElements(By.xpath("//span[@class='destination-title']"));
```

This will find all of the `<span>` elements anywhere on the web page that have an attribute named `class` that's equal to `destination-title`.

You could make this more generic by using a wildcard (`*`) instead of `span`:

```
driver.findElements(By.xpath("//*[@class='destination-title']"));
```

This would find elements whose class was exactly equal to `destination-title`. Unfortunately, modern web applications will sometimes add extra classes to the `class` attribute, so you can't rely on an exact match. A more reliable solution would be to use the `XPath contains()` function, matching elements that have a `class` attribute with a value that contains `destination-title`:

```
driver.findElements(By.xpath("//*[contains(@class,'destination-title')]"));
```

The full power of XPath becomes more apparent when you need to find elements based on their content—something that's not currently supported in CSS. For example, the featured destinations you saw earlier are rendered in HTML like this:

```
<div class="featured-destination"...>
  </img>
  <span class="destination-title">Singapore</span>
  <span class="destination-price">$900</span>
</div>
<div class="featured-destination">...</div>
```

You could find the `<span>` element containing the text `Singapore` using the following XPath expression:

```
//span[.='Singapore']
```

You could take this further. Suppose you need to find the price displayed for the Singapore featured destination. XPath supports relative paths using the `..` notation, so you could find the neighboring `<span>` notation with a class of `destination-price` like this:

```
//span[.='Singapore']/../span[contains(@class,'destination-price')]
```

XPath isn't without its disadvantages. XPath expressions are generally more verbose and less readable than CSS selectors. XPath expressions can also be fragile if they aren't well-crafted. XPath has no native support in Internet Explorer, so tests that use XPath on Internet Explorer may run very slowly. But XPath is more powerful than CSS, and there are cases where XPath will be the only way to reliably identify the elements you're looking for.

USING NESTED LOOKUPS

When you write automated tests with WebDriver, it's important to keep expressions as simple and readable as possible. Simpler expressions tend to be easier to understand and to maintain, and in many cases they're more reliable. One useful strategy when it comes to writing simpler WebDriver code is to use nested lookups.

So far, you've found web elements using the WebDriver instance. But you can also call the `findElement()` and `findElements()` methods directly on `WebElement` instances.

For example, suppose there are several Book links at different places on the page, and you need to click on the Book link in the main menu. You could do this by first finding the main menu using its ID, and then finding the menu entry within the main menu's web element using the `linkText` selector:

```
driver.findElement(By.id("main-navbar"))
    .findElement(By.linkText("Book"))
    .click();
```

← Narrow down the search for elements to the main navbar.

← Click on the link.

← Look for a link within the navbar.

This approach is clear and intuitive and tends to be less error-prone than using complex XPath expressions or CSS selectors.

8.2.3 Interacting with web elements

Interacting with web elements in WebDriver is usually fairly intuitive, and it involves a relatively small number of methods. You can use the `click()` method on any web element (not just buttons and links) to simulate a mouse click. The `sendKeys()` method can be used to simulate user input. And the `getAttributeValue()` and `getText()` methods let you retrieve attribute values and the text contents of a web element.

The Frequent Flyer booking page illustrated in figure 8.10 has many fields that can illustrate these ideas.

TEXT INPUT FIELDS

To enter a value into a text field, you can use the `sendKeys()` method, as shown here:

```
driver.findElement(By.id("from")).sendKeys("Sydney");
```

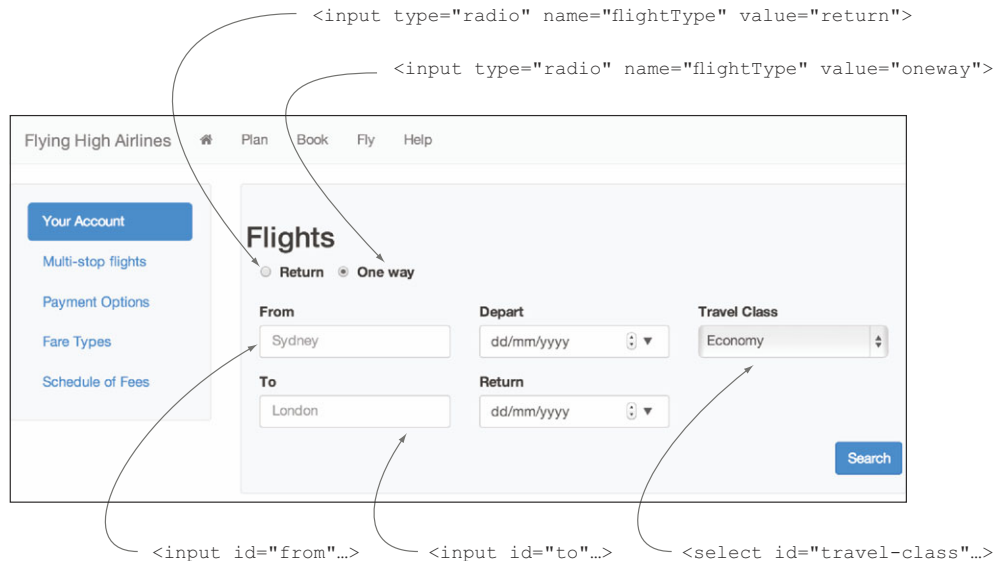


Figure 8.10 The Frequent Flyer booking page

The `sendKeys()` method doesn't set the value of the field; rather, it simulates the user typing the text into the field. If the field contains an existing value, you'll need to use the `clear()` method before entering the new value.

The current value of a text field is stored in the `value` attribute. To retrieve an attribute value, you can use the `getAttribute()` method, as shown here:

```
String fromValue = driver.findElement(By.id("from")).getAttribute("value");
```

This approach will also work for any other form field that uses the `value` attribute, such as check boxes or the newer HTML5 input field types like email and date. The exception is `<textarea>` fields, which don't have a `value` attribute. You can retrieve the contents of a `<textarea>` field by using the `getText()` method.

RADIO BUTTONS AND CHECK BOXES

The simplest way to select a radio button value is to find the radio button you want and to click on it. This can be a little tricky because the `name` attribute isn't unique and the `id` attribute isn't always defined or directly related to the value. You can do this by using a CSS selector that combines the `name` and `value` you want, like this:

```
driver.findElement(  
    By.cssSelector("input[name='flightType'][value='return']")).click()
```

This approach will also work for check boxes, which behave in exactly the same way.

DROP-DOWN LISTS

WebDriver provides a convenient helper class for dealing with drop-down lists. The `Select` class is used to wrap a web element representing a drop-down list to add drop-down-specific methods, such as `selectByVisibleText()`, `selectByValue()`, and `selectByIndex()`. On the booking page, you could set the travel class to "Business" using the following code:

```
WebElement travelClassElt = getDriver().findElement(By.id("travel-class"));  
new Select(travelClassElt).selectByVisibleText("Business");
```

The `Select` class also provides a number of methods that you can use to learn about the current state of the drop-down list, including `getFirstSelectedOption()` and `getAllSelectedOptions()`.

8.2.4 Working with asynchronous pages and testing AJAX applications

Most modern web applications use AJAX in one way or another. AJAX-based JavaScript libraries allow developers to write applications with vastly improved usability and user experience. But the asynchronous nature of AJAX can present challenges when it comes to automated web testing.

In a conventional web application, when you click on a link or submit a form, an HTTP request is sent to the server and a new page is returned. In these cases, WebDriver will automatically wait for the new page to load before proceeding. But with an AJAX application, the web page will send queries to a server and update the page

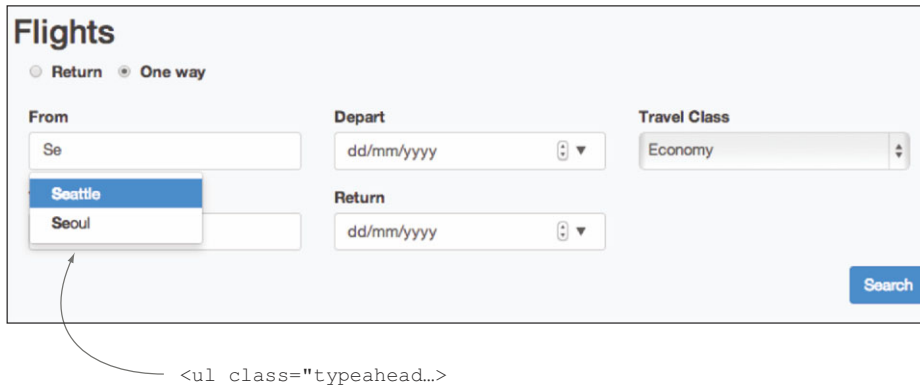


Figure 8.11 Type-ahead fields are a common example of asynchronous lookups.

directly, without reloading. When this happens, WebDriver will not know if or when it needs to wait for updates, which may cause the test to fail unexpectedly.

Fortunately, it's relatively easy to tell WebDriver when it needs to wait and what it should wait for. Your main tools for achieving this are the `Wait` interface and its implementations.

Figure 8.11 illustrates a common case where waiting may be useful. On the Frequent Flyer booking page, the `From` and `To` fields are configured with a type-ahead capability that displays matching cities in a drop-down list as the user types.

The time it takes to display the drop-down list can be unpredictable, as it may depend on the speed of the network and the target server. Depending on the nature of the web page, there are several ways you can handle this sort of delay.

USING THE IMPLICIT WAIT

By default, if WebDriver doesn't find a web element, it will fail immediately. But this behavior is configurable. For example, if you wanted WebDriver to wait for five seconds before declaring forfeit, you could set the implicit wait time to five seconds, as shown here:

```
driver.manage().timeouts().implicitlyWait(5, TimeUnit.SECONDS);
```

This is a blunt tool, as it will apply every time WebDriver looks up an element, which can slow down the tests in other places. A more refined approach is to use *explicit waits*.

USING EXPLICIT WAITS

WebDriver also lets you wait for specific events. The easiest way to do this is to use the `ExpectedConditions` class. This class provides a large number of useful predefined conditions that make the API more convenient to use. These include waiting for elements to be present (or not present), visible (or invisible), clickable, and so forth. For example, the following code waits until the type-ahead list (identified here by its class name) is present on the screen:

When the expected condition is met, obtain this web element.

```
WebDriverWait wait = new WebDriverWait(getDriver(), 5);
WebElement typeaheadList =
    wait.until(ExpectedConditions.presenceOfElementLocated(
        By.className(".typeahead")));
```

Wait for up to five seconds.
Wait until this element is present on the page.

These predefined conditions cover many common situations. But occasionally you'll need to do something more specific. Again, WebDriver offers several options. The `FluentWait` class allows you to create arbitrary wait parameters on the fly using a readable fluent API:

```
Wait<WebDriver> wait = new FluentWait<WebDriver>(getDriver())
    .withTimeout(30, TimeUnit.SECONDS)
    .pollingEvery(1, TimeUnit.SECONDS)
    .ignoring(NoSuchElementException.class);
```

Don't fail if an element isn't found yet.

Wait for up to 30 seconds.

Check the page every second.

You can use this wait object with one of the predefined conditions from the `ExpectedConditions` class, or you can write your own condition. A condition takes the form of a Function object (from the Google Guava library, <https://code.google.com/p/guava-libraries/>), and it typically returns either a `WebElement` (if you're waiting for a web element to become available) or a `Boolean` (if you're waiting for some more general condition). In the following example, you wait until the type-ahead list is present on the page and contains entries:

All Function implementations must implement the `apply()` method.

```
wait.until(typeaheadIsNotEmpty());
...
private Function<WebDriver, Boolean> typeaheadIsNotEmpty() {
    return new Function<WebDriver, Boolean>() {
        public Boolean apply(WebDriver driver) {
            List<WebElement> typeaheadItems
                = driver.findElements(By.cssSelector(".typeahead li"));
            return typeaheadItems.size() > 0;
        }
    };
}
```

1 Wait until the type-ahead list is both present and contains entries.

2 This function takes a WebDriver instance and returns a Boolean.

Return true if the type-ahead list is present and isn't empty. 4

In this code sample, you create your own implementation of the `Function` interface to check the presence and size of the type-ahead list. The `Function` interface here represents a function that takes a `WebDriver` instance and returns a `Boolean` value 2. When this `Boolean` value is true, the test will be able to continue 1. The `Function` interface defines the `apply()` function 3, which you need to implement. In this case, the condition is relatively simple: the implementation retrieves and checks the size of the type-ahead list 4. This implementation could be more complicated if you need to check more involved conditions.

8.2.5 Writing test-friendly web applications

The style and quality of your web application code has a significant influence on how easy or hard it will be to test. Applications with clean HTML code, identifiers, names, and CSS classes for all the significant elements on a page make testing easier

and more reliable. When applications have messy or inconsistent HTML code, the elements can be hard to identify, which results in more complicated and more brittle selector logic.

The technology stack you choose can have a major effect on testability. Frameworks that limit the control you have over the rendered HTML are a major source of difficulty. In Java web application development, for example, some frameworks automatically generate element identifiers for their own use, making it difficult to use the simplest and fastest of the WebDriver selectors.³ Applications that use plugin technologies such as Flash and Silverlight, which are opaque to testing tools like WebDriver, also make testing very difficult.

8.3 Using page objects to make your tests cleaner

Up until now we've explored the WebDriver API using simple code samples. These examples work well to illustrate the various WebDriver methods, but you wouldn't want to write code like this for real-world automated tests. For example, to log on to the Frequent Flyer website, you used the following code:

```
driver.get("http://localhost:8080/#/welcome");
driver.findElement(By.name("email")).sendKeys("jane.smith@acme.com");
driver.findElement(By.name("password")).sendKeys("s3cr3t");
driver.findElement(By.id("signin")).click();
```

This code wouldn't scale well. You'd need to duplicate the same or similar lines for every scenario involving a user logging on, and any change to this logic would need to be updated at every place that it's used.

Another problem is that you're mixing selector logic (`By.name()` and so on) with test data (`jane.smith@acme.com` and so forth), which prevents you from reusing the locators in other tests.

A better approach would be to refactor the selector logic into one place so that it can be reused across multiple tests. You could write a class along the following lines to do this:

```
public class LoginPage {
    private final WebDriver driver;

    public LoginPage(WebDriver driver) {
        this.driver = driver;
    }

    public void open() {
        driver.get("http://localhost:8080/#/welcome");
    }

    public void signinWithCredentials(String userEmail,
                                     String userPassword) {
```

1 Declare a **WebDriver** instance that the page object can use to interact with the browser.

2 Pass in the **WebDriver** instance.

3 Open the login page.

4 Enter user credentials and sign in to the application.

³ Many JSF-based frameworks fall into this category.

```

        driver.findElement(By.name("email")).sendKeys(userEmail);
        driver.findElement(By.name("password")).sendKeys(userPassword);
        driver.findElement(By.id("signin")).click();
    }
}

```

5 Web element selector details.

This class wraps the lines of code you saw earlier **5** into a single method called `signInWithCredentials()` **4**. It also provides an `open()` **3** method to get to the right page. The code needs a `WebDriver` instance **1**, which is provided in the constructor **2**.

Now your test code can focus on the test data and the intent of the actions, rather than on how the individual web elements are found or manipulated:

```

LoginPage loginPage = new LoginPage(driver);
loginPage.open();
loginPage.signInWithCredentials("jane.smith@acme.com", "s3cr3t");

```

Open the login page.

Sign in with the specified credentials.

8.3.1 Introducing the Page Objects pattern

The class we just looked at follows the Page Objects pattern. A page object is a class that models a web page, or part of a web page, and that presents a set of business-focused methods for tests to use, sparing them from the implementation details of the actual HTML page. A page object has two main roles:

- It isolates the technical implementation of the page from the tests, making the test code simpler and easier to maintain.
- It centralizes the code that interacts with a page (or a page component), so that when the web page is modified, the test code only needs to be updated in one place.

In section 7.3, we discussed the typical layers that make up a well-structured automated acceptance test suite:

- The *Business Rules* layer describes the expected business outcomes.
- The *Business Flow* layer relates the user's journey through the application.
- The *Technical* layer interacts directly with the system.

Page objects belong in the Technical layer (see figure 8.12). They provide business-friendly services to the Business Flow layer and implement the interactions with the web pages using the WebDriver API.

Page objects don't have to represent an actual page. In some cases, such as the login page discussed earlier, it makes sense to have a page object dedicated to a page. In other cases, such as a modern JavaScript single-page application, a single HTML page might be represented by page objects for each state or view of the application.

It also makes sense to use page objects to represent important parts of a screen, particularly if those parts are reused from screen to screen. For example, you might use a page object to represent the main menu bar that appears on every screen, or for the list of featured destinations if this appears in several places. These component objects can be nested within other page objects or used independently.

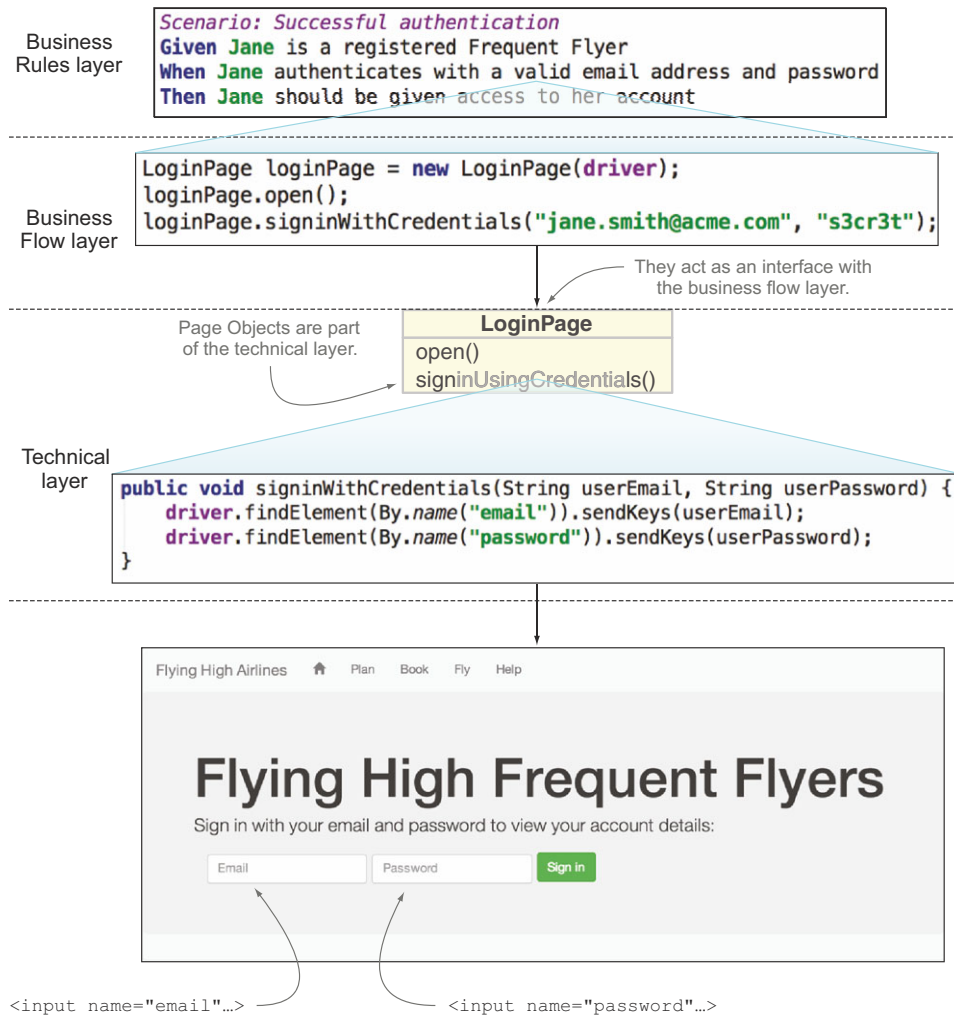


Figure 8.12 Page objects are an important part of the Technical layer.

PAGE OBJECTS IN WEBDRIVER

Although you can write your own page objects from the ground up, it's nice to have some tooling support. The WebDriver API provides excellent built-in support for page objects. In particular, it provides the `@FindBy` annotation to simplify web element lookup. Using this annotation, you could rewrite the `LoginPage` page object like this:⁴

```
public class LoginPage {
    @FindBy(name="email")
    WebElement email;
```

❶ Look up the email field.

⁴ The `open()` method has been excluded from the listing for simplicity.

```

@FindBy(name="password")
WebElement password;

public LoginPage(WebDriver driver) {
    PageFactory.initElements(driver, this);
}

public void signinWithCredentials(String userEmail,
                                  String userPassword) {
    email.sendKeys(userEmail);
    password.sendKeys(userPassword);
}
}

```

② Look up the password field.

③ Initialize the email and password fields.

④ Look up the web elements on the web page before using them.

The `@FindBy` annotation tells `WebDriver` how to look up a `WebElement` field ①, ②. When you mark fields this way, you can use the `PageFactory.initElements()` method in the constructor ③ to instantiate these fields for you. Each time you use these fields ④, `WebDriver` performs the equivalent of a `driver.findElement(By...)` call to bind them to the corresponding element on the web page. The `@FindBy` annotation supports all of the different selector methods available when you use `driver.findElement(By...)` (see table 8.4).

Table 8.4 Different ways to use the `@FindBy` annotation

@FindBy expression	Description
<code>@FindBy(id="welcome-message")</code>	Find by ID
<code>@FindBy(name="email")</code>	Find by name
<code>@FindBy(className="typeahead")</code>	Find by CSS class name
<code>@FindBy(css=".typeahead li")</code>	Find by CSS selector
<code>@FindBy(linkText="Book")</code>	Find by link text
<code>@FindBy(partialLinkText="Book")</code>	Find by partial link text
<code>@FindBy(tagName="h2")</code>	Find by HTML tag
<code>@FindBy(xpath="//span[.='Singapore']")</code>	Find by XPath expression

If the name of the `WebElement` fields in your page object matches either the name or ID of the corresponding HTML element, you can skip the `@FindBy` annotation entirely:

```

public class LoginPage {
    WebElement email;
    WebElement password;

    public LoginPage(WebDriver driver) {
        PageFactory.initElements(driver, this);
    }
    ...
}

```

① You don't need the `@FindBy` annotation here.

In this case, WebDriver automatically instantiates the email and password fields ❶. For the email field, for example, this is the equivalent of first trying `@FindBy(id="email")`, and if that fails, `@FindBy(name="email")`.

The `@FindBy` annotation isn't limited to individual fields; you can also use this notation to retrieve collections of web elements. All you need to do is define your field as a list of `WebElements` instead of a simple `WebElement`, as shown here:

```
@FindBy(css = ".featured .destination-title")
private List<WebElement> featuredDestinations;
```

Match a list of destination titles on the web page.

Each `WebElement` represents an individual title.

WebDriver also provides the `@FindBy`s annotation, which can be used to define nested `@FindBy` annotations.

```
@FindBy({@FindBy(id="main-navbar"), @FindBy(linkText = "Book")})
WebElement bookMenu;
```

This is the equivalent of the nested `findElement()` methods we saw in section 8.2.2:

```
WebElement bookMenu = driver.findElement(By.id("main-navbar"))
    .findElement(By.linkText("Book"))
```

By default, if WebDriver can't find an element on the page to match a `WebElement` field in your page object, it will fail immediately. This is often a good thing, because if WebDriver can't find an element it was expecting to find, then either your test or the application is likely to be broken, and you should be notified about this as quickly as possible. But if you're working with dynamic pages using AJAX-based web elements, you might need to give the application some time for a dynamic field to appear before failing the test.

The type-ahead feature we looked at earlier is a good example of this. You could use an `@FindBy` annotation like the following to retrieve these values. The corresponding page object might look something like this:

```
public class BookingPage {
    @FindBy(css = ".typeahead li")
    private List<WebElement> typeaheadEntries;

    public BookingPage(WebDriver driver) {
        PageFactory.initElements(driver, this);
    }

    public List<String> getTypeaheadEntries() {
        List<String> entries = new ArrayList<String>();
        for(WebElement typeaheadElement : typeaheadEntries) {
            entries.add(typeaheadElement.getText());
        }
        return entries;
    }
    ...
}
```

Type-ahead entries will be stored here.

Initialize the `typeaheadEntries` field.

❶ Extract the text contents of the type-ahead web elements and return them as a list of Strings.

The first time you use this variable in your page object ❶, WebDriver tries to find the matching HTML elements on the page. But due to the asynchronous nature of this feature, the list may not be populated yet, and your test will fail.

To get around this problem, you can configure WebDriver to not fail immediately when it can't find an element, but rather to poll the page repeatedly for a predefined period. You do this by using the `AjaxElementLocatorFactory` class when you initialize the web elements, as illustrated here:

If an element isn't found, poll the web page for up to five seconds.

```
public class BookingPage {
    @FindBy(css = ".typeahead li")
    private List<WebElement> typeaheadEntries;

    public BookingPage(WebDriver driver) {
        PageFactory.initElements(new AjaxElementLocatorFactory(driver, 5),
                                this);
    }

    public List<String> getTypeaheadEntries() {
        List<String> entries = new ArrayList<String>();
        for(WebElement typeaheadElement : typeaheadEntries) {
            entries.add(typeaheadElement.getText());
        }
        return entries;
    }
    ...
}
```

Look up the type-ahead entries, with polling if necessary.

Now, if the type-ahead elements aren't ready straightaway, WebDriver will keep trying for up to five seconds before failing.

In this particular application, this approach will work just fine, but it's not foolproof. For example, suppose you have a checkout page with a total field that's present on the screen but initially contains no data. When the user selects an option (such as opting for travel insurance), an AJAX call will update this field. The web element corresponding to the total field will always be present on the page, but you need to wait for it to be populated with the correct values. In this case, the polling approach won't work, and using the `Wait` interface discussed in section 8.2.4 would be a better strategy.

8.3.2 Writing well-designed page objects

Page objects are highly reusable components, and if they're designed well, they can make your automated tests significantly easier to understand and maintain. A few simple rules can help you go a long way in designing better page objects.

PAGE OBJECTS SHOULD ONLY EXPOSE SIMPLE TYPES AND DOMAIN OBJECTS

The most important design rule for page objects is that a page object should never expose implementation details about the page or component it's encapsulating. Page object methods should accept and return simple types such as strings, dates, Booleans, domain objects, or other page objects. They should never expose WebDriver classes.

For example, the Frequent Flyer flight booking page (see figure 8.13) has many fields that could be encapsulated behind page object calls.

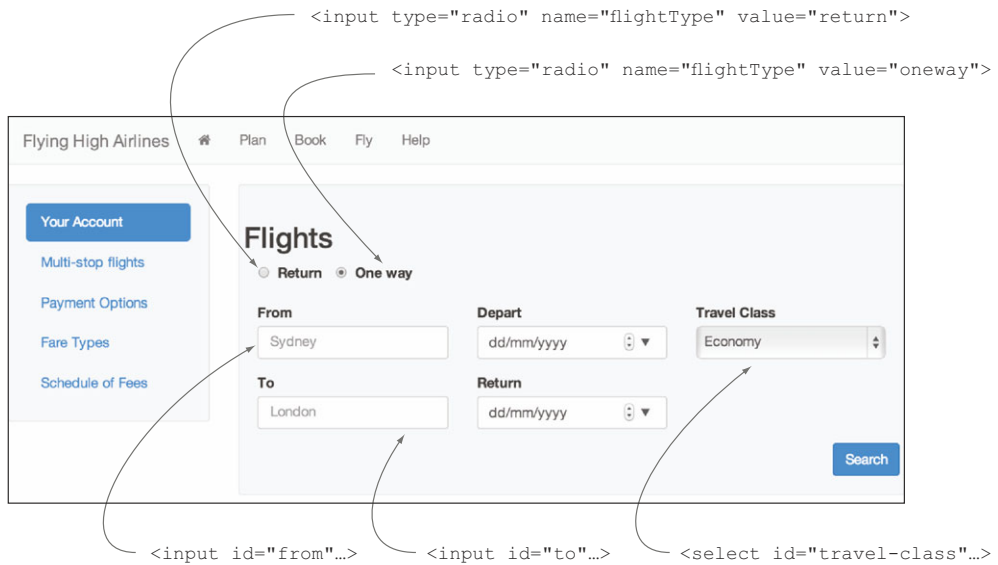


Figure 8.13 The flight booking page has a number of fields that could be read and written to via a page object class.

For example, the from and to fields could be exposed as simple String values:

```
WebElement from;                                     ← The from input field.

public BookingPage setFrom(String fromCity) {
    from.sendKeys(fromCity);
    return this;
}

public String getFrom() {
    return from.getAttribute("value");
}

// Set the from value using a String.
// Return the page object to make it easier to chain set() methods.
// Retrieve the value as a String.
```

In a similar manner, the depart and return fields might be presented as dates:

```
WebElement depart;                                     ← Declare the departure date field.

public BookingPage setDepart(DateTime departDate) {
    depart.sendKeys(departDate.toString("dd/MM/yyyy"));
    return this;
}

public DateTime getDepart() {
    return DateTime.parse(depart.getAttribute("value"),
        DateTimeFormat.forPattern("dd/MM/yyyy"));
}

// Return a DateTime value by parsing the contents of the date field.
// Set the departure date field by formatting the date value.
```

Lists of values should be returned as lists of primitive types (strings, dates, numbers, and so on) or as lists of domain objects. You saw an example of this with the type-ahead values:

```

public List<String> getTypeaheadEntries() {
    List<String> entries = new ArrayList<String>();
    for(WebElement typeaheadElement : typeaheadEntries) {
        entries.add(typeaheadElement.getText());
    }
    return entries;
}

```

Well-designed page objects also accept and return data in the form of domain objects when it makes sense to do so. In many cases, domain objects allow tests to be more expressive and readable than primitive types. For example, suppose you need to verify that the featured destination list contains a deal for Singapore costing \$900 (see figure 8.14). You represent destination deals using a simple Java class called `DestinationDeal`, with a destination and a price.

If you design your page object to return a list of `DestinationDeals`, you could write test code like this:

Create a domain object representing the expected destination deal.

```

DestinationDeal expectedDeal = new DestinationDeal("Singapore", 900);
assertThat(homePage.getFeaturedDestinations()).contains(expectedDeal);

```

Check that the featured destinations on the home page contain the expected deal.

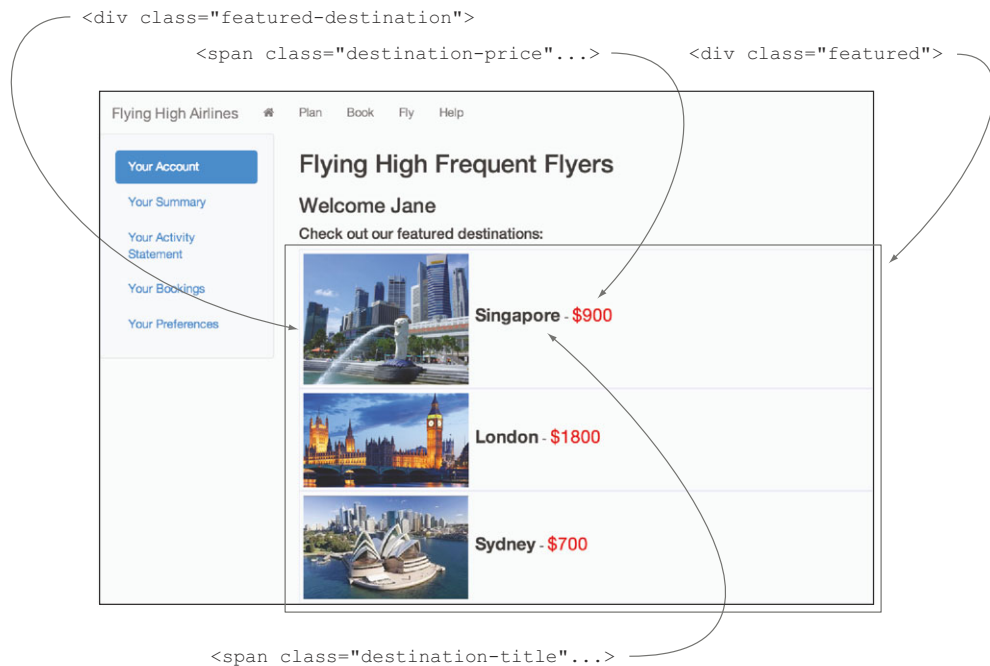


Figure 8.14 The featured destinations in this screen could be represented as a list of domain objects.

The page object needs a `getFeaturedDestinations()` method that returns a list of `DestinationDeals`. Internally, this method needs to convert the data in the featured destinations list into a list of `DestinationDeals`.

Let's walk through how you might implement this method. To start, you could use the `@FindBy` annotation to retrieve the list of featured destination `<div>` blocks:

```
@FindBy(css = ".featured .featured-destination")
private List<WebElement> featuredDestinations;
```

Next, you'd write the `getFeaturedDestinations()` method itself:

Populate the list by converting the web elements to `DestinationDeals`.

```
public List<DestinationDeal> getFeaturedDestinations() {
    List<DestinationDeal> deals = Lists.newArrayList();
    for(WebElement destinationEntry : featuredDestinations) {
        deals.add(destinationDealFrom(destinationEntry));
    }
    return deals;
}
```

Return a list of `DestinationDeals`.

Create a new empty list.

The code here loops through the web elements representing the featured destination `<div>` blocks and converts each to a `DestinationDeal` object. The details of the conversion process are left to the `destinationDealFrom()` method, shown here:

Retrieve the destination title.

1

Retrieve the destination price.

2

```
private DestinationDeal destinationDealFrom(WebElement destinationEntry) {
    String destinationCity = destinationEntry.findElement(
        By.className("destination-title")).getText();
    String priceValue = destinationEntry.findElement(
        By.className("destination-price")).getText();
    int price = Integer.parseInt(priceValue.substring(1));
    return new DestinationDeal(destinationCity, price);
}
```

Convert the price to an integer.

Create a new `DestinationDeal` using these values.

3

This method reads the title 1 and price 2 from the web element and uses them to build a new `DestinationDeal` 3.

PAGE OBJECTS SHOULD REPORT ON PAGE STATE

Page objects are responsible for reporting information about the contents or state of the page (or page component) to the test. The test can then use this information to perform any required checks and assertions; page objects shouldn't contain assertion logic.

For example, on the flight booking page in figure 8.13, there's a Search button. Suppose that this button is disabled until all of the required fields are filled. To test this, you could add a method with an embedded assertion to the corresponding page object, as shown here:

```
public class BookingPage {
    ...
    WebElement search;
    public void searchButtonShouldBeEnabled() {
        assertThat(search.isEnabled(), is(true));
    }
}
```

Represents the Search button.

This will fail if the Search button isn't enabled.

NOTE The preceding snippet uses Hamcrest, the other major Java fluent assertion library.

This isn't a very clean solution. The page object isn't only reporting the state of the page; it's also making assertions about it, which should be the responsibility of the test logic. A better solution would be to have the page object return the state of the Search button, and let the test do the asserting:

```
public class BookingPage {
    ...
    WebElement search;

    public boolean searchButtonIsEnabled() {
        return search.isEnabled();
    }
}
```

Just return the state of the Search button.

Now the test code will be able to decide what assertion is the most appropriate:

```
assertThat(bookingPage.searchButtonIsEnabled(), is(true));
```

This approach provides a better separation of concerns and avoids the risk of having bloated page object classes with assertion methods that are only used by a single test.

There's still one area where you could improve this example. Any test that simply asserts that something is true (or false) runs the risk of providing poor diagnostic information. For example, both of the assertions in the preceding examples will report the following error message if they fail:

```
Expected: is <true>
but: was <false>
```

When a test fails, it's important for diagnostic messages to be informative, and most assertion libraries let you add customized messages. For example, you could rewrite the earlier test to provide a more informative message, as shown here:

```
assertThat("Search button should be enabled",
    bookingPage.searchButtonIsEnabled());
```

There's still a little duplication in this code. Some WebDriver-based libraries also provide extensions that can produce more useful assert messages with even less effort. For example, in Thucydides, you can use the `WebElementState` class to return the state of a web element to the test. You could use a Thucydides page object for this page like this:

```
public class BookingPage extends PageObject {
    WebElement search;
    ...
    public WebElementState searchButton() {
        return $(search);
    }
}
```

The page object extends the Thucydides PageObject base class.

The `$()` method returns information about the current state of the web element.

The `WebElementState` class provides a large number of methods that tests can use to query and make assertions about the state of a web element. For example, using this class, you could rewrite the test logic like this:

```
bookingPage.searchButton().shouldBeEnabled();
```

If this should fail, the error message will be relatively descriptive:

```
Field '<button id='search'>' should be enabled
```

This way, you can have the best of both worlds: flexible, reusable assertions about the state of your pages that you can place in the test logic, and useful error messages when a test fails.

NAVIGATING WITH PAGE OBJECTS

If an action on a page object causes the application to switch to another page, it's sometimes useful to have the page object return a page object representing the new page. For example, you might write a page object to represent the main menu bar that appears at the top of the page:

```
public class MainMenu {
    ...
    public PlanningPage selectPlanFlight() {...}
    public BookingPage selectBookFlight() {...}
    public FlyPage selectFly() {...}
}
```

You could now write fluent and readable test code like this:

```
mainMenu.selectBookFlight().setFrom("Sydney").setTo("Sydney").search();
```

This approach is often a matter of style. It can make the tests quicker and easier to write, but it may mask the navigation logic through the application and make maintenance harder if the navigation logic changes. It's also debatable whether it's the responsibility of the page object to understand the application's navigation logic, or whether it should be left to the test logic to declare what page it expects to see at any point in the test. Many practitioners prefer to leave the test logic to implicitly describe the expected navigation by using the page object that corresponds to the screen they expect to be on.

If there's a possibility that the action won't always go to another page, or that it may go to multiple pages depending on application-logic considerations, then it's unwise to code this navigation logic into your page objects. For example, a login page will go to a welcome page if the login succeeds or stay on the login page if an authentication error occurs. In this case, the page object will have to decide which page it should navigate to, which is effectively embedding business logic into the page object, and if this business logic changes, your page object will break.

WebDriver is an excellent foundation for automated web tests, and many other open source libraries have been built around WebDriver for more advanced or specific usages. In the next section, we'll look at a few of these.

8.3.3 Using libraries that extend WebDriver

As we mentioned in section 8.2, there are many libraries for different platforms that build on and extend WebDriver, including these:

- *Thucydides* (<http://thucydides.info>)—Provides powerful WebDriver support for Java and Groovy tests
- *Watir* (<http://watir.com>)—A powerful Ruby DSL for WebDriver testing
- *WatiN* (<http://watin.org>)—Provides support for automated web testing in C#
- *Geb* (<http://gebish.org>)—A Groovy DSL for WebDriver

Rather than trying to cover each library in detail, I'll highlight two of the more compelling features from some of these libraries that make them so popular among test-automation practitioners.

PAGE OBJECTS

All of these libraries provide support for page objects in various forms, making page objects easier to define and use.

Thucydides takes care of instantiating page objects and managing the WebDriver instance, and it provides a number of useful base methods. A simple Thucydides page object looks similar to the standard WebDriver code we've discussed so far:

Thucydides page objects extend the PageObject class.

```
@DefaultUrl("http://localhost:8080/#/welcome")
public class LoginPage extends PageObject {

    private WebElement email;
    private WebElement password;
    @FindBy(css = ".btn[value='Sign in']")
    private WebElement signin;

    public void signinAs(String userEmail,
                        String userPassword) {
        email.sendKeys(userEmail);
        password.sendKeys(userPassword);
        signin.click();
    }
}
```

Open the page here by default.

Web elements are defined and used in the normal way.

Thucydides will also instantiate any page object instances in the test code, which simplifies the test code further:

```
LoginPage loginPage;
HomePage homePage;
...
loginPage.open();
loginPage.signinAs(user.getEmail(), user.getPassword());

String welcomeMessage = homePage.getWelcomeMessage();
assertThat(welcomeMessage).isEqualTo("Welcome Jane");
```

Page objects are instantiated automatically.

Open the page at the URL defined by the @DefaultUrl annotation.

Specify the action under test.

Verify the expected outcomes.

WatiN also provides support for page objects in C#. A page object written using WatiN looks like this:

WatiN page objects extend the Page base class.

```

[Page(UrlRegex = "localhost:8080/#/welcome")]
public class LoginPage : Page
{
    public TextField Email
    {
        get { return Document.TextField(Find.ByName("email")); }
    }
    public TextField Password
    {
        get { return Document.TextField(Find.ByName("password")); }
    }
    public Button SigninButton
    {
        get { return Document.TextField(
            Find.ByCSSSelector(".btn[value='Sign in']")); }
    }
    public void SignInAs(String userEmail, String userPassword) {
        Email.TypeText(userEmail);
        Password.TypeText(userPassword);
        SigninButton.Click();
    }
}

```

The UrlRegex is used to check that a test is looking at the right page.

TextFields, Buttons, and so on are used to encapsulate the web elements.

The corresponding test code might look something like this:

```

using (var browser = new IE("http://localhost:8080"))
{
    var loginPage = browser.Page<LoginPage>();
    loginPage.SignInAs("jane@acme.com", "s3cr3t");

    var homePage = browser.Page<HomePage>();
    var message = homePage.WelcomeMessage.Text
    Assert.That(message, Is.EqualTo("Welcome Jane"));
}

```

Specify the action under test.

Set up the browser.

Create the page objects.⁵

Check the test outcomes.

As in the Thucydides code from the previous example, the WatiN page objects neatly hide the web page implementation details from the test logic, making the test code cleaner, more concise, and more readable.

FLUENT SELECTORS

Dynamic languages like Ruby and Groovy make it easy to write fluent and readable APIs, and these can be used to write concise, readable expressions for identifying web elements on a page. Geb provides a powerful expression language based on a jQuery-like notation that allows you to write expressions using CSS selectors. The following code shows a few examples of this sort of expression:

```

$("#search").click()
assert $("#search").disabled
$(".featured-destination").find("a")
$(".featured-destination:nth-child(2) a").click()

```

Click on the link in the second featured destination entry.

Click on the search button.

Verify that the search button is disabled.

Find all of the <a> elements inside featured destinations.

⁵ For conciseness, we haven't included the implementation of the HomePage page object here.

```
List<String> menuHeadings = $(".main-navbar").find("li")
                                .collect {it.text()}
```

Retrieve the text values for each main menu item. 5

In ❶, you use a simple CSS selector to find and click on the search button. In the second example ❷, you check whether this search button is disabled. The next example ❸ will find any <a> elements nested inside a featured destination block. The fourth example ❹ uses a more complex CSS expression to identify and click on the link in the second featured destination block. The final example ❺ fetches all of the elements in the main menu and converts them into a list of Strings.

In Java, Thucydides provides its own support for fluent selectors. Though not as extensive as the Geb API, Thucydides still offers quite a few to choose from. The Thucydides equivalent of the Geb expressions shown above would look like this:

```
$("#search").click();
assertThat($("#search").isEnabled()).isFalse();
findAll(".featured-destination a");
$(".featured-destination:nth-child(2) a").click();
$(".main-navbar").thenFindAll("li")
```

Click on the search button.

Verify that the search button is disabled.

Find all of the <a> elements inside featured destinations.

Click on the link in the second featured destination entry.

Retrieve a list of all of the main menu entries. ❶

Thucydides also provides a jQuery-like \$() method, so many of the expressions look similar to the Geb equivalents. Java doesn't have closures, so in the last line ❶ you can use the LambdaJ library to do something similar.

When used well, fluent APIs like these are a boon to readability and expressiveness.

8.4 Summary

In this chapter you learned to automate UI tests for web-based applications:

- Automated web tests are a powerful testing tool, but one that should be used sparingly in order to avoid slowing down your test suite too much.
- The WebDriver API provides a rich and robust foundation for automated web testing.
- The Page Objects pattern can make your web tests cleaner and easier to maintain.
- Libraries, such as Thucydides, Watir, WatiN, and Geb, build on and extend WebDriver with additional features.

In the next chapter we'll focus on techniques for testing the non-UI layers of the application.



Automating acceptance criteria for non-UI requirements

This chapter covers

- Balancing UI and non-UI acceptance criteria
- Where to use non-UI acceptance tests
- Automating acceptance tests for the controller layer of a web application
- Automating acceptance tests that test application code directly
- Automating acceptance tests for remote services
- Automating acceptance tests for nonfunctional requirements
- Discovering application design using non-UI acceptance tests

Although they have their uses, web tests shouldn't be the only tool in your automated acceptance testing toolbox. It's important to know when to use them and when to look for alternative strategies. In this chapter, you'll learn about other ways to automate your acceptance tests that don't involve exercising the user interface (see figure 9.1).

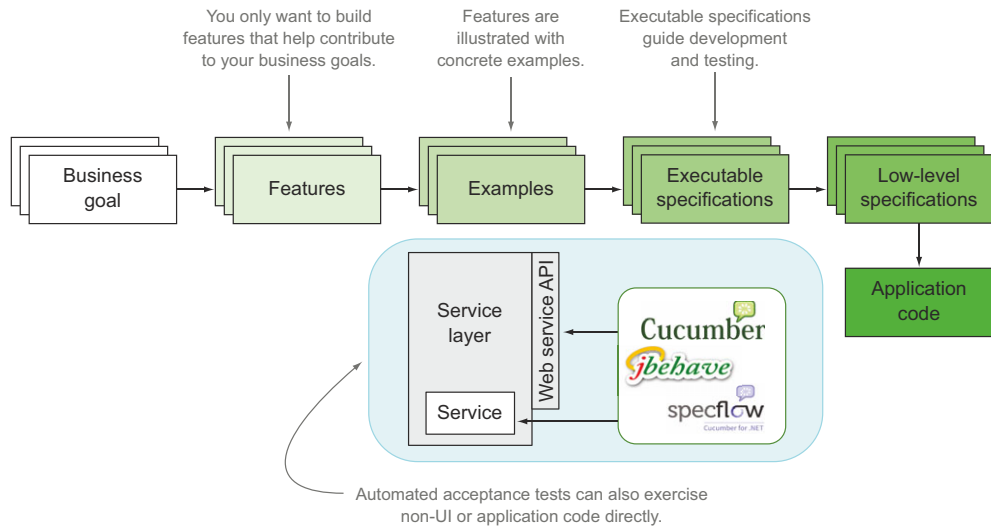


Figure 9.1 In this chapter we'll focus on automating acceptance tests that exercise the non-UI components of your application.

In chapter 8, you learned how to automate acceptance tests using automated web testing tools such as Selenium 2/WebDriver. Web tests are a great way to simulate the user's journey through the system, to illustrate interactions with an application, and to document the key features of your application. Stakeholders can easily relate to web tests, as they're highly visual, intuitive, and map closely to the user experience. Automated web tests can also be used to demonstrate new features, which is a great way to increase confidence in the tests. Automated web tests are also the only way to effectively test business logic that's implemented directly within the user interface (UI).

But you also saw that end-to-end web tests tend to execute significantly more slowly than tests that don't involve the UI. Interacting with a browser adds significant overhead in terms of execution time and system resources. Web tests that rely on a real browser are also more subject to technical or environment-related issues that are hard to control. For example, web tests can fail because the wrong version of a browser is installed on a test machine, or because the browser crashes. Tests that fail for reasons unrelated to the application logic waste development time and resources, and can reduce the team's confidence in the test suite.

Although they can have great value, web tests shouldn't be your only option when it comes to automating your BDD scenarios:

- Most applications need a judicious mix of both automated UI tests and automated tests for non-UI components.
- Non-UI tests can work at different levels of the application, including tests for the controller layer of an MVC application, tests that exercise the business rules implemented in the application code directly, and tests that work with remote services.

- Non-UI tests can also be used to verify nonfunctional requirements, such as performance.
- Implementing non-UI acceptance tests, and the corresponding application code, is also a great way to discover what components your application needs and to design clean, effective APIs within your application.

Let's start with how you should find the correct balance between UI and non-UI acceptance tests.

9.1 Balancing UI and non-UI acceptance tests

The ideal balance between UI and non-UI tests will naturally vary from project to project, depending on the nature of the application being built, the features being developed, and the technologies used. For some requirements, web or UI tests will be a natural fit, but for others, non-UI-based testing is more appropriate. Still other tests may need a mixture of both approaches.

Some web applications choose to keep business logic on the client side to a minimum, with the bulk of the business logic being performed on the server side in a well-defined service layer. This can be a deliberate design decision or a consequence of the chosen technology stack. This approach is illustrated in figure 9.2.

In this sort of application, automated web tests will typically be used to illustrate and verify the user's journey through the application, to ensure that data is submitted to the server correctly, to check that form validation messages are displayed correctly, to see that results from the server are rendered accurately, and so on. The web tests also

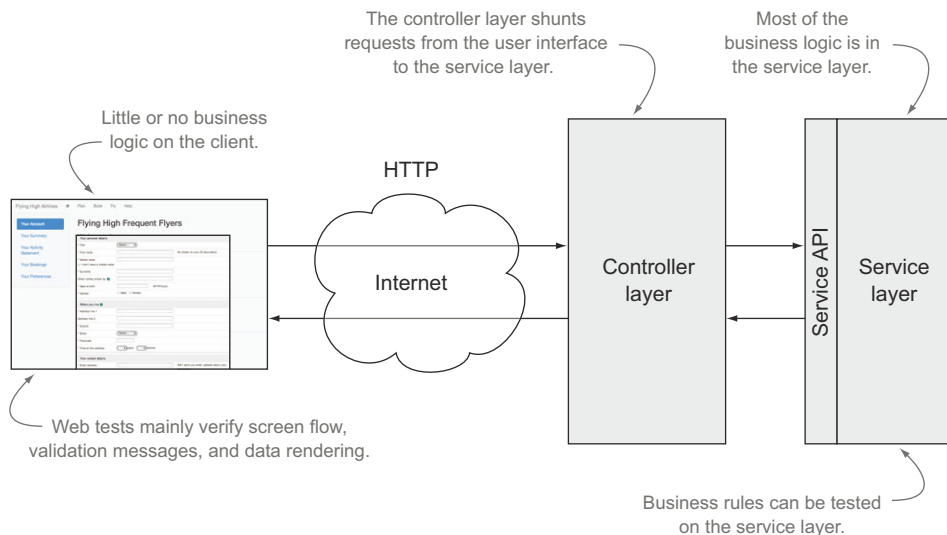


Figure 9.2 When an application has little business logic in the UI, much of the business logic can be tested directly against the service layer.

act as end-to-end tests, verifying the flow of information through the whole system. If the application is designed well, the business logic will be localized in the form of a well-defined service API that can be tested effectively using non-web tests.

If, on the other hand, the application wasn't designed with a clean service layer, business logic may be scattered through several layers of the system, and there may be no easy way to isolate particular operations. This is often the case for legacy applications, or it may be the result of poor architectural choices; in either case, pragmatic teams may use web testing to verify both the UI behavior and the business logic.

It's increasingly rare these days to see an application without any client-side behavior. Modern applications need to provide rich, interactive experiences that can't be achieved with older approaches. This can only be done by including at least some business logic and behavior within the UI layer. Modern JavaScript-based frameworks such as AngularJS and Backbone take this to the point where the client layer can be considered an application in its own right, calling remote web services for tasks that it can't perform locally or to retrieve the data it needs to function (see figure 9.3). Mobile apps often use a similar architecture, with a significant amount of business logic residing within the client application.

In both these cases, the web services used by the rich client application form a clean API that can be effectively tested using non-UI testing, whereas the screen flow and behavioral logic of the UI can be effectively tested using web tests.

In the rest of the chapter, we'll look at a few strategies that can be applied in these different scenarios.

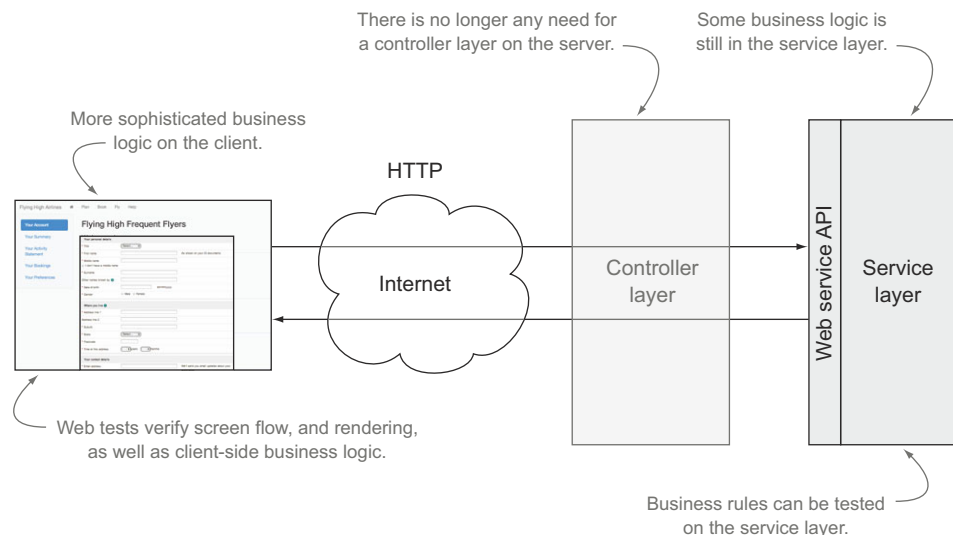


Figure 9.3 Modern single-page apps need more comprehensive tests of the business logic in both the UI and web service layer.

9.2 When to use non-UI acceptance tests

The business logic or business rules of an application describe how the application is expected to deliver business value, as well as the constraints under which it will operate. Business logic looks beyond how a user interacts with the application and focuses on what the user expects to achieve. The UI and user experience play an essential role in helping users get the most out of an application, but if you aren't clear on the underlying business rules and constraints, even the best-designed UIs will be wasted on features that are of limited practical value to your end users.

Business logic includes things like these:

- What are the expected outcomes, in business terms, of a particular user action?
- How does your application or business differentiate itself from its competitors? What makes it better than the legacy application you're replacing?
- What business rules need to be applied?
- What business-related constraints apply to user actions?

For example, suppose you're promoting a special Flying High–endorsed credit card that members can use to earn Frequent Flyer points when they make purchases. Because this is a web-based application with a rich user experience, web tests will play a critical role in your testing strategy. You'll need to verify the user's journey through the screens to apply for a new credit card, check on how successful and failing applications are displayed, ensure appropriate legal texts are agreed to, and so forth.

For example, an automated web test might be a good way to verify how the Frequent Flyer home page encourages members to apply for the new credit card:

```
Scenario: The Frequent Flyers site encourages members to apply for the new
         credit card
Given Joe is a Flying High Frequent Flyer
When Joe views his account home page
Then he should be able to apply for a Flying High Credit Card
```

CUCUMBER EXAMPLES The approaches discussed here are relatively implementation-neutral and tool-agnostic, but I did need to choose a format for the sample code. In the previous chapter we discussed examples built using JBehave and Thucydides, so in this chapter you'll use Cucumber.

Other examples of useful web-based scenarios might describe the user's journey through the application process:

```
Scenario: Joe is eligible for a Flying High Credit Card
Given Joe is a Flying High Frequent Flyer eligible for automatic Credit Card
      approval
When Joe applies for a Flying High Credit Card
Then Joe should be informed that his application was successful
And Joe should receive a confirmation email
And Joe's application should be queued for approval

Scenario: Joe is not eligible for a Flying High Credit Card
Given Joe is a Flying High Frequent Flyer who is not eligible for automatic
      Credit Card approval
```

```

When Joe applies for a Flying High Credit Card
Then Joe should be informed that Flying High will be in touch
And Joe's application should be queued for manual processing

```

Both of these scenarios illustrate key paths through the application and focus on how the user interacts with the website. You could implement them using the approaches we discussed in chapter 8.

However, because these acceptance criteria focus on the user experience and high-level outcomes, they deliberately gloss over some important underlying business logic. In particular, how do you determine if Joe is eligible or not? And depending on the scope of the feature, you may also need to explore other questions. How will the feature attract new Frequent Flyer members, and how will it earn revenue for the organization? This may involve some complex business rules. If you were to write automated web tests for each of these rules, it would slow down and add complexity to the test suite without adding a great deal of reporting value.

For example, income is an important consideration. The Flying High sales manager explains that regular Frequent Flyers with an annual income of over \$120,000 will be approved automatically, whereas those with an income less than \$50,000 will be declined. Those in between will need to be processed manually:

```

Scenario Outline: Credit card eligibility based on income
Given Joe is a regular Frequent Flyer earning <income>
When Joe applies for a Flying High Credit Card
Then his application should be <result>
Example:

```

income	result	notes
120000	automatic	Income >= 120000
100000	manual	
49999	declined	Income < 50000

But Frequent Flyer status also influences eligibility. Depending on a member's status, their application may be accepted automatically, even with a lower income:

```

Scenario Outline: Credit card eligibility based on income and status
Given Joe is a <status> Frequent Flyer earning <income>
When Joe applies for a Flying High Credit Card
Then his application should be <result>
Example:

```

status	income	result	notes
gold	80000	automatic	Automatically approved over \$80000
gold	79999	manual	
gold	49999	declined	
silver	100000	automatic	Automatically approved over \$100000
silver	99999	manual	
bronze	110000	automatic	Automatically approved over \$110000
bronze	109999	manual	

As you can see here, table-based scenarios are a great way to explore business rules, and they can be used to discuss and illustrate both positive and negative scenarios.

Note that none of these examples rely on a particular UI. If these rules are only ever calculated on the server side, they could be tested without having to manipulate the UI at all. This is often the case with acceptance criteria around business rules.

EXERCISE 9.1 The credit card eligibility table in the previous example is incomplete. For example, it doesn't explore examples of eligibility that depend on age or job history. Complete the preceding scenario with additional business rules until you think you have enough to implement a solution.

Discovering examples through conversation

As you've seen, conversation is the cornerstone of BDD, but not all conversations are equal. BDD conversations are more valuable when they're discovering new examples and expanding the team's collective understanding. In practice, the time a team has to work through and clarify acceptance criteria isn't infinite, and this process monopolizes the time and attention of several team members. It's important to focus first on high-value business rules and on areas of uncertainty, and then on features that are better known and less risky.

For example, suppose you're working on a story card related to authentication. If your application uses an existing, well-known authentication mechanism that the team has already worked with, the acceptance criteria for the related business rules can afford to be succinct. You might have a basic scenario such as the following:

```
Given Joe has a valid LDAP account
And Joe has application permissions for application X
When Joe logs on
Then Joe should be given access to the application
```

To this you might add a few counterexamples: invalid username or password, insufficient permissions, and so forth. But if this is familiar territory for the team, there's little benefit spending a lot of time working through these examples together, and the developers may be able to implement the feature using only a small set of examples.

On the other hand, if the application uses a new authentication strategy or technology, it will be valuable to spend more time working through the unknowns. Where are user accounts stored? Will there be single-sign-on between this application and other applications within the organization? What application-specific permissions need to be managed, and who will manage them? And so on.

One of the big benefits of BDD comes from the way conversation can be used to reveal assumptions and flush out uncertainty, expanding your collective understanding of a problem domain. Naturally, this process is more beneficial where there are unknowns to be flushed out, and the law of diminishing returns is very applicable. As a rule of thumb, if you're in a Three Amigos session and the scenarios start to feel like you're "stating the obvious," it's probably time to move on to another requirement.

In fact, although you'll need to have UI-based acceptance criteria that illustrate how users will interact with the application, and to demonstrate and verify any business logic that's implemented within the UI layer itself, acceptance criteria that relate to

business rules implemented on the server may not need to exercise the UI at all. In the rest of this chapter we'll discuss ways to effectively automate acceptance criteria like these without the need for automated UI tests.

9.3 Types of non-UI automated acceptance tests

As you've seen, UI tests have their place when it comes to testing and illustrating user interactions with the system and the user experience as a whole, but it's often inefficient to use them for more detailed business rules. There are a number of strategies that can be used to bypass heavyweight UI tests when automating your acceptance criteria, and many tools can be used to implement these approaches. Some of the more commonly used approaches include

- Testing against the controller-layer application code
- Testing directly against business logic in the application code
- Testing services remotely, such as by invoking web services and thus avoiding the UI layer entirely

Figure 9.4 illustrates these strategies in use with a typical modern web application.

In the following sections, we'll look at each of these options in more detail.

9.3.1 Testing against the controller layer

In some cases, you may be able to implement workflow and validation acceptance criteria without touching the web interface at all. This is particularly true if your application uses one of the many variations of the Model-View-Controller (MVC) architecture. MVC is a widely used architecture pattern that aims at cleanly separating the data an application uses from the way it's presented to the user (see figure 9.5).

If your application uses a variation of the MVC architecture, you may be able to write some of your acceptance tests directly against the controller layer. This is sometimes

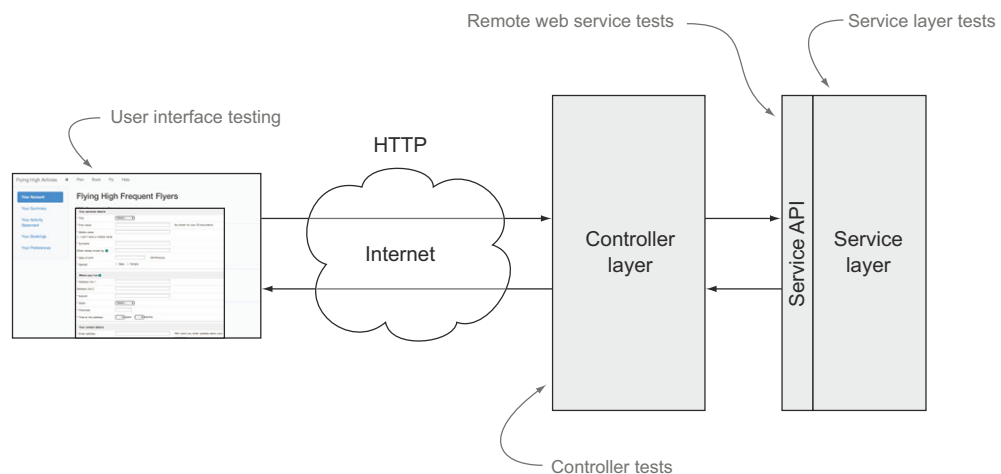


Figure 9.4 Applications need different types of tests for different parts of the system.

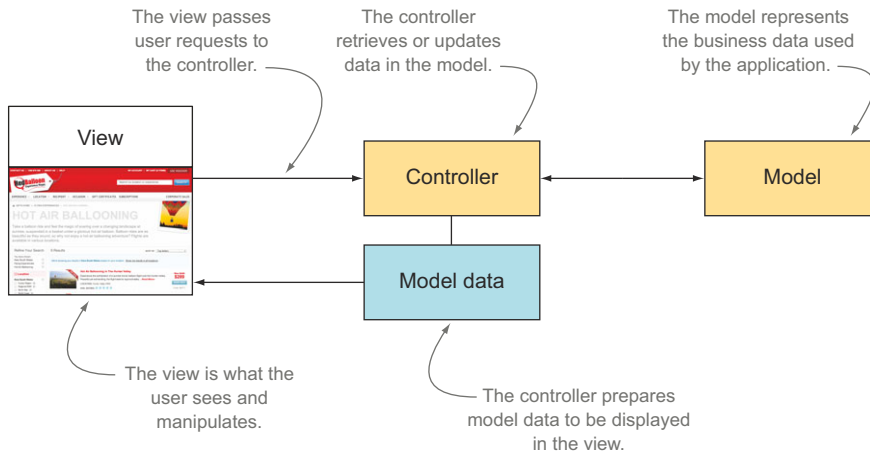


Figure 9.5 The MVC architecture pattern aims at cleanly separating application data from the way it's presented.

referred to as testing “under the skin” of the application. When you automate acceptance tests for a web application at the controller level, for example, you simulate requests arriving at a controller, and then check which page the controller displays and what data the controller places in the HTTP session or request scope.

In the Java world, Spring is an excellent and popular application development framework that provides first-class support for testing the controller layer. The web application module, Spring MVC, makes it easy to both write and test controllers using Java annotations to handle web requests.¹

Suppose you’re writing a Spring MVC application to monitor real-time flights for Flying High. You need a page that displays the current flight status for a particular flight. The corresponding acceptance criteria might look like this:

Feature: Displaying flight status

Scenario: Provide a positive visual queue for on-time flights

Given that flight FH-101 has no reported delays

When I check the flight status

Then I should see that it is on time

And I should see its scheduled arrival time

In a Spring MVC application, the view is typically a JSP page template that renders an HTML page containing data from the model. The controller receives queries from the user, prepares the model data to be displayed, and decides what page to render. For the flight status screen, the controller might receive the flight number as a parameter,

¹ For more information, see section 11.3.6, “Spring MVC Test Framework,” of the Spring Framework Reference Documentation: <http://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/testing.html#spring-mvc-test-framework>.

retrieve the current flight status from the model (most likely via a service layer), and prepare the flight status screen with the corresponding status information.

Cucumber-JVM provides excellent integration with Spring and Spring MVC. You could automate this scenario using Cucumber in Java with the following code:

```
@WebAppConfiguration
@ContextConfiguration("classpath:cucumber.xml")
public class FlightStatusSteps {

    @Autowired
    private FlightStatusService flightStatusService;

    @Autowired
    private MockMvc mockMvc;

    private String flightId;
    private ResultActions resultActions;

    @Given("^that flight (.*) has no reported delays$")
    public void no_reported_delays_for_flight(String flightId)
        throws Throwable {
        this.flightId = flightId;
        flightStatusService.updateStatusForFlight(flightId)
            .to(FlightStatus.ON_TIME);
    }

    @When("^I check the flight status$")
    public void I_check_the_flight_status() throws Throwable {
        resultActions = mockMvc.perform(get("/flights/{flightId}",
            flightId))
            .andExpect(status().isOk());
    }

    @Then("^I should see that it is (.*)$")
    public void I_should_see_that_it_is(FlightStatus expectedStatus)
        throws Throwable {
        resultActions.andExpect(view().name("flightstatus"))
            .andExpect(model().attribute("flightId",
                is(flightId)))
            .andExpect(model().attribute("flightStatus",
                is(expectedStatus.toString())));
    }

    @Then("^I should see its scheduled arrival time of (.*)$")
    public void expect_scheduled_arrival_time_of(String arrivalTime)
        throws Throwable {
        resultActions.andExpect(model().attribute("eta",
            is(arrivalTime)));
    }
}
```

1 Inject a Spring-configured service.

2 Use a Spring helper class to help test the controller layer.

3 Is the correct view used to display the results?

4 Is the retrieved model data correct?

5 Is the arrival time correct?

In this test you use Spring to configure the components and services it needs to retrieve the flight status **1** and set up the test data using an existing service class **2**. Next, you invoke the flight status controller with the `flightId` parameter **4** and check that the controller retrieved the expected values **4** and **5** and that you're redirected to the right screen **3**.

Controller-level testing isn't the exclusive domain of server-side development: it can apply equally to the client side. Let's look at an example. AngularJS (<http://angularjs.org/>) is a popular and elegant JavaScript MVC framework used to build well-designed and easily maintainable single-page applications in JavaScript. It can be compared to other JavaScript MVC frameworks such as Backbone.js and Ember.js. As you'll see, one of the areas in which AngularJS shines is ease of testing. Let's see what a controller-level test would look like in AngularJS.

Suppose you're writing an AngularJS app to monitor real-time flights for Flying High, using the requirements we discussed previously. To do this, you might write an AngularJS controller that talks to a service layer whose job is to provide information about the status of a given flight.

AngularJS provides tight integration with the Jasmine JavaScript testing library (<http://jasmine.github.io>), so for simplicity you'll implement this scenario in Jasmine.² A basic version of an AngularJS test for this scenario that calls the controller directly might look like this:

```
describe("Displaying flight status", function () {
  var controller, scope, stateParams, flights;
  beforeEach(module('flyinghigh'));

  beforeEach(inject(function($rootScope, $controller,
    $stateParams, flightService) {
    controller = $controller;
    scope = $rootScope.$new();
    stateParams = $stateParams;
    flights = flightService
  }));

  it('should provide a positive visual queue for on-time flights',
  function () {
    stateParams.flightId = 'FH-101';

    controller('FlightMonitorController',
      { $scope: scope,
        $stateParams: stateParams,
        flightService: flights });

    expect(scope.flight.title).toBe('Flight 101');
    expect(scope.flight.status).toBe('ontime');
    expect(scope.flight.statusicon).toBe('icon-thumbs-up');
    expect(scope.flight.eta).toBe('13:45');
  });
});
```

This is the application under test.

1 AngularJS injects the services you need.

2 The FlightService object returns the current state of a flight.

3 Pass in the query parameter 'flightId' for the flight under test.

4 Invoke the controller.

5 Check the expected outcomes.

In this test, you simulate an HTTP request that passes the flight ID, FH-101 **3**, to your AngularJS controller. You then call the controller with this parameter **4** and check what data the controller has placed into the page scope **5**. The controller uses the `flightService` object, which AngularJS sets up for you **1**. This makes this test an

² For tighter integration with the scenario text, you could also use Cucumber-JS or Yabba.

integration test: you're assuming that a test database has been configured to return the right status for flight FH-101.

If you wanted to have more direct control over the flight status, you could easily replace the `flightService` object ❷ with your own stub:

```
flightService = {
  getStatus: function(flightNumber) {
    switch(flightNumber)
    {
      case 'FH-101': return 'ontime';
      case 'FH-102': return 'delayed';
    }
  }
};
```

Both would be valid strategies: the first focuses on ensuring that all the components are wired together correctly, whereas the second is more interested in exploring different scenarios.

Testing against the controller layer is certainly a lot faster than web testing, but it does have some limitations. For more involved scenarios, the code can be complicated to set up and hard to maintain compared to an equivalent web test. When you check the routing instructions or error messages produced by a controller component, you have no way of knowing if this routing goes to the correct web page or whether the error messages are rendered correctly. You may need to complement your controller-level tests with a few UI-based tests that illustrate these more visual aspects.

This approach is also closer to unit or integration testing than the end-to-end web tests, which may inspire less confidence from nontechnical team members. Compared to web tests, where they can see visual feedback about the application's behavior, stakeholders will have a much harder time understanding (and therefore trusting) the results of controller-layer tests. When deciding how to implement your acceptance criteria, there's often a balance to be struck between confidence in the tests and speed of execution.

9.3.2 Testing business logic directly

Some application features and some kinds of business logic are relatively independent of the UI. The credit card eligibility rule we discussed earlier is a good example of this:

Scenario Outline: Credit card eligibility based on income and status

Given Joe is a <status> Frequent Flyer earning <income>

When Joe applies for a Flying High Credit Card

Then his application should be <outcome>

Example:

status	income	outcome	notes
gold	80000	automatic	Automatically approved for >= \$80000
gold	79999	manual	
gold	49999	declined	
silver	100000	automatic	Automatically approved for >= \$100000
silver	99999	manual	
bronze	110000	automatic	Automatically approved for >= \$110000
bronze	109999	manual	

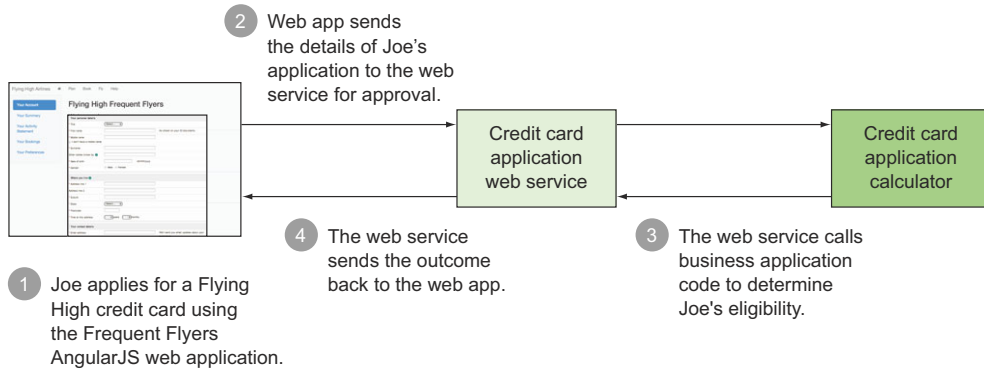


Figure 9.6 The Flying High application invokes a business process through a web service to evaluate credit card applications in real time.

You might implement this feature as illustrated in figure 9.6. Joe applies for a Frequent Flyer credit card online using an AngularJS application ①, which submits his details to a web service to determine whether he is eligible or not ②. The web service in turn calls a business service component that does the actual eligibility calculation ③ and returns the result to the web page ④ to be displayed.

In this case, you need to describe and verify the behavior of the AngularJS application, such as by using the web testing techniques discussed in chapter 8. You also need to document and verify the behavior of the web service, as we discussed in the previous section. But the core business logic around the eligibility calculation is neither in the UI nor in the web service: it's in the Credit Card Application Calculator module. There are many different scenarios that need to be verified, and it's potentially inefficient to test them all through the UI.

A more effective strategy might be to test the UI for each possible eligibility outcome (presuming they're displayed differently on the screen), and test the Credit Card Application Calculator directly to verify that the various status and income combinations produce the expected outcomes. Instead of eight UI tests, you'd have three (one for each eligibility outcome), plus eight tests written directly against the application code.

Other features may operate quite independently of the UI, and it would be difficult or impractical to test them via the UI. Batch processes, file or transaction processing, and back-end services are good examples of this type of scenario. In these cases, automating the acceptance criteria by exercising the application code directly is a viable option.

Let's look at another example. Suppose the Flying High flight-tracking system needs to process baggage registrations coming from the legacy baggage-handling system. This system provides the registrations in a well-defined text-based message format. You need to extract data from these baggage registrations and inject them into your system, routing them to different subsystems depending on whether the baggage

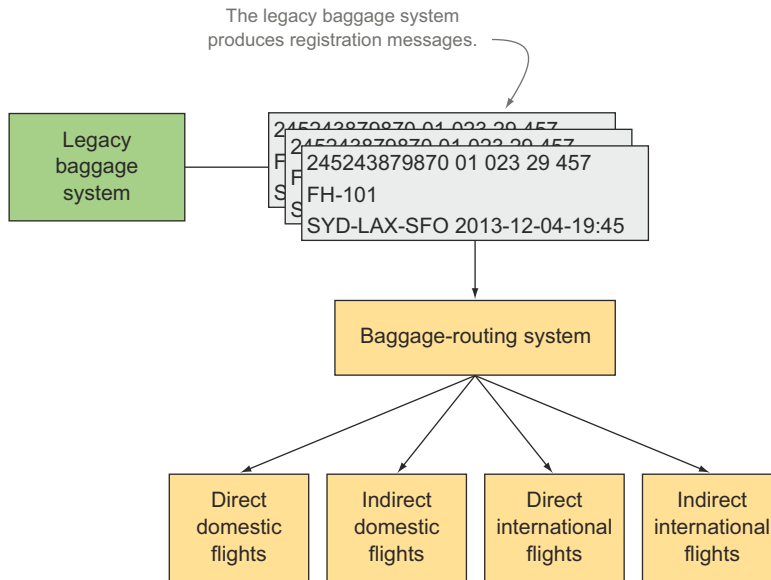


Figure 9.7 The Flying High baggage-routing system takes legacy messages and routes them to different subsystems.

is booked on a domestic or international flight and whether the flight is a direct one or involves transfers (see figure 9.7).

One way you could do this is to embed the actual content of the message in the feature file. One such scenario, implemented using Cucumber in Java, might look like this:

Scenario: Registered baggage should be queued according to its itinerary type

Given a baggage registration message:

"""

245243879870 01 023 29 457

FH-101

SYD-LAX-SFO 2013-12-04-19:457

"""

When the baggage registration is processed

Then the registration details should be:

flight	depart	destination	via	workflow
FH-101	SYD	SFO	LAX	international-transfer

1 The message header

2 The flight number

3 Itinerary details

4 Extract these details from registration message

Alternatively, you could use a set of named, well-defined message files. This approach is similar to the personas we looked at in section 7.2.5; it allows you to make the tests more concise by hiding the message details behind a well-known identifier.

Scenario Outline: Baggage is processed according to its itinerary type

Given a baggage registration message <message>

When the baggage registrations are processed

Then the bags should be placed in the <workflow> workflow

Examples:

message	workflow	
sydney_melbourne.txt	domestic-direct	
sydney_melbourne-hobart.txt	domestic-transfer	
sydney_wellington.txt	international-direct	
sydney_toronto.txt	international-transfer	

In both of these scenarios, there will be a web page that monitors the state of registered bags, but that's not really what's being tested here. Testing application code is faster and more robust than exercising the UI, and the bulk of the business rules can often be safely tested directly against the application code, but such testing may inspire less confidence in the results. A small number of additional end-to-end UI tests can alleviate these doubts and verify that the baggage details are correctly displayed.

These scenarios are typical of many business-focused acceptance criteria, especially scenarios involving file processing, message routing, and so forth. In general, acceptance criteria should be of the end-to-end variety. They should, at least in principle, exercise all of the components of the system and verify how they work together. This is usually a more reliable way to build confidence in the acceptance tests. But testing specific business rules can often be done more efficiently within an isolated context, especially when an acceptance criterion is focused on a specific calculation or operation, with well-defined input parameters and easily identifiable outcomes.

The Cucumber step definitions for the first scenario might look like this:

```
public class BaggageRegistrationSteps {
    String message;
    BaggageRegistration baggageRegistration;
    RegistrationService service;

    @Given("^a baggage registration message:$")
    public void a_baggage_registration_message(String messageText)
        throws Throwable {
        this.message = messageText;
    }

    @When("^the baggage registration is processed$")
    public void the_baggage_registration_is_processed() throws Throwable {
        service = new RegistrationService();
        baggageRegistration = service.registerBaggage(message);
    }

    @Then("^the registration details should be:$")
    public void registration_details_should_be(
        List<BaggageRegistration> expectedDetails)
        throws Throwable {
        BaggageRegistration expected = expectedDetails.get(0);
        assertThat(baggageRegistration).isEqualTo(expected);
    }
}
```

Here you create a new baggage registration service ❶ and use it to transform the message into a `BaggageRegistration` object ❷. Then you check that the generated registration object has the same values that you provided in the Then statement ❸.

Most real-world Java applications would use a dependency injection library, such as Spring or Guice, to manage dependencies between the modules in a cleaner manner. As you saw earlier, Java-based BDD tools like Cucumber and JBehave provide integration with the main dependency-management libraries, allowing you to automatically configure and inject other production service classes into the service you're testing.

9.3.3 Testing the service layer

Many, if not most, modern applications use some kind of service-based architecture. A service is a well-defined piece of functionality that can be called by other components or applications. Services generally act as facades, gateways, or aggregators for lower-level components (see figure 9.8). In many applications, the UI layer can't invoke lower-level business or database components directly, but instead must use one of the services available in the service layer.

Such services can be incorporated within a larger application, or they may take the form of discrete services that can be deployed and redeployed individually as needed and can be reused across multiple applications. This second approach is what's generally understood by the term *Service Oriented Architecture* (SOA). In both cases, a well-designed service architecture allows services to be combined and reused in different parts of an application, or by different applications, to build and deliver new business functionality more quickly and more reliably.

The service layer plays a key role in good system architecture. You've already seen how many web applications are now built around sophisticated client applications, such as modern JavaScript-based single-page applications using frameworks like AngularJS and Backbone. And as you've seen, these client applications have their own complex business logic and behavior, and they invoke web services to obtain the data they need. Many mobile applications are implemented using a similar approach, with a native client application calling remote services.

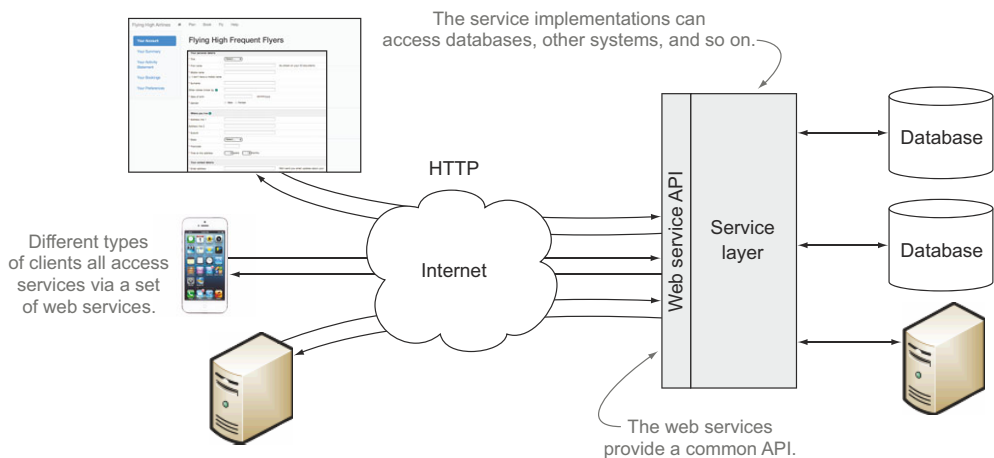


Figure 9.8 An example of an architecture using a service layer

In both these scenarios, the service layer can be considered as a separate API, with specific business goals and requirements. These requirements will be driven by the client applications that consume the services, and which may be developed by different teams. In BDD terms, the client application and the service API should be considered separately, each with its own set of scenarios and executable requirements.

A service-based architecture works well with BDD. By definition, services are designed to be reusable, so it makes sense to define them clearly and cleanly, and to document them with worked examples. When you design a new service using BDD, you begin by describing the requirements in the form of practical examples of how the service will be used. This tends to produce cleaner, more focused services that are easier to understand. If it's hard to find clean scenarios with well-defined input parameters and expected outcomes, then the service being designed may need a little more thought.

This approach also makes the reuse of these services easier. Well-written BDD scenarios give a clear description of what a service is meant to do and how it works, and the automation steps give worked examples of how to interact with a service that new developers can use to get started quickly.

One of the most popular ways to implement an SOA architecture is to use web services. REST web services in particular are becoming increasingly widespread, though the more heavyweight SOAP web services are still widely used.

Let's look at a few examples of working with both REST and SOAP in Java and .NET.

RESTFUL WEB SERVICES WITH CUCUMBER AND JAVA

Imagine you're designing web services for the Flying High Frequent Flyers application. One of the core services might be to display the details of a given flight. This service is required in several different screens and by the back-end payment-processing system:

```
Feature: Retrieve information about a given flight
Scenario: Find flight details by flight number
  Given I need to know the details of flight number FH-101
  When I request the details about this flight
  Then I should receive the following:
```

flightNumber	departure	destination	time
FH-101	MEL	SYD	06:00

For simplicity, this example supposes that flight FH-101 is a well-known entity in the system. The key value in this scenario is discovering precisely what information you need to retrieve about a given flight.

Note that although you're implementing a web service, this scenario is implementation-neutral and would be perfectly understandable by team members not fluent in JSON or XML. It's also a good first approach to the problem that will help the team agree on the goals of a particular web service, its input parameters, and what information the service will provide.

You could automate this scenario using Cucumber in Java as shown here:

```

public class FlightDetailsSteps {

    String flightNumber;
    Flight matchingFlight;

    @Given("^I need to know the details of flight number (.*)$")
    public void flight_number(String flightNumber) throws Throwable {
        this.flightNumber = flightNumber;
    }

    @When("^I request the details about this flight$")
    public void request_flight_details() throws Throwable {
        FlightStatusClient client = new FlightStatusClient();
        matchingFlight = client.findByFlightNumber(flightNumber);
    }

    @Then("^I should receive the following:$")
    public void verify_details(DataTable flightDetails) throws Throwable {
        flightDetails.diff(new ArrayList(matchingFlight));
    }
}

```

Note what flight number you're looking for.

1 Write a simple web service client to access the web service.

2 Compare the expected data with what the web service returned.

The automation code is relatively simple. You use a class written for your tests to access the web service and retrieve the result in the form of a Java object **1**.

To verify the outcomes, you use Cucumber's `DataTable` class **2**, which provides a convenient `diff()` method to compare the contents of a table with the corresponding field values from a list of Java objects, using column headers as field names. This is a concise way of checking that the `Flight` object that you retrieved from the web service matches the values described in the scenario.

When it comes to automating the BDD scenarios that verify services, it's common to use exactly the same libraries that are used to access them in the production code. This makes sense, because the automated steps act as examples for other developers to follow when reusing the services. In this example, you're using JAX-RS and Jersey, the standard Java library used to implement web services and web service clients:

```

public class FlightStatusClient {

    private final String BASE_URL = "http://localhost:8080/rest/flights";

    public Flight findByFlightNumber(String flightNumber) {
        Client client = ClientBuilder.newClient();
        WebTarget webTarget = client.target(BASE_URL).path(flightNumber);
        return webTarget.request().buildGet().invoke(Flight.class);
    }
}

```

Create a new web service client.

Specify the resource path you're invoking.

The URL of the web service.

Retrieve the result and convert it to a Flight object.

This level of feature is sufficient and appropriate for many scenarios. But some of the scenarios you write to describe a service may have a more technical audience than more business-focused scenarios. In some contexts, for example, if you're implementing

a JavaScript UI that relies heavily on JSON, it can make sense to provide sample JSON output as part of the BDD scenarios. In this environment, the target audience of the living documentation will be both the developers who are implementing the UI and the business, which will need to understand the business logic. If the JSON is simple and clean enough to make sense to both parties, there's no reason not to include the JSON output directly in the scenario:

```
Scenario: Should return flight details in JSON form
  Given I need to know the details of flight number FH-102
  When I request the details about this flight in JSON format
  Then I should receive:
    """
    {
      "flightNumber": "FH-102",
      "departure": "SYD",
      "destination": "MEL",
      "time": "06:15"
    }
    """
```

A known flight

The expected JSON result

You could automate these steps in a similar way to the previous example:

```
String receivedJsonData;

@When("^I request the details about this flight in JSON format$")
public void request_details_in_json_format() {
    receivedJsonData = client.findByFlightNumberInJson(flightNumber);
}

@Then("^I should receive:$")
public void should_receive_json_data(String expectedJsonData)
    throws JSONException {
    JSONAssert.assertEquals(expectedJsonData,
        receivedJsonData,
        JSONCompareMode.LENIENT);
}
```

Retrieve the search results as a JSON document.

Compare the retrieved JSON with the expected data using the JSONassert library.

Here you use the JSONassert library (<https://github.com/skyscreamer/JSONassert>) to compare the JSON data you receive from the web service with that given in the feature file. And once again, you delegate the actual code that invokes the web service to the FlightStatusClient class you saw earlier:

```
public String findByFlightNumberInJson(String flightNumber) {
    Client client = ClientBuilder.newClient();
    WebTarget webTarget = client.target(BASE_URL).path(flightNumber);
    return webTarget.request().buildGet().invoke(String.class);
}
```

Return the result in raw JSON format.

This is a simple example, but JSON is relatively readable, and more complicated scenarios are quite possible without compromising readability. The following scenario provides test data to be injected into the flight schedule database:

Scenario: Should return scheduled flights in JSON form

Given the following flights have been scheduled:

flightNumber	Departure	Destination	time
FH-101	SYD	MEL	06:15
FH-102	MEL	SYD	06:30
FH-223	SYD	LAX	06:00
FH-305	MEL	SFO	07:15
FH-234	SYD	LHR	09:25
FH-403	SYD	DBX	14:05

Set up the
test data.

Call the web
service.

When I request the International flights in JSON form

Then I should receive the following flights:

```

"""
[
  { "flightNumber": "FH-223",
    "departure": "SYD", "destination": "LAX", "time": "06:00" },
  { "flightNumber": "FH-305",
    "departure": "MEL", "destination": "SFO", "time": "07:15" },
  { "flightNumber": "FH-234",
    "departure": "SYD", "destination": "LHR", "time": "09:25" },
  { "flightNumber": "FH-403",
    "departure": "SYD", "destination": "DBX", "time": "14:05" }
]
"""

```

Expected
search
results.

Providing sample output in JSON (or XML) form is a more technical approach that's not suitable for all scenarios. But when the audience is technical enough to understand the output, and the JSON format will provide added value over simply returning the information, it can be beneficial to express requirements in this format.

There are many other ways to implement a service architecture, both within the context of a single application and in a broader SOA strategy. In the Java world, EJB 3 and Spring Remoting are other frequently used approaches. Teams using a .NET stack will typically use WCF (Windows Communication Foundation) or WebAPI.

9.4 Defining and testing nonfunctional requirements

BDD isn't limited to purely functional requirements; the approach can also be used very successfully to discover and verify nonfunctional requirements. A *nonfunctional requirement* is traditionally defined as a (generally technical) requirement that's not related to a particular feature but relates to how the application operates as a whole. Typical nonfunctional requirements include performance, stability, accessibility, and security.

Just like functional requirements, nonfunctional requirements should deliver value in some identifiable way, typically by increasing revenue or reducing costs. An unresponsive website will cost an organization in customers and sales. According to Amazon, 100 ms of page latency results in a loss of 1% of sales.³ Google found that a delay of half a second resulted in a traffic drop of 20%.⁴

³ These figures were released in a presentation by Greg Linden, "Make Data Useful" (2006), <https://sites.google.com/site/glinden/Home/StanfordDataMining.2006-11-28.ppt>.

⁴ Marissa Mayer, Web 2.0 Conference, 2006.

BDD is a great way for teams to have a conversation about which nonfunctional requirements really matter for a particular application. BDD helps phrase these nonfunctional requirements in terms that stakeholders can understand and help define, and that can be easily verified and potentially automated.

In addition, not all nonfunctional requirements are equal. Expressing requirements as BDD scenarios can help stakeholders and developers determine what areas are most affected by nonfunctional requirements, such as performance or security.

For example, before doing any performance testing on an application, a team should ask end users and managers about their idea of acceptance performance metrics and the application's anticipated load. Ideally, these expectations should be expressed in a way that's meaningful to the business, but this isn't always the case in practice. Many teams spend time and effort measuring performance metrics such as CPU and memory usage that aren't necessarily related to the underlying business goals and don't provide useful feedback to the business.

Let's look at how this might work in practice. The Flying High Flight Status application relies on receiving timely updates from the flight status message server. One of the selling points of the Frequent Flyer program is its ability to provide clients with fast updates about flight status, including information about delays, expected arrival times, gates, baggage carousel numbers, and so forth.

From discussion with stakeholders and the marketing department, which had conducted market research and studied competing products, it emerged that updates needed to be dispatched within 30 seconds 95% of the time, and within 60 seconds 100% of the time, and that every update needed to be successfully dispatched.

You could express this performance requirement like this:

Feature: Application performance

Scenario: Flight status update performance

Given the flight status server is running

When a production peak hour of updates is sent at 5 times production throughput

Then 95% of the updates should be received within 30s

And 100% of the updates should be received within 60s

And 100% of the updates should be received successfully

Once you've expressed the performance requirements in these terms, you can automate. There are many open source and commercial load-testing tools, and most can be scripted. Popular open source options in the Java world include SoapUI (www.soapui.org), JMeter (<http://jmeter.apache.org/>), and The Grinder (<http://grinder.sourceforge.net>). These tools let you define and execute test scripts that simulate interactions with an application, such as requesting a web page, invoking a web service, and querying an EJB. To simulate load, they can run scripts on a number of remote machines as well as locally. They're also relatively easy to automate and to run programmatically. This makes them easy to integrate into tools such as Cucumber and JBehave.

Load testing like this needs realistic production-like data. In general, the best way to get an idea of the number of messages sent during peak times would be to observe

the current production system, if one exists. Many teams even record a selection of production data for use in load testing. If you're building the entire system from scratch, on the other hand, the team would have to agree on some figures based on the expected usage patterns.

Cucumber, Java, and The Grinder for real-world performance testing

When Tom Howard needed to verify the performance of an application built around a messaging platform for a large electricity company, he chose to use a BDD approach.⁵ To do this, he used The Grinder and Cucumber-JVM to automate a BDD performance scenario similar to the one discussed here. The main Cucumber step invoked a Grinder script directly and waited for it to finish. The Grinder script ran a series of requests based on real production data from a number of worker machines to simulate production-like loads.

Once the script was done, the test results were recorded in a CSV log file. Tom used open source libraries to parse the CSV file and determine the 95th and 100th percentiles, which were in turn used to decide whether the scenario passed or failed.

It might be tempting to write automated tests for more low-level performance metrics such as memory consumption, CPU usage, average response time, and so forth. But this is generally not a cost-effective approach. During performance testing, a system can underperform or fail for many different reasons: insufficient memory, a slow database, network issues, suboptimal code, and so forth. In addition, in the real world, performance issues inevitably come from unexpected places, and it's very hard to predict or test for every possible problem.

But in business terms, poor performance at any point will only really be of concern if it's reflected in the performance requirements described in the BDD scenario, and this will be reflected by a failing BDD scenario.

A more effective approach is to monitor and record low-level performance metrics such as memory and CPU usage, but not to include these in any automated tests. If the BDD scenario does fail, then you've identified a performance issue that can potentially impact the business. In this case, you can use the recorded metrics to investigate and troubleshoot the issue.

9.5 Discovering the design

BDD is an excellent way to design clean, reusable service APIs, as well as clean, reusable APIs at all levels of the application. Well-designed applications are typically organized into layers and components (in design patterns, this principle is often referred to as the *separation of concerns*). If you think of an application in terms of these layers and components, starting from the UI and working down, each layer is the "user" of the next layer down, and any code that uses a component is effectively a user of that component. When the

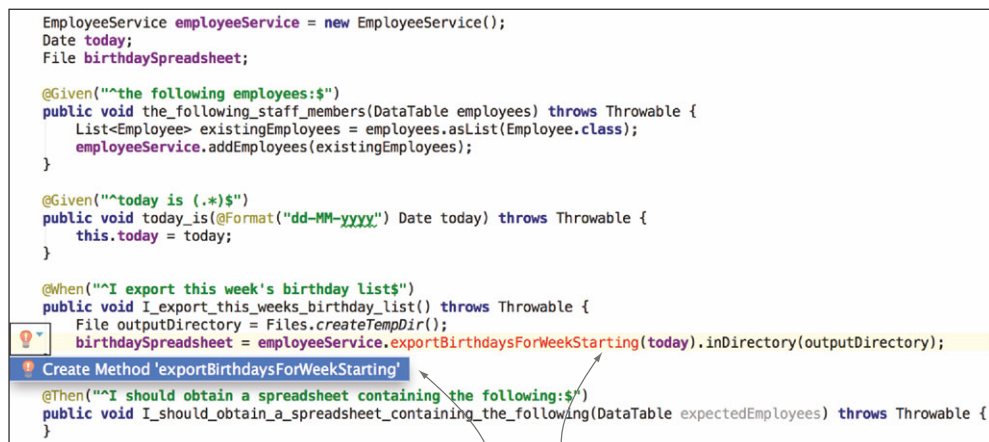
⁵ Tom Howard, "Cucumber-JVM + The Grinder + Math = Performance.Testing.Heaven," <http://windyroad.com.au/2013/09/11/cucumber-jvm-the-grinder-math-performance-testing-heaven>.

user is application code (or, more precisely, the developer writing the application code) rather than a physical person, you're effectively writing an API. In fact, any code that you write can be considered an API for someone else, even if that someone is yourself.

Applying BDD principles to the actual implementation of these layers and components can help you write cleaner, better-designed, and more maintainable code. When you implement new code, you think about what data and services you need and how you'd ideally like to obtain them. If they don't exist yet, you implement your code *as if they do exist*, and then use this code as the starting point for their implementation (see figure 9.9). This is a form of the Feature Injection idea you saw in chapter 3.

For example, imagine you're working on implementing a new feature for the Flying High Human Resources application. The requirement is to export an Excel spreadsheet containing a list of employees who have a birthday in the current week. The corresponding scenario might look like this:

```
Scenario: Export staff birthdays as an Excel spreadsheet
  Given the following staff members:
    | Name | Birthday |
    | Joe  | 10-Mar-1980 |
    | Jill | 18-Dec-1965 |
    | Jack | 20-Dec-1965 |
    | Joan | 20-Nov-1991 |
  And today is 16-Dec-2013
  When I export this week's birthday list
  Then I should obtain a spreadsheet containing the following:
    | Name | Birthday |
    | Jill | 18-Dec-1965 |
    | Jack | 20-Dec-1965 |
```



```
EmployeeService employeeService = new EmployeeService();
Date today;
File birthdaySpreadsheet;

@Given("^the following employees:$")
public void the_following_staff_members(DataTable employees) throws Throwable {
    List<Employee> existingEmployees = employees.asList(Employee.class);
    employeeService.addEmployees(existingEmployees);
}

@Given("^today is (.*)$")
public void today_is(@Format("dd-MM-yyyy") Date today) throws Throwable {
    this.today = today;
}

@When("^I export this week's birthday list$")
public void I_export_this_weeks_birthday_list() throws Throwable {
    File outputDirectory = Files.createTempDir();
    birthdaySpreadsheet = employeeService.exportBirthdaysForWeekStarting(today).inDirectory(outputDirectory);
}

@Then("^I should obtain a spreadsheet containing the following:$")
public void I_should_obtain_a_spreadsheet_containing_the_following(DataTable expectedEmployees) throws Throwable {
}
```

Let the IDE help you
fill in the details.

Write the code you
would like to have.

Figure 9.9 Writing the code you'd like to have. Here, you use the "quick-fix" feature available in most modern IDEs to get the IDE to generate the missing method for you.

As you implement the step methods for this scenario, you'll use the scenario example to drive out the features you need from the next layer down.

This process doesn't stop with the step implementation. When you write the production code, you can do exactly the same thing, discovering what services you need from other components (which may not exist yet), and writing the code you'd like to have. This act of imagining the code that would serve you best is a powerful design practice: what information do you need to get the job done? What services do you need to call? Do they exist, or do they need to be created? What would be the most convenient way to obtain the data you need, and in what form? When you write this *imaginary* code, you're providing an example of how you'd like to use an API that doesn't yet exist. Because you haven't written any implementation code yet, this example will not be polluted by preconceived ideas about what the technical solution should look like; rather, it will be driven by what would be easiest to use from the perspective of the developer using the API.

In this way, BDD at the requirements level (where you use more communication-focused tools, such as Cucumber and JBehave) flows naturally to BDD at a more technical level (where you often use tools more traditionally associated with unit testing). We'll study this lower-level form of BDD in much more detail in the next chapter.

9.6 Summary

In this chapter you learned about different sorts of non-UI acceptance testing, including

- When you should use UI tests and when it's more appropriate to use non-web tests
- How to write BDD scenarios that bypass the UI and exercise the controller layer, service layer, or business logic directly
- How to write BDD scenarios for remote services such as web services
- How to use BDD scenarios to describe and verify nonfunctional requirements, such as performance
- How BDD techniques at the requirements level help encourage clean, well designed APIs, and lead naturally to lower-level BDD practices at the unit- and integration-testing level

In the next chapter, you'll learn how BDD practices are also applicable at the coding level, and how this sort of lower-level BDD relates to Test-Driven Development and more traditional unit-testing practices.

10

BDD and unit testing

This chapter covers

- The relationship between BDD, TDD, and unit testing
- Going from automated acceptance criteria to implemented features
- Using BDD to discover design and explore low-level requirements
- Tools that help you write BDD unit tests more effectively

So far we've focused on Behavior-Driven Development (BDD) as a tool for discovering, illustrating, and verifying business requirements. But BDD doesn't stop at the business requirements level, or once you've automated your acceptance tests. In this chapter you'll learn how BDD principles and tools can help you write better-designed and better-tested application code (see figure 10.1).

The principles of BDD can be effectively applied at all levels of development, and with significant benefits:

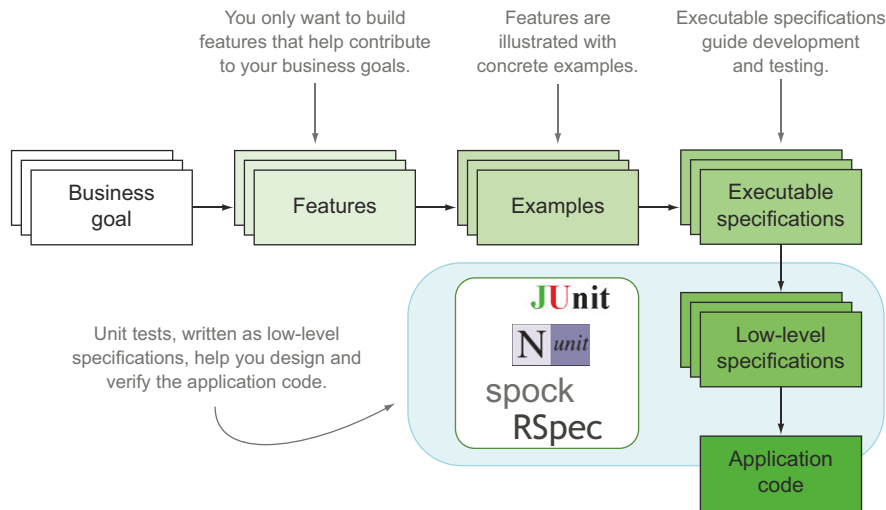


Figure 10.1 In this chapter we'll focus on using BDD practices at the unit-testing level.

- BDD is about writing executable specifications that guide the implementation at all levels of development.
- At a unit-testing level, BDD builds on and extends established TDD practices.
- BDD practitioners use an outside-in approach, using automated acceptance tests and unit tests to drive the implementation of the underlying code.
- You can practice BDD-style unit testing with any tool, but some tools make it easier to write more expressive and more concise unit tests.
- BDD practices at the unit-testing level also help provide living technical documentation of the components and APIs you develop for your application.

Let's start by taking a closer look at the relationship between BDD and TDD.

10.1 BDD, TDD, and unit testing

In this chapter, you'll learn how BDD principles can also help developers write more focused, more effective, more maintainable, and better-documented low-level code, and how unit tests can be used very effectively to express, document, and validate low-level specification and design.

"But isn't that TDD?" I hear you asking. There's often confusion about the distinction between Behavior-Driven Development and Test-Driven Development (TDD). Many developers think of BDD as a technique used for acceptance testing and use "TDD" to refer to lower-level, test-first activities involving unit tests. In fact, things aren't that clear-cut, and the two techniques are deeply intertwined.

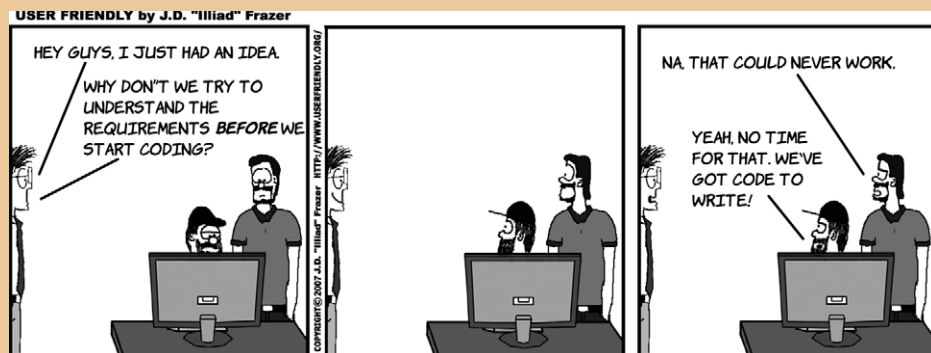
What is Test-Driven Development?

Test-Driven Development (TDD) is a development practice that uses unit tests to specify, design, and verify the code you're writing. Before implementing a piece of functionality, developers write a failing unit test that demonstrates how this functionality should work. At the same time, this failing test also proves that the current implementation doesn't yet support the new functionality. Only then do the developers write the application code. Once the unit test passes, the developers know that the functionality has been successfully implemented. At this stage, they can review their code to tidy things up and fine-tune the design.

TDD has been around for a long time and can be traced back to Extreme Programming practices developed in the late 1990s. TDD relies on two simple principles:

- Don't write any code until you've written a failing test that demonstrates why you need this code.
- Refactor regularly to avoid duplication and keep the code quality high.

The cornerstone of TDD is the idea of writing a unit test before you write the corresponding code. But TDD is much more than just a guarantee that every class has a corresponding set of unit tests. With enough discipline, any experienced developer writing unit tests after writing the code can achieve that outcome as well. TDD's killer feature is that it forces developers to think about the code they're going to write before they write it, in practical and unambiguous terms—you need to understand the functionality before you can write a unit test for it. This way, developers resist the temptation to just start coding something and actively think about what they need to achieve. In this respect, a better term for this practice might be something like “Test-Driven Design.”



TDD's killer feature is that it forces developers to think about the requirements before they start to code. This approach isn't always easy for teams to adopt.

This somewhat counterintuitive idea arguably represents one of the single most significant contributions to software engineering quality that has happened over the last 20 years. TDD offers several benefits:

- Cleaner, better designed code
- Code that's easier and less expensive to maintain

(continued)

- Fewer bugs from the outset
- A comprehensive set of regression tests

Experienced developers will typically think about the design of their code before they do any coding, but expressing this design in the form of unit tests makes this process a lot more concrete. Before implementing any code, TDD practitioners imagine the code “they would like to have,” which tends to result in cleaner, better-designed APIs. These unit tests also become examples of how to use the application code. And because at least one test is written for every new feature, the code tends to be better tested and have significantly fewer bugs.

Code developed using TDD also benefits from a comprehensive set of regression tests, which, coupled with the clean design that TDD encourages, makes the application easier to change and cheaper to maintain. This, along with the lower bug count, reduces maintenance costs and the total cost of ownership significantly.

On the downside, TDD isn't easy to learn, and it requires a lot of discipline, perseverance, and practice to adopt, especially without the guidance of an experienced TDD practitioner or the support of lead developers and management. As with any new technology or method, productivity will initially take a hit. Teams new to TDD will initially deliver more slowly, though the reduction in defects happens almost immediately.¹

10.1.1 BDD is about writing specifications, not tests, at all levels

As you've seen, BDD involves discovering and specifying the behavior of a system, but this concept works just as well for the whole application as it does for an individual class: in both cases, you're specifying behavior.

Low-level BDD is a natural continuation of the BDD principles we've been applying to high-level requirements. In the chapters so far, you've seen how BDD practitioners express high-level requirements in the form of executable specifications. High-level requirements deal with the behavior of the system as a whole from the point of view of the business. Low-level requirements deal with the behavior of a component, class, or API from the point of view of the developer working with them. In both cases you're specifying behavior, and in both cases you can describe this behavior in terms of executable specifications. Only the target audience changes: high-level requirements are typically aimed at the broader team, whereas low-level, more technical requirements are aimed at future developers who will have to understand and maintain the application code.

¹ This seems to be supported by a case study from 2008, in which four teams (all new to TDD) reported 40–90% fewer defects, but observed a 15–35% increase in the initial development time. Nagappan, Maximilien, Bhat, and Williams, “Realizing quality improvement through test driven development: results and experiences of four industrial teams,” *Empirical Software Engineering*, 13, no. 3 (June 2008): 289–302, <http://dl.acm.org/citation.cfm?id=1380664>.

10.1.2 *BDD builds on established TDD practices*

The core practices we’re looking at in this chapter are essentially those of TDD, or are deeply rooted in TDD. Indeed, experienced TDD practitioners have been using these techniques for a long time. In many respects, BDD has formalized the way many successful TDD practitioners have been doing things, emphasizing a few key aspects of advanced TDD practice that make the technique more effective. These practices include

- Using outside-in development to ensure that the code you write is delivering real business value
- Using a shared (or “ubiquitous”) domain language to encourage closer collaboration and understanding within the team
- Using examples to describe this behavior more clearly
- Describing and specifying the behavior of the system at both a high and a more detailed level

You’ll see more of these concepts in the rest of the chapter.

10.1.3 *BDD unit-testing tools are there to help*

The result of this formalization is that a number of more BDD-flavored unit-testing tools have emerged over recent years that make these techniques easier and more intuitive to practice. Tools like RSpec, NSpec, Spock, and Jasmine have flourished over recent years. While there’s no obligation to use these tools (many of our examples will be written simply using JUnit), they can make it easier to write more concise, more expressive, low-level executable specifications.

In the remainder of this chapter, you’ll learn how to apply the BDD principles you’ve seen so far to design and deliver working code. Or, if you prefer, you’ll learn to practice TDD with a BDD flavor. If you don’t have any experience with TDD, don’t worry. I’ll introduce all the key concepts as we go.

10.2 *Going from acceptance criteria to implemented features*

In this section we’ll walk through a typical BDD workflow, going from high-level automated acceptance criteria to application code. We’ll keep the requirements simple, so that we can focus on the process. But before we start, let’s take a look at what we mean by “outside-in” development.

10.2.1 *BDD favors an outside-in development approach*

One of the core practices used in BDD is *outside-in development*. This involves using acceptance criteria to drive the implementation details you need to build business features. You start with the outcomes you expect and use these outcomes to determine what code you need to write.

The process typically iterates over the following steps:

- 1 Start with a high-level acceptance criterion that you want to implement.
- 2 Automate the acceptance criterion as pending scenarios, breaking the acceptance criterion into smaller steps.

- 3 Implement the acceptance criterion step definitions, imagining the code you'd like to have to make each step work.
- 4 Use these step definitions to flesh out unit tests that specify how the application code will behave.
- 5 Implement the application code, and refactor as required.

The last two steps, where you use unit tests to describe low-level behavior and build the corresponding application code, are what many developers would describe as a



Figure 10.2 Going from acceptance criteria to production code with outside-in development

form of TDD. In an outside-in approach, the description of the low-level technical behavior of the application flows naturally from discussing, thinking about, automating, and implementing the high-level acceptance criteria (see figure 10.2).

There are many benefits of outside-in development, but the principle motivations are summarized here:

- Outside-in code focuses on business value.
- Outside-in code encourages well-designed, easy to understand code.
- Outside-in code avoids waste.

OUTSIDE-IN CODE IS FOCUSED ON BUSINESS VALUE

Outside-in code starts with the expected outcomes expressed in business terms, so any code you write can be traced back to some form of business value. The acceptance criteria help keep you focused on where the value is coming from and what you're trying to achieve.

OUTSIDE-IN CODE IS WELL-DESIGNED AND EASIER TO UNDERSTAND

When you write code from the outside in, you write an example of how a method should be used before implementing the method. This makes you think about how the code will be used by other developers, which tends to lead to cleaner API design. These examples also illustrate and document your classes and APIs, making the code much easier to understand for any new developers (or your future self).

OUTSIDE-IN CODE AVOIDS WASTE

If you strictly apply this outside-in approach, any code you write is written with the goal of making an acceptance test pass. If this isn't the case, then either some necessary aspect of the system hasn't been specified in the acceptance criteria, or you're writing code that won't be used in production. BDD won't guarantee that no unnecessary code gets written, but it will highlight potential situations where this might be the case.

Let's see what outside-in development looks like in action with a practical example.

10.2.2 Start with a high-level acceptance criterion

When practicing outside-in development, you usually start with the acceptance criteria of the feature or story that you're currently building. Suppose, for example, that you need to add support for member status levels to your Frequent Flyer application. Flying High Frequent Flyers have different status levels (Bronze, Silver, Gold, and Platinum) depending on how often they fly. Members earn status points with each flight, and when they've earned enough points over the period of a year, they move up to the next level. To encourage travellers to fly with Flying High, members with higher status levels get extra privileges, such as access to lounges, fast lanes, and so forth. At the end of the year, members retain their status level, but their status points are reset to zero.

When implementing a feature, BDD practitioners like to begin with the high-level acceptance criteria, often automating these acceptance criteria using tools like Cucumber, JBehave, and SpecFlow. Using Java and Cucumber, the acceptance criteria might look like this:

Feature: Frequent Flyer status is calculated based on points

As a Frequent Flyer member

I want my status to be upgraded as soon as I earn enough points

So that I can benefit from my higher status sooner

Recall the business goals behind this requirement.

Scenario: New members should start out as BRONZE members

Given Jill Smith is not a Frequent Flyer member

When she registers on the Frequent Flyer program

Then she should have a status of BRONZE

This scenario describes what status members should get when they start out.

Scenario Outline:

Given Joe Jones is a <initialStatus> Frequent Flyer member

And he has <initialStatusPoints> status points

When he earns <extraPoints> extra status points

Then he should have a status of <finalStatus>

This scenario describes the actual status earning process.

Examples: Status points required for each level

initialStatus	initialStatusPoints	extraPoints	finalStatus
Bronze	0	300	Silver
Bronze	100	200	Silver
Silver	0	700	Gold
Gold	0	1500	Platinum

Illustrate the number of points needed for each level.

This is a simple tangible business requirement with a clear value proposition and two acceptance criteria. Still, it's probably too big to write in one pass, so you'd typically implement the individual acceptance criteria one at a time. In any case, the first step is to automate the step definitions for these scenarios.

10.2.3 Automate the acceptance criteria scenarios

The first acceptance criterion describes the initial status level of a new Frequent Flyer member:

Scenario: New members should start out as BRONZE members

Given Jill Smith is not a Frequent Flyer member

When she registers on the Frequent Flyer program

Then she should have a status of BRONZE

Background and context

The expected outcome

The behavior under test

You could automate this scenario using the techniques we discussed in the previous chapters. In Cucumber, for example, the step definitions might look something like this:

Background and context	<pre>@Given("^(\\S*) (\\S*) is not a Frequent Flyer member\$") public void not_a_Frequent_Flyer_member(String name) throws Throwable {...}</pre>
The behavior under test	<pre>@When("^(?:s?)he registers on the Frequent Flyer program\$") public void registers_on_the_Frequent_Flyer_program() throws Throwable {...}</pre>
The expected outcome	<pre>@Then("^(?:s?)he should have a status of (.*)\$") public void should_have_status_of(FrequentFlyerStatus expectedStatus) {...}</pre>

But it's when you actually implement the step definitions that you really discover what code you need to write.

10.2.4 Implement the step definitions

Writing a step definition is an exercise in code design. When you write the step definition, you come in contact with your application code for the first time. Although you may have some ideas about the high-level architecture and design, the step definition code is where the rubber meets the road and your design takes the form of real code for the first time.

NOTE Because this chapter focuses on BDD and TDD unit-testing practices, we'll concentrate on acceptance criteria that manipulate the application code directly. Chapter 8 discusses automating acceptance criteria where a UI is involved.

Of course, not all steps are equal. Some step definitions contain relatively simple test code, whereas others involve more forethought and design effort.

In this case, the first step is very simple indeed:

```
String firstName;
String lastName;

@Given("^((\\S*)) (\\S*) is not a Frequent Flyer member$")
    public void not_a_Frequent_Flyer_member(String firstName,
                                           String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
}
```

1 Store the new member's name here.

2 Note the name of the new member for use in the following step definitions.

In the first step ❶, you need to ensure that Jill isn't already a Frequent Flyer member. There's little to do here except record the name of the prospective member so that you can reuse it in the subsequent steps.

The next step definition ❷ is a little more interesting, as you introduce the domain concept of the Frequent Flyer member. But before you start writing code, you'd typically make sure that you have a reasonable understanding of this domain concept.

10.2.5 Understand the domain model

Although incremental and emergent design works well in many situations, a team practicing BDD doesn't exclude more structured design practices where appropriate. In particular, BDD draws many concepts from Domain-Driven Design, including the idea of a common (or *ubiquitous*) language shared by everyone in the team, from business stakeholders to developers. This common language is founded on a high-level domain model, which can be built up incrementally as features are added, as you'll do in this example. This isn't the only option: many teams, particularly larger ones, often draw up a broader domain model supporting several features early on in the iteration. In a similar way, BDD teams will usually give some forethought to the high-level application design and architecture before the coding starts.

In our example, an important business entity has emerged: the Frequent Flyer member. From conversations with the business, you might learn that a Frequent Flyer

A simple UML diagram representing the Frequent Flyer domain object.

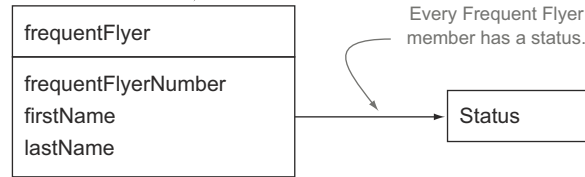


Figure 10.3 The Frequent Flyer domain object as described in discussions with the business

member has a name, a unique Frequent Flyer number, and a current status. Possible status values are Bronze, Silver, Gold, and Platinum. The Frequent Flyer domain model could look like the one in figure 10.3.

Armed with this understanding of the domain, you can now implement the second step definition, which will involve code that manipulates this domain object.

10.2.6 Write the code you'd like to have

When you write a step definition, you write the code you'd like to have if you were a developer using the code. In other words, you imagine the ideal classes and methods for your current needs, and write sample code that illustrates how you'd exercise these classes and methods. This idea of writing the code you'd like to have is an important part of outside-in development.

For example, you might implement the second step definition like this:

```

FrequentFlyer member;

@When("^(?:s?)he registers on the Frequent Flyer program$")
public void registers_on_the_Frequent_Flyer_program() throws Throwable {
    member = FrequentFlyer.withFrequentFlyerNumber("123456789")
        .named(firstName, lastName);
}
  
```

Application code illustrating how to create a new Frequent Flyer member

1

This approach is a great way to design a class, because the code used in the step definitions will be very similar to the code used in the actual application. Writing the code in the step definitions is a way of exploring the objects and methods you'll need to implement a particular acceptance criterion and of discovering the cleanest and most elegant way to do so.

Note that the `FrequentFlyer` class mentioned in the code ❶ doesn't exist yet; you've just specified what it should look like. The next step is to actually implement this code.

10.2.7 Use the step definition code to specify and implement the application code

You can now switch gears and consider the requirements in more granular terms. The high-level requirements expressed in the acceptance criteria often break down into more detailed, low-level requirements.

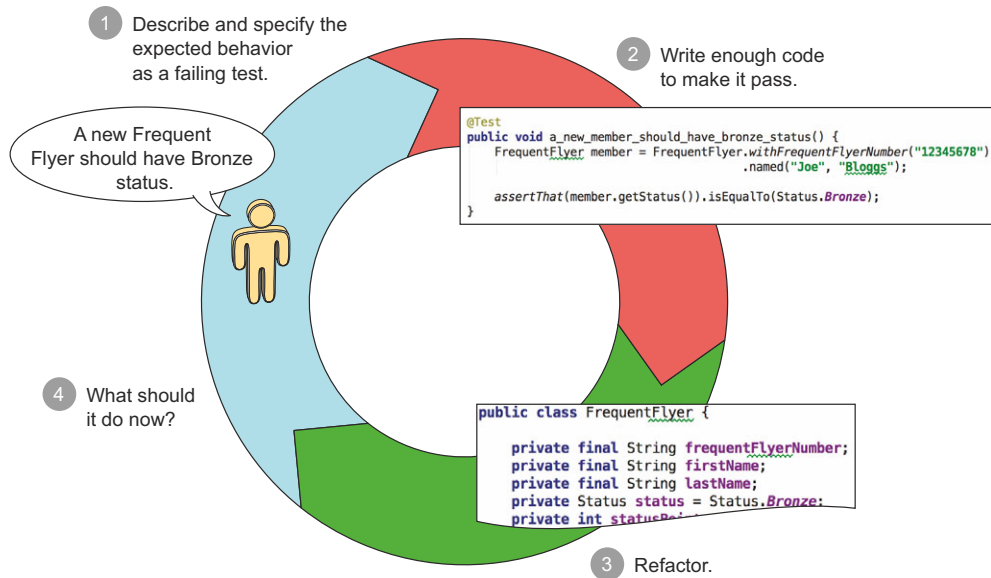


Figure 10.4 Use an incremental approach to progressively describe the behavior of a class.

To flesh out the application code required to implement the high-level step definitions, you write low-level specifications, expressed as unit tests, to progressively describe what your class should do. This approach of using unit tests to drive low-level design is a core TDD practice. The overall process is illustrated in figure 10.4, and can be summarized in the following steps:

- 1 Describe the behavior you need for a small piece of functionality in the form of a failing test.
- 2 Write just enough code to make this test pass.
- 3 Tidy up the code you just wrote, if required, to simplify it, remove duplication, and improve clarity.
- 4 Ask yourself “What else should the class do?” or “What input could I provide that should produce a different outcome?” and repeat the process.

When your acceptance criteria pass, you’re done!

For example, the step definition for registering a new Frequent Flyer member discussed previously looks like this:

```
FrequentFlyer member;

@When("^(?:s?)he registers on the Frequent Flyer program$")
public void registers_on_the_Frequent_Flyer_program() throws Throwable {
    member = FrequentFlyer.withFrequentFlyerNumber("123456789")
        .named(firstName, lastName);
}
```

Application code illustrating how to create a new Frequent Flyer member

1

This code illustrates how to create a new Frequent Flyer member ❶, but it lacks precision. You're assuming that the first name comes before the last name, but this isn't demonstrated explicitly. This sort of detail would be useful for a developer using the FrequentFlyer class to implement other features, but it would be of little interest to the business. In other words, it's a low-level technical requirement. Let's see how each step of this process works.

DESCRIBE THE EXPECTED BEHAVIOR AS A FAILING TEST

To illustrate and document this technical behavior more clearly, you start by illustrating what the method used in the step definition code (❶ in the previous code snippet) is expected to do, in the form of a unit test:

```
public class WhenRegisteringANewFrequentFlyerMember {
    ❶ @Test
    public void should_be_able_to_create_a_new_member() {
        FrequentFlyer member
            = FrequentFlyer.withFrequentFlyerNumber("123456789")
                .named("Jill", "Smith");

        ❷
        ❸
        assertThat(member.getFirstName()).isEqualTo("Jill");
        assertThat(member.getLastName()).isEqualTo("Smith");
        assertThat(member.getFrequentFlyerNumber()).isEqualTo("123456789"); #4
    }
}
```

Identify which detailed technical requirement or feature you're illustrating. ❶

Specify which business requirement you're working on.

Create a new Frequent Flyer member. ❷

Verify the expected outcomes. ❸

This method is more than just a unit test; it's a simple and concise executable specification. You've described the requirement ❶, given an example of how you'd like the FrequentFlyer API to work ❷, and described the expected outcomes of this operation ❸.

WRITE JUST ENOUGH CODE TO GET THE TEST TO PASS

You can now write some code for the FrequentFlyer class, as you see fit, to make this test pass. An important aspect of both TDD and BDD is trying to write the minimum amount of code that will make the test pass. In practice, this may be as little as a single line of code, or it may be a little more for simpler cases or where boilerplate code can be generated by the IDE.

In this case, you could create a simple domain object with a builder class containing the methods required by the API definition:²

```
public class FrequentFlyer {
    private String frequentFlyerNumber;
    private String firstName;
    private String lastName;

    protected FrequentFlyer(String frequentFlyerNumber,
                            String firstName,
                            String lastName) {...}
}
```

Getters omitted for brevity

Private constructor

² You can find the tests and application code discussed in this chapter in the sample code for the chapter.

```

public static FFBuilder withFrequentFlyerNumber(String number) {
    return new FFBuilder(number);
}

public static class FFBuilder {
    private String frequentFlyerNumber;

    public FrequentFlyerBuilder(String frequentFlyerNumber) {
        this.frequentFlyerNumber = frequentFlyerNumber;
    }

    public FrequentFlyer named(String firstName, String lastName) {
        return new FrequentFlyer(frequentFlyerNumber,
                                firstName,
                                lastName);
    }
}
}

```

Using a builder pattern

Testing getters and setters

Experienced BDD (and TDD) practitioners generally avoid testing getter and setter methods in themselves. When you need getters, setters, and constructors, they're exercised by the unit tests that describe the behavior that requires them. Remember, your unit tests are executable specifications and living documentation; they describe what the properties of an object are for and how they're used. If you test them explicitly, you're merely stating that your object needs a "name" or a "status" field, for example, but you're not explaining why or how they should be used.

You now have a builder that will allow you to write code like the following to instantiate a new `FrequentFlyer` object:

```

FrequentFlyer member = FrequentFlyer.withFrequentFlyerNumber("12345678")
    .named("Joe", "Bloggs");

```

REFACTOR IF REQUIRED

Once you've got the test to pass, it's always useful to review your code and see if it can be improved in any way. This includes code-quality considerations such as removing duplicated code, simplifying over-complex algorithms, grouping related lines of code into appropriately named methods, and making sure the variables are well named. But you can also refactor your code from an API perspective: is the API as simple and intuitive as it should be? For example, the `FrequentFlyer` builder might be more readable like this:

```

FrequentFlyer member = FrequentFlyer.withFrequentFlyerNumber("12345678")
    .withFirstName("Joe")
    .andLastName("Bloggs");

```

You should also include the executable specifications themselves in the refactoring activity; for example, does the name of the unit test reflect the functionality and code being illustrated? Are the variable names clear and accurate? And so on.

WHAT SHOULD THE APPLICATION DO NEXT?

When your unit test (or, more precisely, your executable specification) passes, you need to decide what test to write next. In BDD terms, you ask, “What should the application do that it doesn’t already do in order to satisfy the acceptance criteria I’m working on?” The acceptance criteria remind you of what you’re trying to achieve and help you avoid being sidetracked.

In this case, with the preceding builder implemented, there’s little left to explore in the area of creating new Frequent Flyer members, so you can safely move on to the last step in the acceptance criterion, which checks the status of the Frequent Flyer member. You could implement this step definition as shown here:

```
@Then("^(?:s?)he should have a status of (.*)$")
public void should_have_status_of(Status expectedStatus) {
    assertThat(member.getStatus()).isEqualTo(expectedStatus);
}
```

The Status class represents the concept of member status levels.

You need to be able to get a member’s status.

Writing this step definition leads to the discovery of a new domain class (the Status class) and a new method to obtain the status of a member. This requirement would probably be simple enough to implement directly from the step definition, but you could also choose to write a more focused unit test to illustrate the initial member status:

```
@Test
public void the_members_initial_status_should_be_bronze() {
    FrequentFlyer member
        = FrequentFlyer.withFrequentFlyerNumber("123456789")
                        .named("Jill", "Smith");

    assertThat(member.getStatus()).isEqualTo(Status.Bronze);
}
```

Create a Frequent Flyer member.

The initial status should be Bronze.

The implementation for this should be simple. You could implement member status as an enumerated type, like the following:

```
public enum Status { Bronze, Silver, Gold, Platinum; }
```

Then you’d simply add a status attribute and a getter method to the FrequentFlyer class:

```
public class FrequentFlyer {

    private String frequentFlyerNumber;
    private String firstName;
    private String lastName;
    private Status status = Status.Bronze;
    ...
    public Status getStatus() {
        return status;
    }
}
```

The first acceptance criteria, as well as the supporting unit tests, should now pass. Once you've verified this and done any necessary refactoring, you're ready to move on to the next acceptance criterion.

The process continues until you've written and implemented a set of low-level specifications for all of the classes involved in the high-level acceptance criteria. At this point, the acceptance criteria should pass.

10.2.8 How did BDD help?

Working this way gives you much finer control over what you're building and more confidence that the code actually works! Although this was a very simple requirement, the process we walked through did illustrate a number of important points.

THE ACCEPTANCE CRITERIA HELP YOU FOCUS ON VALUE

Starting with an acceptance criterion helps you remember what business value you're trying to deliver and provides a clear and convenient indicator so you know when you're done.

SMALL FEEDBACK LOOPS KEEP YOU ON TRACK

Working in very small feedback loops makes a lot of sense. The more code you write without the support of unit tests to back you up, the more chances you have of getting bogged down. The more code you write without a unit test to verify it, the more likely you'll write code that's not covered by a unit test, which both increases the risk of regressions and makes regression issues harder to isolate and fix when they do occur. For an experienced BDD or TDD practitioner, writing the unit tests first is a little like thinking aloud about the design.

THE UNIT TESTS ACT AS LOW-LEVEL SPECIFICATIONS AND DOCUMENTATION

In executable specifications like the previous ones, you're still describing behavior, but your target audience is different. The business stakeholders won't care about how a Frequent Flyer domain object is created, what its default values are, and so forth. But the developers who have to work with and maintain the code will. The unit tests you write are effectively technical specifications and documentation for other developers.

We've just walked through the process of taking a simple acceptance criterion and using it to produce a piece of working, verified production code. But this was a very simple case, and you could almost go directly from the step definitions to the production code. In the next section, we'll look at how to work with more involved acceptance criteria, where you'll need to use unit tests more actively to design and implement the production code.

10.3 Exploring low-level requirements, discovering design, and implementing more complex functionality

The acceptance criterion you implemented in the previous section was a very simple one, and it didn't require much in the way of unit tests or underlying code complexity. But this is usually not the case. In real-world applications, many acceptance criteria

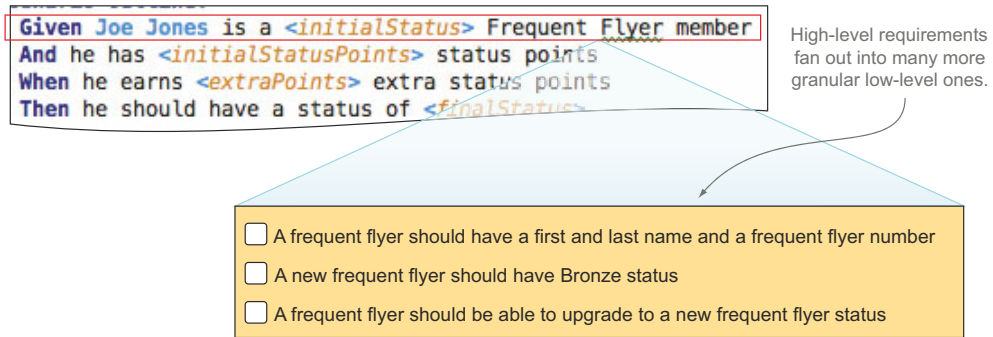


Figure 10.5 High-level requirements in the acceptance criteria fan out into many low-level ones.

hide a large amount of complexity under the hood. In some cases, the automated acceptance test may be sufficient to illustrate and verify this functionality; in other cases, each step in the high-level acceptance criteria may lead to a large number of low-level requirements that you'll express as BDD unit tests (see figure 10.5). This is typical of the BDD work process: as you implement the code required for an acceptance criterion, you'll often discover more low-level requirements that you'll also need to implement.

In this section we'll look at some of the techniques used when you expand high-level acceptance criteria into more detailed unit tests, discovering the classes, methods, and services you need as you go.

The best way to explore this idea is with a practical example. Let's see how this process would play out with the second of the acceptance criteria we introduced earlier in the chapter:

Scenario Outline:

```
Given Joe Jones is a <initialStatus> Frequent Flyer member
And he has <initialStatusPoints> status points
When he earns <extraPoints> status points
Then he should have a status of <finalStatus>
```

Describe how the application should behave.

Examples: Status points required for each level

initialStatus	initialStatusPoints	extraPoints	finalStatus
Bronze	0	300	Silver
Bronze	100	200	Silver
Silver	0	700	Gold
Gold	0	1500	Platinum

Illustrate the behavior with some basic examples.

Once again, the first task is to automate the steps in the acceptance criteria.

10.3.1 Use step definition code to explore low-level design

In this case, the definition for the first step might look like this:

```
@Given("^(.*) (.*) is a (.*) Frequent Flyer member$")
public void a_Frequent_Flyer_member(String firstName, String lastName, Status
    status) {
```

```

member = FrequentFlyer.withFrequentFlyerNumber("12345678")
                        .named(firstName, lastName);
member.setStatus(status);
}

```

1 Create a new member.

2 Update the member's status.

Here you reuse the builder method that was introduced in section 10.2.7 to create a new Frequent Flyer member ❶, and then update the status for this member ❷. In doing so, you discover that you need a setter method for the status field, which you can simply add to the FrequentFlyer class. A simple setter method like this would not justify a separate unit test, as the automated acceptance test is enough to illustrate how this code works.

The next step definition introduces a new concept: you need to keep track of a member's status points:

And he has <initialStatusPoints> status points

Status points are accumulated for each flight, but the business rules around how many status points are earned for a given flight are complex and vary depending on many factors (the length of the flight, the cabin category, special deals, and so forth). However, a key principle of BDD, and agile development in general, is to keep things simple until you have a good reason to make them more complicated. In this case, the simplest solution for the FrequentFlyer class would be to store the current number of status points. Using this approach, the step definition might look like this:

```

@Given("^(:s?)he has (.*?) status points$")
public void earned_status_points(int statusPoints) {
    member.setStatusPoints(statusPoints);
}

```

Again, this is a simple method that doesn't justify any dedicated unit tests.

The next step involves updating the status points of a member account:

When he earns <extraPoints> status points

Once again, writing the step definition gives you the opportunity to describe the ideal API you'd like to use to perform this operation. For example, you could use a fluent API approach like the one here:

```

@When("^(:s?)he earns (.*?) extra status points$")
public void earn_extra_status_points(int points) {
    member.earns(points).statusPoints();
}

```

This is a more sophisticated API, so you'd typically specify its behavior in more detail in a low-level executable specification, like this one:

```

@Test
public void a_member_should_be_able_to_earn_extra_status_points() {
    FrequentFlyer member =
        FrequentFlyer.withFrequentFlyerNumber("12345678")
                      .named("Joe", "Jones");
    member.setStatusPoints(100);
}

```

Given Joe Jones
is a member
with 100
status points...

```

member.earns(150).statusPoints();
assertThat(member.getStatusPoints()).isEqualTo(250);
}

```

When Joe earns 150 points...

Then he should have a total of 250 points.

This is still pretty simple code. The final step in the acceptance criterion is the interesting part, where you check the business logic around status points and status levels:

Then he should have a status of <finalStatus>

You've already implemented the step definition for this step:

```

@Then("^(?:s?)he should have a status of (.*)$")
public void should_have_status_of(Status expectedStatus) {
    assertThat(member.getStatus()).isEqualTo(expectedStatus);
}

```

But this is where things get a little more interesting. The acceptance criterion will still not pass, which indicates that the implementation is incomplete. When a member earns additional status points, their status should also be updated if enough points have been accumulated.

You could express this in more technical terms by writing the following unit test:

```

@Test
public void should_obtain_a_new_status_when_enough_points_are_earned() {
    FrequentFlyer member
        = FrequentFlyer.withFrequentFlyerNumber("12345678")
        .named("Joe", "Bloggs");

    member.setStatusPoints(100);
    member.earns(200).statusPoints();
    assertThat(member.getStatus()).isEqualTo(Status.Silver);
}

```

When a new Frequent Flyer member...

With 100 status points...

Earns 200 status points...

Then the member should get Silver status.

This is a low-level example of how the application code updates member status. Note that you've just expressed one of the examples from the acceptance criterion as a unit test. Although this unit test is a little redundant (after all, it tests exactly the same thing as one of the examples in the acceptance criterion), it's still useful, because it acts as the starting point for implementing the status upgrade functionality. It also concisely documents how this piece of functionality is implemented.

10.3.2 Working with tables of examples

Many acceptance criteria use example tables to summarize a number of related scenarios, as you saw in this scenario:

Scenario Outline:

```

Given Joe Jones is a <initialStatus> Frequent Flyer member
And he has <initialStatusPoints> status points
When he earns <extraPoints> status points
Then he should have a status of <finalStatus>

```

Describe how the application should behave.

Illustrate the behavior with some basic examples.

Examples: Status points required for each level

initialStatus	initialStatusPoints	extraPoints	finalStatus
Bronze	0	300	Silver
Bronze	100	200	Silver
Silver	0	700	Gold
Gold	0	1500	Platinum

When you implement this feature, you'll use these examples to guide your implementation. You may even expand on these examples to include edge cases or boundary conditions that are important from a technical perspective but of little interest to the business. At the unit-testing level, there are several approaches you can take.

Some developers write a single unit test to illustrate the technical implementation, and let the acceptance test verify the bulk of the examples. This works fine if the tests run quickly and the implementation is identical or very similar for each example. But when you're incrementally building a solution, it can be more convenient to have a unit-test case for each example, adding a new unit test for each example as you expand your implementation to cater to the different cases.

As you saw in the previous section, you can do this easily enough by writing separate unit tests for every example. While this works fine, it can be long-winded and repetitive, and can make refactoring harder.

Many unit-testing tools support example-driven testing to some extent. For example, both JUnit and NUnit support parameterized tests, which allow you to pass a table of values into a single unit test. The following example is in JUnit.

Listing 10.1 A data-driven unit test in JUnit

```
@RunWith(Parameterized.class)
public class WhenEarningStatusLevels {
```

```
    @Parameters
    public static Collection pointsPerStatus() {
        return Arrays.asList(new Object[][] {
            {Bronze, 0, 100, Bronze},
            {Bronze, 0, 300, Silver},
            {Bronze, 100, 200, Silver},
            {Silver, 0, 700, Gold},
            {Gold, 0, 1500, Platinum}
        });
    }
}
```

The test data goes here.

```
Status initialStatus, finalStatus;
int initialPoints, earnedPoints;
```

The test data values are stored in these fields.

```
public WhenEarningStatusLevels(Status initialStatus,
                                int initialPoints,
                                int earnedPoints,
                                Status finalStatus) {
    this.initialStatus = initialStatus;
    this.initialPoints = initialPoints;
    this.earnedPoints = earnedPoints;
    this.finalStatus = finalStatus;
}
```

The test data is passed into the unit test via the constructor.

```

@Test
public void should_earn_new_status_based_on_point_thresholds() {
    FrequentFlyer member
        = FrequentFlyer.withFrequentFlyerNumber("12345678")
            .named("Joe", "Jones")
            .withStatusPoints(initialPoints)
            .withStatus(initialStatus);

    member.earns(earnedPoints).statusPoints();

    assertThat(member.getStatus()).isEqualTo(finalStatus);
}
}

```

The test is run once for each row of data.

More modern unit-testing tools like Spock and Spec2 (see section 10.4.3) provide built-in support for data tables. For example, using Spock (which we looked at briefly in chapter 2), the unit test in listing 10.1 might look like the following listing.

Listing 10.2 A data-driven unit test in Spock

```

class WhenEarningStatus extends Specification {

    def "should earn status based on the number of points earned"() {
        given:
        def member = FrequentFlyer.withFrequentFlyerNumber("12345678")
            .named("Joe", "Jones")
            .withStatusPoints(initialPoints)
            .withStatus(initialStatus);

        when:
        member.earns(earnedPoints).statusPoints()

        then:
        member.status == finalStatus

        where:
        initialStatus | initialPoints | earnedPoints | finalStatus
        Bronze       | 0             | 100          | Bronze
        Bronze       | 0             | 300          | Silver
        Bronze       | 100          | 200          | Silver
        Silver       | 0             | 700          | Gold
        Gold         | 0             | 1500         | Platinum
    }
}

```

Create a new Frequent Flyer member.

The member earns some points.

Check the status.

The test data used in the previous steps comes from here.

In both these examples, using data tables makes the unit test easier to read, extend, and maintain.

10.3.3 Discover new classes and services as you implement the production code

So far, we've looked at several ways to describe what you'd like your application code to look like and how it should behave. You've described how you want to be able to upgrade a member's status in the code, but you haven't written any corresponding

application code. To get status upgrades to work, and the tests to pass, you need to write some application code. For example, you could make the following changes to the `FrequentFlyer` class:

```
StatusService statusService;

public void setStatusPoints(int statusPoints) {
    this.statusPoints = statusPoints;
    updateStatusLevel();
}

private void updateStatusLevel() {
    setStatus(statusService.statusLevelFor(statusPoints));
}
```

← You need a service to tell you what status can be obtained for a given number of points.

Update the status level to the appropriate level. ←

Here you’ve discovered the need for a new service (`StatusService`) and method (`statusLevelFor()`). This service needs to provide the status level that corresponds to a given number of status points.

Notice how the principle of outside-in development isn’t limited to tests. When you write implementation code, you can use the same principle, writing the code you’d like to have, and use this process to discover the classes and methods that you need. Whenever you write a piece of code that doesn’t yet exist, you’re discovering a new low-level requirement.

In general, when you come across something you need from another class or method, you have several choices:

- Implement the class or method immediately.
- Implement a minimum version of the class, and come back to it later.
- Defer implementation by using a “fake” class (a stub or mock) until your current test works, and then go back to implement the class.

Each of these approaches has advantages and trade-offs; experienced practitioners typically know how to use each, and how to pick the most appropriate approach for a given situation. In the following sections we’ll look at each of these approaches briefly.

10.3.4 *Implement simple classes or methods immediately*

Specifying and implementing immediately works well for simple, obvious implementations that can be done quickly. This is what you did in the previous section, where you implemented the `Status` class and the `getStatus()` method of the `FrequentFlyer` class directly, without needing any dedicated unit tests.

Oftentimes when you start to implement an apparently simple method or class, you realize that it’s more complicated than you initially thought. If a class has any nontrivial behavior, it’s a good idea to describe this behavior through “executable specification”-style unit tests.

To do this, you need to put your current test on standby while you implement the new method or class. For example, in JUnit, you might use the `@Ignore` annotation to temporarily skip this test:

```

@Ignore
@Test
public void should_obtain_a_new_status_when_enough_points_are_earned() {
    ...
}

```

← This test will now be skipped.

You could then implement the `StatusService` class or interface, and the `statusLevelFor()` method, using a test-driven approach (an example of doing this is illustrated in section 10.3.7). Once the method works, you'd come back and remove the `@Ignore` annotation, to ensure that the method works as expected in its original context.

This approach is simple and intuitive. But it does mean that you'll momentarily have more than one unit test in progress, and you'll need to remember to go back and restore the skipped test later on.

10.3.5 Use a minimal implementation

If the service class looks too complicated to implement in one go, another option is to code a minimal implementation of the real service class, and to use that for your test. This can help you get a feel for an API and how the classes should interact. Once you've finished with the `FrequentFlyer` code, for example, you'd come back and flesh out the status service implementation. Technically, this is an integration test rather than a unit test, which may have performance implications down the track.

10.3.6 Use stubs and mocks to defer the implementation of more complex code

Another approach, commonly used in outside-in development, is to use a stub or mock class to act as a placeholder for the real implementation, and at the same time to describe the behavior you expect of the new class. A stub or mock class is essentially a class you can control for testing purposes that stands in for a real class.

Using stubs and mocks helps you focus on the test at hand and avoids having several unit tests in progress at the same time. It also helps you clarify the roles and responsibilities of the classes and services in your application, which is very useful for larger, more complex applications. But it does add some overhead and complexity to your test classes.

USING MOCKS AND STUBS For a more detailed discussion of using mocks and stubs to discover and define application design, be sure to read the seminal book *Growing Object-Oriented Software, Guided by Tests*, by Steve Freeman and Nat Pryce (Addison Wesley Professional, 2009).

Let's see what this approach would look like for the `StatusService` classes.

MAKE SURE YOU CAN PROVIDE A CUSTOM VERSION OF THE SERVICE TO YOUR CLASS

Remember, you need the `StatusService` class to tell you what status a `FrequentFlyer` member should achieve with a given number of points:

```

StatusService statusService;
...
private void updateStatusLevel() {
    setStatus(Status.statusLevelFor(statusPoints));
}

```

You’ve also discovered that the `FrequentFlyer` class needs the `StatusService` to do its job, so you’ll need to make sure the `FrequentFlyer` class has an instance of `StatusService` available to use. There are many ways you could do this, such as using dependency injection frameworks like Spring or Guice in Java. But for testing purposes, you need to be able to provide your own custom version of the `StatusService` class:

```

protected FrequentFlyer(String frequentFlyerNumber,
                        String firstName,
                        String lastName,
                        Status status,
                        int statusPoints,
                        StatusService statusService) {
    ...
    this.statusService = statusService;
}

```

Pass a `StatusService` object in the constructor.

SPECIFY FOCUSED BEHAVIOR USING MOCK SERVICES

When you use this approach, a unit test focuses on specifying the behavior of a particular class or method and describes what it expects of any other classes it needs. This makes it very clear what other services your class needs to work. More interestingly, it gives you a chance to define the “contract” between your class and the service class you’re calling.

Mocking libraries exist for all modern testing frameworks, and you can also write your own stubs and mocks if you prefer. The following code uses the popular Mockito library for Java (<https://code.google.com/p/mockito/>):

```

@Test
public void should_cumulate_points_with_each_flight() {
    // GIVEN
    StatusService statusService = mock(StatusService.class);
    when(statusService.statusLevelFor(300)).thenReturn(Status.Silver)

    FrequentFlyer member = new FrequentFlyer("12345678", "Joe", "Bloggs",
                                             statusService);

    // WHEN
    member.earns(100).statusPoints();
    member.earns(200).statusPoints();

    // THEN
    assertThat(member.getStatusPoints(), is(greaterThanOrEqualTo((300))));
    assertThat(member.getStatus(), is(Status.Silver));
}

```

Use a mock version of the status service.

Use the mock status service for this `FrequentFlyer` member.

Return “Silver” for a value of 300.

Here you assume that the status service works, and that it returns the correct status when you ask about a given number of points. So this test could read, “Assuming that the status service tells us that the status level for 300 points is Silver, when a member

earns 300 points, then they should obtain a status of Silver.” Once you have this test working, you could proceed to implement the real version of this class.

This test is now totally independent of any particular implementation of the `StatusService` class. The test will run quickly and require no setup, and if it fails you’ll know that the problem is likely to be in the `FrequentFlyer` class and not in the `StatusService` class.

Of course, this now means that you need to write a separate specification for the `StatusService` class. In the next section you’ll see an example of some of these low-level specifications.

10.3.7 Expand on low-level technical specifications

In the previous section, you saw how mocks can be used to define contracts between classes. After these contracts are defined, you need to come back to ensure that your service class respects this contract. Likewise, in section 10.3.5 we discussed using a minimal implementation of the status service, but not one that would be sufficient for the full acceptance test. In both cases, you need to come back to complete the implementation.

Suppose you’re implementing this as an in-memory service. You might start off by defining how this service behaves for zero points:

```
StatusService statusService = new InMemoryStatusService();

@Test
public void should_stay_on_bronze_for_zero_points() {
    assertThat(statusService.statusLevelFor(0), is(Status.Bronze));
}
```

The status level for zero points should be Bronze.

Low-level specifications often cover cases that aren’t included in the acceptance criteria, such as boundary conditions like this one. Acceptance criteria are high-level examples that don’t need to cover every possible input and expected outcome, only the ones of significant business value. Often you’ll have more technical edge cases that you’d like to document that aren’t included in the acceptance criteria.

Once you’ve implemented enough code to make the code work for zero points,³ you might proceed to another edge case:

```
@Test
public void should_stay_on_bronze_up_to_299_points() {
    assertThat(Status.statusLevelFor(299), is(Status.Bronze));
}
```

What is the status level for 299 points?

Then you might proceed to one of the acceptance criteria examples:

```
@Test
public void should_earn_silver_for_300_points() {
    assertThat(Status.statusLevelFor(300), is(Status.Silver));
}
```

What is the status level for 300 points?

³ We won’t dwell on the implementation details here, but you can find a sample solution in the sample code for this chapter.

You'd continue with this process until you're confident that your class does everything it needs to do to satisfy the acceptance criteria you're working on.

The exact number of unit tests you need to write will vary from case to case. You could repeat this process for each status level, including edge cases, boundary conditions, and so on. This would make troubleshooting easier, but it could introduce some duplication between the acceptance tests and the unit test because the status levels are also described in the acceptance criteria; if the requirements change for the acceptance tests, the unit tests would need updating as well. Alternatively, if the `StatusService` used a database table, and the logic was identical for every status level, you might simply illustrate one status level and let the acceptance test do the work for the others.

Wording your requirements well: low-level BDD vocabulary

As you may have noticed in the examples so far, BDD practitioners often like to express requirements using the word “should.” For example,

```
"a new frequent flyer should have Bronze status"
```

or

```
"should be able to upgrade a frequent flyer status"
```

This vocabulary isn't an accident. Although some developers prefer more definitive wording like “must” or “will,” and produce fine executable specifications and living documentation doing so, the reasoning behind the BDD “should” are worth considering.

Language influences thought patterns. The words you use influence the way your message is perceived. Consider the following subconscious dialogue that our brains go through when reading a requirement:

Requirement: A new Frequent Flyer member status must be Bronze.

Brain: Yes Sir, very good Sir, right away Sir!

or

Requirement: A new Frequent Flyer member status should be Bronze.

Brain: Should it? Are there times when it might not be Bronze? What about special deals from marketing where they start out as Silver? What about if they're transferring points from a partner airline?

Only one word has changed, but the way we receive the information is quite different. The word “should” invites a question: “should it?” “Should” helps us justify why we want a class to provide a particular service, or to perform a particular task. Should also implies that any requirement, at any level, can be questioned, leaving communication channels much more open.

This use of the word “should” is, incidentally, quite different from the way the word is used in more traditional requirements documentation. In old-style formal specifications, and in legal documents, words like “must,” “should,” and “may” often have very precise meanings. For example, in the IETF recommendations, “must” or “shall”

(continued)

refers to a mandatory feature, “should” refers to a recommended one, and “may” refers to something that’s optional.⁴ The goal of the IETF recommendations is essentially to distinguish the really important requirements from the “nice-to-haves.”

BDD doesn’t use the word “should” in this way. By the time a feature or requirement gets to the stage of writing BDD-style acceptance criteria, there isn’t much room for optional requirements. If it’s in the acceptance criteria, it needs doing. If you discover that one of the acceptance criteria isn’t essential, you can safely remove it.

You’ve now seen what a typical BDD/TDD development process looks like. BDD unit testing is more an approach than a particular toolset, but there are tools and techniques that can make practicing BDD unit testing easier. We’ll look at a few of these in the next few sections.

10.4 Tools that make BDD unit testing easier

Many tools and techniques can smooth the transition to BDD-style unit-testing practices and make the unit tests you write more descriptive and easier to maintain:

- By adopting a descriptive coding style, you can practice BDD unit tests with traditional unit-testing tools such as JUnit and NUnit.
- Other unit-testing tools, such as RSpec, NSpec, and Jasmine, put the emphasis on writing low-level executable specifications in the form of unit tests, which encourages a more rigorous, test-first approach.
- More recent BDD unit-testing tools such as Spock and Spec2 allow for more expressive and powerful low-level specifications, including example-driven specifications.

Let’s start by looking at what you can do with traditional unit-testing tools.

10.4.1 Practicing BDD with traditional unit-testing tools

It can sometimes be less intimidating for teams starting out with BDD to keep their existing unit-testing tools. In addition, teams often have a large investment in existing unit tests using a traditional unit-testing library, and they’re understandably reluctant to migrate to a new toolset. In this section, we’ll look at a few simple steps you can take to start your journey down the road of BDD unit testing without changing your current unit-testing toolset.

WRITE TEST METHOD NAMES THAT DESCRIBE THE BEHAVIOR

The first step toward a more BDD-style approach to unit testing can be as simple as giving your tests more meaningful names. Traditionally, many unit tests use the convention

⁴ See Network Working Group, RFC 2119, “Key words for use in RFCs to Indicate Requirement Levels,” www.ietf.org/rfc/rfc2119.txt.

of starting test method names with the word “test.” For example, the following JUnit tests follow this convention:

```
@Test
public void testAddStatusPoints() {...}

@Test
public void testTransferPoints() {...}
```

Some earlier unit-testing libraries required you to respect this convention, and many development tools still encourage it by letting you generate a unit test for each method in your class, prefixed with the word “test.” And this convention does make it easy to see where a particular method has been tested.

But from a BDD perspective, this approach has some limitations. First, it assumes that the method under test already exists, or that you know what it should be called from the outset. It also binds a test to a particular method, which limits the ways you might want to test a given method and makes for more work when refactoring.

In BDD you focus on specifying the *behavior* of a class, rather than just testing its methods. After all, the methods are simply a way of getting the class to perform some particular task or behavior. To reflect this, instead of naming a test after the method it’s testing, you name the test based on what you expect the class to do. For example, in .NET you might write tests like the following:

```
[TestFixture]
public class WhenUpdatingStatusPoints
{
    [Test]
    public void ShouldBeAbleToAddStatusPointsEarnedFromAFlight()
    {...}

    [Test]
    public void ShouldUpdateStatusWhenEnoughStatusPointsAreEarned()
    {...}
}
```

These test names are designed to be part of the living technical documentation, so clarity and readability are key. Some Java and .NET practitioners prefer to use a more readable notation for test names based on underscores instead of CamelCase. For example, in JUnit you could write the following:

```
@Test
public void should_be_able_to_add_status_points_earned_from_a_flight() {
    ...
}

@Test
public void should_update_status_when_enough_status_points_are_earned() {
    ...
}
```

Other practitioners use a slight variation on this approach that respects the use of the “test” prefix. Instead of using the word “should,” they use “test that.” This works well

for unit-testing libraries that still require the “test” prefix on unit-test method names, as illustrated in the following Python examples:

```
class WhenUpdatingStatusPoints(unittest.TestCase):
    def test_that_status_points_can_be_added(self):
        ...

    def test_that_status_is_updated_when_enough_points_are_earned(self):
        ...
```

In all of these cases, you’re describing what a class should do, rather than testing individual methods. This makes the tests read more like technical documentation for the class that describes and illustrates the intended usage of the class, rather than just listing method names prefixed with the word “test.” It also makes the test cases more robust: method names, and how you use them, are much more likely to change during refactoring than the expected behavior of the class.

USE TEST CLASS NAMES THAT PROVIDE CONTEXT

In a similar vein, BDD practitioners like to name the test classes in a more meaningful way. The name of a BDD-style unit-test class isn’t related to the class being tested (which, when you create the test class, may not even exist). Rather, it gives the context of the behavior being described.

A useful trick for doing this is to prefix the test class name with the word “When.” This approach, combined with the more expressive test names discussed earlier, makes the tests read much more like technical specifications:

```
public class WhenUpdatingMemberStatusPoints {
    @Test
    public void should_be_able_to_add_status_points_earned_in_a_flight() {
        ...
    }
    @Test
    public void should_update_status_when_enough_status_points_are_earned() {
        ...
    }
    ...
}
```

◀— The test class name describes the feature or context.

◀— The test names describe the detailed requirements.

These techniques let you obtain many of the benefits of BDD unit testing without needing to change your unit-testing tools. But BDD unit-testing tools that are more focused, such as RSpec for Ruby, NSpec for .NET, and Spock for Java and Groovy, make things a little easier. They make it simpler and more natural to write unit tests in the form of clear, readable, low-level executable specifications. In the next section, we’ll look at how they can help.

10.4.2 Writing specifications, not tests: the RSpec family

Inspired by the concepts of BDD, in the mid-2000s a new generation of low-level BDD tools emerged that put a greater distance between “executable specifications” and

“tests.” When you write a test in JUnit or NUnit, it’s difficult to avoid mentioning it somewhere. You use the `@Test` annotation, for example, or you make assertions about the result value. These conventions come from the mindset of writing unit tests once the code is complete, which, as you’ve seen, doesn’t reflect how BDD and TDD work. Even if you try to adopt more BDD-style naming conventions, xUnit tests are still focused on *verification* rather than *specification*. In BDD, the idea is to use unit tests to implement low-level executable specifications, not to test code that has already been written.

Members of the Ruby community have traditionally been very early adopters of BDD practices and tools, and in 2005 RSpec, the first BDD unit-testing tool, was released (<http://rspec.info>). RSpec is fairly typical of a whole family of low-level BDD tools, so we’ll look at RSpec first, and then see what other similar libraries exist for other languages.

WRITING LOW-LEVEL EXECUTABLE SPECIFICATIONS IN RUBY WITH RSpec

The main idea behind RSpec is, in true BDD fashion, to think in terms of describing application behavior rather than verifying code. In RSpec, rather than speaking in terms of tests, you talk about *specifications*, expressed in terms of executable examples.

For example, to specify the requirement that a Frequent Flyer should initially have a Bronze Frequent Flyer status, you could write the following specification:

```
describe FrequentFlyer do
  it 'should initially have Bronze status' do
    frequentFlyer = FrequentFlyer.new
    expect(frequentFlyer.status).to eq('BRONZE')
  end
end
```

What should it do or have?

You’re talking about the Frequent Flyer domain object.

Create a new Frequent Flyer object.

You expect the status to be Bronze.

This descriptive, example-based approach steers away from mentioning the word “test” at all; it’s more in line with the BDD philosophy of living documentation and executable specifications. This can be a great help in getting developers into the BDD mindset of writing specifications rather than tests.

RSpec provides many features that make it easier to structure these executable specifications in a sensible manner. For example, you can group requirements into different contexts to give more background to each set of granular specifications:

```
describe FrequentFlyer do
  context 'when the frequent flyer account is first created' do
    it 'should initially have Bronze status' do
      frequentFlyer = FrequentFlyer.new
      expect(frequentFlyer.status).to eq('BRONZE')
    end
  end

  context 'when a new member starts to fly with Flying High' do
    it 'should earn points for each flight' do
      frequentFlyer = FrequentFlyer.new
      frequentFlyer.earn_status_points(100)
    end
  end
end
```

1 Given...

Group requirements by context.

Contexts can contain several related requirements.

```

    expect(frequentFlyer.status_points).to eq(100)
  end

  it 'should upgrade member status when enough points are earned' do
    frequentFlyer = FrequentFlyer.new

    frequentFlyer.earn_status_points(100).earn_status_points(200)

    expect(frequentFlyer.status).to eq('SILVER')
  end
end
end

```

Contexts can contain several related requirements.

1 Given...

2 When...

3 Then...

Specifications written this way should stay simple and concise, respecting the BDD approach of favoring simplicity and communication. If you study the structure of these specifications, you may notice that they actually respect a “Given ... When ... Then” style. For example, in the last specification above, the *Given* creates a new Frequent Flyer object ❶, the *When* is the action you’re demonstrating ❷, and the *Then* is the expected outcome ❸. Even in the more concise RSpec style, respecting this structure will make your specifications much cleaner and easier to understand.

These specifications are not only quite readable, they’ll also produce reports that describe the low-level specifications in a more structured way than a list of unit-test results. For example, running `rspec` from the command line would produce a summary of the requirements in text form:

```

$rspec --format documentation

FrequentFlyer
  when the frequent flyer account is first created
    should initially have Bronze status
  when a new member starts to fly with Flying High
    should earn points for each flight
    should upgrade member status when enough points are earned

```

RSpec has many other interesting features that we don’t have time to discuss here. Since RSpec was released, other languages have also adopted RSpec-style testing libraries for BDD unit testing, including NSpec for .NET and a number of libraries for JavaScript. As an example, let’s look at one of these.

EXECUTABLE SPECIFICATIONS IN JAVASCRIPT WITH JASMINE

JavaScript is playing an increasingly important role in modern application development, both on the client side, and, with the rise of JavaScript application platforms like Node.js, on the server side.

Although more traditional unit-testing tools such as QUnit are widely used in the JavaScript world, there are several BDD-style JavaScript unit-testing libraries around. The best known of these are Jasmine (<http://jasmine.github.io>) and Mocha (<http://visionmedia.github.io/mocha/>). As far as BDD goes, both these libraries have a very similar style. You’ve already seen Jasmine in action in chapter 9, where you used it to test an AngularJS web application. Let’s take a closer look at how it works.

Jasmine lets you describe granular requirements in a format similar to RSpec. It supports declarative, example-based specifications, nested contexts, and its own mocking library. Here's a simple example of some requirements expressed using Jasmine:

```
describe('Frequent Flyers', function() {
    var frequentFlyer;

    beforeEach(function() {
        frequentFlyer = require('../lib/frequent_flyer');
    });

    describe("Managing Frequent Flyer statuses", function() {
        it("should initially have Bronze status", function() {
            expect(frequentFlyer.getStatus()).toBe('Bronze');
        });
        it("should initially have no status points", function() {
            expect(frequentFlyer.getStatusPoints()).toBe(0);
        });
    });

    describe("Cumulating Frequent Flyer points", function() {
        it('should earn points for each flight', function() {
            frequentFlyer.earnStatusPoints(100);
            frequentFlyer.earnStatusPoints(50);

            expect(frequentFlyer.getStatusPoints()).toBe(150);
        });

        it('should upgrade member to next status level when enough points
            are earned', function() {
            frequentFlyer.earnStatusPoints(300);

            expect(frequentFlyer.getStatus()).toBe('Silver');
        });
    });
});
```

1 A set of related specifications.

2 Nested requirements.

3 it marks a specification.

4 Jasmine uses "expect" to describe expected outcomes.

This is the object under test.

Do this before each specification.

As you can see, Jasmine specifications have a layout that's very similar to the layout of RSpec specifications. Groups of related specifications are marked by the `describe` function **1**, which can contain specifications or more nested `describe` functions **2**. Each specification is marked by the `it` method **3**, which is made up of a title and a function containing the specification test code. Jasmine uses the `expect()` function **4** to describe expected outcomes, though as you'll see in section 10.5.2, this isn't your only option.

EXECUTABLE SPECIFICATIONS IN .NET WITH NSPEC

You've seen how you can write NUnit tests in a more BDD style. But for teams who prefer a purer BDD approach, NSpec is an elegant BDD library that lets you write RSpec-style specifications directly in .NET (<http://nspec.org/>).

A sample set of specifications can be seen here:

```
public class WhenUpdatingStatusPoints : nspec
{
    FrequentFlyer member;
```

```

void before_each()
{
    member = new FrequentFlyer();
}

void earning_status_points()
{
    context["When the frequent flyer account is created"] = () =>
    {
        it["should have BRONZE status"] = () =>
            member.getStatus().should_be(Status.Bronze);
        it["should have 0 status points"] = () =>
            member.getStatusPoints().should_be(0);
    };

    context["When cumulating Frequent Flyer points"] = () =>
    {
        it["should earn points for each flight"] = () =>
        {
            member.earnStatusPoints(100);
            member.earnStatusPoints(50);
            member.getStatusPoints().should_be(150);
        };

        it["should get upgrade when enough points are earned"] = () =>
        {
            member.earnStatusPoints(00);
            member.getStatus().should_be(Status.Silver);
        };
    };
}
}

```

“should”-style assertions

A new specification context

Specifications within the context

This is very close, at least in spirit, to the RSpec and Jasmine examples you saw earlier. As with those other tools, you describe your specifications in natural language and group them within contexts, using very clean “should”-style assertions.

10.4.3 Writing more expressive specifications using Spock or Spec2

Tools like RSpec, Jasmine, and NSpec are great ways of expressing BDD-style low-level specifications, but they do have some limitations. For example, there are times when it would be nice to have a more explicit “Given ... When ... Then” structure. Although good practitioners do this with whatever tool they’re using, it can often help teams to work more consistently if the structures are more visible.

In addition, example-driven specifications can often be expressed more concisely using examples. You saw the use of table-based examples with tools like JBehave and Cucumber, but they’re equally useful at the unit-testing level.

RSpec was released over eight years ago, and more recently a new generation of BDD unit-testing tools has emerged. These newer tools provide the expressiveness of Gherkin, including support for example tables and clear “Given ... When ... Then” structures, without the overhead of maintaining separate feature files and step-definition classes.

They do this by mixing the code with the specification definitions, as is done in RSpec, but by also providing support for much richer BDD language structures.

There aren't many unit-testing BDD tools that support these features yet, but there are a few. In Scala, for example, Specs2 is one such tool (<http://etorreborre.github.io/specs2/>). And if you're working with Java or Groovy, you can use Spock (<https://code.google.com/p/spock/>). Both tools can be used very effectively to test Java code. Let's take a closer look at Spock.

Spock is a powerful and very expressive BDD unit-testing tool that lets you write clean and readable executable specifications in Groovy. Here's a simple example:

All Spock specifications extend the Specification class.

```
class WhenManagingFrequentFlyerMembers extends Specification {

    def "a new frequent flyer should have Bronze status"() {
        given:
            def member = FrequentFlyer.withFrequentFlyerNumber("12345678").
                named("Joe", "Bloggs")

        when:
            def status = member.status

        then:
            status == FrequentFlyerStatus.BRONZE
    }
}
```

1 Given

2 When

3 Then

Notice how this example is structured. The specification is organized into clear *Given* ①, *When* ②, and *Then* ③ sections. Of course, Spock, like Gherkin, gives you a fair bit of flexibility in how you organize your “Given ... When ... Then” blocks, but they're nevertheless present and clearly marked.

Spock also makes example-driven testing very easy.

```
def "should upgrade status when enough status points are acquired"() {
    given: "a frequent flyer member with some points"
        def member = FrequentFlyer.withFrequentFlyerNumber("12345678").
            named("Joe", "Bloggs").
            withStatusPoints(initialPoints).
            withStatus(initialStatus)

    when: "he earns some extra points on a flight"
        member.earns(extraPoints).statusPoints()

    then: "he may or may not be upgraded to a new status"
        member.getStatus() == expectedStatus

    where:
        initialStatus | initialPoints | extraPoints | expectedStatus
        BRONZE        | 0             | 299         | BRONZE
        BRONZE        | 0             | 300         | SILVER
        SILVER        | 0             | 699         | SILVER
        SILVER        | 0             | 700         | GOLD
        GOLD          | 0             | 1499        | GOLD
        GOLD          | 0             | 1500        | PLATINUM
}
```

1 Precondition

2 Action

3 Expected outcome

4 Examples

This specification, using a format that's quite similar to the Gherkin equivalent, feeds test data in from the examples table ④. The column headers are injected into the *Given* ①, *When* ②, and *Then* ③ steps, where they're used to do the actual test.

This example also adds some extra information about each step. The texts following the given, when, and then labels are optional, but they're often used to make the intent of the specification clearer. Sometimes a bit of extra explanation goes a long way. Other times the code itself is sufficient.

Spock also supports a more lightweight syntax that's useful for example-based testing. For example, you could describe the minimum points required to upgrade to the next status level like this:

```
class WhenCheckingMinimumStatusPoints extends Specification {
  def "should know the minimum points required for each status level"() {
    expect:
      FrequentFlyerStatus.statusLevelFor(points) == expectedStatus
    where:
      points | expectedStatus
      0      | BRONZE
      299    | BRONZE
      300    | SILVER
      699    | SILVER
      700    | GOLD
      1499   | GOLD
      1500   | PLATINUM
  }
}
```

The status level
obtained with a given
number of points

The expected
statuses for a
range of
point values

The approach used by tools like Spock and Specs2 hits a sweet spot between expressive executable requirements and technical documentation. When using tools like Cucumber or JBehave, you keep the feature files separate from the step definitions, since the scenarios are discussed, defined, and owned by the team as a whole. Adding implementation code to the mix would make these feature files harder for non-developers to understand. But the more granular, low-level executable requirements we've looked at in this chapter need to combine technical documentation, business justification, and sample code.

10.5 Using executable specifications as living documentation

From a BDD perspective, writing a good unit test is an exercise in good communication. When you practice BDD, you think of every unit test as a low-level specification that illustrates some aspect of how a class or component behaves. You've seen ways to help ensure that the intent of these low-level specifications is clearer. But the implementation of your test is also sample code that illustrates how a particular requirement is satisfied, or how a particular goal is achieved. The code inside your tests doesn't just exercise the application; it documents *how to exercise* the application.

Not keeping your test code clean and easy to understand has very practical consequences. If an old test fails when you make a change elsewhere in the code base, it's essential to understand both what the test was trying to demonstrate and how it was doing so. Only by understanding this will you be in a position to decide what to do with the failing test.

An important part of this is keeping your test code simple. Complex, convoluted test code is hard to understand and maintain, and is more difficult for others (and yourself, later on!) to understand.

One very effective way to make your code easier to read and to understand is to use fluent coding practices.

10.5.1 *Using fluent coding to improve readability*

Fluent coding can help you write code that communicates your intent more effectively. We say that code is *fluent* when it reads more like a natural-language sentence than like something written for a compiler.

For example, suppose you need to create a flight for your tests. Using conventional code in Java or .NET, you might write something like this:

```
Flight lastPlaneOut = new Flight("FH-525", "Hong Kong", "Sydney")
```

Although it's globally clear what this code does, there's still room for doubt. For example, you could probably guess that the first parameter represents the flight number, but does the flight leave from Hong Kong or Sydney?

With a little more effort, you could design the testing API so that the tests might read more like this:

```
Flight lastPlaneOut = Flight.number("FH-525").from("Sydney")  
                                .to("Hong Kong");
```

The idea of fluent code isn't language-specific. Dynamic languages like Ruby and Groovy, for example, support relatively fluent constructions natively:

```
def lastPlaneOut = new Flight(from: "Sydney",  
                              to: "Hong Kong",  
                              number: "FH-525")
```

The benefits of fluent code also apply to assertions. A well-written assertion tells you at a glance what a test is trying to demonstrate. It should be simple and obvious. You shouldn't have to decipher conditional logic or sift through `for` loops to understand what the code is doing. In addition, any nontrivial logic in a test case increases the risk of the test itself being wrong.

In recent years there has been a rise in the popularity of tools and techniques that make it easier to write more fluent code, both for production code and for tests. In the testing space, in particular, many libraries support fluent assertions in different languages. There are two main flavors to fluent assertions. The first typically uses the word “assert,” whereas the second uses terms like “should” or “expect.”

The first approach comes from a more traditional unit-testing background and focuses on testing and verification. Indeed, you typically make an assertion about something that has already happened, or a result that has already been calculated. The second is more BDD-centric: the words “should” and “expect” describe what you think the application should do, regardless of what it does currently, or if it even exists.

Let's look at some examples of fluent assertion libraries in different languages.

10.5.2 Fluent assertions in JavaScript

JavaScript has a number of libraries that can help make your assertions more expressive. You’ve seen, for example, the built-in `expect()` method that comes with Jasmine:

```
expect(frequentFlyer.getStatus()).toBe('Silver');
```

But this sort of expressiveness isn’t limited to Jasmine. `Should.js` (<https://github.com/visionmedia/should.js/>) and `Chai` (<http://chaijs.com>) are other well-known libraries that support similar features.

Chai is probably the most flexible of these, supporting both the “expect” and “should” formats, as well as the old-school `assert`. Chai focuses on using method chaining to make the assertions fluid and readable. For example, the Chai equivalent of the Jasmine `expect` statement would be the following:

```
var expect = require('chai').expect
...
expect(frequentFlyer.getStatus()).to.equal('Silver');
```

Import the Chai
expect function.

Expect assertions
are similar to those
in Jasmine.

As you might expect, Chai supports a rich collection of assertions and can chain multiple assertions together. For example,

```
var obtainableStatuses = ['Silver', 'Gold', 'Platinum']
...
expect(obtainableStatuses).to.have.length(3).and.to.include('Gold')
```

Chai also supports the more BDD-style `should` assertion, as illustrated here:

```
var expect = require('chai').should();
frequentFlyer.getStatus().should.equal('Bronze');
obtainableStatuses.should.have.length(3).and.include('Silver');
```

Note that you’re calling the
`should()` function, not importing it.

Should can now be
called on any object.

All the assertion methods can be
called using either `expect` or `should`.

Both styles are equally expressive, so the choice is largely a question of style and personal preference.

10.5.3 Fluent assertions in static languages

Fluent assertion libraries also exist for static languages such as Java and .NET, although they’re generally a bit less expressive than their dynamic equivalents.

Java, for example, has several fluent assertion libraries. The two best known are `Hamcrest` (<https://github.com/hamcrest/JavaHamcrest>) and `FEST-Assert` (<https://github.com/alexruiz/fest-assert-2.x/wiki>). More recent versions of `JUnit` come with a similar constraint-based `assert` model, and .NET developers can also use the very rich `Fluent Assertions` library (<https://github.com/dennisdoomen/FluentAssertions>).

All of these libraries move away from the older-style assert methods and let you express your expectations in a more fluent and concise manner. In particular, they propose a number of higher-level assertions on collections and can be extended to work with domain objects, which helps you to avoid having to put too much logic within your unit tests.

Imagine you want to say that the member status should initially be Bronze. In traditional JUnit, you'd write something like this:

```
assertEquals(BRONZE, member.getStatus());
```

The parameter order and the somewhat clumsy wording make this sort of assertion less than ideal. It doesn't read fluently or naturally, which limits your ability to express your expectations easily and quickly.

An equivalent Hamcrest assertion, on the other hand, would look like this:

```
FrequentFlyer member = FrequentFlyer.withFrequentFlyerNumber("12345678")
    .named("Joe", "Bloggs");
assertThat(member.getStatus(), is(FrequentFlyerStatus.BRONZE));
```

FEST-Assert does something similar, but using a different syntactic structure:

```
assertThat(member.getStatus()).isEqualTo(FrequentFlyerStatus.BRONZE);
```

In NUnit, you could write something like this:

```
Assert.That(member.getStatus(), Is.EqualTo(FrequentFlyerStatus.BRONZE));
```

The .NET Fluent Assertions library uses a more natural style built around the word "should:"

```
Member.getStatus().Should().Be(FrequentFlyerStatus.BRONZE);
```

All of these libraries propose a rich set of matchers, including a number of convenient operations on lists. For example, to check that the list of unachieved statuses contains both Gold and Platinum, you could write the following Hamcrest assertion:

```
assertThat(member.getUnachievedStatuses(), hasItems(GOLD, PLATINUM));
```

In FEST-Assert, the equivalent would be similar:

```
assertThat(member.getUnachievedStatuses()).contains(GOLD, PLATINUM);
```

With Fluent Assertions, the assertion might look like this:

```
member.getUnachievedStatuses().Should().Contain(GOLD).And.Contain(PLATINUM);
```

All of these libraries also allow more complex expressions. For example, suppose you wanted to verify that all of the Frequent Flyers in a particular group were under the age of 18. You could express this quite elegantly in Hamcrest like this:

```
List<Integer> memberAges = ...;
assertThat(memberAges, everyItem(lessThan(18)));
```

You could also do something similar in NUnit:

```
Assert.That(memberAges, Has.All.LessThan(18));
```

Alternatively, using the .NET Fluent Assertions library, you could write something like this:

```
memberAges.Should().Contain(item => item < 18);
```

Traditionally, higher-level assertions like this would often require loops and nontrivial logic in the unit tests. This isn't only risky; it's also often enough to discourage developers from testing nontrivial outcomes. By making it easier for developers to express their expectations effectively, fluent assertion libraries contribute to more meaningful and higher quality executable specifications.

Hamcrest and FEST-Assert play similar roles in Java-based BDD. Hamcrest is more flexible and easier to extend, but FEST-Assert has a simpler syntax and is a little easier to use. The constraint-based assert model in NUnit is similar, and it has a particularly rich library of assertions. And the Fluent Assertion library proposes an expressive style of assertions with a very BDD feel to it. All of these are a vast improvement on the traditional assert statements.

Fluent assertion libraries are in no way specific to BDD, and they can be used to make any unit tests easier to understand. But their emphasis on readability, expressiveness, and communication makes them well aligned with the BDD philosophy.

10.6 Summary

In this chapter you learned about practicing TDD techniques using a BDD style:

- Low-level BDD can be considered as an extension of classic TDD, or even “TDD practiced very well.”
- BDD acceptance criteria lead to lower-level BDD unit tests, and help you discover the detailed application design.
- BDD unit-testing tools exist for virtually all modern languages and platforms, with the RSpec family of tools being very widespread.
- More recent BDD tools like Spock and Specs2 are more powerful and expressive.
- Fluent assertion libraries make it easier to express your expectations clearly.

In the next chapter, we'll look at how BDD fits into the broader picture of project management and reporting, and how to get the most out of your living documentation.

Part 4

Taking BDD Further

In part four, we'll look at some more advanced aspects of BDD and see how they fit into the overall build and release cycle.

If you've gotten this far, you've seen all of the core aspects of BDD, with one important exception: living documentation. Communication and feedback are essential parts of the BDD process, and living documentation is an important part of the communication process. In chapter 11, you'll see how to produce high-quality living documentation out of your automated acceptance criteria. We'll introduce concepts such as Feature Readiness and Feature Coverage, which can be used to keep tabs on project progress and overall product quality. We'll also see how well written BDD unit tests serve as effective technical documentation.

Finally, in chapter 12, we'll look at how BDD fits into the overall build process and the role it plays in Continuous Integration and Continuous Delivery.

11

Living Documentation: reporting and project management

This chapter covers

- What we mean by “living documentation”
- Keeping track of project progress using feature readiness and feature coverage
- Organizing your living documentation
- Technical living documentation

In this chapter, we’ll focus on an important part of BDD that you need to understand if you’re to get the most out of whatever BDD strategy you adopt. You’ve seen how BDD encourages teams to express requirements in terms of executable specifications that can be run in the form of automated tests. These executable specifications become the definitive reference (often referred to as the “source of truth”) for the current set of application requirements. The definitive form of these executable specifications is generally source code, so they fit neatly into the overall development process and drive the automated tests. The reports generated by the automated tests refer back to the original executable specifications. These reports, which combine the original specifications, acceptance criteria, and test results, are what we call *living documentation*.

11.1 *Living documentation: a high-level view*

BDD reports don't simply provide a list of test outcomes, in terms of passing or failing tests. First and foremost, BDD reports document and describe what the application is expected to do, and they report whether or not the application actually performs these operations correctly. When you drill down into the details, a BDD report also illustrates *how* a particular feature or functionality is performed, from the user's perspective.

Living documentation targets a broad audience (see figure 11.1). You've seen how BDD encourages teams to collaborate to define acceptance criteria in the form of concrete examples, scenarios, and executable specifications, and how these guide the development and delivery of the features being built. As features are delivered, the living documentation ties the features back to the original requirements, confirming that what was delivered corresponds to what the team originally discussed.

In this way, BDD reporting completes the circle that started with the initial conversations with business stakeholders. The stakeholders, business analysts, testers, and anyone else that participated in the conversations leading up to the scenarios

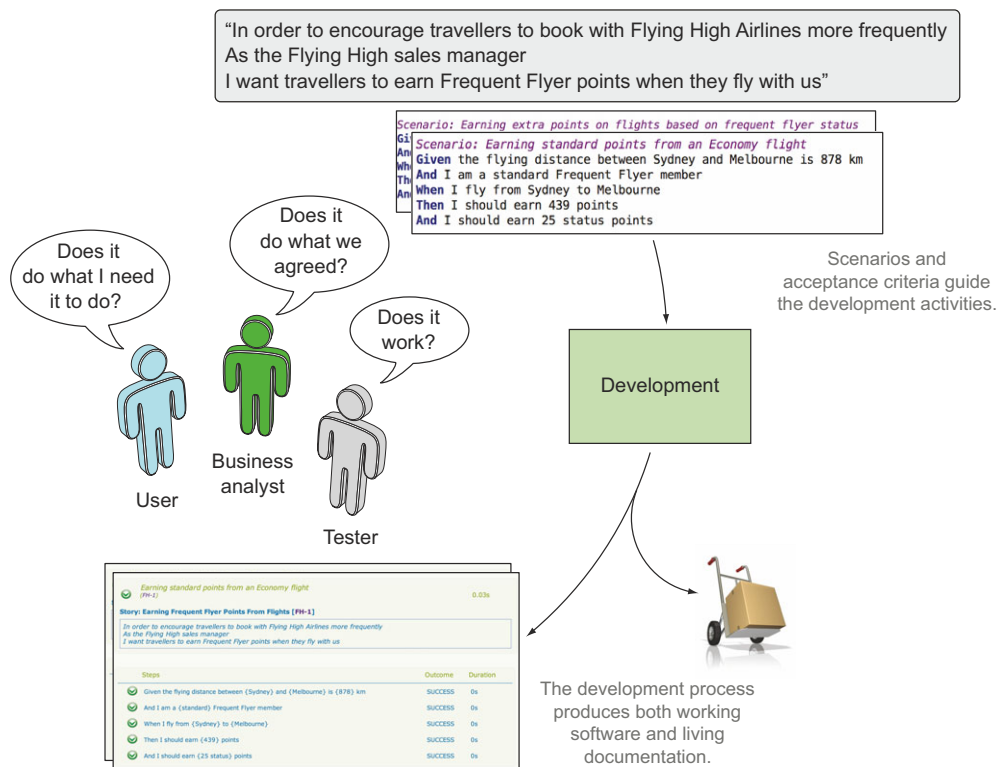


Figure 11.1 Living documentation provides feedback to the whole team, but in particular to business analysts, testers, and business stakeholders.

and executable specifications see the conversations they had, and the examples they discussed appear as part of the generated reports. This feedback loop is a great way to get buy-in from business stakeholders, who are often much more keen to contribute actively when they see the results of their contributions verbatim in the living documentation. In addition, because the reports are generated automatically from the automated acceptance criteria, it's a fast and efficient way of providing feedback once it's set up (see figure 11.2).

Testers also use the living documentation to complement their own testing activities, to understand how features have been implemented, and to get a better idea of the areas in which they should focus their exploratory testing.

The benefits of living documentation shouldn't end when a project is delivered. When organized appropriately, living documentation is also a great way to bring new

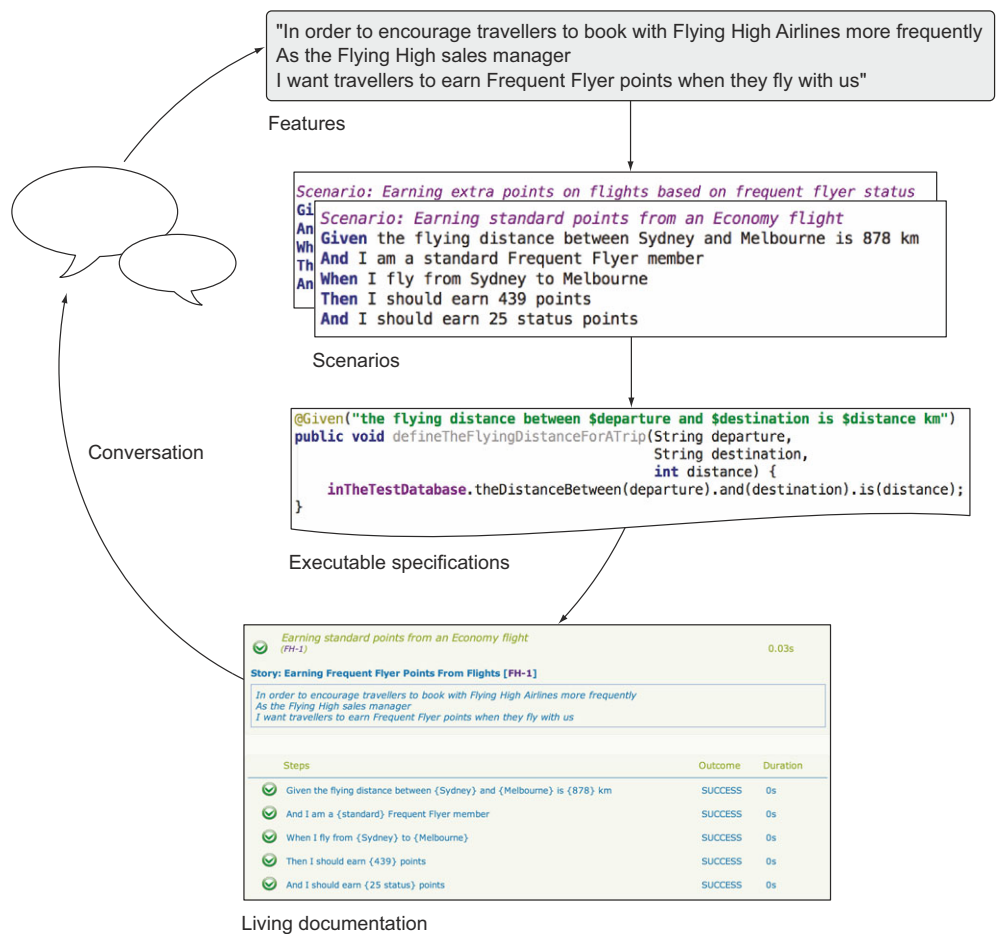


Figure 11.2 Living documentation is generated automatically from the executable specifications, which simplifies reporting and accelerates the feedback cycle.

team members up to speed not only with what the application is supposed to do, but also how it does so. For organizations that hand over projects to a different team once they go into production, the benefits of this alone can be worth the time invested in setting up the living documentation reporting.

But living documentation goes beyond describing and illustrating the features that have been built. Many teams also integrate their BDD reports with agile project management or issue-tracking systems, making it possible to focus on the state of the features planned for a particular release. Teams that do this typically rely on the living documentation to produce their release reports, if they aren't generated automatically from the BDD reports.

In the rest of this chapter, we'll look at some of these different aspects of living documentation in more detail. I'll illustrate many of the principles using Thucydides reports and a few other similar tools, but the principles aren't specific to any particular toolset.

11.2 Are we there yet? Reporting on feature readiness and feature coverage

One of the core concepts in BDD is something we'll call *feature readiness*. Features, in this sense, are just pieces of functionality that the stakeholders care about. Some teams use user stories in this role, and others prefer to distinguish user stories from higher-level features. The principle is the same in both cases: when the development team reports on progress, stakeholders are less interested in which individual tests pass or fail and are more interested in what functionality is ready to be deployed to production.

11.2.1 Feature readiness: what features are ready to deliver

In BDD terms, a feature can be considered *ready* (or *done*) when all of its acceptance criteria pass (see figure 11.3). If you can automate all of these acceptance criteria, then the automated test reports can give you a simple, concise view of the state of the features you're building. At this level, you're more interested in the overall result of all of the scenarios associated with a feature than with whether individual scenarios pass or fail.

Most BDD tools provide at least some level of reporting on feature readiness, where scenario results are aggregated at the feature level. For example, Thucydides provides feature-level reports, either directly with JBehave or with test results imported from other tools such as Cucumber, SpecFlow, and Behave. SpecFlow also provides comprehensive built-in reporting. Cucumber provides only basic feature reporting out of the box, but tools like Cucumber Reports (www.masterthought.net/section/cucumber-reporting) provide more presentable reports.

In figure 11.4, for example, you can see a feature report for a Cucumber project, generated using the Cucumber Reports library. Here the status of each feature is reported based on the overall result of the corresponding scenarios.

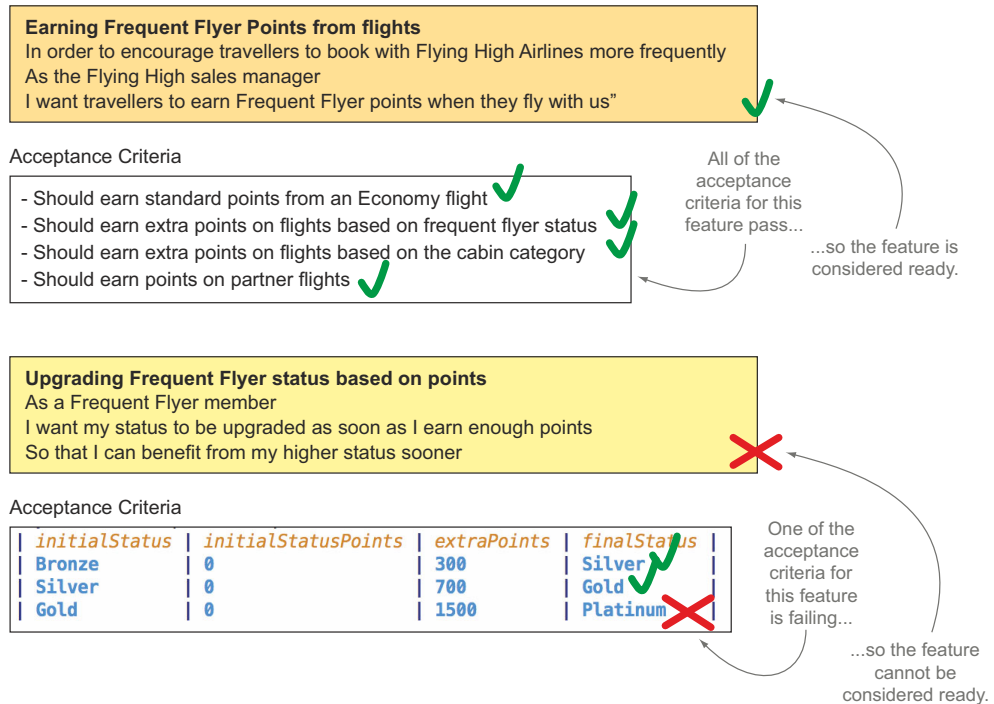


Figure 11.3 A feature can only be considered ready when all of its acceptance criteria pass.

Succinct reports like the one in figure 11.4 are a good way to get an overview of the current state of the features under development, without drowning in the details of each individual scenario.

11.2.2 Feature coverage: what requirements have been built

Feature-readiness reporting can go beyond simply aggregating conventional automated test results. Ideally, feature readiness should also take into account requirements that haven't yet been implemented and for which no automated tests exist. We'll call this more comprehensive form of feature readiness *feature coverage*.

Feature coverage tells you how many acceptance criteria have been defined and automated for each requirement. It also tells you what requirements have no automated acceptance criteria. Figure 11.5 illustrates this idea. It shows reports from a Thucydides project configured to organize requirements in terms of epics and stories. The Spending Points epic hasn't been started yet, so it has no automated scenarios and no test results. The Authentication epic has one implemented story, but two others exist only in the backlog. All of these still need to appear in the report so that the team can get a clearer overall view of the project status.

For example, in figure 11.5, all of the automated acceptance criteria associated with the Authentication epic pass. However, the coverage metric is only around 33%

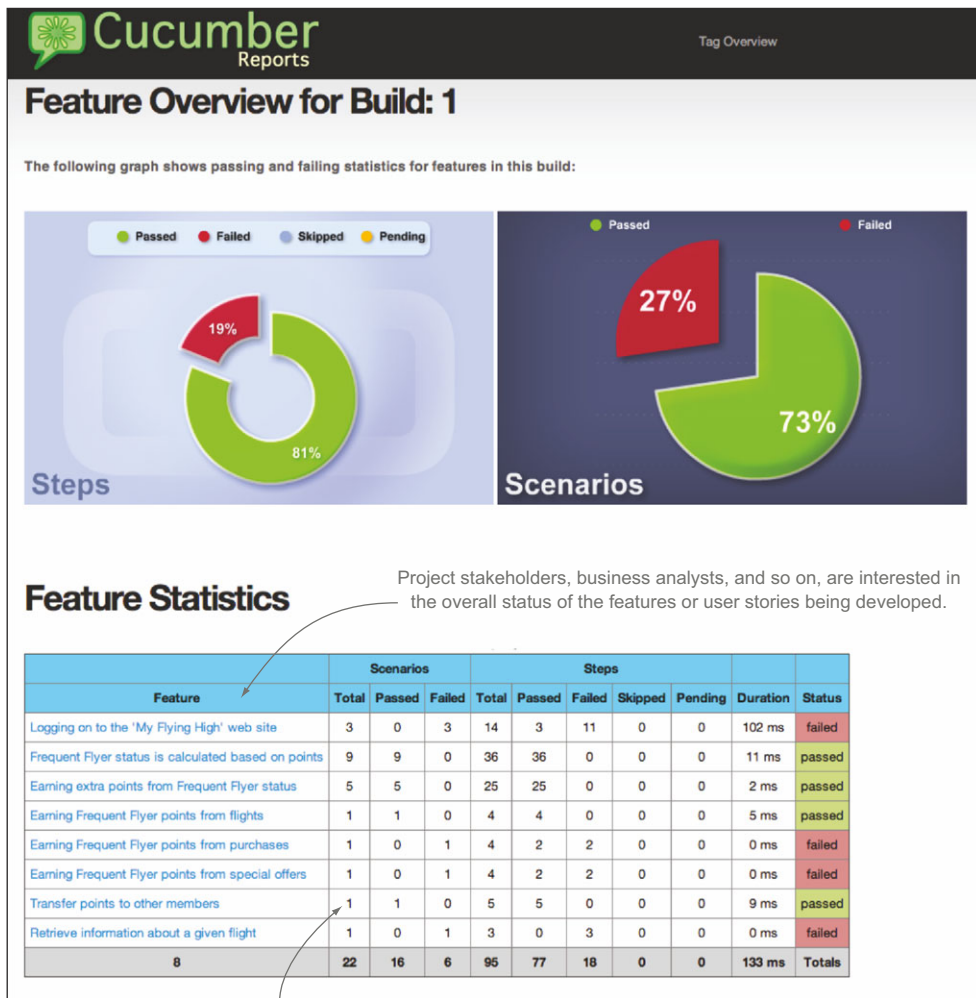


Figure 11.4 Feature readiness reports on the status of features or stories as a whole, rather than on individual tests.

because only one of three user stories associated with this epic has been implemented. Although all of the tests are green, this epic is only around a third complete.

Feature coverage isn't the same as code coverage. Traditional code coverage reports how many lines of code were exercised during the unit and integration tests. It can be a useful metric to tell developers what parts of the code base haven't been well tested. It's more limited when it comes to telling how well an application is tested at the unit-testing level, as it doesn't, in itself, vouch for the quality of the tests.



Figure 11.5 Feature-readiness reports can also report on higher-level concepts such as capabilities and epics.

The idea behind feature coverage is to give a fairer overall picture of project progress than you'd get by just reporting on test results. Feature coverage reports from the point of view of the requirements that have been defined, rather than the tests that have been executed. Of course, there's a caveat: a feature-coverage report will only be as thorough as the number of overall requirements it knows about. As you've seen, BDD practitioners, and Agile projects in general, avoid defining more detailed requirements up front than absolutely necessary. Stories and scenarios will typically be available for the current iteration, but not for much more; beyond that, the product backlog will typically contain higher-level stories and epics. This is reflected in the feature-coverage reports. For example, in figure 11.5, Spending Points is an epic with no stories or scenarios associated with it.

To produce this sort of high-level report, the BDD reporting tool needs knowledge about the application requirements beyond what can be obtained from the test reports.

Test results can tell you what features were tested, but they can't tell you which features have no tests at all. One popular way to achieve this is to integrate the BDD reporting process with a digital product backlog.

11.3 Integrating a digital product backlog

In Agile projects, *task boards* are frequently used to keep track of project activity. A task board is a physical board or wall containing index cards that represent user stories, tasks, bug fixes, and other activities the team must undertake to complete the project. A simple example can be seen in figure 11.6.

The exact layout of a task board is very flexible and is generally different for each team and project, but the general principle is always the same. During an initial planning session, work is broken down into the user stories, tasks, and so on that need to be delivered during the current iteration. Each task or user story goes in a column on the board, based on its status (not started, in progress, done, and so forth). Each day, team members get together in front of the task board to discuss the status of the task they're working on, possibly moving the cards from one column to another. The main advantage of this format is to let the whole team see at a glance what everyone is working on, making it easier to coordinate work and troubleshoot blockages.

Physical boards are excellent communication facilitators, and they provide great visibility for the current work in progress. But they do have their limitations. For example, they aren't optimal for teams that aren't colocated, they can be time-consuming to

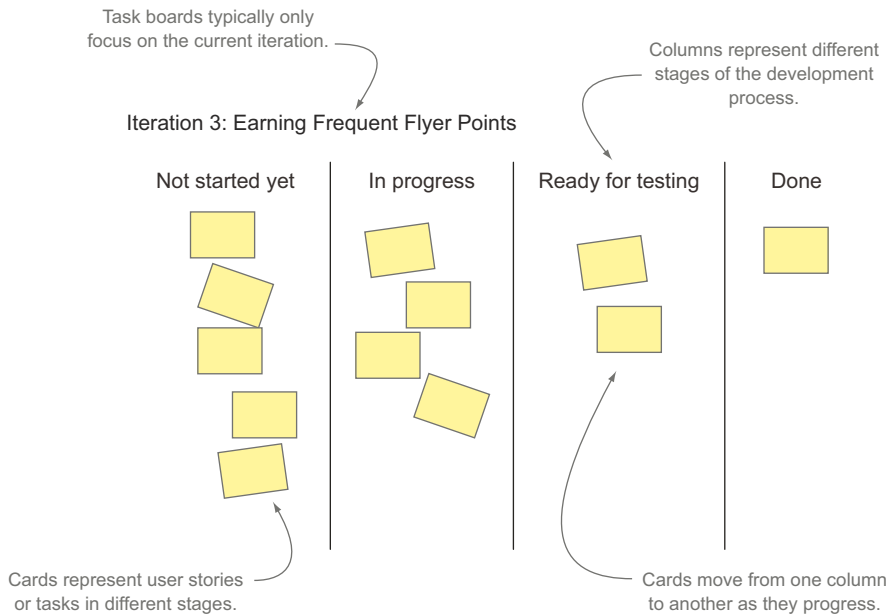


Figure 11.6 A task board is a visual representation of the activities currently going on in a project.

maintain, and there are many useful metrics that can be more easily tracked and visualized if the tasks are recorded electronically.

Because of these constraints, some teams prefer to keep track of work in some sort of issue-tracking or agile project-management system, even if they still use a physical board for day-to-day organization and visibility. Some practitioners refer to this as a “digital product backlog.”¹ A digital product backlog can save you time and effort by calculating and reporting burn-down charts and other metrics automatically. It’s also easier to attach extra information to a card without cluttering up the board.

In this situation, a team member (for example, the scrum master in a scrum team) will typically update the digital product backlog items based on the outcomes of the daily meeting around the physical board.

Storing user stories in an electronic system also has a number of advantages when it comes to integrating with BDD tools. In the simplest form, this integration may just include links to the corresponding card in the agile project-management software.

But a well-integrated system can do more. The BDD reporting tool can retrieve information from the digital product backlog and integrate this information into the reports. For example, the narrative description of a story need only be recorded in the digital product backlog and doesn’t need to be duplicated in the feature file. A good reporting tool will be able to, at a minimum, add links in the living documentation to the corresponding items in the digital product backlog.

The simplest way to integrate scenarios with a digital product backlog is to use tags. Features and user stories are captured in the product backlog tool, where they have a unique number. The acceptance criteria are stored in the form of BDD feature files. Within these feature files, you use a specific tag to indicate the number of the corresponding item in the product backlog. For example, Thucydides uses the @issue tag for this purpose; if this tag is present, Thucydides will fetch the narrative text from the electronic product backlog and add a link to the corresponding item in the generated reports. This approach is illustrated in figure 11.7.

Reporting tools like Thucydides can also look up information about how the user stories are organized (in terms of epics, features, capabilities, and so forth). Some teams even update the status of the cards in the issue-tracking system based on the outcomes of the automated acceptance criteria, so if an acceptance criterion fails, the status of the corresponding card will be updated and the team notified.

11.4 Organizing the living documentation

If it’s to be useful to the team as a whole, living documentation needs to be presented in a way that’s easy to understand and navigate. For large projects, a flat list of features can quickly become unwieldy. Fortunately, there are many ways to structure living documentation so that it’s easier to navigate and, as a result, is more valuable. Here, we’ll look at two of the most common approaches:

¹ This is the term we’ll use going forward, because it’s a lot easier to say than “agile project-management software.”

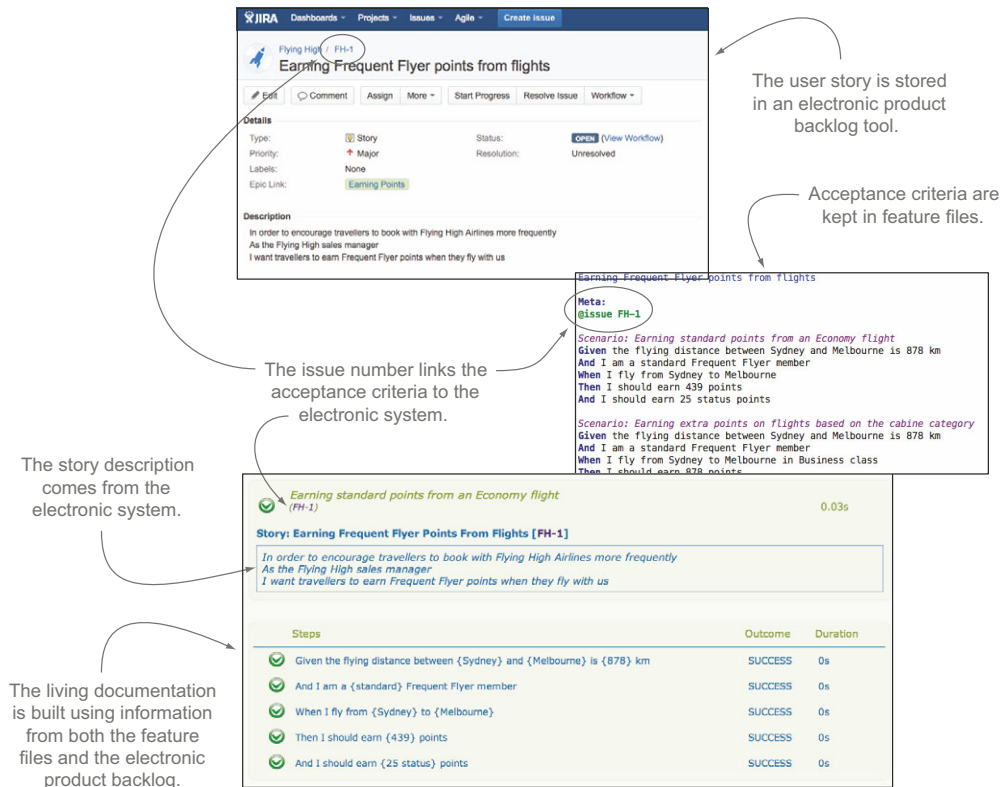


Figure 11.7 Reports are produced using information from both the executable specifications and from the electronic product backlog.

- Organizing living documentation to reflect the requirements hierarchy of the project
- Organizing living documentation according to cross-functional concerns by using tags

Note that these choices are not exclusive. A good set of living documentation should be flexible, and it should be possible to easily display documentation in different ways, based on the needs of the project and of the reader. Let's start with a look at how you can organize your living documentation in terms of the project requirements structure.

11.4.1 Organizing living documentation by high-level requirements

Grouping features by high-level requirements such as epics or capabilities is a good alternative to a flat list of features. You've seen an example of this approach in the Thucydides reports, where a high-level report summarizes the feature status by epic (see figure 11.8). This works well when the living documentation reporting is integrated with an electronic product backlog, because the structure of the living documentation will automatically reflect the structure used in the backlog tool.

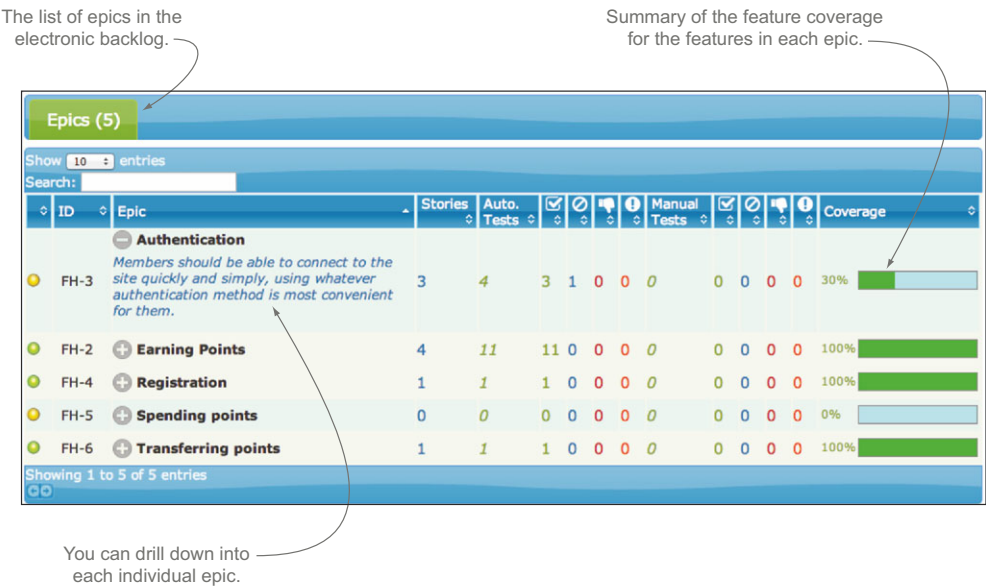


Figure 11.8 One useful way to organize living documentation is in terms of high-level requirements, such as epics or capabilities.

Sometimes the structure that comes from this sort of hierarchical organization is a little too rigid. As you’ll see in the next section, tags can offer a more flexible option.

11.4.2 Organizing living documentation using tags

There are times when it’s useful to group features in a more free-form manner. For example, you might want to examine all of the features related to particular nonfunctional requirements, such as security or performance, or all of the features that need to integrate with a particular external system.

Tags are an easy way to identify nonfunctional requirements like this. For example, the following Thucydides/JBehave scenario has been tagged with the “security” nonfunctional requirement:

```
Registering for a Frequent Flyer account online
Meta:
@issue FH-13
>tag non-functional:security

Scenario: Registering online for a new Frequent Flyer account
Given Jane is not a Frequent Flyer member
When Jane registers for a new account
Then Jane should be sent a confirmation email with her account number
And Jane should receive 500 bonus points
```

Tag these scenarios with the “security” nonfunctional requirement.

These features won’t necessarily be grouped in a single capability or epic; in fact, they’re typically scattered across the whole application. But it’s still often useful to report on all of the features that are impacted by this nonfunctional requirement.

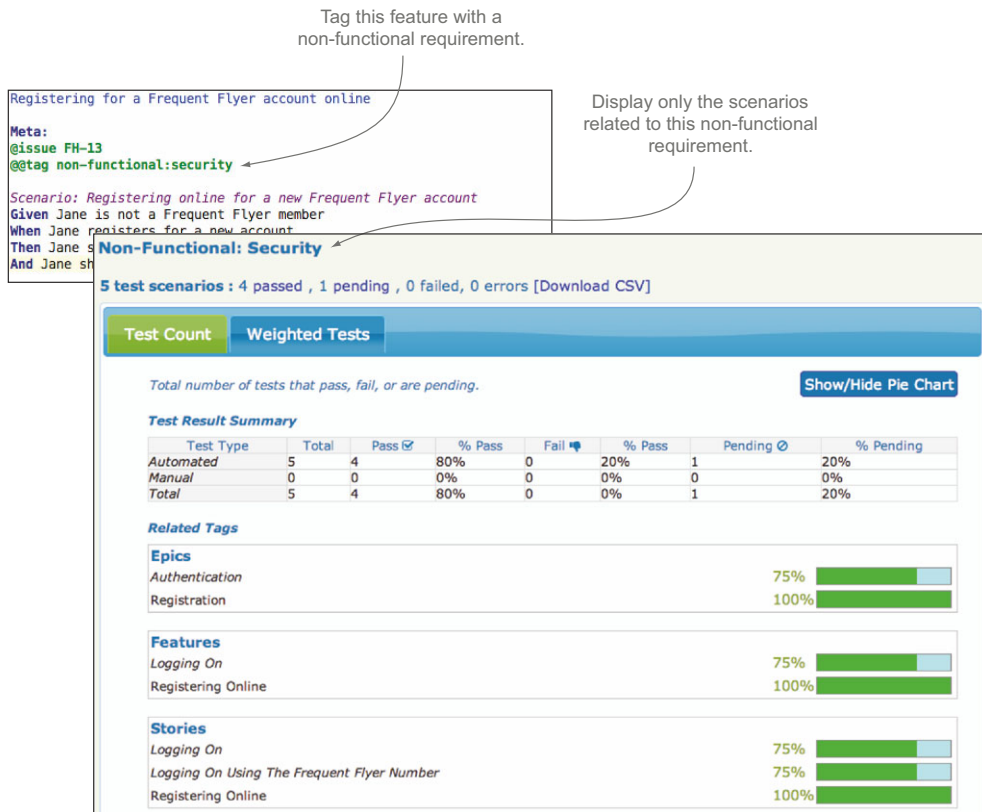


Figure 11.9 Reports are produced using information from both the executable specifications and data from the electronic product backlog.

One simple way to generate a tag-specific report is to run the scenarios using the tag as a filter. The only problem with this approach is that you have to run the tests again each time you need a report for a different tag.

Some more advanced BDD reporting tools let you view the features related to a given tag directly in the normal living documentation. In figure 11.9, for example, you can see a view in a Thucydides report summarizing the outcomes of only the scenarios related to the “security” nonfunctional requirement.

11.4.3 *Living documentation for release reporting*

Living documentation is a great way to get a view of the overall project status from the point of view of the requirements. But for larger projects, the quantity of information can be a little overwhelming. It’s useful to be able to focus on the work being done in the current iteration and mask out work that’s already been completed, or work that’s scheduled for future releases.

There are many ways you can do this. One of the simplest is to use tags to associate features or individual scenarios with particular iterations. For example, you

could assign a feature to iteration 1 in Cucumber or SpecFlow by giving it an appropriate tag:

```
@iteration-1
Feature: Logging on to the 'My Flying High' website
  Frequent Flyer members can register on the 'My Flying High' website
  using their Frequent Flyer number and a password that they provide
  ...
```

You could then run a separate batch of acceptance tests for the features containing the @iteration-1 tag, focusing only on the features scheduled for this iteration. This will produce a report containing only the scenarios and features that have been planned for this release, giving a clearer understanding of how ready the application is for release.

Many teams manage iterations and releases in an external software tool. For example, any electronic backlog software will let you assign stories, epics, and so forth to a particular release, and produce release notes based on this information. In this situation, a good strategy is to integrate the living documentation with the electronic backlog software. For example, figure 11.10 shows an example of a release report generated using details from the project release data stored in JIRA.

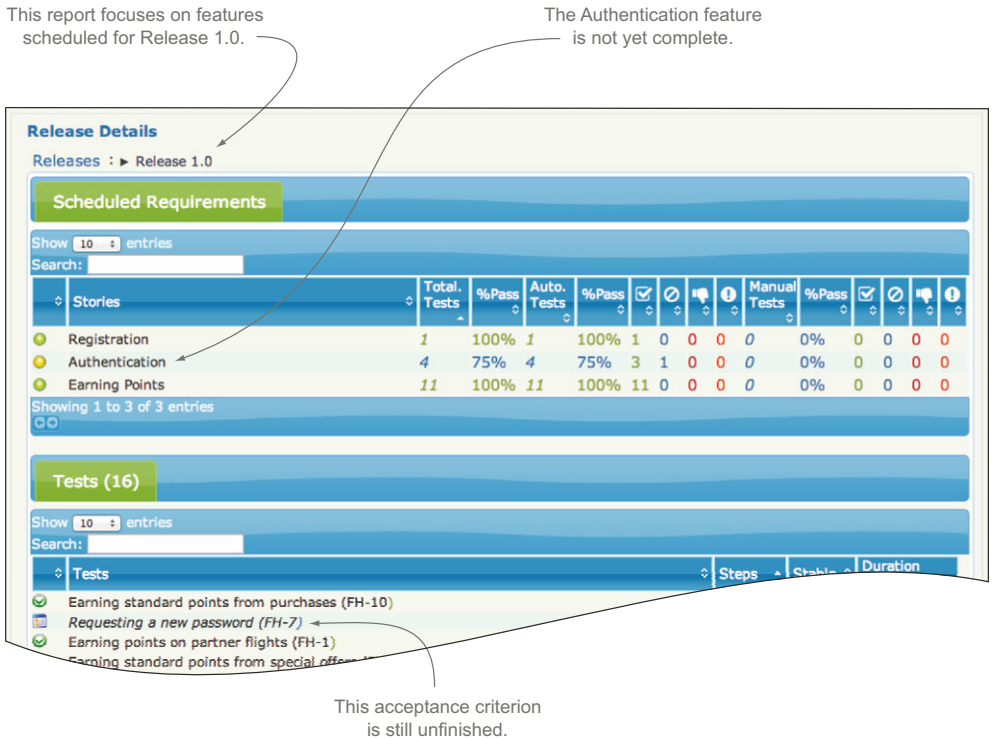


Figure 11.10 A release report focuses on requirements that have been scheduled for a particular release or iteration.

The advantage of this sort of integration is that releases need only be managed in one place. It also makes it easier for release managers to prepare release notes based on the current state of the automated acceptance criteria.

You don't have to stick with a single approach for your living documentation. In practice, teams often use a mix of several strategies for different purposes or different audiences. For example, a requirements-based organization is more effective for documenting what the application does as a whole and how it does it, whereas release-focused reporting is more relevant when reporting on progress and preparing release notes.

11.5 *Providing more free-form documentation*

Living documentation should aim to be just that: documentation. It should be organized in a way that makes it easy to use and simple to find the information you're looking for. It should also contain enough information to make it useful, albeit lightweight, documentation.

A user story is typically described as a “placeholder for a conversation,” with a deliberate minimum of information. But when the acceptance criteria are fleshed out, teams often find it useful to have a little more descriptive text around a feature and story than just the “in order to ... as a ... I want” text. This can include more background about the requirement, references to more detailed business rules, legislative requirements, and so forth.

There are several ways to provide this sort of additional information. In Cucumber and the other Gherkin-based tools, for example, the description section of a feature file is effectively free text, so teams often put quite detailed descriptive texts directly in the feature file.

Teams integrating with an electronic backlog system often store more detailed documentation in the backlog itself. This may even include links to external sources such as an enterprise wiki for additional details or documents. When using a reporting tool like Thucydides, the description text in the electronic backlog will appear in the living documentation, including any hyperlinks, so no information is lost in the reporting process.

Some automated acceptance-testing tools such as FitNesse and Concordion (<http://concordion.org>) give you even more flexibility in how you organize the living documentation. FitNesse (<http://fitnesse.org/>) uses wiki pages to let the team, including business analysts and even users, write their own acceptance criteria in a tabular format (see figure 11.11).

Developers then implement the corresponding acceptance tests using “fixture” classes, typically in Java. These classes interact with the wiki tables and call the actual production code to perform the tests. For example, a fixture class for the scenario in figure 11.11 might look like this:

```
public class StatusBoundaries extends ColumnFixture {
    public String initialStatus;
    public int initialStatusPoints;
    public int extraPoints;
```

```

public String finalStatus() {
    finalStatus = ...
    return finalStatus;
}

```

← Invoke the production code to determine the final status value.

Concordion uses a similar approach, except that the requirements are expressed in the form of simple HTML pages rather than on a wiki.

Both these approaches give you a great deal of flexibility in how you build your living documentation. But this flexibility does come at a cost—you need to organize and maintain the documentation pages, and the links between the pages, by hand, and it's much harder to do the sort of high-level aggregation reporting you've seen in the previous sections.

11.6 Technical living documentation

Living documentation doesn't stop with the high-level requirements. An important part of the low-level BDD practices you saw in chapter 10 involves writing unit and integration tests in the form of low-level executable specifications, which form a major part of the technical documentation in a BDD project.

You've seen how high-level acceptance tests describe and specify the behavior of the application as a whole. You've also seen how BDD unit tests describe and specify

Wiki-based

EarningStatusPoints
UpgradingStatusFromEarnedPoints

Tests Executed OK Test Edit Add Tools

✓ Assertions: 9 right, 0 wrong, 0 ignored, 0 exceptions (0.139 seconds)

Contents:

► Page Configuration Expand All Collapse All

Frequent Flyer members earn status points with each flight. When they earn enough points, they are immediately upgraded to the next status level.

initialStatus	initialStatusPoints	extraPoints	finalStatus?
Bronze	0	300	Silver
Silver	0	700	Gold
Gold	0	1500	Platinum

They earn points progressively, so most flights will not earn them enough points to upgrade their status.

initialStatus	initialStatusPoints	extraPoints	finalStatus?
Bronze	200	99	Bronze
Bronze	200	100	Silver
Silver	600	99	Silver
Silver	600	100	Gold
Gold	1400	99	Gold
Gold	1400	100	Platinum

Front Page | User Guide
 root (for global path's, etc.)

Decision tables used to illustrate the requirements

Free-form requirements text

Figure 11.11 In FitNesse, you document requirements in a wiki.

the behavior of an individual class, component, or API. In both cases the approach is similar; only the audience and the toolset change.

For technical living documentation, sophisticated reporting capabilities are less important than code readability and clarity. The primary audience of technical documentation is the developer who will need to maintain the code later on. Developers are used to reading code, so technical documentation in the form of well-organized, readable, annotated code samples is generally quite sufficient to help them understand the finer points of the code base. Many teams complement the low-level living documentation with light, higher-level architectural documentation, stored, for example, on a project wiki.

11.6.1 Unit tests as living documentation

Some tools make it a little more natural to express unit tests as living documentation. Spock, for example, is a great way to write low-level BDD tests for Java or Groovy projects. As you saw in chapter 10 (section 10.4.3), a simple Spock specification might look like this:

A short text explains what each step is doing.

```
def "Find flight details by flight number"() {
    > given: "I need to know the details of flight number FH-101"
        def flightService = new FrequentFlyerFlightService();
        def flightNumber = "FH-101"
        def airport = "MEL"
    > when: "I request the details about this flight"
        def flightDetails = flightService.findFlightByNumber(airport,
                                                                flightNumber);
    > then: "I should receive the correct flight details"
        flightDetails.flightNumber == "FH-101" &&
        flightDetails.departure.name == "Melbourne" &&
        flightDetails.destination.name == "Sydney" &&
        flightDetails.time == "06:00"
}
```

But writing unit tests that make good technical documentation relies more on an attitude than on using a particular tool. The following NSpec specification, for example, also does a great job of explaining what feature it's describing and illustrating how an API should be used:

```
public class WhenUpdatingStatusPoints : nspec
{
    FrequentFlyer member;

    void before_each()
    {
        member = new FrequentFlyer();
    }

    void earning_status_points()
    {
        context["When cumulating Frequent Flyer points"] = () =>
```

```

    {
        it["should earn points for each flight"] = () =>
        {
            member.earnStatusPoints(100);
            member.earnStatusPoints(50);
            member.getStatusPoints().should_be(150);
        };

        it["should upgrade status when enough points are earned"] = () =>
        {
            member.earnStatusPoints(300);
            member.getStatus().should_be(Status.Silver);
        };
    };
};

```

You can use more conventional unit-testing tools like JUnit and NUnit to achieve a similar level of readability by using readable class and method names. The following NUnit test, for example, uses nested classes to organize low-level specifications in a more meaningful way.

```

[TestFixture]
public class WhenUpdatingStatusPoints
{
    public class WhenTheFrequentFlyerAccountIsCreated
    {
        [Test]
        public void it_should_have_zero_points() {...}

        [Test]
        public void it_should_have_Bronze_status() {...}
    }

    public class WhenCumulatingStatusPoints
    {
        [Test]
        public void should_earn_points_for_each_flight() {...}

        [Test]
        public void upgrade_status_level_with_enough_points() {...}
    }
}

```

As illustrated in the Microsoft Visual Studio screenshot in figure 11.12, this makes the test classes read less like a list of test methods and more like a hierarchically organized specification document.

When a code base grows large, it can be hard to find the unit test that illustrates a particular feature. Many teams prefer to organize their unit tests in a package structure that mirrors the application structure. This approach has the advantage of making it easier to find the test class for a particular test, although this advantage is much less compelling with today's development environments, where it's very easy to find all the places in the code base where a particular class has been used.

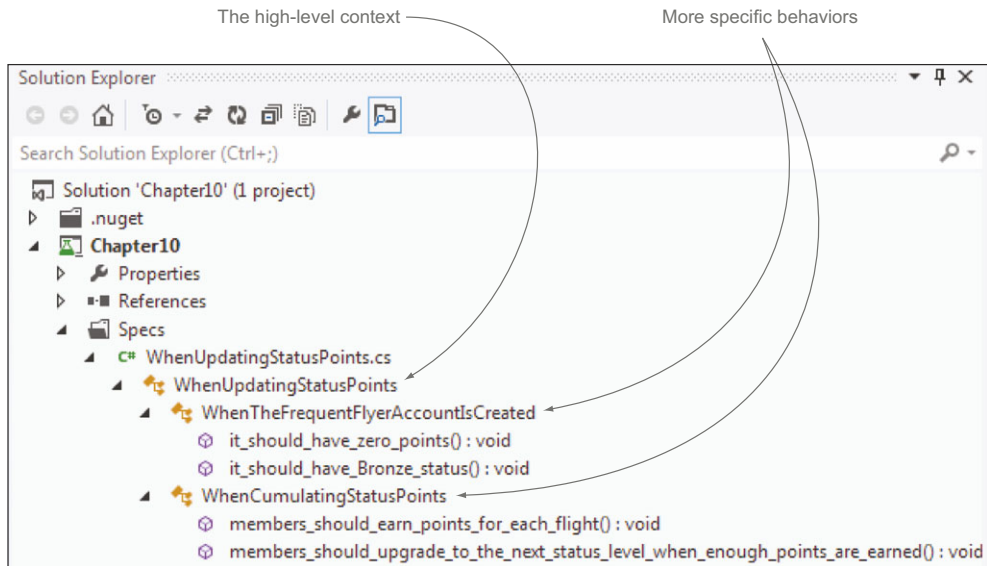


Figure 11.12 Structuring your unit tests well can make them read like living documentation within your IDE.

This approach dates from the early days of unit testing. One of the original motivations was that Java classes in a given package could access protected fields of other classes in that package, so if the unit test class was in the same package as the class being tested, you could access protected fields as part of the unit tests.

From a BDD perspective, the argument for this approach is less compelling. A BDD-style unit test should provide a worked example of how to use a class and how the class is expected to behave. If you need to access protected variables, you may be binding your test code too tightly to the implementation, which runs the risk of making the unit tests more brittle.

This approach also assumes a very tight coupling between the classes under test and the features (or behavior) those classes are implementing. If you refactor a class, change its name, or break it apart into several smaller classes, for example, the requirement that these classes implement shouldn't change.

Having a test class for each production class can also hamper refactoring. A modern IDE will tell you at a glance if a method or a class isn't being used anywhere. Unless it's part of an API for an external client, a method that's never used anywhere can generally be deleted, which results in less code to maintain going forward.

For all these reasons, many organizations apply a looser association between test classes and production classes. For example, some teams find that test packages or directories organized in terms of functional slices are often easier to navigate, especially when you come back to a code base after a long period.

11.6.2 Living Documentation for legacy applications

Many organizations have large legacy applications that are still very much in use and still need regular maintenance, updates, and releases. Many of these applications have few unit or integration tests and low test coverage, and they're often lacking technical or functional documentation.

The lack of automated tests for this sort of application makes it harder and riskier to deliver new features or bug fixes quickly. Releases are delayed by long testing cycles and regression issues. But for many organizations, rewriting the entire application isn't a viable proposition.

BDD offers some possible approaches that can relieve these symptoms and help teams to deliver changes to their legacy applications more safely and efficiently. One popular strategy that many teams adopt is to retrofit the legacy application with high-level acceptance tests, typically web tests for a web application. These acceptance tests both describe and document the existing system and help reduce the risk of regressions when new features are introduced. The tests are written in the same style as the automated acceptance criteria you've seen elsewhere in this book, and often with the same BDD-focused tools (such as Cucumber, SpecFlow, and JBehave). The BDD reporting capabilities of these tools are a great way to document and communicate how people believe the application should behave.

Retrofitting unit testing is traditionally very difficult to do effectively, because unit tests written too long after the code was written tend to be quite superficial. BDD unit tests specify the behavior of classes and components within these applications and are often the only place that this behavior is documented. When these specifications don't exist, it can be difficult to invent them after the fact, as it involves a deep understanding of how each class or component is expected to behave.

Despite these difficulties, BDD unit tests can provide excellent technical documentation, even for legacy applications. Some teams with mission-critical legacy applications that still require frequent and significant changes use BDD tests to document the most critical or high-risk parts of their application first, before expanding into less critical functionality. Unit tests are written in the spirit of documenting the current application behavior and giving code samples for how to use each class. This is a great way to provide technical documentation for the more critical parts of the application and to build up high-quality test coverage.

11.7 Summary

In this chapter you learned what living documentation means in a BDD project:

- Living documentation is a way of automatically reporting both what an application is intended to do and how it does it.
- Living documentation provides information about features and requirements in addition to individual test results.

- If you store your requirements in a digital product backlog tool, it's useful to integrate the BDD reporting from the automated acceptance criteria with the requirements from the product backlog tool.
- Living documentation can be organized in many different ways, depending on what information needs to be retrieved.
- Living documentation can also be used to document the lower-level technical components of your system.

In the next and final chapter, you'll see how BDD fits into the broader build process.

12

BDD in the build process

This chapter covers

- Executable specifications and the automated build process
- The role of BDD in continuous integration (CI) and continuous delivery
- Using a CI server to publish living documentation
- Using a CI server to speed up your automated acceptance tests

The ultimate aim of BDD is to deliver more valuable software with less waste. But the business value of a new feature isn't truly realized until it's deployed into production for users to use. So if you want to deliver business value to users faster, you need to be able to deploy features quickly and efficiently.

Deploying a feature into production is typically a fairly complicated process. You need to build the application from source code and run the automated unit, integration, and acceptance tests. You need to bundle it up into a deployable package. You may do performance tests, code quality checks, and so forth. And you'll typically deploy it into a test or UAT environment for testers and users to verify before deploying it into production.

USER ACCEPTANCE TESTING (UAT) UAT is a dedicated environment many organizations use to allow end users to test a new version of an application before it goes into production.

Automation is the key to an efficient deployment process. Any automated steps in this process will be faster and more reliable than the manual equivalents. Indeed, fast deployment relies on minimizing the number and length of manual steps in the build/release cycle.

Manual testing is usually the largest and slowest of these steps, so test automation can have a significant impact on the time required for manual testing. The automated acceptance criteria and living documentation produced by BDD also help build up confidence in the quality of the application, letting testers focus on exploratory testing and spend less time on repetitive scenario-based testing.

In this chapter, we'll take a closer look at the role BDD plays in the overall build and deployment lifecycle and how it can help streamline the automated delivery process.

12.1 Executable specifications should be part of an automated build

The executable specifications we've seen in the previous chapters, both for acceptance criteria and for lower-level technical specifications, aren't designed to be run by hand on an ad hoc basis. Although it's certainly convenient to be able to run individual tests from within an IDE, this isn't their main purpose. Rather, they're intended to be run automatically, as part of an automated build process. Individual acceptance criteria scenarios give a partial view of an application's behavior. Only when they're run together, as a comprehensive suite, can they give a full picture of the current state of the application (see figure 12.1).

For this reason, executable specifications need to work well in the context of an automated build. Whether they're implemented as low-level unit tests or end-to-end functional tests, they should respect a certain number of constraints. In particular,

- Each executable specification should be self-sufficient.
- Executable specifications should be stored in version control.
- You should be able to run the executable specifications from the command line, typically using a build script.

Let's look at these constraints in more detail.

12.1.1 Each specification should be self-sufficient

Some teams try to organize their executable specifications into *suites* that need to be executed in a precise order. For example, an initial acceptance criterion might describe how to look up an item in an online catalog, a second might then illustrate how to purchase this item, and a third might illustrate requesting a refund of this same item.

This is generally not a good idea. Executable specifications shouldn't depend on other specifications to prepare test data or to place the system in a particular state.

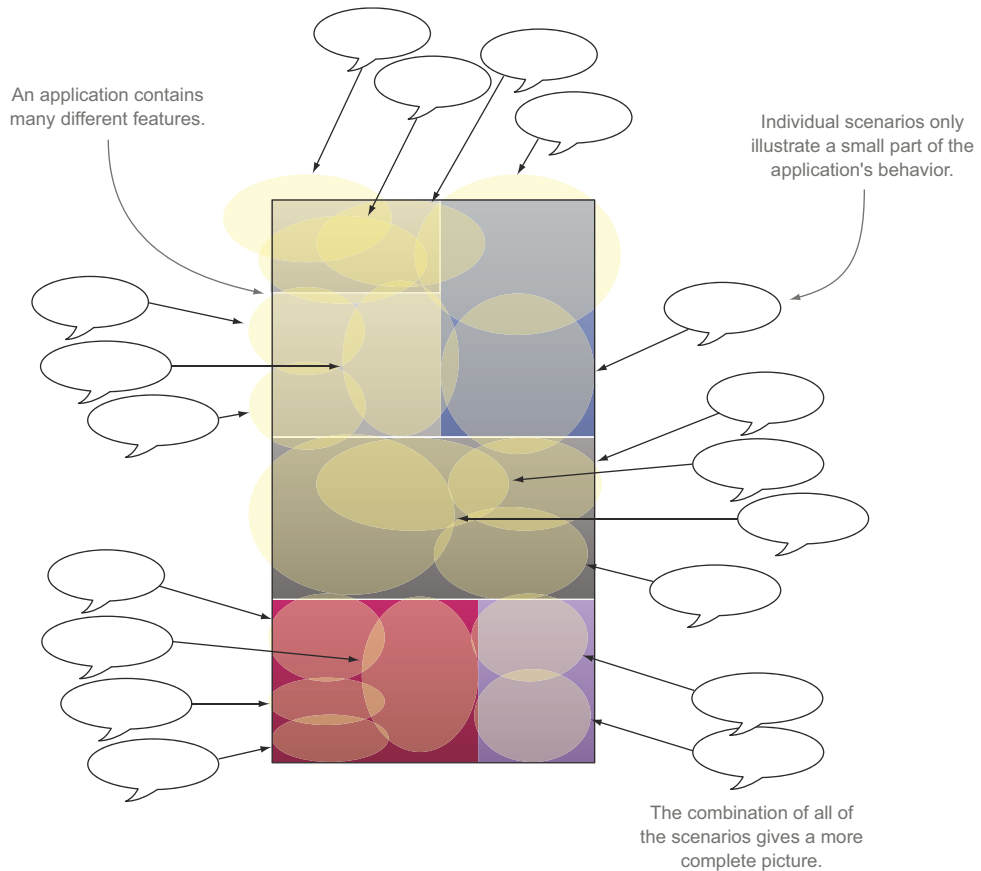


Figure 12.1 Only by considering all of the acceptance criteria together can you get a global picture of how the application is expected to behave.

Each specification should be able to run in isolation, and each specification should set up the test environment in the initial state it requires.

This is important for many reasons. If one of the acceptance criteria in a suite fails, the other acceptance criteria that depend on it will also fail, simply because the system isn't in the initial state they expect. This results in misleading reports that record working features as being broken and makes it harder to troubleshoot real issues.

In addition, when they're run as part of an automated build, executable specifications won't necessarily be run in a guaranteed order. Tests may be run in parallel batches, or simultaneously on different machines. Even on a single machine, many test libraries don't guarantee that tests will be executed in a predetermined order.

For these reasons, executable specifications shouldn't make any assumptions about which other specifications have (or haven't) been previously executed; if they need the system to be in a particular state, they should set it up themselves. Doing so results in more flexible and more robust test suites overall.

12.1.2 Executable specifications should be stored under version control

A given set of automated acceptance criteria is designed to run against a specific version of an application. If a new feature is added to the application, there must be a new version of the acceptance criteria that illustrates this feature. If developers are working simultaneously on different versions (or different branches) of the application code base, there should be matching different versions of the acceptance criteria (see figure 12.2).

In addition, the automated build process needs to be able to run the right set of executable specifications for a given version of the application, whether it's the latest build of the source code, a specific release candidate build, or a bug fix for a previous release.

Some tools let you store acceptance criteria scenarios away from the source code, in a separate database or managed in a separate application. The motivation behind these tools is to make it easier for non-developers to access (and take ownership of) the acceptance criteria scenarios. However, this approach poses a number of problems. For one thing, it makes the acceptance criteria much harder to associate with a

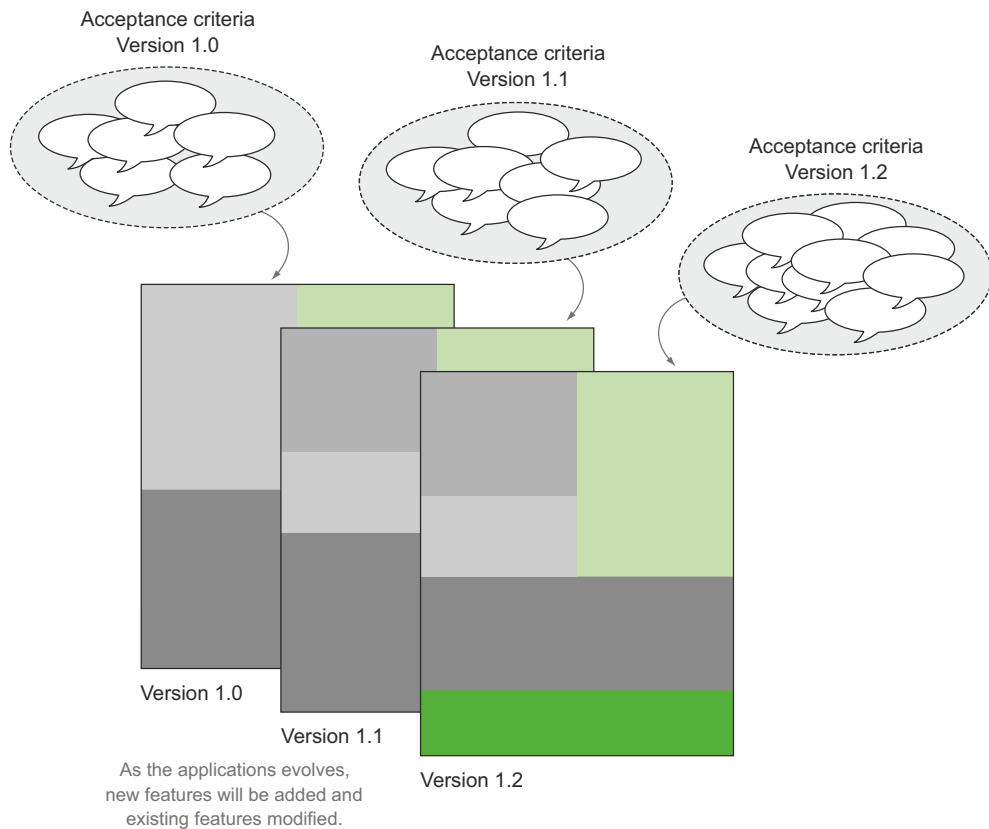


Figure 12.2 Acceptance criteria evolve alongside the application.

particular version of the application, which in turn makes the acceptance criteria harder to integrate into a robust automated build system.

In addition, this approach can also encourage business analysts or other less technical team members to create scenarios in isolation. When developers come to implement these scenarios, they inevitably need to “tweak” them to make them easier to automate, so the automated scenarios end up being slightly different from the ones the business analysts originally proposed. This whole workflow breaks down the collaboration aspect of BDD that’s so essential to its success.

For these reasons, it’s important to store your executable specifications in version control. In fact, automated acceptance criteria should be considered a form of source code and stored in the same source code repository as your application code.

12.1.3 You should be able to run the executable specifications from the command line

You also need to be able to execute your acceptance criteria from the command line, typically using a build script. Graphical interfaces are convenient, but build automation needs scripting. And build automation is at the heart of the continuous integration and delivery strategies we’ll discuss in the rest of this chapter.

There are many build-scripting tools, and your choice will typically depend on the nature of your project. In the Java world, you might use Maven, Gradle, or Ant. For a JavaScript-based project, you could use Grunt or Gulp. In .NET, it might be MSBuild or NAnt, and so on. In many cases, you could also launch the tests directly with the tool you’re using (for example, using Cucumber with Ruby), although build scripts often give you more control over the execution environment.

All of the tools we’ve discussed in this book can be executed on the command line, or from within a build script, but many organizations still use more heavyweight test automation tools that integrate poorly with command-line execution and build automation. Without this capability, it becomes very hard to integrate the automated acceptance criteria into the automated build process, which in turn makes continuous integration and continuous delivery strategies very hard to implement.

12.2 Continuous integration speeds up the feedback cycle

One of the most important principles underlying agile and lean software development practices is feedback. Fast feedback loops are essential to reducing wasted effort and delivering valuable features efficiently. Being informed of an issue is the first step in resolving it. Delay represents risk: the faster you know about a potential problem, the better position you’re in to adjust your actions accordingly. Even positive feedback is useful; when you can confirm that a particular strategy or problem solution is successful, you can build on this for subsequent work.

TDD or low-level BDD, for example, provides feedback within seconds or minutes, whereas automated acceptance tests, which typically take longer to run, can provide feedback within minutes or tens of minutes. In a project that relies entirely on manual

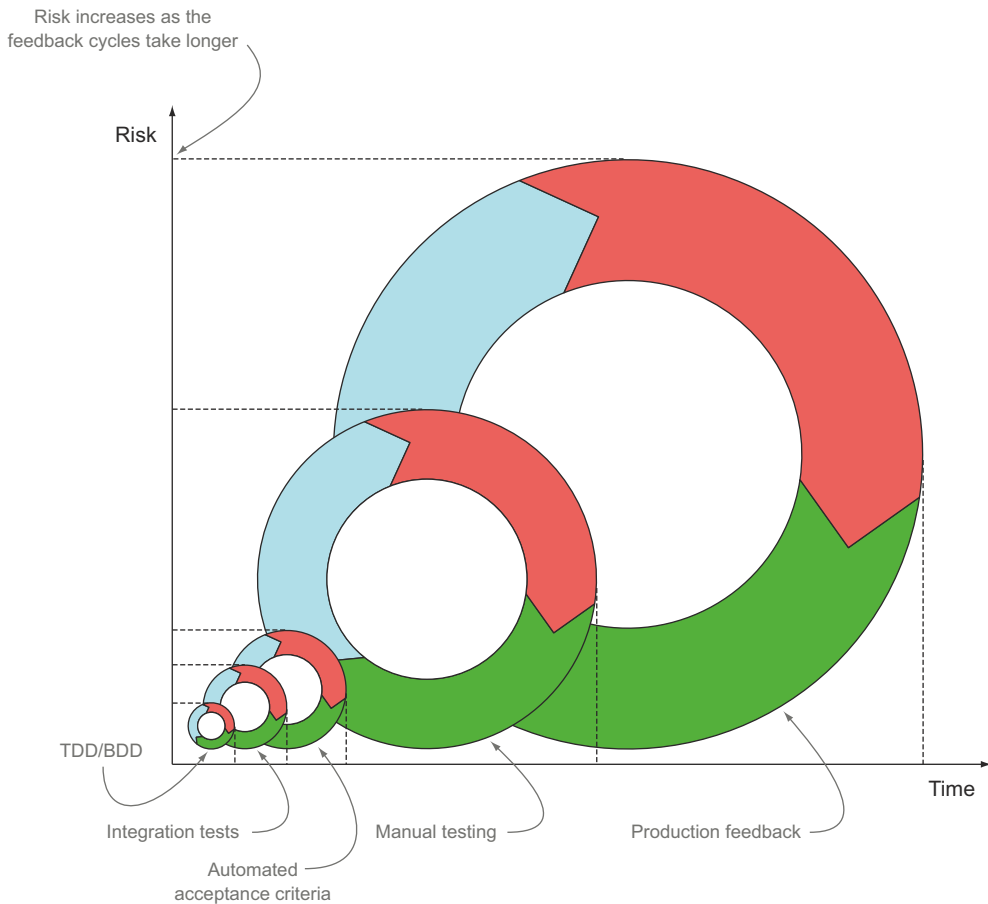


Figure 12.3 All forms of tests provide feedback, but the longer a test takes to provide feedback, the higher the risk.

testing, developers might need to wait for days, weeks, or even months to get this same feedback (see figure 12.3).

The principal goal of continuous integration (CI) is to provide fast feedback on the state of the build process. A CI server is an application that continually monitors a project's source code repository for changes. Whenever a change is committed to version control, the CI server kicks off a build to compile and test this version of the application. This ensures that all of the automated tests are run against each new version of the code base. If anything goes wrong, the team is immediately notified. In teams that practice CI well, the status of the build is taken very seriously, and if a build breaks, the developer responsible will immediately stop work and fix the problem.

Figure 12.4 illustrates the results of a set of automated acceptance criteria run under Cucumber, as displayed on Jenkins, a popular open source CI server (<http://jenkins-ci.org/>).

The CI server displays the results of the latest automated acceptance criteria.

It also records the history of previous test runs.

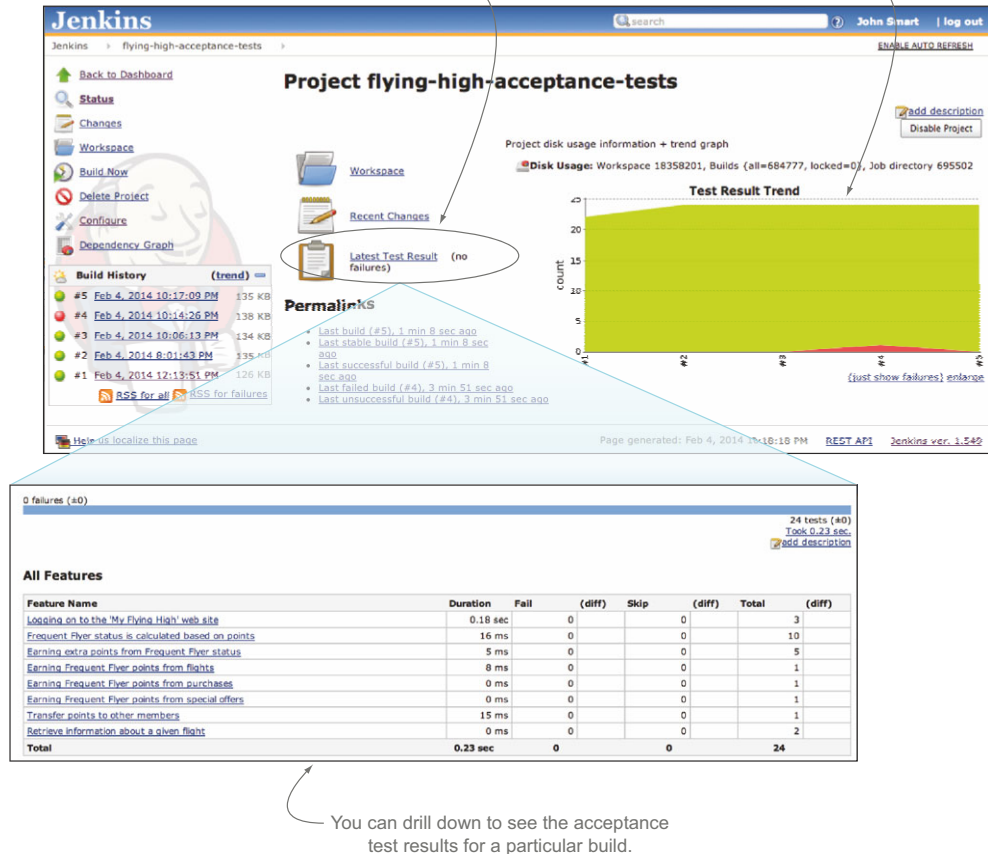


Figure 12.4 Automated acceptance criteria are typically run on a CI server, such as Jenkins.

CI relies heavily on good automated testing: without automated testing, a CI server is little better than an automated compilation checker. But with reliable and comprehensive automated tests, the team can use the CI build results to evaluate the current state of the application. If every feature has a meaningful set of automated unit and acceptance tests, then the automated test reports will give a meaningful picture of the current state of the application.

For teams practicing BDD, a CI server also acts as a platform for automatically building and publishing living documentation. The full set of automated acceptance criteria is executed for each change made to the application. Issues are raised quickly, but even when all goes well, the living documentation generated from the acceptance criteria is updated and published for each change to the application. This provides much faster feedback for business analysts, testers, and even business stakeholders about the current state of the application. They don't have to wait for the developers

to announce that a new feature has been delivered, or demonstrate how it works; they can look at the latest version of the living documentation and see for themselves.

12.3 Continuous delivery: any build is a potential release

Continuous delivery takes CI a step further. For a team that practices continuous delivery, any build is a potential release. To be deemed “release-ready,” a particular version of an application must successfully pass through a number of *quality gateways*—unit tests, integration tests, acceptance tests, performance tests, code quality metrics, and so on. In continuous delivery, an executable version of the application is built and packaged very early on in the build process, and this same packaged version is passed through each of the quality gateways. This approach streamlines the process, because the application doesn’t need to be rebuilt at each stage. If it successfully passes all of the gateways, it can be deployed to production if and when the business decides to do so.

An example of a build pipeline is illustrated in figure 12.5. Here, an initial build compiles the application and runs unit and integration tests. If these pass, a binary version of the application is prepared: this is the release candidate for this version.

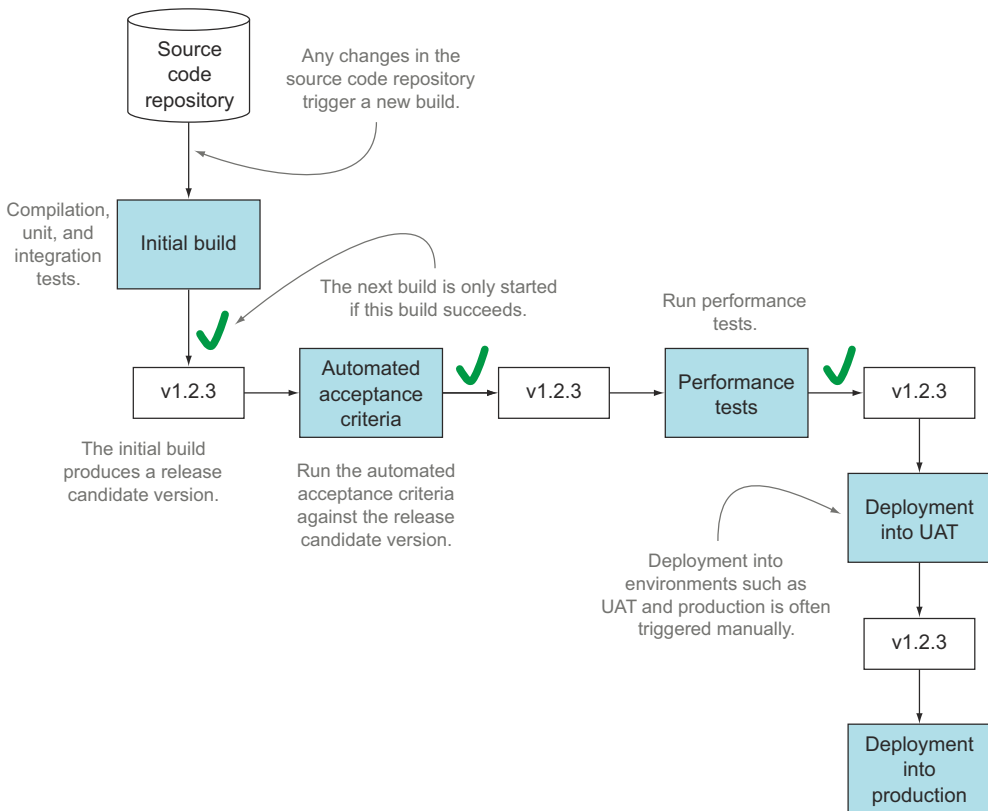


Figure 12.5 A simple build pipeline

The release candidate is passed to subsequent build jobs, each of which acts as a quality gateway. In the pipeline illustrated here, the quality gateways are the automated acceptance criteria and the performance tests. If the release candidate passes both of these gateways successfully, it can be deployed to the User Acceptance Testing (UAT) environment. Deployment to UAT and production is triggered manually—the process is still automated, but the team decides when the latest release candidate should be deployed.

In a more traditional release process, when the code is deemed ready, a special “release build” will produce a release candidate version of the application. This release candidate version will need to pass through a similar set of quality gateways before it can be released into production, but it will generally be rebuilt from source code at each stage.

In both cases, the build pipeline is typically made up of a series of quality gateways, which run different sorts of tests and checks. Common examples of quality gateways include

- A simple, fast build that compiles and runs the unit and integration tests (in that order). This is designed to provide feedback to developers. Using BDD naming styles makes it easier to troubleshoot problems when they do occur.
- A longer-running build job that runs the automated acceptance criteria. This build job also produces and publishes the living documentation.
- A build job that verifies code quality metrics such as code coverage or coding standards.
- Build jobs that verify performance or other nonfunctional requirements (discussed in section 9.4).

Many modern CI servers now support build pipelines to various degrees. Figure 12.6 illustrates the build pipeline illustrated in figure 12.5 running on a Jenkins server using the Build Pipeline plugin (<https://wiki.jenkins-ci.org/display/JENKINS/Build+Pipeline+Plugin>).

Teams practicing BDD are in a good position to implement continuous delivery strategies. Continuous delivery relies on an extremely high level of confidence in the automated test suite: if you can’t trust the automated tests to verify your application, you need to introduce a manual testing step, or possibly several manual testing steps, into the pipeline. Many organizations do indeed have manual testing steps, but the number and length of these steps has a significant impact on the time it takes a given release candidate to get into production. The more confidence the team has in the automated tests, the less time needs to be spent on these manual testing stages.

This is one of the reasons that BDD practitioners place such a strong emphasis on communication and readability when writing automated acceptance scenarios. Testers need to be able to understand what the automated acceptance tests are verifying, and to trust that they’re testing these aspects of the application effectively. Otherwise, they’ll effectively duplicate the verification work done by the automated tests, which

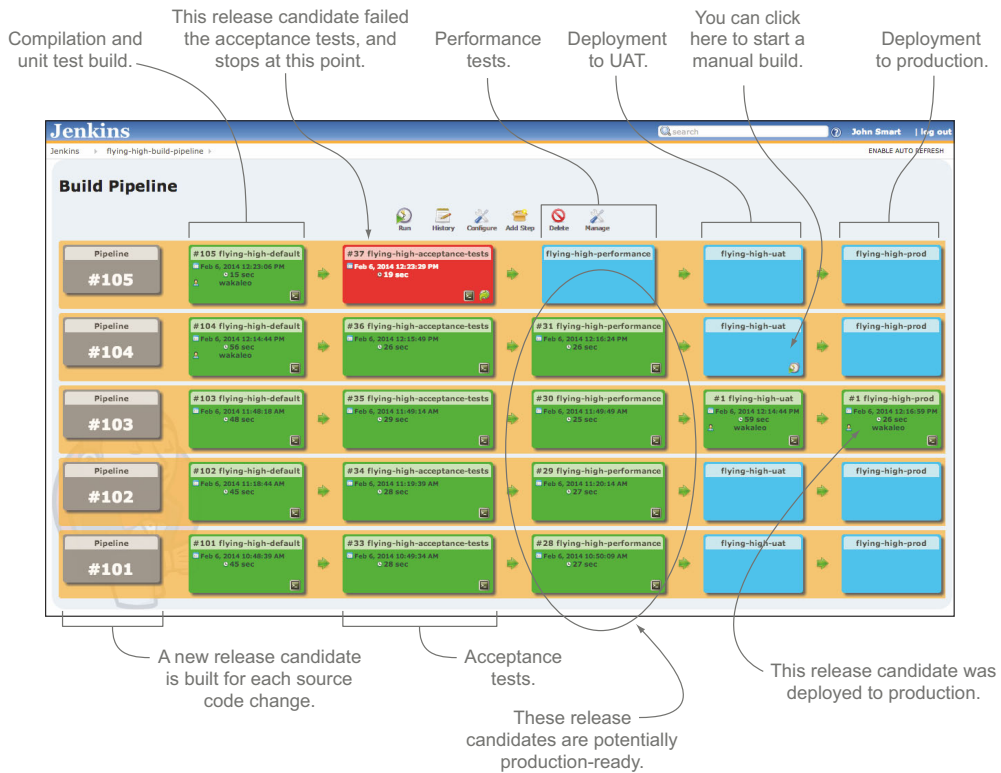


Figure 12.6 A build pipeline implemented using Jenkins

wastes time, slows down the deployment pipeline, and diverts testers from doing more value-added testing such as exploratory testing.

But this trust can't be blind—testers need to be able to see both what scenarios the automated acceptance criteria are testing and how they're testing these scenarios. The key to this trust is effective living documentation. In the next section, you'll see how CI can help make this living documentation easily accessible to testers and other team members.

12.4 *Continuous integration used to deploy living documentation*

You learned about the benefits of living documentation in chapter 11. But living documentation isn't useful if it's not up to date, or if you need to produce a new version manually whenever you need to consult it. Effective living documentation needs to be generated automatically for each new version as part of the CI build. And it needs to be easy to access, so that anyone who needs to see the latest version can do so with minimal effort.

Let's look at a couple of ways that BDD teams make living documentation available.

12.4.1 Publishing living documentation on the build server

This simplest way to publish up-to-date living documentation is to store it directly on the CI build server. Almost all of the BDD tools we've discussed in this book produce HTML reports, and most CI servers let you store HTML reports produced as part of a build. Some CI servers, such as Jenkins, also provide dedicated plugins for BDD tools, such as Cucumber and Thucydides, that let you publish the reports on the CI server website more easily (see figure 12.7).

This approach is easy to set up and is relatively low maintenance. But accessing the living documentation on a CI server isn't always as easy as it should be, particularly for team members or users not familiar with using a CI build server.

Some CI servers, such as Jenkins, do make it easy to use a fixed URL that will always access the latest successful build for a project, or, for example, the latest set of

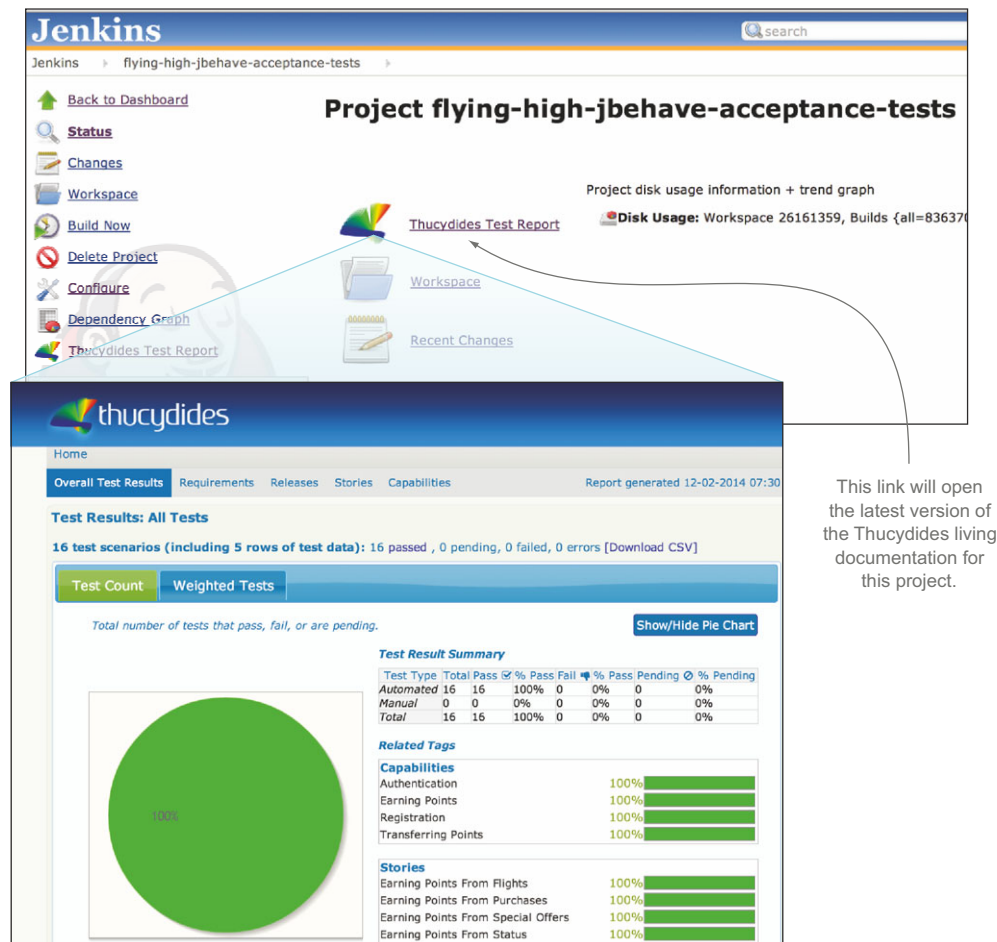


Figure 12.7 The Thucydides Jenkins plugin makes it easy to access the latest Thucydides reports.

Thucydides reports. But another way to make access simpler is to use a dedicated web server for the living documentation.

12.4.2 Publishing living documentation to a dedicated web server

Some teams choose to publish living documentation to a dedicated web server. Any web server will do, because the reports are just static HTML. This way, team members don't need to log on to the build server or find the right build job; they can access the documentation simply and directly.

This approach requires a little more initial configuration to deploy the reports, but most CI servers let you deploy files to a remote server easily enough. For example, Jenkins has several plugins that make it easy to deploy files across FTP, SFTP, and so on.

It can be useful to store previous versions of the living documentation for future reference. A common strategy is to publish the latest living documentation on a dedicated web server, and to store older versions on the build server.

12.5 Faster automated acceptance criteria

As you've seen, automated acceptance criteria are an excellent way to provide rich and meaningful feedback about the state of a project. And compared to manual tests, automated acceptance criteria are certainly fast. But in terms of build automation, automated acceptance tests (along with performance tests and load tests) are often among the slowest of the automated tests you're likely to run. The value of feedback is proportional to the speed with which you receive it, so faster feedback is always preferable.

When automated acceptance tests take too long to run, the whole development and release process suffers. A full release process that takes 15 to 30 minutes, for example, provides reasonably fast feedback for developers and makes it easier to streamline the release process and get new features or bug fixes deployed as quickly as they're implemented. But if your full test suite takes three hours to run, developers will often have to wait until the following day to get feedback about their changes, and same-day releases of changes and bug fixes becomes much harder.

In this section, we'll look at a few ways you can speed up your automated acceptance tests, and as a result speed up your delivery process.

12.5.1 Running parallel acceptance tests within your automated build

One way to speed up tests is to configure the acceptance criteria to run in parallel directly within the automated build. Modern machines with multicore processors have plenty of processing power that can be harnessed to accelerate your test suite. Exactly how you do this varies greatly depending on what BDD toolset you're using, and different tools have different levels of support for parallel processing.

If you're implementing your automated acceptance criteria using JBehave and Thucydides with Maven, for example, you can use the parallel execution capabilities that come with the Maven JUnit test runner Surefire (<http://maven.apache.org/surefire/maven-surefire-plugin/>) used to execute the JBehave scenarios. In figure 12.8, you can see a typical project structure using JBehave and Thucydides.

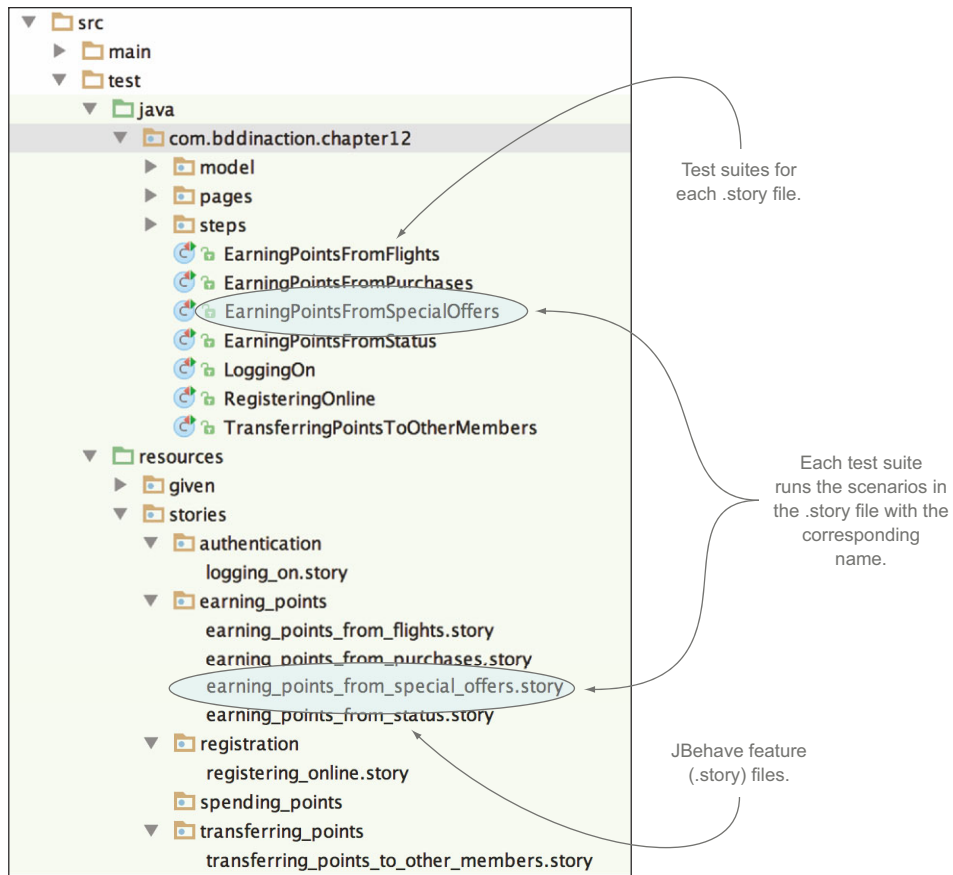


Figure 12.8 A Thucydes/JBehave project with a test suite for each feature (.story) file

In this example, each feature file has its own test suite class. The following is an example of the test suite class for the `earning_points_from_flights.story` file:

```
import net.thucyides.jbehave.ThucyidesJUnitStory;

public class EarningPointsFromFlights extends ThucyidesJUnitStory {}
```

This will run scenarios in the `earning_points_from_flights.story` file.

For larger projects, you might have a test suite class per higher-level feature or capability, with the stories being organized into folders accordingly. For example, the following test suite class would run all of the stories in the `earning_points` directory:

```
public class EarningPoints extends ThucyidesJUnitStories {
    public EarningPoints() {
        this.findStoriesCalled("**/earning_points/*.story");
    }
}
```

This will run all the .story files in the `earning_points` directory.

In both cases, you’re running the tests through JUnit.

One way you can configure the automated build to run these tests in separate parallel threads is to use the `<forkCount>` option, as shown here:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.16</version>
  <configuration>
    <forkCount>4</forkCount>
    <reuseForks>true</reuseForks>
    <includes>
      <include>com/bddinaction/chapter12/*.java</include>
    </includes>
  </configuration>
</plugin>
```

Run the tests in four parallel JVM processes.

Don't create a new process for each new test.

Run all of the acceptance test stories.

This runs each test class in a separate JVM process. You can also use JUnit’s built-in `<parallel>` option, but this runs all of the tests in parallel threads within a single JVM process, which is less robust and better suited to running unit tests in parallel.

The `<forkCount>` option will also work fine with the Maven Failsafe plugin, a popular alternative Maven plugin used for integration testing.¹

Similar approaches can work with any tool that uses a JUnit-based test runner, though with some caveats. If you’re using Cucumber-JVM, you can configure Cucumber-JVM test suites similar to the Thucydides test runners shown previously to run a subset of the Cucumber feature files. For example, the following class will run all of the feature files with the `@earning_points` tag:

```
@RunWith(Cucumber.class)
@Cucumber.Options(tags = {"@earning_points"},
                  format = {"json:target/cucumber/earning_points.json"})
public class EarningPoints {}
```

Run only scenarios with this tag.

Store the reports in JSON format in this file.

Similarly, the following class will only run features related to Authentication:

```
@RunWith(Cucumber.class)
@Cucumber.Options(tags = {"@authentication"},
                  format = {"json:target/cucumber/authentication.json"})
public class Authentication {}
```

Note how each of these test suites writes the test results to a different file. This ensures that the test results can be produced independently, but it means you’ll need to merge the results from all of these JSON files after the tests have finished. If you’re using a CI tool with Cucumber integration, such as Jenkins or Bamboo, this can be easily done directly from within the CI build job itself.

Not all BDD tools offer this level of support for parallel testing, and even when this is the case, parallel testing on a single machine won’t scale beyond a certain point. For

¹ See “Fork Options and Parallel Test Execution” on Apache’s “Maven Failsafe Plugin” page: <http://maven.apache.org/surefire/maven-failsafe-plugin/examples/fork-options-and-parallel-execution.html>.

example, if you're running web tests, running too many browser sessions simultaneously on a single machine can slow down the machine and cause tests to fail due to browser crashes or memory limitations. For large projects, a more scalable approach is to run parallel builds on more than one machine.

12.5.2 Running parallel tests on multiple machines

Today, hardware is cheap and virtual machines are easy to set up. Many organizations configure their testing infrastructure so that they can devote a battery of physical and virtual machines to testing when they need to run an intensive set of tests, and then release them once the tests are finished. Continuous integration servers like Jenkins, Bamboo, and TeamCity make it easy to dispatch build jobs across several machines to better distribute the load.

You can use this distributed capability to speed up your automated acceptance criteria. This approach is illustrated in figure 12.9. The acceptance criteria **1** need to be divided into groups that can be distributed across different build jobs, possibly on different machines **2**.

Different teams use different strategies to group acceptance criteria in this way. Tags and directory structures are both popular options. If you're using Cucumber, you

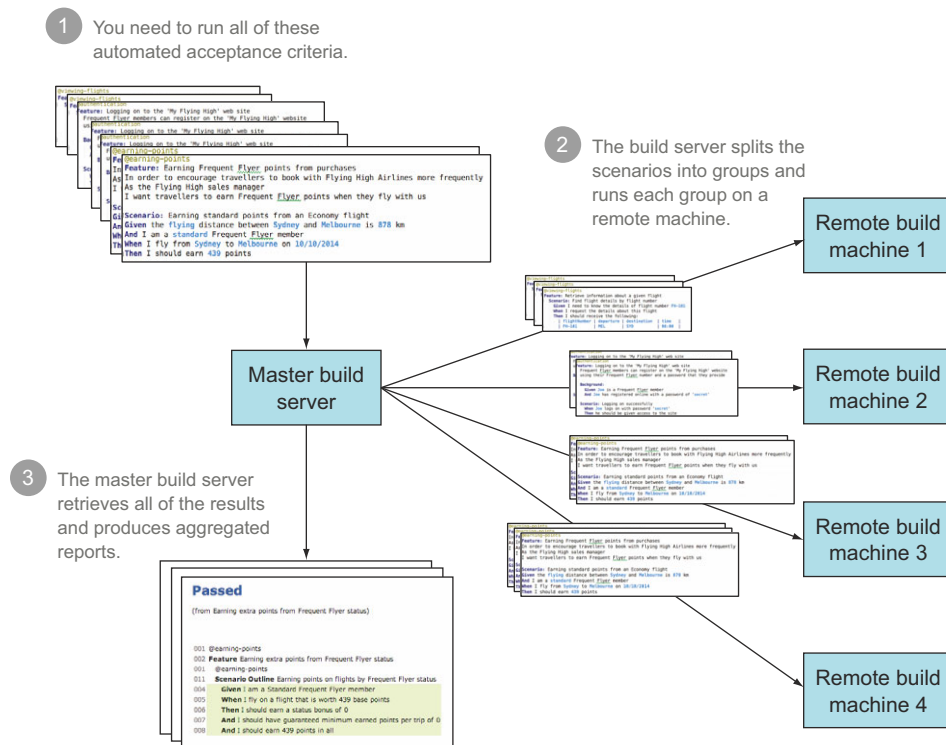


Figure 12.9 Automated acceptance criteria can be run faster by splitting them into groups that are run on different machines.

can take an approach similar to the one you saw with the JBehave/Thucydides project in section 12.5.1, with a test-runner class to run the scenarios for a given tag or tags:

```
@Cucumber.Options(tags = {"@earning_points"},
                    format = {"json:target/cucumber/earning_points.json"})
public class EarningPoints {}
```

In this configuration, each parallel build job would be configured to run a different test runner, typically by passing a system property from the command line. Using Maven, for example, you could run something like this:

```
mvn test -Dtest=EarningPoints
```

Using JBehave, you'd typically use the metafilter system property directly:

```
mvn test -Dmetafilter=+earning_points
```

Virtually any modern CI tool will let you run build jobs in parallel like this, although the functionality (and even the vocabulary used) varies from tool to tool. In Bamboo, for example, you'd set up a build plan to deploy your application. This build plan would be made up of a number of stages: compilation and unit tests, integration tests, automated acceptance criteria, performance tests, and so forth. Within the automated acceptance criteria stage, you can run several jobs in parallel, with each job running a subset of the acceptance criteria.

If you're using Jenkins, you can use a multi-configuration project type for a more centralized approach. Multi-configuration projects let you run a build job multiple times with different parameters, either on the main build server or across a number of remote machines. The multi-configuration build job lets you define a list of parameter values to be passed to the automated build script, and it runs a separate build job for each parameter value. These build jobs are run in parallel, either on the master build server or on remote build machines.

Multi-configuration build jobs work well for automated acceptance criteria. For example, rather than having a separate build job for each tag you want to execute, you could use a multi-configuration build job and provide a set of values for the test or metafilter parameter. To do this, you'd first create a multi-configuration build job in Jenkins and add a User-Defined Axis in the Configuration Matrix section (see figure 12.10).

When Jenkins runs this multi-configuration build, it'll start a build job for each value you place in the User-Defined Axis field, passing in the values you define in the user-defined axis as system properties. So in the example in figure 12.10, it will start five child jobs respectively running the following commands:

```
mvn test -Dtest=Authentication
mvn test -Dtest=EarningPoints
mvn test -Dtest=StatusPoints
mvn test -Dtest=TransferringPoints
mvn test -Dtest=ViewingFlights
```

Create a new multi-configuration build.

Define a value for each test runner you want to run in parallel.

The 'test' parameter defined in the User-Defined Axis will be passed into the child build jobs.

After the jobs are finished, collect the results and generate a report.

Figure 12.10 A Jenkins multi-configuration build job lets you run several variations of a build job in parallel and then aggregate the results.

The results of each of these child jobs are shown in the overall build job status page (see figure 12.11).

Multi-configuration builds are an excellent way to speed up automated acceptance criteria, and they're very easy to scale, though they're a Jenkins-specific feature. In the next section we'll look at another approach to this problem that will work on any build server.

12.5.3 Running parallel web tests using Selenium Grid

If you're running web tests using WebDriver (see chapter 8), another strategy you can use is to set up a Selenium Grid server (<http://docs.seleniumhq.org/projects/grid/>).

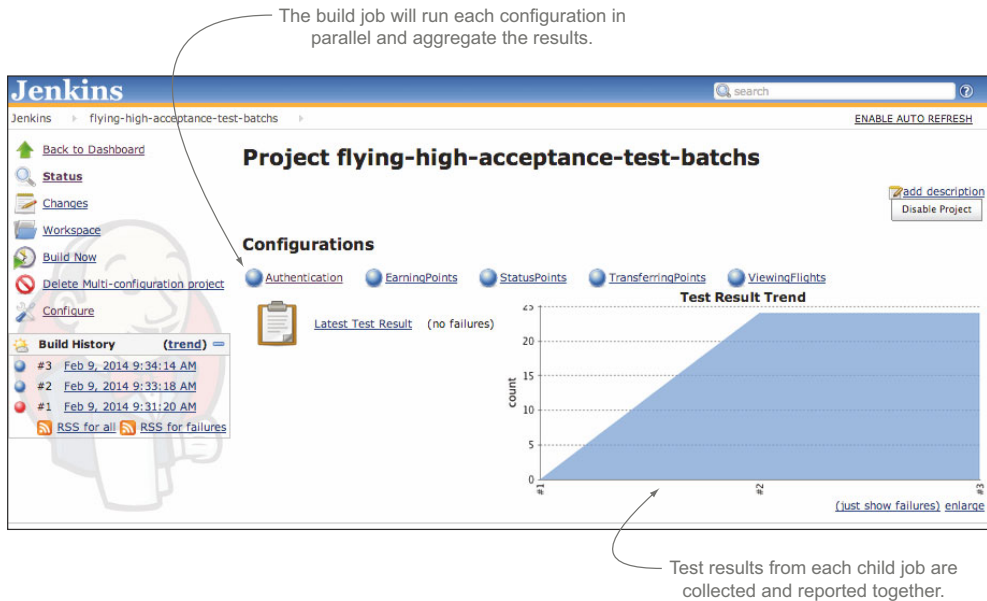


Figure 12.11 The results of each child job are displayed on the main job page.

Selenium Grid lets you set up remote machines that you can use to farm out automated web tests. Teams often use Selenium Grid to run web tests on different browsers, devices, and operating systems, or to distribute the load in large automated web testing suites. You can use Selenium Grid to run tests on browsers on Windows, Linux, and OS X, and even on mobile devices running Android and iOS.

Unlike the CI-based approach we looked at earlier, the tests are often executed on a single machine (typically the build server). But when the web tests are executed, they won't open a browser locally; instead, they'll delegate to the Selenium Grid server, which will open and control a browser on a remote machine (see figure 12.12). You can configure any browser supported by the operating system on the node machine, including PhantomJS.

Note that although Selenium Grid does make it easy to run web tests on different operating systems and browsers, it won't in itself run the web tests in parallel. For this to happen, you still need to configure the tests to run in parallel, such as by using the strategy discussed in section 12.5.1.

A Selenium Grid server is easy to set up.² The first thing you'll need is a machine to act as your Selenium Grid Hub. Choose one with a decent amount of memory and CPU power—although this machine doesn't run the browsers itself, WebDriver tests are memory-hungry, and if the server doesn't have enough resources, the tests will fail unexpectedly.

² See the Selenium Grid2 wiki for detailed instructions: <https://code.google.com/p/selenium/wiki/Grid2>.

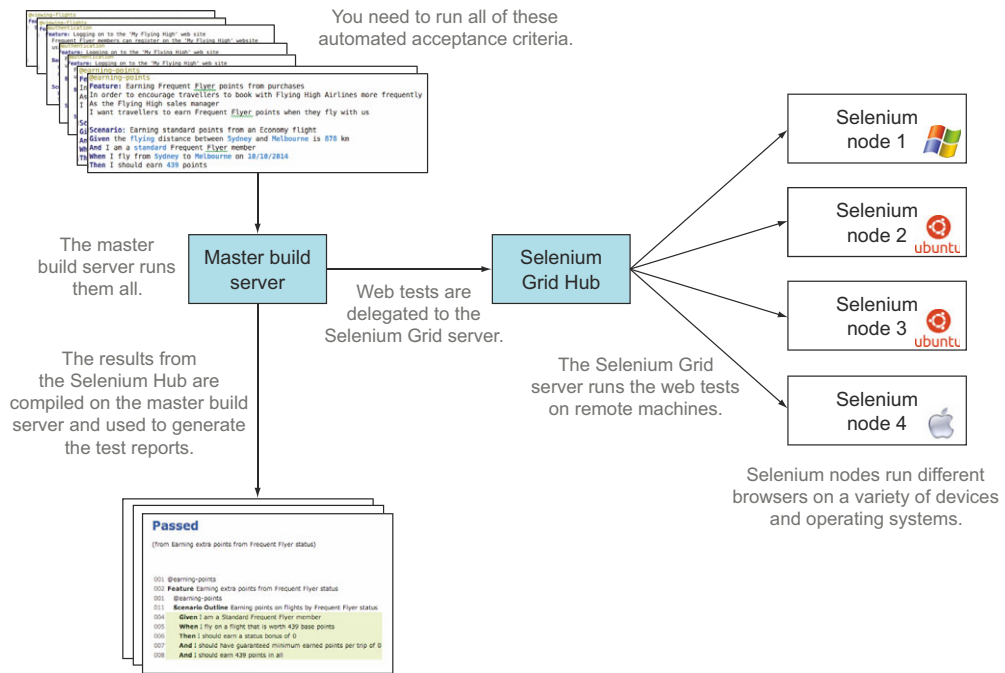


Figure 12.12 Selenium Grid lets you run WebDriver-based tests on a number of remote machines.

The Selenium Hub binary comes as part of the Selenium Server JAR file that you download from the Selenium site (<http://docs.seleniumhq.org/download/>). To start it, you execute the JAR file with the `-role hub` option, as shown here:

```
java -jar selenium-server-standalone-2.39.0.jar -port 4444 -role hub
```

This will run the Selenium Hub on port 4444.

Before you can use this hub for any tests, you need to register some Selenium node machines to run the tests. From the machine you want to run the actual tests on, you start up the Selenium Server with the `-role node` option, typically along with other options related to what browsers are installed on the node and the maximum number of each browser to open simultaneously on the node. For example, if the Selenium Hub is running on `selenium.acme.org`, you could set up a Linux node running up to four instances of Firefox and Chrome like this:

```
java -jar selenium-server-standalone-2.39.0.jar
-role node
-hub http://selenium.acme.org:4444/grid/register
-browser browserName=firefox,maxInstances=4
-browser browserName=chrome,maxInstances=4
```

Specify the URL of the hub machine. →

← **Run the same JAR file as the Hub.**

← **This time you're running a node.**

← **It can also run up to four instances of Chrome.**

← **This server can run up to four instances of Firefox.**

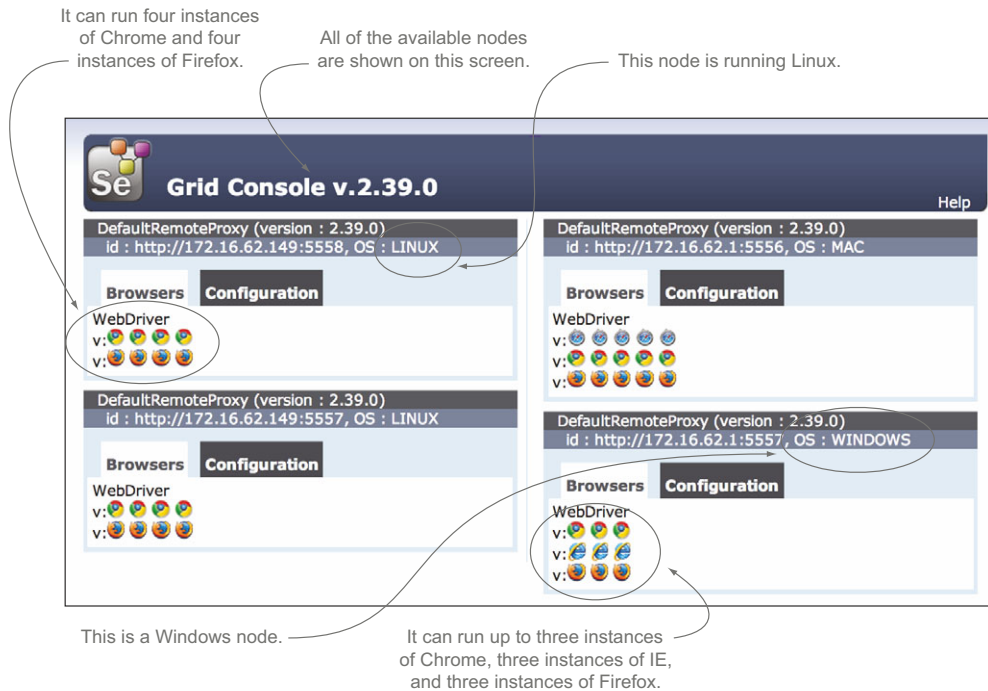


Figure 12.13 The Selenium Grid console page shows all the available Selenium nodes, what OS they're running, and which browser instances they're hosting.

There are many other options available, which are documented on the Selenium Grid website. For more complex configurations, you can also place the configuration options in a JSON file.

Once you've registered a few hubs, you can monitor them on the Selenium Hub console page, as illustrated in figure 12.13.

When the nodes are running, you can start dispatching your tests to the Selenium Hub instead of running them locally. If you're dealing with native Selenium, you'll need to use a `RemoteWebDriver` to connect to the Selenium Hub. In Java, for example, you might do something like this:

Creates a new RemoteWebDriver instance and proceeds as normal

```
DesiredCapabilities capabilities = new DesiredCapabilities();
capabilities.setBrowserName("chrome");
capabilities.setPlatform(Platform.LINUX);
URL hubUrl = new URL("http://my.selenium.hub:4444/wd/hub");
WebDriver driver = new RemoteWebDriver(hubUrl, capabilities);
```

1 Determines which Selenium node should be used

The URL of the Selenium Hub machine

The `DesiredCapabilities` object **1** is used to provide a description of the environment you'd like to run this test in. These parameters are used to send the web test to the most appropriate Selenium node machine. Then you create a `RemoteWebDriver`

instance using these preferences and the URL of the Selenium Hub, and use this WebDriver instance to run your tests.

If you're using Thucydides, you just need to provide the `webdriver.remote.url` system property, and Thucydides will run all of the scenarios on the Selenium Hub automatically.

If you provide undemanding desired driver capabilities (for example, you might just be interested in running your tests on Firefox), then Selenium Grid will transparently dispatch your web tests to as many compatible Selenium node machines as possible. This, combined with the parallel testing strategies we saw earlier, is a simple and effective way to scale your tests.

If you have more involved web-testing requirements (for example, you may need to test on a number of specific browsers), then Selenium Grid combined with the Jenkins multi-configuration build job (see figure 12.10) that you saw earlier is a great fit. But this time you'd use two User-Defined Axis fields: one for the test sets you want to execute, and another for the browsers you want to use. The resulting build job will run each test under every browser (see figure 12.14).

Partition the scenarios.

Run each scenario in each of these browsers.

Configuration Matrix	firefox	chrome	explorer	safari
Authentication				
EarningPoints				
StatusPoints				
TransferringPoints				
ViewingFlights				

Latest Test Result: (no failures)

Figure 12.14 Multi-configuration build jobs work well with Selenium Grid.

All of the strategies we've discussed here require a little organization and planning to set up efficiently. But when done well, they can lead to significantly faster and more scalable automated acceptance criteria, which in turn leads to a faster build and release process.

12.6 Summary

In this chapter you learned about the role of BDD in the build and deployment process:

- Your executable specifications need to be part of an automated build process.
- Automated executable specifications can be run on a CI server to provide faster and more reliable feedback than could be achieved with manual testing alone.
- Automated executable specifications are an essential part of any CI process.
- CI can also be used to ensure that a project's living documentation is always up to date and easily available.
- You can speed up automated acceptance criteria by running them in parallel, either on a single machine or across several machines.
- Selenium Grid is a specialized tool that makes it easy to run WebDriver-based tests in parallel on multiple machines.

If you've made it this far, congratulations! We've covered a lot of material in this book, going right across the spectrum from requirements discovery to automated tests and living documentation. I'll try to distill all that we've discussed into a few short paragraphs.

12.7 Final words

BDD is based on a few simple principles, many of which are closely aligned with broader Agile principles:

- Describe behavior, don't specify solutions.
- Discover the behavior that will deliver real business value.
- Use conversations and examples to explore what a system should do.

Conversation and collaboration is key. As Liz Keogh is fond of saying, "Having conversations is more important than capturing conversations is more important than automating conversations." In the first part of this book, you learned a number of techniques and approaches that can help facilitate these conversations and capture useful examples in an unambiguous form.

But automation has huge value too. Once you're satisfied that you've understood a requirement sufficiently, automation is where the rubber meets the road and the examples turn into executable specifications. At the high-level requirements level, you learned how to express examples in a form that can be easily automated with tools like Cucumber and JBehave, and how to automate them in a clean, robust, and sustainable way. In chapter 10 you saw how writing executable specifications at the unit-testing level really is a no-brainer. And automation is the cornerstone of the continuous integration and delivery strategies we looked at in this chapter.

One of the outcomes of both high-level and low-level executable specifications is up-to-date requirements documentation that can be viewed by the whole team at any time. This is what we call living documentation, and it's what we looked at in chapter 11.

At the end of the day, BDD is about streamlining the whole development process and delivering value. And it's much more a mindset than a particular toolset. With that in mind, adopt and adapt the practices you find valuable within your organization. I hope you'll find as much success with it as I have!

Symbols

@ (at symbol) 172
! (exclamation point) 124
\ (backslash) 172
(hash character) 123
<...> notation 20
| (pipe symbol) 125

A

acceptance criteria
 automated tools for 40
 defined 33
 examples and 67
 going from examples to 39–40
 implemented features from
 automating scenarios 267
 BDD advantages 274
 high-level acceptance
 criterion 266–267
 implementing step
 definitions 268
 specifying application
 code 269–274
 understanding domain
 model 268–269
 writing ideal code 269
 See also automating UI tests
acceptance tests 30–31
Acceptance-Test-Driven Development. *See* ATDD
Acceptance-Test-Driven
 Planning 15
actors 80

@After annotation 187–188
after_all hook 188
after_feature hook 188
after_scenario hook 187
after_tag hook 188
[AfterFeature] attribute 188
@AfterScenario annotation 187
[AfterScenario] attribute 187–188
@AfterStories annotation 188
@AfterStory annotation 188
[AfterTestRun] attribute 188
Agile methodology 30
 describing features 36
 user stories and 95
AJAX (Asynchronous JavaScript
 and XML)
 explicit waits 220
 implicit waits 220
 overview 219
AjaxElementLocatorFactory
 class 227
And step 122–123
AngularJS 178, 204, 246
annotating scenarios with
 tags 137–139
Ant 325
Appium 202
 apply() function 221
Asynchronous JavaScript and
 XML. *See* AJAX
at symbol (@) 172
ATDD (Acceptance-Test-Driven
 Development) 15
automated acceptance tests 88
 automating setup process

 initialization hooks 186–190
 overview 185–186
 scenario-specific data 190–192
 using personas 192–194
implementing features 46–55
layers of abstraction
 Business Flow layer 196–198
 Business Rules layer 195–196
 overview 194–195, 199–200
 Technical layer 198–199
nonfunctional requirements 255–257
non-UI
 overview 236–239
 testing business logic 247–251
 testing controller layer 243–247
 testing service layer 251–255
 types of 243
 when to use 240–243
overview 181–183
QA and 56
running parallel tests
 within build 332–335
 on multiple machines 335–337
 using Selenium Grid 337–343
separation of concerns 257–259
well-written 183–185

- automating scenarios
 - with Behave
 - combining steps 168
 - embedded tables 168–169
 - installing 167
 - project structure 167
 - running scenarios 169
 - step definitions 167–168
 - tables of examples 169
 - with Cucumber-JS
 - feature files for 175–176
 - running scenarios 177–178
 - setting up 175
 - step definitions 176–177
 - with Cucumber-JVM
 - passing tables to steps 164–165
 - pattern variants 163–164
 - pending steps 166
 - project setup 162–163
 - scenario outcomes 166
 - sharing data between steps 165–166
 - step definitions 163
 - tables of examples 165
 - with JBehave
 - installing 154–156
 - passing tables to steps 158
 - pattern variants 159–160
 - scenario outcomes 160–161
 - sharing data between steps 157
 - step definitions 156–157
 - tables of examples 158–159
 - overview 143
 - with SpecFlow
 - adding feature files 170–171
 - running scenarios 171–172
 - setting up 170
 - sharing data between steps 173
 - step definitions 172–173
 - tables of examples 173–174
 - step definitions
 - example-based scenarios 151–152
 - keeping methods simple 145–146
 - maintaining state 149–150
 - overview 143–145, 147–148
 - passing parameters to 148–149

- scenario outcomes 152–153
 - using table data from 150–151
- Thucydides and 154
- automating UI tests
 - with headless browsers 204
 - how much testing to do 205
 - page objects
 - exposing simple types and domain objects 227
 - fluent selectors and 234
 - libraries for 233
 - navigating with 232
 - overview 222
 - Page Objects pattern 223
 - reporting on page state 230
 - WebDriver API and 224
 - Selenium WebDriver
 - AJAX applications 219
 - element interaction methods 218
 - identifying elements 210
 - overview 206
 - test-friendly web applications 221
 - using Java API 207
 - too many tests and 203

B

- Backbone.js 204
- Background keyword 132
- backlog 64, 308–309
- backslash (\) 172
- Bamboo 335
- BAU (Business as Usual) 56
- BDD (Behavior-Driven Development)
 - advantages of 28–29, 274
 - conversation and 242
 - disadvantages of
 - acceptance tests and 30–31
 - Agile methodology
 - needed 30
 - high business engagement 29
 - silos development and 30
 - lifecycle 39–56
 - overview 3–7
 - principles of
 - embracing uncertainty 18
 - executable specifications 21–23

- focusing on features that deliver business value 16–17
 - illustrating features with concrete examples 18–20
 - living documentation 26
 - low-level specifications 23–25
 - maintaining
 - applications 27–28
 - overview 15–16
 - specifying features 17
 - requirements analysis using 14–15
 - software quality and aligning with business goals 9–10
 - overview 7–9
 - unknown data 10–12
 - TDD and 12–14
 - unit testing and
 - built on TDD practices 264
 - overview 261–263
 - specifications vs. tests 263
 - tools available for 264
- @Before annotation 187–188
- before_all hook 187
- before_feature hook 187
- before_scenario hook 187
- before_tag hook 188
- [BeforeFeature] attribute 187
- @BeforeScenario
 - annotation 187
- [BeforeScenario] attribute 187–188
- @BeforeStories annotation 187
- @BeforeStory annotation 187
- [BeforeTestRun] attribute 187
- Behat 40
- Behave 119
 - automating scenarios with
 - combining steps 168
 - embedded tables 168–169
 - installing 167
 - project structure 167
 - running scenarios 169
 - step definitions 167–168
 - tables of examples 169
 - initialization hooks in 189
- Behavior-Driven Development. *See* BDD
- [Binding] attribute 172
- browsers, headless 204
- Build Pipeline plugin 329

- build process
 - continuous delivery 328–330
 - continuous integration 325–328
 - executable specifications in overview 322
 - running from command line 325
 - self-sufficient specifications 322–323
 - using version control 324–325
- living documentation
 - publishing on build server 331–332
 - publishing on dedicated web server 332
- running parallel tests
 - within build 332–335
 - on multiple machines 335–337
 - using Selenium Grid 337–343
- Business as Usual. *See* BAU
- Business Flow layer 196–198, 223
- business goals
 - capabilities 82–84
 - defined 67
 - determining 73–76
- Feature Injection
 - gathering information through 65–66
 - hunting business value 64–65
 - injecting features with most value 65
 - overview 63–64, 66–67
- Flying High Airlines
 - example 63
- Impact Mapping 76–80
- importance of
 - understanding 67–68
- overview 70–71
- Purpose-Based Alignment Model
 - differentiating features 85–86
 - minimum impact features 86
 - overview 84–85
 - parity features 86
 - partner features 86
- revenue and 72–73
- stakeholder needs 80–82

- vision statement 68–69
- vision statement templates 69–70
- writing effective 71–72
- business logic, testing 247–251
- Business Rules layer 195–196, 223
- business value
 - features with highest 65
 - hunting 64–65
- But step 122
- By.cssSelector() method 213
- By.id() method 213
- By.xpath() method 216

C

- capabilities
 - business goals and 82–84
 - defined 67
 - features and 90–93
 - illustrated 89
- Cascading Style Sheets. *See* CSS
- Chai 295
- change control 74
- CHAOS Report 4
- Chef 142, 186
- classes
 - focusing on behavior of 286
 - naming test 287
- click() method 211, 218
- comments in scenarios 123–124
- communication 184
- conceptualizing 101
- Concordion 314
- concrete examples. *See* examples
- contains() function 217
- continuous delivery 141, 328–330
- continuous deployment 142
- continuous integration 141, 325–328
- controller layer 243–247
- conversation
 - about examples 111
 - discovering examples through 242
- Crossing the Chasm: Marketing and Selling High-Tech Products to Mainstream Customers* 69
- CSS (Cascading Style Sheets) 212
- Cucumber
 - executable scenarios and 115
 - Gherkin and 19
 - testing RESTFUL web services 252–255

- Cucumber-JS
 - feature files for 175–176
 - running scenarios 177–178
 - setting up 175
 - step definitions 176–177
- Cucumber-JVM
 - initialization hooks in 188–189
 - passing tables to steps 164–165
 - pattern variants 163–164
 - pending steps 166
 - project setup 162–163
 - scenario outcomes 166
 - sharing data between steps 165–166
 - step definitions 163
 - tables of examples 165

D

- database, initializing 186
- DDD (Domain-Driven Design) 14, 268
- declarative vs. imperative 131
- deferring implementation 281–283
- Deliberate Discovery 110–111
- dependencies, avoiding between scenarios 133–134
- dependency injection 251
- deployment process and BDD
 - continuous delivery 328–330
 - continuous integration 325–328
 - executable specifications in automated build process overview 322
 - running from command line 325
 - self-sufficient specifications 322–323
 - using version control 324–325
- living documentation
 - publishing on build server 331–332
 - publishing on dedicated web server 332
- running parallel tests
 - within build 332–335
 - on multiple machines 335–337
 - using Selenium Grid 337–343
- diff() method 253

differentiating features 85–86
 domain model 268–269
 Domain-Driven Design. *See* DDD

E

embedded tables for
 Behave 168–169
 @end-to-end tag 189
 environment.py module 189
 epics 99–100
 examples
 conversations about 111–113
 illustrated 90
 illustrating features with
 100–106
 turning into executable
 specifications 115–118
 understanding features
 using 67
 See also scenarios
 exclamation point (!) 124
 executable scenarios. *See* scenarios
 executable specifications
 in automated build process
 overview 322
 running from command
 line 325
 self-sufficient
 specifications 322–323
 using version control
 324–325
 defined 44
 implementing features 42–46
 as living documentation
 fluent assertions in
 JavaScript 295
 fluent assertions in static
 languages 295–297
 fluent coding 294
 principles of BDD 21–23
 turning examples into 115–118
 ExpectedConditions class
 220–221
 Expeditionary Combat Support
 System 3
 explicit waits 220

F

failure of software projects 4
 feature files
 for Cucumber-JS 175–176
 description for 119–120

organizing 136
 for SpecFlow 170–171
 storing scenarios in 135–136
 features
 breaking into manageable
 pieces 93–95
 breaking into stories 37
 capabilities and 90–93
 conversations about
 examples 111–113
 coverage of requested
 305–308
 defined 16, 67, 87
 Deliberate Discovery 110–111
 delivering business value
 16–17
 describing 35–37
 differentiating features 85–86
 illustrated 89
 illustrating stories with
 examples 38
 illustrating with concrete
 examples 18–20
 illustrating with
 examples 100–106
 implementing
 automated tests 46–55
 executable specifications
 42–46
 going from examples to
 acceptance criteria
 39–40
 setting up Maven and
 Git 40–41
 tests as living
 documentation 55–56
 injecting
 defined 62
 gathering information
 through 65–66
 hunting business value
 64–65
 overview 63–64, 66–67
 using those with most
 value 65
 minimum impact features 86
 overview 88–90
 parity features 86
 partner features 86
 providing highest ROI 84–86
 readiness of 304–305
 Real Options principle
 delaying until last responsi-
 ble moment 110
 options expire 109
 options have value 108–109
 overview 107–108
 specifying 17
 user stories and
 describing features using
 95–98
 differences between 98–99
 epics 99–100
 features covering
 multiple 100
 feedback and tests 185
 FEST-Assert library 215, 295
 @FindBy annotation 224–225
 findElement() method 210,
 217
 Firefox 210
 FitNesse 314
 five-why technique 74
 fluent assertions
 in JavaScript 295
 in static languages 295–297
 .Net Fluent Assertions
 library 295–297
 fluent coding 294
 fluent selectors 234
 FluentWait class 221
 Flying High Airlines
 example 63
 <forkCount> option 334
 Frequent Flyer website 208
 Freshen 167
 Function interface 221

G

Geb 233
 getAllSelectedOptions()
 method 219
 getAttribute() method 219
 getAttributeValue()
 method 218
 getFirstSelectedOption()
 method 219
 getters/setters 272
 getText() method 215,
 218–219
 Gherkin 19, 122, 142, 163, 167
 Git 40–41
 GitHub 40
 Given ... When ... Then
 structure 52, 122, 129–130
 @Given annotation 47, 156
 given keyword 39
 [Given] attribute 172
 GivenStories keyword 133

Google Chrome 210
 Gradle 325
 Groovy 47, 50
 Grunt 325
 Guava 221
 Guice 251
 Gulp 325

H

Hamcrest 295
 hash character (#) 123
 headless browsers 204
 high-level acceptance
 criterion 266–267
 high-level requirements
 310–311
 hooks 137
 HtmlUnit 204, 210

I

IBM Federal Systems Division
 team 24
 Impact Mapping 62, 76–80
 implementing features from
 acceptance criteria
 automating scenarios 267
 BDD advantages 274
 high-level acceptance
 criterion 266–267
 implementing step
 definitions 268
 specifying application
 code 269–274
 understanding domain
 model 268–269
 writing ideal code 269
 implicit waits 220
 initialization hooks
 in Behave 189
 in Cucumber-JVM 188–189
 in JBehave 188
 overview 186–188
 in SpecFlow 189–190
 injecting features
 gathering information
 through 65–66
 hunting business value
 64–65
 overview 63–64, 66–67
 using those with most
 value 65
 Internet Explorer 210
 @issue tag 309

J

Jasmine 246, 289–290
 Java
 automating scenarios with
 Cucumber-JVM
 passing tables to steps
 164–165
 pattern variants 163–164
 pending steps 166
 project setup 162–163
 scenario outcomes 166
 sharing data between
 steps 165–166
 step definitions 163
 tables of examples 165
 automating scenarios with
 JBehave
 installing 154–156
 passing tables to steps 158
 pattern variants 159–160
 scenario outcomes
 160–161
 sharing data between
 steps 157
 step definitions 156–157
 tables of examples 158–159
 JavaScript
 automating scenarios with
 CucumberJS
 feature files for 175–176
 running scenarios 177–178
 setting up 175
 step definitions 176–177
 fluent assertions in 295
 JavaScript Object Notation. *See*
 JSON
 JBehave
 and keyword 122
 automating scenarios with
 installing 154–156
 passing tables to steps 158
 pattern variants 159–160
 scenario outcomes
 160–161
 sharing data between
 steps 157
 step definitions 156–157
 tables of examples
 158–159
 executable scenarios and 115
 GivenStories keyword 133
 initialization hooks in 188
 syntax for 42
 Jenkins 326, 335

JIRA 120, 137
 JMeter 256
 JSON (JavaScript Object
 Notation) 254
 JSONassert 254
 JUnit 50, 317
 -junit option 169

K

Karma test runner 178
 known entities 192, 194

L

layers of abstraction
 Business Flow layer 196–198
 Business Rules layer 195–196
 overview 194–195, 199–200
 Technical layer 198–199
*Leader's Guide to Radical Manage-
 ment, The* 17
 legacy applications 319–320
 Lettuce 167
 living documentation
 additional background for
 stories 314–315
 comments not part of 124
 executable specifications as
 fluent assertions in
 JavaScript 295
 fluent assertions in static
 languages 295–297
 fluent coding 294
 feature coverage 305–308
 feature readiness 304–305
 organizing
 by high-level
 requirements 310–311
 overview 309–310
 using tags 311–312
 overview 302–304
 principles of BDD 26
 product backlog 308–309
 publishing
 on build server 331–332
 on dedicated web
 server 332
 release reporting 312–314
 technical
 for legacy
 applications 319–320
 overview 315–316
 unit tests as 316–318
 tests as 55–56

low-level requirements
 deferring implementation
 using stubs and mocks
 281–283
 discovering new classes and
 services 279–280
 exploring using step defini-
 tion code 275–277
 implementing simple classes
 or methods
 immediately 280–281
 overview 274–275
 using minimal
 implementation 281
 using tables of examples
 277–279
 low-level specifications 23–25

M

maintaining applications
 56–58
 market differentiation
 criteria 85
 Maven 40–41, 154, 325
 Meta keyword 137
 minimum impact features 86
 mission critical criteria 85
 Mocha 289
 mocks 281–283
 models 65
 MSBuild 325
 must as keyword 284
 MVC (Model-View-
 Controller) 243

N

NAnt 325
 NASA space shuttle 24
 .NET
 adding feature files
 170–171
 running scenarios 171–172
 setting up 170
 sharing data between
 steps 173
 step definitions 172–173
 tables of examples 173–174
 Node.js 175, 208
 nonfunctional
 requirements 255–257
 NSpec 290–291
 NuGet 170
 NUnit 317

O

objective-driven
 management 75
 Octopus Deploy 142
 open() method 223
 Opera 210
 outcomes, scenario
 for Cucumber-JVM 166
 for JBehave 160–161
 overview 152–153
 outside-in development
 approach 264–266

P

page objects
 exposing simple types and
 domain objects 227
 fluent selectors and 234
 libraries extending 233
 navigating with 232
 overview 222
 Page Objects pattern 223
 reporting on page state 230
 using with WebDriver API 224
 parallel tests, running
 within build 332–335
 on multiple machines 335–337
 using Selenium Grid 337–343
 <parallel> option 334
 parameters, passing to step
 definitions 148–149
 parity features 86
 partner features 86
 pattern variants
 for Cucumber-JVM 163–164
 for JBehave 159–160
 @Pending annotation 48
 pending steps 166
 personas 192–194
 PhantomJS 204, 207, 210
 Pip 167
 pipe symbol (|) 125
 PMOs (project management
 offices) 69
 pom.xml file 162
 project management offices. *See*
 PMOs
 project objectives 35
 project vision 66
 publishing living documentation
 on build server 331–332
 on dedicated web server 332
 Puppet 142, 186

Purpose-Based Alignment Model
 differentiating features 85–86
 minimum impact features 86
 overview 84–85
 parity features 86
 partner features 86
 purpose of 63
 Python
 combining steps 168
 embedded tables 168–169
 installing 167
 project structure 167
 running scenarios 169
 step definitions 167–168
 tables of examples 169

Q

QA team 56
 quality gateways 328
 Queensland Health Depart-
 ment payroll system 9
 QuickTest Professional 203

R

Real Options principle
 delaying until last responsible
 moment 110
 options expire 109
 options have value 108–109
 overview 107–108
 Record-Replay style scripting
 tools 203
 reflection 101
 regression testing 141
 regular expressions 163
 release reporting 312–314
 requirements analysis 14–15
 breaking features down into
 stories 37
 describing features 35–37
 illustrating stories with
 examples 38
 RESTFUL web services 252–255
 revenue and business goals
 72–73
 ROI (return on investment)
 84–86
 RSpec 287–289

S

Safari 210
 Scala 47

- Scenario keyword 120
- scenarios
 - automating 143
 - automating with Behave
 - combining steps 168
 - embedded tables 168–169
 - installing 167
 - project structure 167
 - running scenarios 169
 - step definitions 167–168
 - tables of examples 169
 - automating with Cucumber-JS
 - feature files for 175–176
 - running scenarios 177–178
 - setting up 175
 - step definitions 176–177
 - automating with Cucumber-JVM
 - passing tables to steps 164–165
 - pattern variants 163–164
 - pending steps 166
 - project setup 162–163
 - scenario outcomes 166
 - sharing data between steps 165–166
 - step definitions 163
 - tables of examples 165
 - automating with JBehave
 - installing 154–156
 - passing tables to steps 158
 - pattern variants 159–160
 - scenario outcomes 160–161
 - sharing data between steps 157
 - step definitions 156–157
 - tables of examples 158–159
 - automating with SpecFlow
 - adding feature files 170–171
 - running scenarios 171–172
 - setting up 170
 - sharing data between steps 173
 - step definitions 172–173
 - tables of examples 173–174
 - avoiding dependencies
 - between scenarios 133–134
 - avoiding duplication
 - among 126
 - comments in 123–124
 - data for automated acceptance tests 190–192
 - defined 117
 - describing 120–121
 - feature file description 119–120
 - Given ... When ... Then
 - structure 121–123, 129–131
 - organizing
 - annotating scenarios with tags 137–139
 - feature file can contain multiple scenarios 136
 - feature file organization 136
 - storing in feature file 135–136
 - providing background and context 132–133
 - step definitions
 - example-based
 - scenarios 151–152
 - keeping methods simple 145–146
 - maintaining state 149–150
 - overview 143–145, 147–148
 - passing parameters to 148–149
 - scenario outcomes 152–153
 - using table data from 150–151
 - tables in
 - of examples 125–128
 - individual steps 125
 - title for 120
- SDS (system design specification) 25
- selectByIndex() method 219
- selectByValue() method 219
- selectByVisibleText() method 219
- selectors, CSS 212–213
- Selenium Grid 337–343
- Selenium WebDriver
 - AJAX applications
 - explicit waits 220
 - implicit waits 220
 - overview 219
 - element interaction methods
 - for check boxes 219
 - for drop-down lists 219
 - overview 218
 - for radio buttons 219
 - for text input fields 218
 - identifying elements
 - by link text 212
 - nested lookups 217
 - overview 210
 - using CSS 212
 - using XPath 215
 - overview 206
 - page objects and
 - exposing simple types and domain objects 227
 - fluent selectors and 234
 - libraries extending 233
 - navigating with 232
 - overview 222
 - Page Objects pattern 223–224
 - reporting on page state 230–232
 - using with WebDriver API 224–227
 - test-friendly web
 - applications 221–222
 - using Java API 207–210
 - sendKeys() method 211, 218
 - separation of concerns 257–259
 - service layer
 - RESTFUL web services 252–255
 - testing 251–252
 - Service Oriented Architecture. *See* SOA
 - setters, testing 272
 - should as keyword 284
 - Should.js 295
 - siloed development 30
 - SMART acronym 71
 - SOA (Service Oriented Architecture) 251
 - SoapUI 256
 - software quality and BDD
 - aligning with business goals 9–10
 - overview 7–9
 - unknown data 10–12
 - SpecFlow 40, 115, 119
 - automating scenarios with
 - adding feature files 170–171
 - running scenarios 171–172
 - setting up 170
 - sharing data between steps 173
 - step definitions 172–173
 - tables of examples 173–174
 - initialization hooks in 189–190
 - Specification by Example 15
 - specifications, Spock 50

specifying application code
 adding new features 273–274
 describing expected behavior
 as failing test 271
 making test pass 271–272
 refactoring 272
 Specs2 291–293
 Spock 50, 291–293
 Spring 251
 stakeholders 80–82
 state, maintaining between step
 definitions 149–150
 step definitions
 for Behave 167–168
 for Cucumber-JS 176–177
 for Cucumber-JVM 163
 example-based
 scenarios 151–152
 exploring low-level
 requirements 275–277
 implementing features 268
 for JBehave 156–157
 keeping methods simple
 145–146
 maintaining state 149–150
 overview 143–145, 147–148
 passing parameters to
 148–149
 scenario outcomes 152–153
 for SpecFlow 172–173
 using table data from
 150–151
 step library 197
 @Steps annotation 197
 stories
 breaking features down
 into 37
 defined 33
 illustrating with examples 38
 stubs 281–283
 Surefire 332
 system design specification. *See*
 SDS

T

tables of examples
 for Behave 169
 for Cucumber-JVM 165
 for JBehave 158–159
 low-level requirements
 using 277–279
 overview 125–128
 for SpecFlow 173–174

tabular parameters 158
 tags
 annotating scenarios
 with 137–139
 organizing living documenta-
 tion using 311–312
 task boards 308
 TDD (Test-Driven Development)
 BDD and 12–14
 BDD built on practices of 264
 overview 261–263
 TeamCity 335
 Technical layer 198–199, 223
 technical living documentation
 for legacy applications
 319–320
 overview 315–316
 unit tests as 316–318
 @Test annotation 288
 test suite 183
 Test-Driven Development. *See*
 TDD
 testing
 getters/setters 272
 parallel
 within build 332–335
 on multiple machines
 335–337
 using Selenium Grid
 337–343
 See also automated acceptance
 tests
 testing UI
 with headless browsers
 204–205
 how much testing to do
 205–206
 page objects
 exposing simple types
 and domain objects
 227–230
 fluent selectors and
 234–235
 libraries for 233–234
 navigating with 232
 overview 222–223
 Page Objects pattern
 223–224
 reporting on page
 state 230–232
 WebDriver API and
 224–227
 Selenium WebDriver
 AJAX applications
 219–221

 element interaction
 methods 218–219
 identifying elements
 210–218
 overview 206–207
 test-friendly web
 applications 221–222
 using Java API 207–210
 too many tests and
 203–204
 The Grinder 256
 @Then annotation 47, 156
 then keyword 39
 Then step 122, 131
 [Then] attribute 172
 Thucydides 120, 136, 154, 157,
 233
 tools, unit testing
 Jasmine 289–290
 NSpec 290–291
 RSpec 287–289
 Specs2 291–293
 Spock 291–293
 traditional 285–287
 train timetable application
 determining project
 objectives 35
 implementing features
 automated tests 46–55
 executable
 specifications 42–46
 going from examples to
 acceptance criteria
 39–40
 setting up Maven and
 Git 40–41
 tests as living
 documentation 55–56
 maintenance 56–58
 overview 33–34
 requirements analysis
 breaking features down
 into stories 37
 describing features 35–37
 illustrating stories with
 examples 38
 @transfers annotation 169
 Transport for NSW 73
 Twill 204

U

UAT (User Acceptance
 Testing) 322, 329
 ubiquitous 268

- UI tests
 - with headless browsers 204–205
 - how much testing to do 205–206
 - page objects
 - exposing simple types and domain objects 227–230
 - fluent selectors and 234–235
 - libraries for 233–234
 - navigating with 232
 - overview 222–223
 - Page Objects pattern 223–224
 - reporting on page state 230–232
 - WebDriver API and 224–227
 - Selenium WebDriver
 - AJAX applications 219–221
 - element interaction methods 218–219
 - identifying elements 210–218
 - overview 206–207
 - test-friendly web applications 221–222
 - using Java API 207–210
 - too many tests and 203–204
 - uncertainty 88
 - unit testing
 - acceptance criteria to implemented features
 - automating scenarios 267
 - BDD advantages 274
 - high-level acceptance criterion 266–267
 - implementing step definitions 268
 - specifying application code 269–274
 - understanding domain model 268–269
 - writing ideal code 269
 - BDD and
 - built on TDD practices 264
 - overview 261–263
 - specifications vs. tests 263
 - tools available for 264
 - executable specifications as living documentation
 - fluent assertions in JavaScript 295
 - fluent assertions in static languages 295–297
 - fluent coding 294
 - importance of 24
 - low-level requirements
 - deferring implementation using stubs and mocks 281–283
 - discovering new classes and services 279–280
 - exploring using step definition code 275–277
 - implementing simple classes or methods immediately 280–281
 - overview 274–275
 - using minimal implementation 281
 - using tables of examples 277–279
 - outside-in development approach 264–266
 - TDD and 261–263
 - tests as technical living documentation 316–318
 - tools for
 - Jasmine 289–290
 - NSpec 290–291
 - RSpec 287–289
 - Specs2 291–293
 - Spock 291–293
 - traditional 285–287
 - U.S. Air Force 3
 - User Acceptance Testing. *See* UAT
 - user stories
 - defined 88
 - describing features using 95–98
 - epics and 99–100
 - features and 98–99
 - features covering multiple 100
 - illustrated 89
 - UX (user experience) 192
-
- V**
- value, business 64–65
 - version control 324–325
 - vision statement 62, 68–69
 - vision statement templates 69–70
-
- W**
- W3 web site 215
 - Wait interface 220
 - wasted effort 28
 - Waterfall-style projects 74
 - WatN 233
 - Watir 233
 - WebElement class 210
 - WebElementState class 232
 - Webrat 204
 - @When annotation 47, 156
 - when keyword 39
 - When step 122, 130–131
 - [When] attribute 172
 - will as keyword 284
-
- X**
- XPath 215–217
-
- Y**
- YAGNI (You Ain't Gonna Need It) principle 109

BDD IN ACTION

John Ferguson Smart



You can't write good software if you don't understand what it's supposed to do. Behavior Driven Development (BDD) encourages teams to use conversation and concrete examples to build up a shared understanding of how an application should work and which features really matter. With an emerging body of best practices and sophisticated new tools that assist in requirement analysis and test automation, BDD has become a hot, mainstream practice.

BDD in Action teaches you BDD principles and practices and shows you how to integrate them into your existing development process, no matter what language you use. First, you'll apply BDD to requirements analysis so you can focus your development efforts on underlying business goals. Then, you'll discover how to automate acceptance criteria and use tests to guide and report on the development process. Along the way, you'll apply BDD principles at the coding level to write more maintainable and better documented code.

What's Inside

- BDD theory and practice
- How BDD will affect your team
- BDD for acceptance, integration, and unit testing
- Examples in Java, .NET, JavaScript, and more
- Reporting and living documentation

No prior experience with BDD is required.

John Ferguson Smart is a specialist in BDD, automated testing, and software lifecycle development optimization.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/BDDinAction

“Delivers a thorough treatment of the current state of BDD tools.”

—From the Foreword by Dan North, Creator of BDD

“Learn BDD from top to bottom.”

—Dror Helper, CodeValue

“The first complete step-by-step guide to BDD.”

—Marc Bluemner
liquidlabs GmbH

“Many useful techniques, tools, and concepts to make you more productive.”

—Karl Métivier
Facilité Informatique