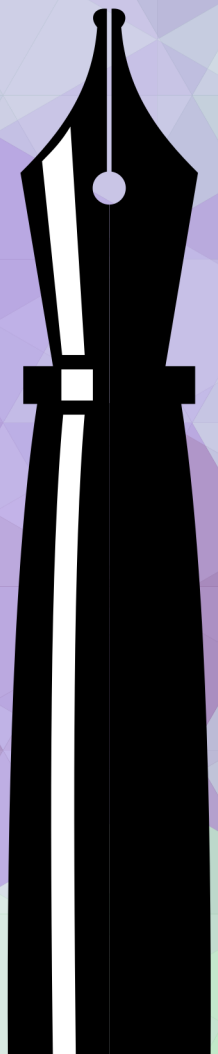
THE BDD BOOKS

Formulation

Document examples with Given/When/Then



> Required reading for anyone embarking on their BDD journey
>
> — Daniel Terhorst-North

Seb Rose and Gáspár Nagy

Forewords by Angie Jones and Daniel Terhorst-North

The BDD Books – Formulation

Document examples with Given/When/Then

Seb Rose and Gáspár Nagy

This book is for sale at http://leanpub.com/bddbooks-formulation

This version was published on 2021-04-30

Tweet This Book!

Please help Seb Rose and Gáspár Nagy by spreading the word about this book on Twitter!

The suggested hashtag for this book is #bddbooks.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#bddbooks

Praise for Formulation

"This is a very important book for all BDD practitioners. Seb and Gáspár present actionable tips that will help you get the most out of scenarios and feature files, by capturing the examples in a way that's easy to communicate, understand and maintain. I especially liked the part on organising good documentation."
– Gojko Adzic, author of Running Serverless, Impact Mapping, Specification by Example and a few more books... Working on Narakeet.

"I like the way the authors put collaboration at the heart of this book. In particular the stories they tell about a fictional team and how they go from examples to successfully formulated Gherkin. It gives you a flavour of how BDD should feel when you're doing it well, alongside detailed and useful advice. Any development team looking to improve the way they work with user stories and test automation would benefit from applying the ideas in this book."
– Emily Bache, technical agile coach, author of Technical Agile Coaching with the Samman method

"If you are interested in BDD and more importantly understanding requirements, you should absolutely check this one out. As always Seb and Gáspár take the time to write in a clear and actionable way"
– Abby Bangser, site reliability engineer

"Writing good quality BDD scenarios is something of an art. It looks easy on the surface, but teams too often struggle to get it right. Seb and Gáspár have written a clear and easy to read book, packed with good habits and useful tips, that manages to capture the essence of what goes into well written Given/When/Then scenarios. If you want to master the art of writing high quality BDD scenarios, this book is an invaluable resource: you should read it."
– John Ferguson Smart, author of BDD in Action and founder of the Serenity Dojo (http://www.serenity-dojo.com/)

"What a great follow-up book to BDD: Discovery. As I was reading BDD: Formulation, I realized how little I knew about some aspects of BDD. This book isn't only about how to write Gherkin, but provides great ideas about how to think about writing and organizing tests. The examples throughout are a great way to emphasize what you

mean. Thank you so much for sharing your knowledge with the community."
– Janet Gregory, co-author of three agile testing books, including Agile Testing Condensed

"There is a famous quote that goes like this: 'If I had more time, I would have written a shorter letter'. This applies to Gherkin document too. Well-written Gherkin documents are not only short, they also serve three purposes: specification, acceptance testing and living documentation. Without a style guide, this is hard to do. Over the past decade, Seb and Gáspár have collected and catalogued the essential elements of Gherkin style and presented them in this book. It is the definitive guide for writing succinct and clear Gherkin documents."
– Aslak Hellesøy, creator of Cucumber

"Formulation aptly closes the gap between team members' discussions around scenarios and the automation that results, helping them formalize their thoughts into readable business language... even if they started with the tools! The problems faced by the WIMP team are realistic, and widespread. The solutions are simply presented, with examples (of examples) that help to illustrate how easy and effective BDD can be. I highly recommend this book for learners picking up BDD for the first time, for practitioners looking to hone their craft, and for experts looking for vocabulary and concepts to pass on their knowledge and keep the culture of conversation alive."
– Liz Keogh, Lean, agile and Cynefin coach

"Seb and Gáspár elegantly convey advice from their extensive BDD experience on how to transform an understanding of behavior into readable scenarios. The book is suitable for BDD beginners to start their journey. It's useful as well to those who have created many feature files to find out how Seb and Gáspár's guidance may make better scenarios."
– Ken Pugh, ATDD/BDD at Ken Pugh, Inc.

"Formulating specs that stand the test of time is deceptively difficult. Seb and Gáspár break down the essential ingredients, provide many patterns for writing great specs and cover the practicalities of applying them to real world scenarios.

This book is a treasure trove of good ideas that will help people of all levels of experience."
– Tom Roden, Neuri Consulting, co-author of 50 Quick Ideas to Improve Your Tests

"We know that software people spend more time reading the code and tests more than they spend time writing. This book is a guide for how to make that reading easier. Chock full of helpful ideas for how to write scenarios, and what to leave out, you will learn to write scenarios people want to read. Even better, when your team uses these guidelines, everyone will see the value of the various scenarios. Use the ideas here to improve everyone's understanding of your products so you can improve the products themselves."

– ***Johanna Rothman, author, Modern Management Made Easy books***

Contents

Foreword by Angie Jones . . . . . **i**

Foreword by Daniel Terhorst-North . . . . . **ii**

Preface . . . . . **iii**
The WIMP project iv
Who this book is for iv
Why you should read this book v
How to read this book v
Rules and examples vii
BDD needs skilled testers vii
Why you should listen to us viii
Online resources ix
Acknowledgments ix

Chapter 1 – What is formulation? **1**
1.1 – Where does formulation fit into BDD? 1
1.2 – Shared understanding 2
1.3 – Two types of scenarios 3
1.4 – Many formats 3
1.5 – Gherkin overview 5
1.6 – Living documentation 6
1.7 – What we just learned 7

Chapter 2 – Cleaning up an old scenario **8**
2.1 – The old scenario 8
2.2 – Keep your scenarios BRIEF 10
2.3 – Using example maps to provide focus 13

2.4 – Document the essence of the behaviour 18
2.5 – Scenarios should read like a specification 20
2.6 – Use real data when it provides clarity 21
2.7 – Communication, not testing 24
2.8 – Illustrative scenarios 25
2.9 – What we just learned 27

Chapter 3 – Our first feature 28
3.1 – Feature files 28
3.2 – A sample feature file 28
3.3 – Gherkin basics 31
3.4 – The feature file 32
3.5 – Rules 34
3.6 – Scenario structure 34
3.7 – Multiple contexts 36
3.8 – Keeping context essential 37
3.9 – Is it a *Given* or a *When*? 38
3.10 – Multiple outcomes 39
3.11 – Conjunctions always need consideration 40
3.12 – Data tables 41
3.13 – Scenario outlines 42
3.14 – Keep tables readable 48
3.15 – Readable blocks of text 48
3.16 – What we just learned 49

Chapter 4 – A new user story 51
4.1 – Restricting customers using a blocklist 51
4.2 – Write it upwards 53
4.3 – Too many cooks 54
4.4 – Quotation marks 58
4.5 – There's no "I" in "Persona" 59
4.6 – Does repetition matter? 61
4.7 – Readability trumps ease of automation 62
4.8 – Background 62
4.9 – Unformulated examples 65
4.10 – Commenting in feature files 66

4.11 – Setting the context . . . 68
4.12 – Staying focused . . . 71
4.13 – Formulation gets faster . . . 75
4.14 – Incremental specification . . . 78
4.15 – Manual scenarios . . . 79
4.16 – Who does what and when . . . 81
4.17 – What we just learned . . . 82

Chapter 5 – Organizing the documentation . . . 84
5.1 – User stories are not the same as features . . . 84
5.2 – Separation of concerns . . . 87
5.3 – Documentation evolves . . . 90
5.4 – Documenting the domain . . . 92
5.5 – Tags are documentation too . . . 95
5.6 – Journey scenarios . . . 99
5.7 – Structuring the living documentation . . . 104
5.8 – Documenting shared features . . . 109
5.9 – Targeted documentation . . . 111
5.10 – What we just learned . . . 112

Chapter 6 – Coping with legacy . . . 114
6.1 – BDD on legacy projects . . . 114
6.2 – Incremental documentation . . . 115
6.3 – Making use of manual test scripts . . . 117
6.4 – What we just learned . . . 119

What's next . . . 121
Where we've got to . . . 121
What's left to cover . . . 121
How else we can help . . . 121

Appendices . . . 122
Gherkin cheat-sheet . . . 123
Gherkin jump-list . . . 124
Formulation "smells" jump-list . . . 125
Formulated feature files . . . 127

Bibliography **135**

Foreword by Angie Jones

I'm a big believer of the phrase "show, don't tell", especially as it relates to software development. Many experts are constantly telling us what we should be doing but offer very few examples that actually show us how to do so.

Gherkin is no different. A simple internet search results in article after article explaining how teams are claiming to practice Behaviour Driven Development (BDD) but are falling short on the core intention of this process, which is to communicate early in the development process to ensure the team is building the right thing. Very few of these articles touch on how to effectively do so.

I was delighted to learn that Seb and Gáspár took on the challenge of addressing this gap by providing an entire book explaining how to effectively write good Gherkin scenarios that will assist in your overall BDD efforts.

They do a wonderful job of sharing stories to give context to their reasoning. They draw from their wealth of experience to thoroughly cover the rich Gherkin language. They take no shortcuts on explaining the how and the why.

This book is for teams who are committed to collaborating in an effective manner to build quality software. If applied, these comprehensive lessons are sure to get any software team on the right track, no matter what their previous experience has been with BDD.

Buckle up! You're in for an exciting ride!

Angie Jones

Foreword by Daniel Terhorst-North

Back in 2003, when I first tried framing test-driven development without the word "test", I had no idea where it would lead. Along the way I've been lucky enough to meet brilliant folks like Seb Rose and Gáspár Nagy who, over nearly two decades, have been instrumental in evolving BDD into a rich and detailed landscape of collaboration and value creation.

Arguably, formulation is the least well understood of the BDD disciplines. Much has been written about collaboration and shared understanding, and automation is where most people start with BDD. To my mind, formulation is where the real power and leverage of BDD lies.

It takes skill and experience to capture, "the context, the whole context, and nothing but the context", of a scenario to describe the desired behaviour of a digital product. Gáspár and Seb have done a masterful job in providing a comprehensive and accessible treatment of a complex topic, with plenty of examples and illustrations to light the way.

I found myself nodding along with each chapter and each story. The sidebars alone are solid gold. As an example, I don't know whether it was deliberate but their section on the *BRIEF* acronym is a self-referential mini-masterclass. It uses the *Business Language* of BDD, works through a *Real* example, is *Intention-revealing*, only contains *Essential* information, and is *Focused* on one thing.

I recognise much of my own experience in these pages, but beyond that an articulation of things that are, "obvious when you say them out loud". This book should be required reading for anyone embarking on their BDD journey, and will act as a timely refresher and an indispensable reference for those further along.

Daniel Terhorst-North

Preface

Behaviour Driven Development (BDD) is an agile approach to delivering software, comprising three *core BDD practices*: discovery, formulation, and automation.

- Discovery – creates a shared understanding of the requirements through collaboration, typically achieved through a structured conversation centered on rules and examples.
- Formulation – examples of system behaviour are documented using business terminology.
- Automation – the documentation is automated, creating living documentation that verifies the system's behaviour.

The **BDD Books series** guides you through the end-to-end adoption of BDD, including specific practices needed to successfully drive development using collaboratively authored specifications and living documentation. This is the second book in the BDD Books series and presents a deep dive into the BDD practice of *formulation* – the writing of executable specifications in a business readable format.

The first book covered the BDD practice of *discovery* and we strongly recommend that you read *Discovery – Explore behaviour using examples (Book 1)* [Nagy2018] first. Once you've practiced *formulation* using the principles we outline in this book, you can read *Automation with SpecFlow (Book 3)* [NagyInPrep].

Many development teams come to BDD through the desire to improve their test automation. Improved test automation is one of the significant outcomes of following a BDD approach, but it is a *downstream* outcome. Unless you adopt the practices in the order described (*discovery*, *formulation*, *automation*) you will not gain the expected benefits.

Conversely, you will achieve significant improvements in your software development activities just by practicing *discovery* on its own. Add *formulation* and you'll get the extra benefits that come from growing a truly ubiquitous language through an active review and feedback process. *Automation* then turns the scenarios into an

executable specification that guides development and provides a safety net during maintenance, plus business readable *living documentation* of the system that is guaranteed to be up-to-date.

Teams that follow the advice we present have the best chance of creating valuable living documentation that will guide development, engage business stakeholders and reduce the cost of maintenance and enhancements.

The WIMP project

The BDD Books series follows an imaginary team developing the *Where Is My Pizza* (WIMP) product. WIMP is a pizza-delivery management application for a large pizza company. The application will allow clients to track the real-time location of their order(s). The WIMP team is made up of the following team members, the first letter of their names indicating their role in the team:

- **P**atricia – the Product Owner (PO)
- **D**ave – a developer
- **D**ishita – a developer
- **I**an – an intern
- **T**racey – a tester
- **U**lisses – a UX specialist

In *Formulation*, we follow the WIMP team as they practice their formulation skills using Gherkin, the specification format that is understood by Cucumber and SpecFlow.

Who this book is for

This book is written for everyone involved in the specification and delivery of software (including product owners, business analysts, developers and testers). You don't need any previous BDD experience. The book describes how all stakeholders need to be involved in the creation of a product's specification. How you get involved will depend on your skills, your other time commitments and a host of other factors,

but the involvement of all concerned is essential. Whether you come up with the words, do the typing or provide constructive feedback, you will find this book indispensable.

It's worth noting that while *Discovery* [Nagy2018] is completely tool agnostic, this book is focused on tools that understand the Gherkin syntax. This still encompasses a large number of tools, including Cucumber[1], SpecFlow[2], JBehave[3], Behave[4], and Behat[5].

Why you should read this book

Your team may understand what needs to be delivered when they come out of a requirement workshop, but what about people that weren't in the meeting, or the attendees' future selves when they are tasked with fixing a defect a year down the line? In this book we show you how to capture that understanding using Gherkin. Gherkin allows you to write specifications using your own business language that is understandable by everyone on the team, but sufficiently structured that it can also be understood by automation tools.

Because Gherkin's structure is so simple it's easy to write, but it's harder to ensure that it's written in a way that is easy to understand, easy to maintain, and valuable enough for non-technical team members to actively participate in its authoring. This is the art of formulation that we teach in this book.

How to read this book

This book contains plenty of tips and tricks to help you write better BDD scenarios, as well as pointing out a handful of practices to avoid. To make it easier to digest, we have grouped our advice according to the kind of problem that our WIMP team is trying to solve.

[1] https://cucumber.io/
[2] https://specflow.org/
[3] https://jbehave.org/
[4] https://github.com/behave/behave
[5] https://docs.behat.org/

- Chapter 1, "What is formulation?", introduces the concept of formulation, its role among the common BDD practices (discovery, formulation, automation) and defines the most important elements of terminology.
- Chapter 2, "Cleaning up an old scenario", shows how a "bad" BDD scenario can be fixed and introduces the BRIEF acronym that captures the six core principles of good scenarios.
- Chapter 3, "Our first feature", guides you through a complete feature file that the WIMP team created. Through this we show the connection between the BDD scenarios and the requirements discussed during discovery. We also learn about the fundamental elements of a feature file.
- Chapter 4, "A new user story", guides you through the WIMP team's important discussions and decisions as they formulate a new scenario. We show how the resulting agreements lead to faster progress for subsequent scenarios.
- Chapter 5, "Organizing the documentation", focuses on the challenges that teams often experience when they have many feature files. As the WIMP team builds up their feature file structure we clarify the difference between story and feature-based structuring, explain how the scenarios can be efficiently used as documentation, and explore formulating shared features and journey scenarios.
- Chapter 6, "Coping with legacy", discusses how BDD can be introduced into a legacy project. You can see what incremental strategies might work and how to deal with existing manual test scripts.

In each chapter, the WIMP team faces new problems and considers what techniques to apply. To emphasize that there are no general "best practices", we follow the team's discussions as they consider alternatives, side-effects, and trade-offs. Remember however, that this is a simplified example and therefore some discussions might be incomplete.

All the information you need to follow along is in the text of the book, but if you prefer you can download the source code for each chapter (see *Online resources*, later in this chapter).

In addition to the six chapters, the Appendices contain a Gherkin cheat-sheet to get a quick overview of the Gherkin features. This is followed by two jump-lists that help you find details in the book related to Gherkin elements and formulation "smells". The Appendices also contain the final feature files that the WIMP team created.

Rules and examples

Discovery is achieved in a collaborative session during which the delivery team explore their understanding of a story's requirements using concrete examples. At minimum, the business, development and test perspectives should be represented. Our recommendation is that the workshop should use *Example Mapping*[6].

The scope of a *user story* is defined by the *rules* that will be implemented by its delivery. *Rule* is a synonym for *business requirement* and *acceptance criterion.*

Each rule is illustrated by several *concrete examples* that ensure there is no ambiguity in the rule's interpretation. Each example describes a single concrete instance of a rule being applied by documenting the expected outcome that should result from a specific action taking place in a given context.

Examples should be captured in any format that is relevant and concise (such as bullet lists, wire frames, flow diagrams or truth tables). In this book, we will show you how to formulate concrete examples into business readable documentation. Once automated, this becomes *living documentation* that can be relied on to accurately describe the actual behaviour of the system.

Together, the rules and examples specify the expected behaviour of the system.

BDD needs skilled testers

Testers continue to play a critical role in teams that adopt a behaviour-driven way of working. Still, we often hear about organizations that seem to believe that test automation reduces the need for testers. We feel that it's important to reiterate that this is not true. There may come a time when artificial intelligence will be able to automate all aspects of software development, but for the time-being we continue to rely on skilled, human professionals at all stages of specification and delivery.

Despite the huge benefit of test automation, organizations should do everything they can to retain skilled testers, especially those with extensive domain knowledge, because:

[6] https://cucumber.io/blog/bdd/example-mapping-introduction/

- a tester's experience and perspective is essential during discovery;
- there are a wide range of specialist test techniques that are valuable throughout the development lifecycle, and;
- exploratory testing requires deep knowledge of testing and the problem domain.

It undoubtedly makes sense to offer testers training in development skills, in the same way that it makes sense to offer developers training in testing skills. However, the organization should recognize that domain knowledge is too valuable to squander. Offers of cross-skilling should be voluntary, both in theory and in practice.

Why you should listen to us

Gáspár is the creator of SpecFlow, the most widely used BDD framework for .NET.

He is an independent coach, trainer and test automation expert focusing on helping teams implementing BDD and SpecFlow through his company, Spec Solutions. He has more than 20 years of experience in enterprise software development as he worked as an architect and agile developer coach.

He shares useful BDD and test automation related tips on his blog[7] and on Twitter (@gasparnagy). He edits a monthly newsletter[8] about interesting articles, videos and news related to BDD, SpecFlow and Cucumber.

He also works on an open-source Visual Studio extension for SpecFlow, called Deveroom[9] and on a tool that can synchronize scenarios to Azure DevOps, called SpecSync[10].

Seb has been a consultant, coach, designer, analyst and developer for over 40 years. He has been involved in the full development lifecycle with experience that ranges from architecture to support, from BASIC to Ruby.

During his career, he has worked for companies large (e.g. IBM, Amazon) and small, and has extensive experience of failed projects. He's now Continuous Improvement Lead with SmartBear, helping apply the lessons he has learned to internal development practices and product roadmaps.

[7] http://gasparnagy.com
[8] http://bddaddict.com
[9] https://github.com/specsolutions/deveroom-visualstudio
[10] https://www.specsolutions.eu/services/specsync/

He's a regular speaker at conferences, a contributing author to *97 Things Every Programmer Should Know* (O'Reilly) and the lead author of *The Cucumber for Java Book* (Pragmatic Programmers).

He blogs at cucumber.io[11] and tweets as @sebrose.

Together Seb and Gáspár have over 60 years of software experience which they put to good use developing and delivering training and coaching for organizations worldwide. If you would like to talk about the services they can provide, please get in touch at services@bddbooks.com.

Online resources

- BDD Books series: http://bddbooks.com
- Resources for this book: http://bddbooks.com/resources/formulation
- Figures from the book: http://bddbooks.com/resources/formulation/figures
- WIMP project files (feature files): https://github.com/bddbooks/bddbooks-formulation-wimp

Acknowledgments

This book would not have been possible without the help of: Gojko Adzic, Emily Bache, Abby Bangser, Lisa Crispin, Gary Fleming, Markus Gärtner, Janet Gregory, John Ferguson Smart, Aslak Hellesøy, Claude Hanhart, Kevlin Henney, Angie Jones, Heidi Kinsey, Liz Keogh, Ailsa Laing, Cyrille Martaire, Rob McBryde, Ken Pugh, Tom Roden, Johanna Rothman, Daniel Terhorst-North, Joe Wright, and Matt Wynne.

Seb Rose and Gáspár Nagy, 2021

[11] https://cucumber.io/blog

Chapter 1 – What is formulation?

Formulation is the process of turning a shared understanding of how a system should behave into a business readable specification. On one level this is a trivial activity, since formulation usually produces specifications that are very close to natural language. However, like BDD itself, formulation, while simple, is not easy.

In this chapter, we briefly recap on where formulation fits into BDD before diving into the details in the following chapters.

1.1 – Where does formulation fit into BDD?

BDD is an approach to software development that emphasizes the collaborative aspect of software development by linking requirements, documentation, and tests. It is made up of three practices, shown in Figure 1, which illustrates the order that the BDD practices should be applied to *each small increment* of functionality in your application. So, by the time your team is formulating scenarios for a *user story*, that story should already have been analyzed collaboratively (in discovery). It's the concrete examples generated during discovery that form the raw material that will be formulated into business readable scenarios.

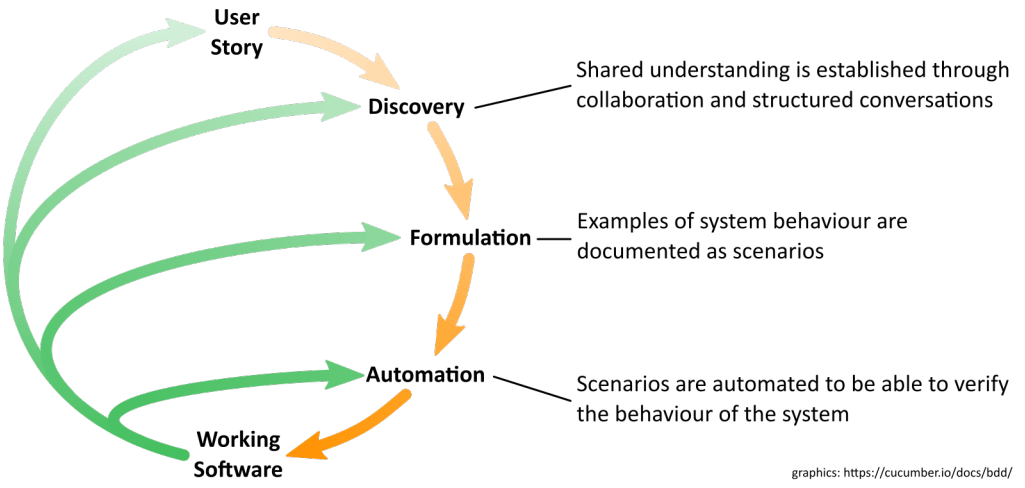


Figure 1 – BDD practices

A more detailed description of the full BDD process has been published online[12].

1.2 – Shared understanding

The primary purpose of formulation is to document a shared understanding of the system's expected behaviour, using unambiguous terms rooted in the business domain. For this reason, formulation needs to be a collaborative activity that requires input from representatives from business and delivery. The shape of the collaboration varies between organizations (and sometimes between teams), but without it you can have no confidence that the language used will convey the same meaning to all stakeholders.

The business language used should be appropriate to the domain being documented. If you are developing a pizza delivery application, then you will probably use words such as *"Order"*, *"Delivery address"*, *"Customer"*. On the other hand, if you are creating a library that performs complex mathematical transformations, you should use terms from the mathematical domain, such as *"Matrix"*, *"Transpose"* and *"Normalization"*.

[12] http://bddbooks.com/articles/bdd-tasks-and-activities.html

Domain Driven Design

If different members of the team use different terms to describe the same concepts, they will have to pay the price of translating whenever they collaborate. They also open themselves to the risk of a faulty translation, which can lead to errors.

Eric Evans wrote the influential book, *Domain Driven Design* [Evans2004], in which he introduced the idea of a *Ubiquitous Language*: a language structured around the business domain and used by all team members to connect all the activities of the team with the software. Although BDD and DDD are very different, they both emphasize the value of a shared language, rooted in the terminology of the business, used throughout the development process.

1.3 – Two types of scenarios

In *Discovery* [Nagy2018] we explained that each concrete example captured during discovery should focus on illustrating a single rule (a.k.a. requirement, acceptance criteria). The example may then be formulated into a business readable scenario in order to specify, document and test that system behaviour. It naturally follows that scenarios created from a concrete example will themselves be very granular, because they focus on illustrating a single rule. We call these *illustrative scenarios*.

The majority of scenarios will be illustrative scenarios because they are ideal for communicating the detailed behaviour of a system. However, they don't give a broad, end-to-end view of how the business processes work. For this we need longer scenarios that describe a complete user journey, which we call *journey scenarios*. Journey scenarios should be the tiny minority of your scenarios – there should be just enough of them to ensure that a reader can understand how the system delivers useful outcomes. We describe journey scenarios in Section 5.6, "Journey scenarios".

1.4 – Many formats

Some organizations have used formalized formats in their requirement documents for many years. Some of these formats can be thought of as formulated scenarios,

but they typically fall short of our needs, because they have not been designed with automation in mind.

The BDD approach provides the insight that formulated scenarios have two distinct benefits:

1. Document a shared understanding of the system's behaviour;
2. Automatically verify that the system behaves as expected.

For a software tool to automatically verify the system's behaviour using scenarios, they must be formulated in a way that makes them machine readable.

There are several widely used formats that are understood by software automation tools. One of the earliest is FIT[13], created by Ward Cunningham, which allows users to formulate scenarios as tables. A Wiki-based user interface was grafted onto it by Robert C. Martin to create FitNesse[14]. These tools are primarily driven by tabular specifications.

The Robot Framework[15] supports hierarchical, composable keywords to allow the writing of business readable test cases.

Additionally, there are teams that use developer level tools (RSpec[16], Jasmine[17], Cypress[18]) which embed the natural language specifications within the code. Since the expectations have to be written as source code, it is harder for business representatives to contribute to its creation and maintenance, but it can produce business readable documentation as part of the test execution result.

The Cucumber family of tools understand Gherkin, a structured natural language syntax based upon the Given/When/Then format pioneered by Chris Matts. This semantic frame is useful even if your team does no automation. It has even been useful for organizations that do no software development at all. Meanwhile, organizations that choose to adopt all three practices of BDD find that it's easy to automate their business readable Gherkin *feature files*. The loose coupling between

[13] http://fit.c2.com/
[14] http://www.fitnesse.org/
[15] https://robotframework.org/
[16] http://rspec.info/
[17] https://jasmine.github.io/
[18] https://www.cypress.io/

the feature files and the underlying automation code (in *step definition* files) provides a way for documentation that can be used to test the system's behaviour.

This book deals exclusively with Gherkin, although many of the principles described here are applicable to other tools as well.

1.5 – Gherkin overview

Gherkin is an extremely simple syntax to learn and requires absolutely no knowledge of programming. You'll be introduced to most of the Gherkin keywords as you read through this book and there is excellent online documentation[19]. One of the features that makes Gherkin attractive internationally is its extensible support for multiple spoken languages, including those that utilize non-Latin alphabets (such as Greek, Chinese and Russian).

Gáspár's story: The best part of Cucumber

Somebody asked how the Cucumber project became so popular. For me, the answer is clear: the developers realized that in order to make this style of BDD successful, the Gherkin language should be a standalone tool, and not the proprietary language of the Ruby-based Cucumber tool itself.

This attitude has helped many other open-source projects to provide BDD support – for almost every platform and programming language – using a common specification format!

The Gherkin language is simple and stable: it has had only one major change in the last 10 years. I think we can say that, nowadays, Gherkin is the de-facto BDD specification language.

Gherkin is very stable, with a very small set of keywords. A new keyword ("Rule") was introduced as a backwards compatible change in Gherkin v6. At time of writing, not all Cucumber implementations support the "Rule" keyword.

[19] https://cucumber.io/docs/gherkin/reference/

Although we use the new *Rule* keyword in this book, we will also provide alternatives that will work with earlier versions of Gherkin online (see *Online resources* in the Preface).

1.6 – Living documentation

Writing software documentation is a thankless task. It is hard to write, and as soon as you do write it, the software changes and you have to re-write it. This may be why one internet wit came up with the following saying: "If software is useful, you will have to change it. If it is useless, you will have to document it."

Seb's story: A thought experiment

I often ask my teams, "Which should you prefer, incorrect documentation or no documentation?". What's your own answer?

Now ask yourself, "Which have we got?".

Most teams understand the cost of incorrect documentation and yet that's exactly what most of them have. They're in a situation where trusting the documentation is very risky, so they regularly refer back to the source code and the running system. In which case, what's the value of your documentation?

If you're not confident that your documentation is correct … delete it!

One of the major benefits of adopting all three BDD practices (discovery, formulation, automation) is that your organization will benefit from having *living documentation*. By automating the scenarios that make up your documentation, your team will be alerted whenever the implementation and documentation diverge.

A divergence can happen for one of three reasons:

1. The documentation describes behaviour that has not yet been implemented yet.
2. There is a defect in the code.
3. The documentation is out of date.

Being proactively informed of a divergence means that you can keep your documentation in sync with your system. Your teams and your customers can be confident that the documentation is always correct.

1.7 – What we just learned

In this chapter we have introduced the term formulation, the main topic of our book, which is the process of turning a shared understanding of how the system should behave into a business readable specification.

Formulation is the middle of the three BDD practices: discovery, formulation and automation. During discovery, we collect concrete examples to help us understand the requirements. These examples can then be formulated into scenarios that can be automated later.

The exact format of the formulated scenarios depends on the automation tool that you plan to use. There are many BDD automation tools available, but in this book we focus on the Cucumber family of tools, which use the Gherkin format. With the support of these tools, any documentation that has been written using the Gherkin format can be automated – on nearly any platform, using almost any programming language.

The formulated scenarios should be considered as part of the documentation of the system. When they are automated, by running them as tests, they can tell you when the implementation diverges from your expectations – so it becomes living documentation that your whole team can rely on.

Chapter 2 – Cleaning up an old scenario

Gherkin is used by thousands of organizations all over the world, and it is common to find long, complex and unreadable scenarios in their projects. Unfortunately, long, complex and unreadable scenarios do nothing to promote shared understanding across your organization and lead to constant maintenance efforts for your team. Cleaning up such a scenario not only alleviates these problems, but also provides a good opportunity to learn more about your domain.

In this chapter you're going to learn the basics of formulation by following the WIMP team as they take a poorly formulated scenario and make it better. We look in more depth at applying BDD to legacy projects in Chapter 6, "Coping with legacy".

2.1 – The old scenario

The Where Is My Pizza (WIMP) application lets customers place orders that they will collect from the restaurant (customer-collection). The customer can also choose to pay for the order when they collect it (pay-on-collection). There have been some problems with orders that are never collected or paid for. Patricia, the product owner (PO), has been asked to deliver a feature intended to reduce this problem, so she's trying to understand the existing implementation of the customer-collection and pay-on-collection features.

In preparation for a requirement workshop Patricia, Tracey and Dishita sit down to look at the existing scenarios that illustrate the customer-collection order process. They only find one scenario (see Listing 1 below), which was written when the project used scenarios for testing, rather than as a way of supporting collaboration and BDD.

WIMP team members

To make it easier to follow, their initials describe their role:

- **D**ave – developer
- **D**ishita – developer
- **I**an – intern
- **P**atricia – product owner
- **T**racey – tester
- **U**lisses – user experience

Patricia projects the scenario for everyone to read.

Listing 1 – The old scenario

```
Scenario: Order Test
  Given the time is "11:00"
  Given a customer goes to "http://test.wimp.com/"
  And they fill in "Margherita" for "SearchText"
  When they press "Search"
  Then they should see "Margherita" within "SearchResults"
  And they select "Medium" from "Size"
  When they press "Add to basket"
  Then they should see "1 item" in "BasketItemCount"
  When they click "Checkout"
  And they select "Collect" from "DeliveryInstructions"
  And they select "Pay on Collection" from "PaymentOption"
  And they fill in "Marvin" for "OrderName"
  And they fill in "12334456" for "ContactPhoneNumber"
  When they press "Submit order"
  Then they should see "SuccessMessage"
  Then they should not see "ErrorMessage"
  And they should see "Thank you for your order!" within "SuccessMessage"
  And they should see "11:20" within "CollectionTime"
  And they should see "$14" within "TotalAmount"
```

Gherkin basics

Gherkin allows organizations to write business readable specifications that can also be used as automated tests. It is written in *feature files*, which are plain text files with the `.feature` extension.

Each feature file contains one or more *scenarios*. Each scenario is made up of one or more *steps*. Each step starts with one of five keywords: *Given*, *When*, *Then*, *And*, *But*

Given, *When*, *Then* introduce respectively the context, action, and outcome section of a scenario. (See *Discovery* [Nagy2018, Chapter 3]). *And* and *But* are conjunctions that continue the preceding section. We cover the majority of Gherkin syntax in Section 3.1, "Feature files" and Section 3.3, "Gherkin basics". See *Gherkin jump-list* in the Appendices for full details.

"I don't think I've ever seen this scenario before," says Patricia. "It's not as easy to read as the ones we usually write."

"You are right," replies Tracey, "It wasn't written to illustrate a specific *rule*. I wrote this as an automated regression test. There are quite a lot like this that we've been wanting to rewrite."

"It's really long and involves lots of different *rules*," says Dishita. "That may be why it's so hard to maintain. Last time this scenario failed we spent ages trying to work out what went wrong."

"Shall we try to see if we can improve it, then?" asks Patricia. "Maybe we should try to apply the BRIEF principles to it."

2.2 – Keep your scenarios BRIEF

Over the years that Gherkin has been in use, an approach to writing scenarios has evolved. Because Gherkin is very close to natural language it's very easy to learn, but just like writing reports or stories, it takes practice to do it well. There are three main goals that we try to keep in mind when writing scenarios, which give rise to six principles encapsulated by the BRIEF acronym.

The goals

Scenarios should be thought of as documentation, rather than tests. We write scenarios to illustrate and clarify the expected behaviour of the system. The goal is to be descriptive, not exhaustive.

Scenarios should enable collaboration between business and delivery, not prevent it. Scenarios should be written so that they can be understood by everyone that contributes to the creation and evolution of the system.

Scenarios should support the evolution of the product, rather than obstruct it. Scenarios that illustrate a specific behaviour should not need to be changed when unrelated behaviour changes.

The principles

The following six principles work together to support the goals described above. To make them easier to remember, we've arranged it so that the first letter of each principle makes up an acronym, *BRIEF*, which is itself the sixth principle.

B	**Business language**	Business terminology aids cross-discipline collaboration
R	**Real data**	Using actual data helps reveal assumptions and edge cases
I	**Intention revealing**	Describe the desired outcomes, rather than the mechanism of how they are achieved
E	**Essential**	Omit any information that doesn't directly illustrate behaviour
F	**Focused**	Each scenario should only illustrate a single rule
	Brief	Shorter scenarios are easier to read, understand and maintain

Business language: The words used in a scenario should be drawn from the business domain. Software systems are to deliver business value, so it is business terminology that must be understood by all involved in delivering the system. Therefore, we should use the language of the business to enable collaboration and ensure alignment.

Real data: In *Discovery* [Nagy2018, Section 3.1], we explained that examples should

use concrete, real data. This helps expose boundary conditions (a.k.a. edge cases) and underlying assumptions early in the development process. When writing scenarios, we should also use real data whenever this helps *reveal intention.*

Intention revealing: Scenarios should reveal the *intent* of what the actors in the scenario are trying to achieve, rather than describing the *mechanics* of how they will achieve it. We should start by giving the scenario an intention revealing name, and then follow through by ensuring that every line in the scenario describes *intent, not mechanics.*

Essential: The purpose of a scenario is to illustrate how a rule should behave. Any parts of a scenario that don't directly contribute to this purpose are *incidental* and should be removed. If they are important to the system, they will be covered in other scenarios that illustrate other rules. Additionally, any scenarios that do not add to the reader's understanding of the expected behaviour have no place in the documentation.

Focused: Most scenarios should be focused on illustrating a *single rule.* It's easier to achieve this if you derive your scenarios from examples captured during a *requirement workshop.*

Last, but not least, scenarios should be brief, as well as BRIEF.

Brief : We suggest you try to restrict most of your scenarios to five steps or fewer. This makes them easier to read and much easier to reason about.

In the rest of this chapter, you'll see the team apply these principles. In subsequent chapters, we'll go into more detail and see BRIEF applied to the writing of new scenarios. We'll indicate which of the BRIEF principles the team applies in each section by listing them after this symbol:

Business language, **E**ssential

Seb's story: Acronyms

Gáspár and I have been explaining the concepts behind BRIEF for many years, but

> we never managed to create a memorable acronym. At one of our "Writing Better BDD Scenarios" workshops at the European Testing Conference in 2018 we had Gojko Adzic in the audience. He seemed to enjoy the workshop, but spent quite a bit of time scribbling on a piece of paper. At the end of the workshop he came and told us that he had tried (but failed) to turn our bullet points into an acronym. "It worked for the INVEST acronym[a] by Bill Wake."
>
> We took his advice, and BRIEF is the result. Thanks for the encouragement, Gojko!
>
> [a] https://en.wikipedia.org/wiki/INVEST_(mnemonic)

2.3 – Using example maps to provide focus

Business language, **I**ntention revealing, **F**ocused

The team take another look at the old scenario (Listing 1).

> "The name of this scenario is *'Order Test'*, which doesn't really tell us anything about its intentions," says Patricia. "What shall we call it?"
>
> "It's hard to say," says Dishita, "Because it wasn't created to illustrate a specific rule. It should probably be split into several scenarios."
>
> "I can't even remember what the exact requirements were for customer collections," says Tracey. "Maybe we should do a quick example map to make sure we understand how the system works right now."

Seb's story: Lift and shift

I was helping a team who had been tasked with reimplementing an existing application on a new platform. The Product Owner thought that it was sufficient to tell the team "just make it do what the old system did." When he agreed to attend an example mapping workshop, he soon realized that not only did he not understand what the old system actually did, but he also didn't want to replicate some of its behaviour in the new system.

In fact, working with code where the requirements aren't clear is one of the hardest jobs in software, because you're never sure *why* a piece of logic exists or *who* depends on it. So, if you ever hear someone say that your next piece of work is "*just* a lift and shift" ensure that time is allocated to discover the existing system's current behaviour.

The team spend some time refreshing their understanding of customer-collection orders and end up with an example map (Figure 2).

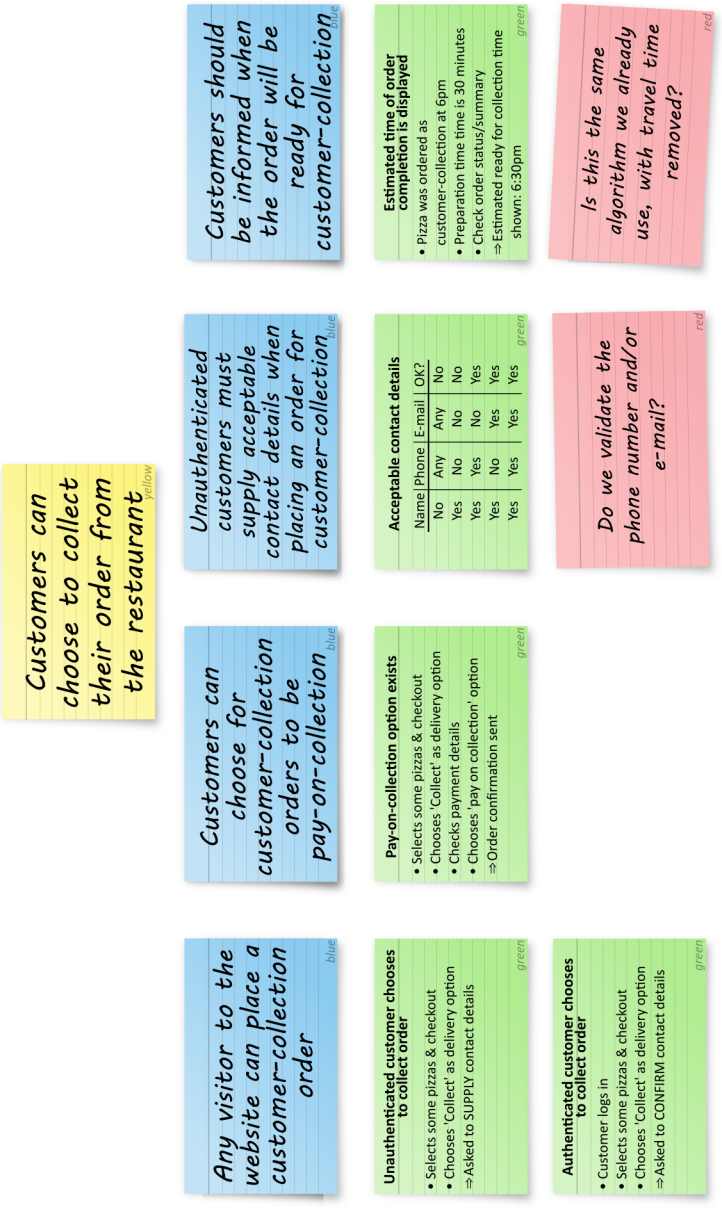


Figure 2 – Customer-collection example map

Creating the example map took them about 20 minutes, but they felt it to be a valuable investment. Now they have the map on the desk in front of them, they can turn back to the scenario projected on the wall and continue with their analysis (see Listing 1).

> "It looks like the important bits of this scenario are that the customer will *collect* the order and that they will *pay on collection*," says Patricia.
>
> "They're submitting their name and phone number, which means that they can't be logged in," says Tracey.
>
> "So, we could call it, *'Not logged in customer chooses to pay on collection'*. That seems to reveal the *intention* of this scenario," suggests Dishita.
>
> "Yes, but no one would talk about a, 'not-logged-in customer'. They're more likely to talk about an 'authenticated' or 'unauthenticated' customer," says Patricia.
>
> "That's true," agrees Dishita, "But authentication is a technical term. Shouldn't we avoid those?"
>
> "Authentication is a technical concept, but it's a term that is familiar to our business colleagues," says Tracey.
>
> "I agree," says Patricia. "Let's talk about authenticated and unauthenticated customers in our scenarios and see what feedback we get."

Listing 2 – An intention revealing name

```
1 ~~Scenario: Order Test~~
2 Scenario: Unauthenticated customer chooses to pay on collection
```

> "Now," she continues, "If we're focusing on the pay-on-collection rule, which steps in the scenario are still essential?"

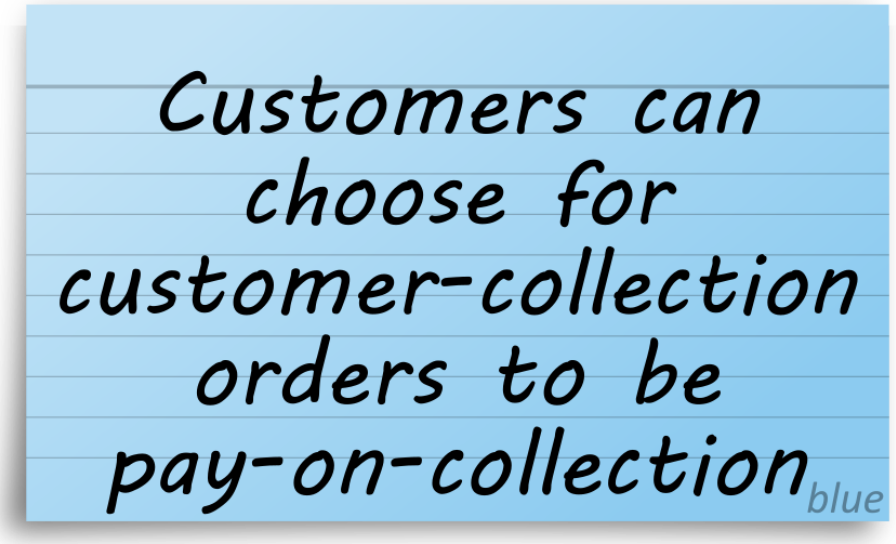


Figure 3 – Pay-on-collection rule

"Let's start by thinking about what the expected outcome is," says Tracey.

"The expected outcome is that the user should be able to choose to pay on collection. The rest is just the normal order finalization," Patricia replies.

"Right! But what did the customer do to be able to see this option?" asks Tracey.

"They've put some food in their basket, gone to checkout, and chosen to collect the order themselves," says Dishita.

"Sounds good," says Patricia. She edits the scenario, removing order finalization steps from the end of the scenario and turning the last step into a *Then*:

Listing 3 – First edit

```
Scenario: Unauthenticated customer chooses to pay on collection
  Given the time is "11:00"
  Given the customer goes to "http://test.wimp.com/"
  And they fill in "Margherita" for "SearchText"
  When they press "Search"
  Then they should see "Margherita" within "SearchResults"
  And they select "Medium" from "Size"
  When they press "Add to basket"
  Then they should see "1 item" in "BasketItemCount"
  When they click "Checkout"
  And they select "Collect" from "DeliveryInstructions"
- And they select "Pay on Collection" from "PaymentOption"
  Then "Pay on Collection" should be available from "PaymentOption"
- And they fill in "Marvin" for "OrderName"
- And they fill in "12334456" for "ContactPhoneNumber"
- When they press "Submit order"
- Then they should see "SuccessMessage"
- Then they should not see "ErrorMessage"
- And they should see "Than you for your order!" within "SuccessMessage"
- And they should see "11:20" within "CollectionTime"
- And they should see "$14" within "TotalAmount"
```

What the team just did

In this short conversation, the team:

- Gave the scenario an intention revealing, business readable name,
- Focused on a single rule,
- Started by identifying the expected outcome, and
- Deleted all outcomes that don't directly illustrate the rule.

2.4 – Document the essence of the behaviour

Essential, Brief

"It's still a bit long, isn't it? The whole bit about searching for a pizza and adding it to the basket doesn't help illustrate the pay-on-collection rule. I think we can remove all of that," says Dishita.

"Yes. There's lots here that isn't *essential*," agrees Patricia.

"The application could live on any URL, so there's no need to specify it in this documentation," Dishita continues.

"The time of day isn't important either. Well, not for this rule anyway," says Tracey.

"All that detail would probably cause confusion rather than illustrate anything," says Dishita.

Patricia edits the scenario again, leaving:

Listing 4 – Second edit

```
Scenario: Unauthenticated customer chooses to pay on collection
  Given the time is "11:00"
  Given the customer goes to "http://test.wimp.com/"
  And they fill in "Margherita" for "SearchText"
  When they press "Search"
  Then they should see "Margherita" within "SearchResults"
  And they select "Medium" from "Size"
  When they press "Add to basket"
  Then they should see "1 item" in "BasketItemCount"
  Given the customer has "1 item" in "BasketItemCount"
  When they click "Checkout"
  And they select "Collect" from "DeliveryInstructions"
  Then "Pay on Collection" should be available from "PaymentOption"
```

"That's certainly *brief*! But it's going to break the existing automation," says Tracey.

"You're right," says Dishita. "But for now, let's focus on ensuring the scenarios document the essence of the expected behaviour. I'll work with you to fix the automation once we're all happy with the scenarios."

What the team just did

In this section, the team:

- Removed incidental details from the scenario,
- Ensured that the scenario is brief enough to easily get feedback.

2.5 – Scenarios should read like a specification

Business language, **I**ntention revealing

> "This is looking much better, but the scenario still isn't really written in 'Business language'," says Patricia. "What is `BasketItemCount` anyway?"
>
> "It's the name of the HTML element that displays the number of items the customer has in their basket. It is an implementation detail," says Dishita.
>
> "The *name* of the HTML element is an implementation detail, but there's an underlying business concept, which is the number of items in the basket," says Tracey.
>
> "Yes. The same is true about `DeliveryInstructions` and `PaymentOption`," says Dishita.
>
> "Using the word 'click' also implies that the UI implementation is a link. That's specifying the *mechanics* of how the system should be implemented rather than revealing the *intent* of what it's expected to do," adds Tracey.

Patricia passes the keyboard to Dishita, who fixes the scenario:

Listing 5 – Third edit

```
Scenario: Unauthenticated customer chooses to pay on collection
  ~~Given the customer has "1 item" in "BasketItemCount"~~
    Given the customer has 1 item in their basket
  ~~When they click "Checkout"~~
  ~~And they select "Collect" from "DeliveryInstructions"~~
    And they have chosen to collect their order
    When they proceed to payment instructions
  ~~Then "Pay on Collection" should be available from "PaymentOption"~~
    Then they should be able to choose to pay on collection
```

"That's really succinct. It doesn't depend on the UI, or even on the detail of the checkout flow. It specifies *intent* using *business language* and (mainly) *focuses* on the pay-on-collection rule," says Patricia. "There's just one more thing that worries me …"

What the team just did

This time the team has:

- Replaced technical terms (from the *solution domain*) with *business language* from the *problem domain*,
- Replaced mechanistic terms (such as "click") with intention revealing descriptions.

2.6 – Use real data when it provides clarity

Real data, **E**ssential

"… why does the scenario say there is '1' item in the basket? I don't think it's relevant. The behaviour would be the same no matter how many items are in the basket."

"Yes, we could replace '1' with 'some': *'Given the customer has some items in their basket'*. But wouldn't that contradict the *real data* part of *BRIEF*?" asks Dishita.

"Since the behaviour is the same no matter how many items are in the basket, the actual number doesn't help *reveal intent*," says Patricia. "What would help reveal intent?"

"A customer can't checkout with an empty basket," says Tracey. "Let's just say that the basket is not empty."

Patricia and Dishita agree, so Tracey makes a small change to the scenario:

Listing 6 – Fourth edit

```
Scenario: Unauthenticated customer chooses to pay on collection
  ~~Given the customer has "1 item" in "BasketItemCount"~~
  Given the customer's basket is not empty
  And they have chosen to collect their order
  When they proceed to payment instructions
  Then they should be able to choose to pay on collection
```

"Do we even need to mention that the basket is not empty?" asks Dishita. "You can't get to the *'payment instructions'* step with an empty basket anyway."

"So, by saying that the customer is at the *'payment instructions'* step, it's *implicit* that the basket is not empty," says Tracey.

"You're right – it doesn't help illustrate the pay-on-collection rule. And there are other scenarios that illustrate the rules that control the checkout flow," says Patricia.

Seb's story: Real data as a distraction

The *real data* principle encourages us to use actual data, because that "helps reveal assumptions and edge cases". However, using actual data such as the type of pizza being bought or the cost of the basket can also be distracting.

I was working with a team that was processing geo-spatial coordinates. Most of their scenarios used the same set of coordinates because the team had got used to them. When a new team member wrote a scenario that used a different set of coordinates, the product owner got confused. He wanted to understand why the new set of coordinates resulted in a different behaviour. In fact, the change in behaviour had nothing to do with the coordinates – they were incidental information – but the fact that all the scenarios had this level of detail had led him to believe that the coordinates were important.

It's generally a good idea to include actual data in a scenario when you first write it. Once the scenario is written, ask yourself whether the behaviour would be different if the actual data used changed. If not, then replace the data with a more abstract, descriptive term.

A scenario should clarify a rule, not distract a reader!

Tracey deletes the first line of the scenario:

Listing 7 – Fifth edit

```
Scenario: Unauthenticated customer chooses to pay on collection
  Given the customer's basket is not empty
  Given the customer has chosen to collect their order
  When they proceed to payment instructions
  Then they should be able to choose to pay on collection
```

"That's looking good," says Patricia and they all agree.

What the team just did

Now the team has:

- Identified inessential *real data*,
- Deleted an implicit statement.

2.7 – Communication, not testing

Intention revealing, **F**ocused

"This scenario isn't just for unauthenticated customers any more," says Tracey. "The documented flow is the same for customers regardless of whether they're authenticated or not."

"You're right," agrees Patricia and she edits the scenario name.

Listing 8 – Focus on communication

```
1 ~~Scenario: Unauthenticated customer chooses to pay on collection~~
2 Scenario: Customer chooses to pay on collection
```

"It may be more accurate, but now we can't test that the system behaves the same way irrespective of whether the customer is authenticated or not," says Tracey.

"A scenario's main purpose is to document behaviour, not to test it," says Dishita, "So Patricia should decide on that basis."

"I agree with Dishita," says Patricia, "This does document the behaviour. Since we don't specify whether the customer is authenticated or not, it's implicit that the behaviour is the same for both."

> "So, we'll say 'authenticated customer' or 'unauthenticated customer' when the behaviour varies, but just 'customer' when their authenticated status is not relevant to the system's behaviour?" asks Dishita.
>
> "That sounds fine to me," confirms Patricia.
>
> "The automation code will still need to choose to use an authenticated or unauthenticated customer," says Tracey.
>
> "That's true," says Dishita. "There are a few ways that we could handle this, but we don't need to make that decision now."
>
> "I'm happy with the scenario as it is," says Patricia. "If we need to refine it based on feedback, then we can do that."

Now that the team has a better understanding of the customer-collection feature, they can begin to consider the requirements for the new feature that will manage the risk of unpaid customer-collection orders.

What the team just did

The team has:

- Renamed the scenario to ensure that it accurately describes its intention,
- Ensured focus on the rule being illustrated.

2.8 – Illustrative scenarios

It's important to remember that the primary purpose for writing scenarios in natural language is to allow all stakeholders to collaborate, irrespective of their technical background. That tools like Cucumber and SpecFlow allow us to turn these into automated tests is a valuable *side effect*, not the primary motivation. That's why we should always concentrate on writing scenarios that are understandable by all stakeholders.

However, it's not enough to write understandable scenarios, we also need to make it easy to understand *why* each scenario is important. We have previously explained that each concrete example should illustrate a single rule. Each of the scenarios that we write should be based on a single concrete example, so we call them *illustrative scenarios*.

We use several heuristics when writing illustrative scenarios. The aim is to write short, easy-to-read, easy-to-understand scenarios. Every scenario will be reviewed by several people, so brevity is not just a nice-to-have property. Long scenarios make reviewing hard and reviewers unhappy.

Keeping scenarios short requires effort identifying business domain abstractions, as described by Daniel Terhorst-North[20]. This effort will be rewarded with improved maintainability. Focusing on domain abstractions will decouple your scenarios from implementation details and reduce the impact in the face of changing user interface or customer workflow.

Gáspár's story: Learning with usable outcomes

I have often participated in meetings that are similar to the ones the WIMP team just had. I find these meetings valuable, but from time to time I hear people saying: "Is it really worth spending *so much time* on a single scenario?"

As an answer I explain that the goal here was not to fix the scenario, but to learn about the problem. The scenario helped us to guide the learning process and, as a bonus, has been transformed into a useful scenario that we can use as it is.

If you spend enough time making *that single scenario* good, writing the subsequent scenarios about the same topic will be surprisingly fast. This is an aspect of *deliberate discovery* (as written about by Daniel Terhorst-North[a] and Liz Keogh[b]) – learning is the constraint, so we should prioritize learning.

So, yes it is worth it.

[a] https://dannorth.net/2010/08/30/introducing-deliberate-discovery/
[b] https://lizkeogh.com/2012/06/01/bdd-in-the-large/

[20] https://dannorth.net/2011/01/31/whose-domain-is-it-anyway/

2.9 – What we just learned

In order to improve their understanding and provide a solid starting point for developing a new feature, the WIMP team has decided to clean up an old, poorly formulated scenario.

By following their discussion you saw how the six BRIEF principles of good scenarios can be applied. First, they had to analyze the rules of the feature to reveal its real intention. This helped them to see which parts of the scenario were essential and which were incidental (and could be removed).

As a result of these changes, the scenario became much simpler, but it was still using technical solution details to describe the different steps. Although everyone in the meeting could make a good guess what those technical details meant, they spent time finding the underlying business concept for each of them. With that understanding, they rewrote the scenario using business language, making the scenario easier to read and less dependent on solution details that might change. Using business language is certainly useful for recording a shared understanding, but the process of discussing the language has another benefit: the terminology is refined, making it easier to discuss the upcoming feature request.

Rephrasing a scenario that has already been automated might cause additional work, because the automation logic needs to be adapted accordingly. This is an effort that pays off quickly though, so the team kept focusing on the communication aspect of the scenario, rather than trying to minimize the automation effort.

With all these changes, they finally produced a scenario that is not only BRIEF and brief, but also a good illustration of a business requirement.

Chapter 3 – Our first feature

In the last chapter we explained the basics of writing better scenarios by watching the team clean up an old, poorly written scenario. In this chapter we're going to review some of the other scenarios that they wrote as part of that process and learn more about the structure of Gherkin.

3.1 – Feature files

Gherkin requires you to organize your scenarios into feature files. There's no limit to how large (or small) your feature files are, but you should always remember that your goal is well structured, readable documentation.

Each feature file should focus on a single area of functionality in your product. As features get more complicated, you may find it useful to spread the documentation over several feature files. In Chapter 5, "Organizing the documentation", we talk in depth about how to structure documentation as it grows. For this chapter, we're going to limit discussion to a single feature file.

The feature file that the team has just written describes how the system should enable a customer to place an order for customer-collection. The scenarios in this feature file are *cohesive* – they all illustrate related behaviour of the software.

3.2 – A sample feature file

The team has written scenarios for the example map that they built in the last chapter (see Chapter 2, Figure 2) and replaced the old scenario completely. The scenarios have been collected into a *feature file*, which is reproduced in Listing 9 and is also available at https://github.com/bddbooks/bddbooks-formulation-wimp/tree/main/ch3-beginning.

Have a look through the feature file and the example map. Can you see how they relate to each other? Do the scenarios conform to the BRIEF acronym we introduced in the previous chapter?

There are several elements that you may not recognize yet – they will all be explained in this chapter.

Listing 9 – Customer-collection feature file (`customer_collection.feature`)

```
Feature: Customers can collect their orders

Customers can choose to collect their order from the restaurant:
- whether they are authenticated or not
- whether they pay on order or pay on collection

Rule: Any visitor to the website can place a customer-collection order

  Scenario: Unauthenticated customer chooses to collect order
    Given the customer is unauthenticated
    When they choose to collect their order
    Then they should be asked to supply contact details

  Scenario: Authenticated customer chooses to collect order
    Given the customer is authenticated
    When they choose to collect their order
    Then they should be asked to confirm contact details

Rule: Customers can choose for customer-collection orders to be
                                                          pay-on-collection

  Scenario: Customer chooses to pay on collection
    Given the customer has chosen to collect their order
    When they choose to pay on collection
    Then they should be provided with an order confirmation

Rule: Unauthenticated customers must supply acceptable contact details
                             when placing an order for customer-collection

  Scenario Outline: Contact details supplied ARE acceptable
    Given the customer is unauthenticated
    And they've chosen to collect their order
```

```
33      When they provide <acceptable contact details>
34      Then they should be asked to supply payment details
35
36      Examples:
37        | description           | acceptable contact details |
38        | Name and Phone        | Name, Phone                |
39        | Name and Email        | Name, Email                |
40        | Everything provided   | Name, Phone, Email         |
41
42    Scenario Outline: Contact details supplied are NOT acceptable
43      Given the customer is unauthenticated
44      And they've chosen to collect their order
45      When they provide <unacceptable contact details>
46      Then they should be prevented from progressing
47      And they should be informed of what made the contact details
48                                                                  unacceptable
49
50      Examples:
51        | description | unacceptable contact details |
52        | No Name     | Phone, Email                 |
53        | Only Name   | Name                         |
54        | Only Email  | Email                        |
55        | Only Phone  | Phone                        |
56
57  Rule: Customer should be informed when their order will be ready for
58                                                        customer-collection
59
60    Scenario: Estimated time of order completion is displayed
61      Given the customer placed an order
62        | time of order | preparation time |
63        | 18:00         | 0:30             |
64      When they choose to collect their order
65      Then the estimated time of order completion should be displayed as
66                                                                      18:30
```

3.3 – Gherkin basics

Gherkin is an extremely simple language. Its motivation is to create documentation that reads like natural language, but is structured enough to enable automation tools (such as Cucumber and SpecFlow) to process it.

Keywords

Gherkin keywords can be divided into two categories – those that *must* be followed by a colon (*block keywords*), and the others (*step keywords*).

The block keywords (which *must* be followed by a colon) are:

- *Feature* – introduces the feature being described.
- *Background* – describes context common to all scenarios in this feature file.
- *Rule* – describes a business rule illustrated by the subsequent scenarios.
- *Scenario* – illustrates a single system behaviour.
- *Scenario Outline* – template for generating several similar scenarios.
- *Examples* – present a table of data used in conjunction with a scenario outline.

The step keywords are:

- *Given* – describes the context for the behaviour.
- *When* – describes the action that initiates the behaviour.
- *Then* – describes the expected outcome.
- *And* – used to combine steps in a readable format.
- *But* – used to combine steps in a readable format.

In this chapter, we will look at the usage of most Gherkin keywords. We will cover the remaining keywords, and explore some advanced usage of Gherkin, in Chapter 4, “A new user story”.

Natural language

One of the strengths of Gherkin is that we can write documentation that can be automated by a computer, while remaining readable by a non-technical audience. You should be aware of two aspects of Gherkin that support this goal of readability.

1. Gherkin keywords are available in over 70 languages. If your native language is not supported yet, it is easy to add it.
2. Some languages define multiple variants for some (or all) of the keywords. For example, in Spanish the translation of *Given* has given rise to four variants: *"Dado"*, *"Dada"*, *"Dados"* and *"Dadas"*.

In this book we will exclusively use the English word forms listed above. The English variants and other languages supported can be found in the Gherkin documentation[21].

Strict reading of the text

Since scenarios are used for automation, you have to be aware that consistent usage of spaces, punctuation and capitalization is even more important than in normal written text. For example, a scenario needs to start with "Scenario:" not "Scenario :" and the context keyword is written "Given" not "given". Any inconsistencies will need to be corrected when the scenario is automated.

3.4 – The feature file

Now, let's take a closer look at the feature file the team wrote, with the goal of explaining the basic syntax you'll need to know. You can get more details from the online documentation[22] or get a quick overview using the *Gherkin cheat-sheet* in the Appendices.

[21] https://cucumber.io/docs/gherkin/languages/
[22] https://cucumber.io/docs/gherkin/reference/

A feature file must start with the *Feature* keyword. This is the place to give the feature file a descriptive name. The name of this feature is "Customers can collect their orders":

```
Feature: Customers can collect their orders

Customers can choose to collect their order from the restaurant:
- whether they are authenticated or not
- whether they pay on order or pay on collection
```

As you can see, the lines between the *Feature* keyword and the next keyword can be used for a textual description of the content of the feature file. This is called a *feature description.*

This feature file shows that textual descriptions can be useful, and the team have provided extra background information for anyone reading the file. They could also have provided links to external source of information, such as wireframes, process diagrams, JIRA issues or wiki pages.

Before *Rule*

Prior to Gherkin version 6, it was usual to list the rules that were being illustrated in the feature description. Gherkin 6 introduced the *Rule* keyword which allows for much clearer association between the rule and the scenarios that illustrate it.

Cucumber remains backwards compatible and doesn't require you to use the *Rule* keyword. To see a version of the feature file written without using the *Rule* keyword, check https://github.com/bddbooks/bddbooks-formulation-wimp/tree/main/final-without-rule-keyword.

Adding this sort of information does come with a risk – Gherkin completely ignores it when processing your feature file. This means that it's the team's responsibility to ensure that the information is correct and updated as required. This contrasts with the scenarios themselves that, when automated, become *living documentation* that automatically confirm their correctness.

3.5 – Rules

The *Rule* keyword was created to allow feature files to more closely reflect example maps as introduced by Matt Wynne[23] and described in *Discovery* [Nagy2018, Chapter 2].

```
1 Rule: Any visitor to the website can place a customer-collection order
```

The purpose of introducing this keyword was to associate scenarios directly with the rule they were created to illustrate. This change was made without introducing any breaking changes to existing feature files. So, if you never use the *Rule* keyword you can continue to specify and document your software products using Gherkin.

Whenever you use the *Rule* keyword in a feature file, subsequent scenarios are assumed to illustrate that rule until the next *Rule* keyword is encountered. So, the rules and scenarios in the feature file in Listing 9 are associated in exactly the way you would expect when looking at the example map in Chapter 2, Figure 2.

If you want to include scenarios that aren't associated with any rule, you must introduce them at the beginning of the feature file *before* the first occurrence of the *Rule* keyword. One possible reason for wanting to write a scenario that isn't associated with a rule is described in Section 5.6, "Journey scenarios".

3.6 – Scenario structure

Each scenario describes a single concrete example of system behaviour, which can (and should) be followed by a descriptive name. In *Discovery* [Nagy2018, Section 2.5] we suggest using the *Friends* naming scheme that copies the way that episodes of the *Friends* sitcom were named, for example "The One Where Rachel Finds Out". The words after "The One Where…" happen to be very good titles for scenarios.

Each scenario is made up of one or more *steps*. Steps are introduced by one of the step keywords outlined in the following table.

[23] https://cucumber.io/blog/example-mapping-introduction/

Keyword	Purpose	Guidelines
Given	sets up the *context*	We usually use the past tense (or passive voice) for a *Given* step, because it describes the state the system is already in before the behaviour being illustrated is exhibited.
When	causes an *action* to take place	This describes the stimulus or event that causes the system to behave in a particular way. There should be only a single *When* step in a scenario.
Then	expected *outcome* of the action	There should be only a single *Then* step in a scenario, unless the rule being illustrated results in multiple, related outcomes.
And, But	conjunctions	When used after *Given*, *When*, or *Then* these simply act as a continuation.

Our feature file has a number of simple scenarios in it. They start with a single *Given* statement that sets the context. Then comes a single *When* statement that describes the action. Finally, there's a single *Then* statement that describes the expected outcome.

Listing 10 – Customer chooses to collect order

```
Scenario: Unauthenticated customer chooses to collect order
  Given the customer is unauthenticated
  When they choose to collect their order
  Then they should be asked to supply contact details
```

This scenario is easy to read and understand. It clearly illustrates that customers don't need to be authenticated to be able to place orders for customer-collection.

Seb's story: The customer is always right

Daniel Terhorst-North (the creator of BDD) and Liz Keogh recommend using the word *should* in *Then* steps. For example *"**Then** the till should print a receipt"*.

This has nothing to do with the use of the terms "must", "shall", "will", "should"

> in formal requirement documents[a]. Instead, as Liz explains in her blog[b], she uses "should" so that the reader is more likely to ask themselves, "should it really behave like that?" The use of "should" to describe the outcome has become idiomatic, but not everyone finds it beneficial.
>
> For some of my clients, using "should" clashes with their traditional requirement engineering practice. This is especially true in regulated industries.
>
> One of my clients explained it this way in their internal style guidelines: "Our customers expect us to be confident that our system works as expected. For that reason, *Then* steps should confidently assert the expected behaviour instead of discussing how the system 'should' function".
>
> There are good reasons that the BDD community uses "should" in *Then* steps, but it is not mandatory. The scenarios should be consistent with the use of business language in the organization that they are being written for. This is definitely a situation in which the customer gets to decide.
>
> [a] https://reqexperts.com/2012/10/09/using-the-correct-terms-shall-will-should/
> [b] https://lizkeogh.com/2005/03/14/the-should-vs-will-debate-or-why-doubt-is-good/

3.7 – Multiple contexts

Sometimes it's helpful to supply more context than can comfortably be contained in a single *Given* step:

```
1  Given the customer is unauthenticated
2  And they've chosen to collect their order
```

Here you see the use of the step keyword *And* acting as an extension to provide two pieces of context information. Both these steps are essential, because the rule that is being illustrated is *"Unauthenticated customers must supply acceptable contact details when placing an order for customer-collection"*. We have to ensure the context is described correctly, and this means we must state that:

- the customer is unauthenticated,

- the customer has chosen to collect their order.

We could have compressed this into a single step:

```
Given the customer is unauthenticated and they've chosen to collect their
                                                                      order
```

In this scenario, we find that the single-step form is harder to read and understand, so we prefer using the *And* conjunction. However, you will need to make this decision on a case-by-case basis.

3.8 – Keeping context essential

One of the most common anti-patterns that we see is scenarios that contain too many context steps. We understand that business processes are often complicated, but the steps in a scenario should include only information that is essential to illustrating a specific rule.

If you find yourself with more than one context step, spend some time thinking about the rule that is being illustrated. For each context step, ask yourself: "Does the logic of this rule require this information?". If not, then remove it from the scenario.

In our application, the customer will not be allowed to checkout with an empty basket. We could explicitly add a context step to place item(s) in the basket, but the rule that this scenario illustrates does not depend on the contents of the basket. So, adding any steps that refer to the basket would go against the *essential* principle of the BRIEF acronym.

Many people find it hard to let go of these inessential details. They argue that these actions would need to take place before the system could get into the required state. They reason that the automation code will need to perform these actions, so it will be easier to list them in the scenario. Remember, our goal should be to write scenarios that are easy to understand and easy to maintain.

Inessential details will burden the reader with information that has no bearing on the rule that is being illustrated, so it will be harder to understand. In the *Given* step

you should declare the state of the system, not explain how the system got into that state.

Inessential details also make our scenarios brittle. If some rule that governs one of the inessential details changes, the scenario may break, even though the rule that the scenario illustrates has not changed. Automation is not the concern of the scenario, so we should favour ease of maintenance over ease of automation.

3.9 – Is it a *Given* or a *When*?

It can sometimes be difficult to decide whether a piece of information is part of the context or the action that invokes the expected behaviour. For example, let's consider the rule, *"customer must supply acceptable contact details when placing an order for collection"*. Try to categorize each of the following statements as context (*Given*), action (*When*), or outcome (*Then*):

- Customer is unauthenticated.
- Customer provides unacceptable contact details.
- Customer chooses to collect their order.
- Customer is prevented from progressing.

Most people agree that being *unauthenticated* is part of the context and that *they should be prevented from progressing* is the outcome. However, *providing unacceptable contact details* and *choosing to collect the order* both look like actions that the customer has to take, so should they both be *When* steps?

Listing 11 – Multiple actions

```
Given the customer is unauthenticated
When they provide unacceptable contact details
And they choose to collect their order
Then they should be prevented from progressing
```

Realistically, the customer cannot perform both actions at exactly the same time. We have to choose the action that directly results in the order being refused, based on our understanding of the business domain. For WIMP, choosing to collect the order is part of the context, because otherwise the customer will not be asked for contact details.

Listing 12 – Single action (contact details)

```
Given the customer is unauthenticated
And they choose to collect their order
When they provide unacceptable contact details
Then they should be prevented from progressing
```

Other businesses may not validate the contact details unless the customer chooses to collect the order, in which case the decision would be different.

Listing 13 – Single action (collection)

```
Given the customer is unauthenticated
And they provide unacceptable contact details
When they choose to collect their order
Then they should be prevented from progressing
```

Treat the advice for a scenario to only contain a single *When* step as an *enabling constraint*. It forces you to consider the detailed behaviour of the system and accurately identify the context for each behaviour. If you still can't decide which is the action that causes the behaviour, maybe they are both context and there's a missing action that still needs to be discovered.

3.10 – Multiple outcomes

The *focused* principle of the BRIEF acronym tells us that each scenario should only illustrate a single rule. Usually this means that we only have a single outcome (*Then*) step. However, sometimes multiple outcomes are tightly bound together. In these cases it can be necessary to have multiple outcome steps.

```
Then they should be prevented from progressing
And they should be informed of what made the contact details unacceptable
```

When a user provides unacceptable contact information, our feature file specifies that they should be prevented from progressing through checkout and informed of the error(s) detected. These two outcomes are tightly coupled. Both outcomes are intrinsic to illustrating the rule that *"Unauthenticated customers must supply acceptable contact details when placing an order for customer-collection"*.

3.11 – Conjunctions always need consideration

When you review your formulated scenarios, you should always pay careful attention to the use of conjunctions (and, or) within the text of a single step. For example:

- the customer has authenticated *and* added a pizza to their basket,
- the customer chooses pay-on-collection *or* pay-on-delivery.

There are no situations where the "or" conjunction is acceptable. It introduces a non-determinism to the example that should be avoided at all costs. Instead, split the scenario in two, with one covering each side of the "or" clause.

The "and" conjunction is much more nuanced, as you will have noticed from some of the sections above. You should still carefully consider whether the use of "and" is appropriate, and that consideration will vary depending on whether the conjunction appears in a *Given*, *When*, or *Then* step.

- *Given* – see Section 3.7, "Multiple contexts" section above.
- *When* – the guidelines in Section 3.9, "Is it a Given or a When?" (above) state that, "There should be only a single *When* step in a scenario". The presence of an "and" in an action step is a sign that, either:
  - part of the action (*When*) should actually be considered as context (*Given*), or
  - you have yet to identify a suitable abstraction that describes the action, e.g. "provide user name and password", could be expressed as, "provide authentication credentials".
- *Then* – as explained in Section 3.10, "Multiple outcomes" (above) there are occasional situations in which multiple separate but tightly coupled outcomes are expected. This is not as common as people think. If you spot an "and" in an outcome step, consider whether, either:
  - the outcomes might actually illustrate different rules, in which case a similar scenario should be written to focus on each rule, with the outcome modified accordingly, or

– you have yet to identify a suitable abstraction that describes the outcome without need for a conjunction.

Conjunctions are a common part of natural language, but when they occur within the text of a step, you should always treat this as a cause for careful consideration.

3.12 – Data tables

A central goal of Gherkin is to ensure feature files remain readable by everyone. Tables were included in Gherkin to enhance readability in two distinct ways, which we will describe below. Both are useful, but neither are strictly necessary – you can always write any scenario without using tables at all.

Put simply, Gherkin tables are a way of structuring information associated with the preceding step as a two-dimensional grid. You can think of them as a simplistic equivalent to a spreadsheet.

Data tables and readability

Data tables are very useful when used appropriately, but can seriously harm readability when they get too large. See Section 3.14, "Keep tables readable"

When an order is placed using our application, the customer will be told when it will be ready for collection. This is calculated by adding the order preparation duration to the time at which the order was placed.

Assuming that an order is placed at 18:00 and that it will take 30 minutes to prepare, we could specify this data in a purely textual format:

```
Given the customer placed an order at 18:00 containing items that take 30
                                                              minutes to prepare
```

We find that this sort of information is easier to read when presented in tabular format:

```
Given the customer placed an order
  | time of order | preparation time |
  | 18:00         | 0:30             |
```

There are times when tables are even more useful. Consider providing pricelist information for the part of WIMP that calculates the total cost of an order:

```
Given the restaurant price list is
  | item       | price |
  | Margherita | 7.99  |
  | Calzone    | 9.99  |
  | Funghi     | 8.99  |
```

Imagine how much harder it would be to read the pricelist above specified in sentences.

3.13 – Scenario outlines

Scenario outlines demonstrate a second very different use of data tables to improve readability. We'll start by explaining the mechanics of scenario outlines, before going on to examine the two situations where you might find yourself using them.

How scenario outlines work

A scenario outline is effectively a *template*. When Cucumber/SpecFlow encounter a scenario outline they expand it into multiple scenarios based on the number of rows in the *examples table*. The *Examples* table below has four rows – one header row and three example rows:

Listing 14 – Scenario outline

```
Scenario Outline: Describe what remains of a pizza
  Given a diner cuts their pizza into <slices per pizza> slices
  When they eat <slices eaten> slices
  Then they should have <remainder> slices left

Examples:
  | description       | slices per pizza | slices eaten | remainder |
  | Not hungry at all | 6                | 0            | all       |
  | Quite hungry      | 6                | 4            | 2         |
  | Very hungry       | 8                | 8            | no        |
```

The angle brackets in a scenario outline are *parameter placeholders*. Gherkin matches the text of each parameter placeholder in the scenario outline with the text in the header row of the examples table. Then, for each non-header row in the examples table, it creates a new scenario by substituting the parameter placeholder with the text from the examples table. So, the scenario outline and examples table above are exactly equivalent to:

Listing 15 – Scenario outline equivalent representation

```
Scenario: Describe what remains of a pizza
  Given a diner cuts their pizza into 6 slices
  When they eat 0 slices
  Then they should have all slices left

Scenario: Describe what remains of a pizza
  Given a diner cuts their pizza into 6 slices
  When they eat 4 slices
  Then they should have 2 slices left

Scenario: Describe what remains of a pizza
  Given a diner cuts their pizza into 8 slices
  When they eat 8 slices
  Then they should have no slices left
```

Use of the *Scenario Outline* keyword has become optional

Recent versions of Gherkin treat the *Scenario* and *Scenario Outline* keywords as synonyms. The presence or absence of an *Examples* table is all that is needed for Gherkin to decide how to process the steps within the scenario.

In the examples table in Listing 14 there is a column with heading "description", but there's no `<description>` in the scenario outline. Gherkin simply ignores any column whose data doesn't get used in the outline. This makes it useful for including extra information that makes it easier for a human reader of the feature file to understand *why* that particular example was included. Of course you don't need to name this column "description", and you may want to include more than one column.

Put the description column first

The main reason for putting the description in the first column is that it helps with readability. The reader's focus is immediately drawn to the purpose of each row.

There's another reason to put descriptions in the first column. Some BDD tools (e.g. SpecFlow) use the data in the first column to make it easier for teams to differentiate between the scenarios created from the scenario outline.

Data variations

There are some behaviours that we naturally think about by considering the variation of the input data. Calculations are a common example, and the *"Describe what remains of my pizza"*, scenarios above are one form of calculation. For these types of behaviours it feels natural to describe the examples using tables of data right from the start, rather than writing multiple scenarios.

In the team's feature file there are two scenario outlines:

```
Scenario Outline: Contact details supplied ARE acceptable
  Given the customer is unauthenticated
  And they've chosen to collect their order
  When they provide <acceptable contact details>
  Then they should be asked to supply payment details

  Examples:
    | description           | acceptable contact details |
    | Name and Phone        | Name, Phone                |
    | Name and Email        | Name, Email                |
    | Everything provided   | Name, Phone, Email         |

Scenario Outline: Contact details supplied are NOT acceptable
  Given the customer is unauthenticated
  And they've chosen to collect their order
  When they provide <unacceptable contact details>
  Then they should be prevented from progressing
  And they should be informed of what made the contact details
                                                              unacceptable

  Examples:
    | description | unacceptable contact details |
    | No Name     | Phone, Email                 |
    | Only Name   | Name                         |
    | Only Email  | Email                        |
    | Only Phone  | Phone                        |
```

These are data-driven validation situations, where we want to document what permutations are acceptable. Again, the tabular format feels natural and illustrates the intended behaviour concisely and readably.

Gáspár's story: Examples are not for free

I was once called by a client to help them fix some performance problems that they were having with their BDD automation. If you have ever tried to fix performance problems with a large codebase, you know that it's often not a quick job, so I was a bit nervous when I arrived. It quickly became apparent that their feature files contained a huge set of scenario outlines, each with a large examples table. They

used scenario outlines for everything – I could not find a single "normal" scenario.

In their examples tables, they used a special first column, like the "description" used by the WIMP team, but it contained only numbers: 1, 2, 3, … up to 30 (that was the usual example count). To get a better understanding of the context, I picked one example row and asked what was interesting about it. "It's just a test", was the answer they gave me which was very disappointing because the table didn't just have 30 rows, but also 20 columns, fully loaded with data.

I spent the next hour analyzing the table and trying to find patterns in the data variations. My conclusion was that many of the example rows were essentially illustrating the same behaviours. In many cases they even contained exactly the same values, although this was hard to spot because the similar rows were not close to each other. Once we removed the duplicates, the examples tables were roughly half the size.

By halving the example set we also halved the time it took to run the automation. This fixed the performance problem (of that scenario outline at least), but the bigger problem remained. It's easy to add more examples to a scenario outline, without considering what the *value* of each example is. The team may believe that "a few more tests" is reason enough, but when you have to fix the chaos caused by them you will soon realize: examples do not come for free.

Similar wording

Sometimes you find that several scenarios are very similar, with just one or two words that differ between them. In these cases it can be hard to see the differences between them. This is another situation where a scenario outline can come in useful.

In our feature file there are two scenarios that are quite similar:

Listing 16 – Similar scenarios

```
Scenario: Unauthenticated customer chooses to collect order
  Given the customer is unauthenticated
  When they choose to collect their order
  Then they should be asked to supply contact details

Scenario: Authenticated customer chooses to collect order
  Given the customer is authenticated
  When they choose to collect their order
  Then they should be asked to confirm contact details
```

They could be written as a scenario outline:

Listing 17 – Scenario outline emphasizes differences

```
Scenario Outline: Customer chooses to collect order
  Given the customer is <authenticated-or-not>
  When they choose to collect their order
  Then they should be asked to <action-needed> contact details

  Examples:
    | authenticated-or-not | action-needed |
    | authenticated        | confirm       |
    | unauthenticated      | supply        |
```

We have found that this use of scenario outlines is very attractive to new users of Gherkin, but our suggestion would be to write simple scenarios to start with. If you find that you have several scenarios that are very similar, then consider compressing them using a scenario outline.

In the end though, the question that needs to be asked is, "Will a scenario outline be easier to read than multiple similar scenarios?". Which do you find more readable?

- The team's choice of two scenarios (*"Unauthenticated customer chooses to collect order"*, *"Authenticated customer chooses to collect order"*).
- The single scenario outline (*"Customer chooses to collect order"*).

3.14 – Keep tables readable

Tables exist to enhance readability. When they get too big they have the opposite effect, so remember to keep them small.

As a rule of thumb if you can't easily fit the entire scenario or scenario outline (including the examples table) onto a single laptop screen, it's probably too big. Remember that *it should fit both vertically and horizontally*, so don't allow the tables to have too many columns or rows in them!

Aligning the columns

Tables are much easier to read if you keep the separators ("|"), aligned vertically. Doing this by hand can be a pain, but many modern IDEs provide functionality that lines them up automatically.

3.15 – Readable blocks of text

There's one more feature of Gherkin that exists to preserve readability – the *doc string*. Doc strings become useful when you need to specify textual information in a scenario that won't fit on a single line.

Consider an error message that should be displayed over several lines, such as:

Listing 18 – Expected error message

```
Oops, there was a problem with your order:

  Your address is outside the store's delivery area.
```

This can easily be represented in a feature file using a doc string:

Listing 19 – Using doc string for multi-line text

```
Then the following error message should be displayed
  """
  Oops, there was a problem with your order:

    Your address is outside the store's delivery area.
  """
```

The text between the two triple quotes is treated as a single piece of data, associated with the preceding step. Whitespace in the text is preserved, allowing you to specify exactly the format that is expected. For extra flexibility, Gherkin uses the starting position of the triple quote as the left margin, so any indentation is removed from every line in the doc string.

Double quotes or backticks

Doc strings can be delimited by either backticks (```) or double quotes ("""). However, a doc string's opening and closing delimiters must be identical.

Doc strings aren't needed very often, but when you do need to specify long blocks of text it's a powerful mechanism that preserves readability.

3.16 – What we just learned

In this chapter we saw the WIMP team create a complete Gherkin feature file based upon an example mapping session they did earlier. By reviewing the feature file, we've been introduced to many important elements of the formulation process.

We started by comparing an example map with a feature file formulated from it. We saw that each example had given rise to a scenario which preserved and expanded, in business language, the details captured on the example card.

The concrete feature file also helped us to explore the core syntactical elements of the Gherkin language, like the keywords and the basic structure that flows from the use of the *Feature*, *Rule* (in Gherkin 6), *Scenario*, and *Scenario Outline* keywords.

We also reviewed the anatomy of Gherkin scenarios that use *Given*, *When*, and *Then* steps to mirror the Context-Action-Outcome structure of an example. The BRIEF principles were used to give guidelines about how many *Given*, *When* or *Then* steps we should have in a scenario.

Scenarios are part of our documentation, so readability should always be our primary concern. This feature file has shown us how Gherkin tables can contribute to this. Data tables can be used to attach tabular formatted data to a particular step. The examples tables turn a scenario into an artifact that can, in a compact and readable format, specify and verify the behaviour defined by data variations. Doc strings are another Gherkin construct that aid readability on the rare occasion that a multi-line text element is needed.

Both data tables and scenario outlines are useful elements of the Gherkin language, but they can also be abused. A data table with 20 columns or a scenario outline with 30 examples will not improve the quality of your product. On the contrary, it will obscure the shared understanding of the expected behaviour that is essential to prevent bugs from creeping in.

Although we have seen the core elements of the Gherkin language, we haven't seen everything yet. In the following chapters we will see a few more Gherkin constructs and discuss how you can structure and manage the feature files of your entire product.

Chapter 4 – A new user story

In the last chapter we reviewed the scenarios that the team wrote from the *"customer-collection"* example map. In this chapter we'll watch the team work in a behaviour-driven way, dealing with a new requirement. This will allow us to explore some more pro-tips concerning formulation and the need to listen to feedback.

4.1 – Restricting customers using a blocklist

The business team is concerned by orders placed for *pay-on-collection* that have never been collected (and so, never paid for). They have suggested that the system should mark these customers as *restricted* and that *restricted* customers should not be offered *pay-on-collection* as an option. Patricia created a story to capture this idea which has been explored at several requirement workshops. The team split the original story into several smaller stories and Patricia picked the highest priority story to bring into the iteration. The example map for the story is shown in Figure 4.

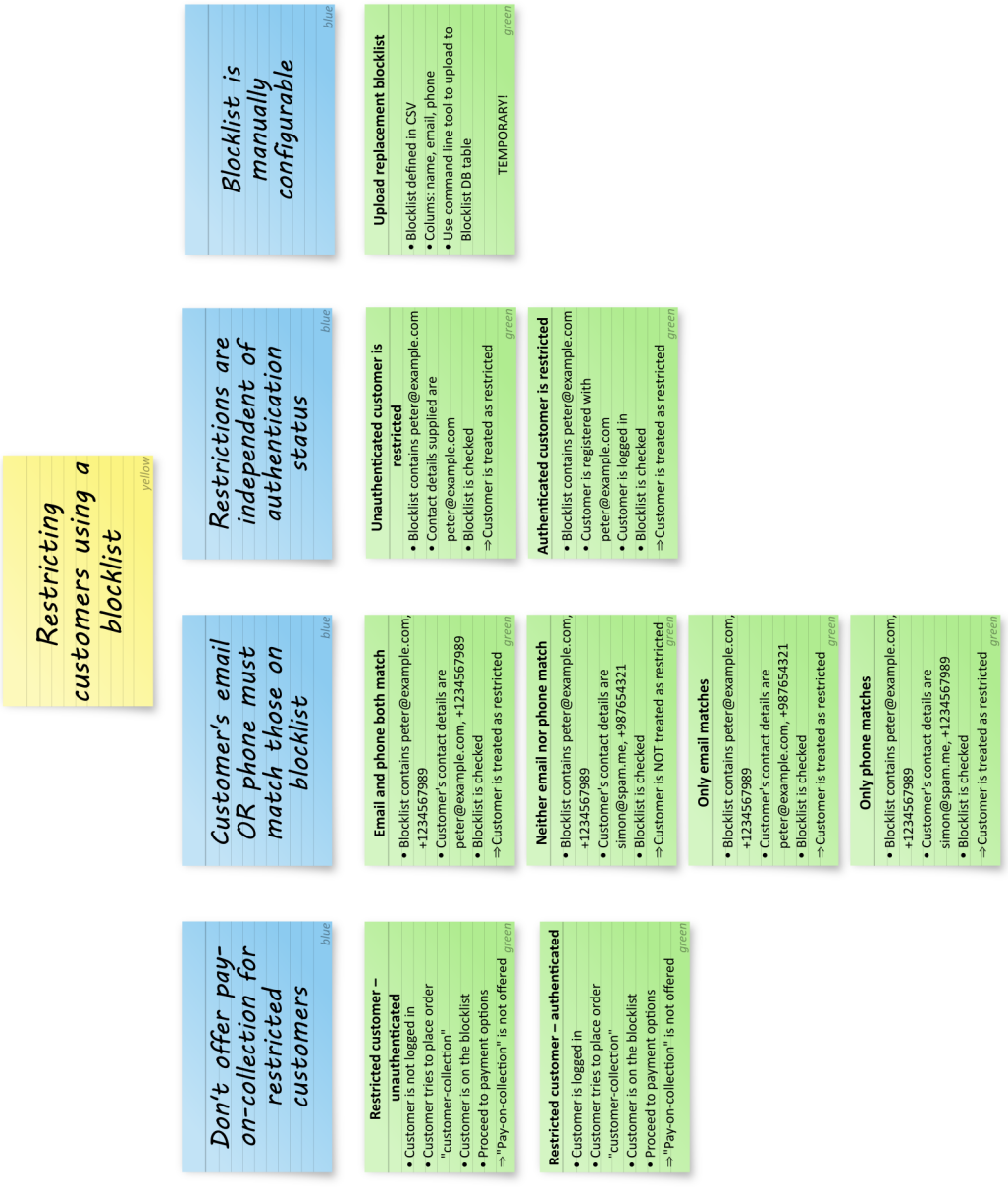


Figure 4 – Blocklist example map

We join the team as they begin to formulate the examples.

4.2 – Write it upwards

> Patricia sets the scene: "We've decided to deliver the blocklisting functionality in several small stories. This first story will deliver the core functionality."
>
> "Which scenario shall we formulate first?" asks Tracey.
>
> "Let's start with the first scenario illustrating, *'Don't offer pay-on-collection for restricted customers'*," says Patricia. "That's the one that I'm most interested in."

The team looks at the first concrete example:

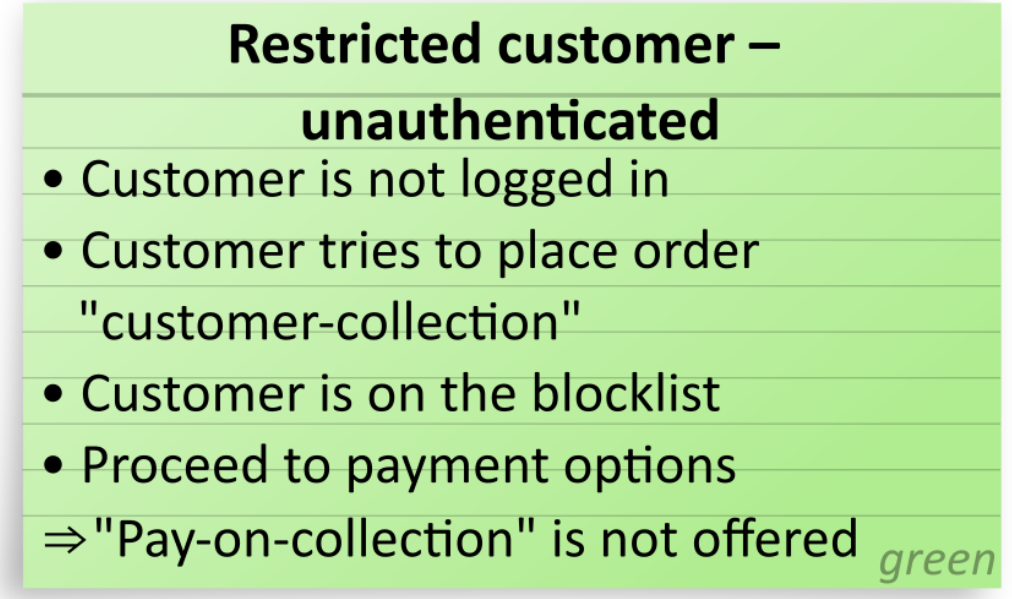


Figure 5 – Restricted customer - unauthenticated

"How about, *'**Given** I am unauthenticated'*," says Dave. "*'**And** I place an order,'*, *'**And** I choose to collect the order'*."

"That's going to make for quite a long scenario," says Dishita. "I'm not sure all those context statements are *essential*. Could we start with the outcome and work our way back to the context?"

Seb's story: Start with the end in mind

"Start with the end in mind," was listed by Steven Covey as one of the habits in his book *The 7 Habits of Highly Effective People* [Covey2005]. Aslak Hellesøy told me how this inspired him to try writing scenarios backwards. By starting with the outcome he found it easier to be sure that there was a clear causal connection between each section of the scenario. In turn, that made it much easier to eliminate inessential context statements.

Later he rephrased this piece of advice as, "Write it upwards".

The team agrees to give this a try.

4.3 – Too many cooks

"Alright then, let's start at the end. What's the expected outcome?", asks Ian. "Will it be, *'**Then** pay-on-collection should not be offered'*?"

"Maybe, *'**Then** should not be allowed to choose pay-on-collection'*," says Dishita.

"*'**Then** I should be required to pay for my order before completion'*," suggests Dave.

Writing is hard, and collaborative writing is even harder. So who should be responsible for formulating the concrete examples uncovered during discovery?

When considering this, there are several questions that need to be answered:

- Why not formulate during the discovery session?
- Shouldn't the whole team formulate together?
- Don't the Product Owners (POs)/Business Analysts (BAs) own the specifications?
- Shouldn't the developers/testers focus on the implementation?
- Isn't it more efficient for one person to do it?

Let's take a look at each of these questions.

Why not formulate during the discovery session?

During discovery we want to uncover misunderstandings and hidden assumptions. The goal is to ensure that the whole team has a shared understanding of how a part of the system is intended to behave. To do this, concrete examples are used to facilitate fast, efficient communication between all participants.

To keep the communication fast and efficient, the concrete examples must be easy to capture. Formulation is a separate practice, with different goals and a more contemplative pace. Attempting to combine discovery and formulation will slow things down and limit the engagement of the team.

Shouldn't the whole team formulate together?

Since the scenarios will be written in a shared language (often called the ubiquitous language), it seems obvious that the whole team should be involved in writing them. However, we've found that most teams choose to delegate formulation to a subset of the team – while maintaining collective ownership of the resulting scenarios.

The first reason to formulate with fewer people is for efficiency. If everyone in the team has an opinion on how an example should be written then the discussion can be very time-consuming. Conversely, if some people don't engage in the discussion, then what benefit is there from having them involved?

The second reason to delegate formulation is that asynchronous reviewing gives the other team members the time and space to read through the scenarios at their own pace and offer constructive feedback. This can also give confidence that the scenario will be understood by people who weren't involved in the product's development.

The challenge is ensuring that the other team members do actually review all scenarios.

Asynchronous reviewing has the potential to become deprioritized, degenerating into a checkbox activity that doesn't provide the vital feedback that is needed. The best way to get feedback is in real-time. If that is not possible, asynchronous feedback can be adequate, as long as the need for meaningful feedback is recognized as an essential part of the team's work.

Don't the Product Owners (POs)/Business Analysts (BAs) own the specifications?

The goal of the PO and BA is to share their business knowledge with the delivery team. This is essential for the delivery team to have enough context to make sensible decisions during development and testing. To confirm that this knowledge has been understood, there needs to be a feedback process.

When scenarios are written by members of the delivery team, they will then be reviewed by the PO/BA, which provides exactly the feedback that is needed. If the scenarios correctly reflect concrete examples, and use appropriate business terminology, then you can be confident that the business and delivery sides of the team really do have a shared understanding.

The PO/BA is still responsible for the specifications – but the work of formulation is shared.

Shouldn't the developers/testers focus on the implementation?

There's a common misconception that the most valuable thing developers can do is write code. A similar misconception exists for the best use of a tester's time. In fact, to deliver value the delivery team must understand the business context of the requirement, so that they can make appropriate decisions.

Writing scenarios is an ideal way for developers and testers to confirm their understanding of the business requirements and the business terminology.

Isn't it more efficient for one person to do it?

If formulation is more efficient when done by a subset of the team, there's an argument that it might be most efficient when done by just one person. In our experience, all activities are improved when undertaken by a pair. In the case of formulation, one of the pair should have business perspective, while the other has a more technical perspective. We have found that pairing a tester (business perspective) with a developer (technical perspective) is often ideal, but your context may be different.

However you choose to delegate formulation, it is always essential that the resulting scenarios are reviewed in detail by team members from other disciplines.

Should the same people always do the formulation?

You should aim to spread the formulation work across the team. If formulation is always done by the same team members, the team will not have a sense of shared ownership of the specification, and misunderstandings and ambiguities will go unnoticed.

To maximize knowledge sharing, formulation should be done by the team members that have the most to learn about the behaviour being specified. During formulation, they articulate their understanding of the behaviour. Then the rest of the team can refine and correct that understanding by giving constructive feedback.

> "Maybe this is an example of, 'too many cooks spoil the broth'[24]," says Patricia. "Perhaps we should delegate formulation to a smaller group."

> "Dishita and I could formulate this example map," suggests Tracey. "Then you could all review what we've done."

Dave, Ian and Patricia get up looking relieved, and leave Tracey and Dishita to continue formulating.

[24] https://en.wiktionary.org/wiki/too_many_cooks_spoil_the_broth

4.4 – Quotation marks

Dishita and Tracey soon have a scenario:

Listing 20 – Pay-on-collection restricted scenario using quotes

```
Scenario: Restricted "unauthenticated" customer not offered
                                                "pay-on-collection" option
  Given I have been restricted
  And I am "unauthenticated"
  When I place an order for "customer-collection"
  Then "pay-on-collection" should not be offered as a payment option
```

"We've written some of the terms in quotation marks," says Dishita. "I don't think they're easier to read like that. Is there any other benefit?"

"I've seen a lot of examples online that use quotation marks to indicate that the value inside the quotes can vary," replies Tracey. "In this scenario, I intended the quotes to show that the terms had a specific meaning within the system, but we don't need to use the quotes."

Quotation marks and snippets

One of the features of Cucumber and SpecFlow is the ability to automatically generate *snippets* to make the job of automation slightly easier. The text enclosed within quotation marks is treated as something that may vary from scenario to scenario, and the snippet that is generated reflects that assumption. Although this is quite a cool feature, it turns out that the engineer usually needs to edit the generated snippet anyway, so there really isn't that much benefit gained from using quotes for this purpose.

As always, consider the readability of the scenario that you are formulating. It will only be automated once, but if you do your job well, it will be read many, many times.

Tracey quickly removes the quotation marks:

Listing 21 – Pay-on-collection restricted scenario without quotes

```
Scenario: Restricted unauthenticated customer not offered
                                                    pay-on-collection option
  Given I have been restricted
  And I am unauthenticated
  When I place an order for customer-collection
  Then pay-on-collection should not be offered as a payment option
```

"That does make it more readable," says Tracey.

4.5 – There's no "I" in "Persona"

"This scenario is looking a lot better," says Dishita, "but I still have one worry. Who is 'I'?"

"'I' is the restricted customer," says Tracey. "Why do you ask?"

"It just seems that 'I' could be misinterpreted. Could we be more specific?"

"We could create a restricted customer persona called Rona," suggests Tracey. "Then we could say *'**Given** Rona is unauthenticated'*. Would that be better?"

"It would, but it's quite an investment creating a persona. Maybe we could just make the noun phrases more descriptive by replacing 'I' with 'customer'," says Dishita.

Listing 22 – Pay-on-collection restricted scenario using *Customer*

```
  Scenario: Restricted unauthenticated customer not offered
                                                  pay-on-collection option
    Given a restricted customer
    And the customer is unauthenticated
    When the customer places an order for customer-collection
    Then pay-on-collection should not be offered as a payment option
```

> "That is clearer," says Tracey, "but I've seen people use the 'I' form on the Internet."
>
> "'I' might work in some cases, but since our application has several different user roles, it's not appropriate in this project," says Dishita.

Use of the first person "I" in scenarios is common throughout industry and is advocated in many tutorials. However, we encourage people to avoid this way of formulating scenarios, because it encourages imprecision. In particular, consider that the reader of a scenario will identify themselves as "I". This leads to the meaning of a scenario being dependent on the person who is reading it, which is exactly what we don't want to happen.

Instead, we recommend that all scenarios are written in the third person, identifying all actors by role or persona. In this case Dishita has correctly observed that creating a persona just for this scenario would be an excessive investment, especially since the role description is perfectly adequate. Personas also come with the added overhead of requiring all readers of the scenario to understand all their attributes, which is usually only practical if we restrict ourselves to a few, well-known personas.

> "This looks really good," says Tracey. "The *Given* steps are in the past tense – indicating that they have happened before the action takes place. There's a single *When* step in the present tense. And the *Then* step is described with 'should', which helps keep interpretation of any failure open."

4.6 – Does repetition matter?

In the previous scenario (Listing 22) there's repeated use of the word *"customer"*. You wouldn't expect to see this sort of language in a novel or a newspaper article, so should it be acceptable in our business readable specification?

An alternative way to structure this scenario could be:

Listing 23 – Pay-on-collection restricted scenario using *they*

```
Scenario: Restricted unauthenticated customer not offered
                                                  pay-on-collection option
  Given a restricted customer
  And they are unauthenticated
  When they place an order for customer-collection
  Then pay-on-collection should not be offered as a payment option
```

This format is closer to natural usage of the English language, and would be considered more readable by many.

Gendered pronouns

English uses gendered pronouns ("he" and "she") when talking about a single individual. In situations where the gender is incidental to the behaviour being described, it is now common to use "they". As the Merriam-Webster website[25] makes clear: "…*they* has been in consistent use as a singular pronoun since the late 1300s."

However, using "they" is not always possible. In the scenario above, it is clear that "they" refers to the customer, but in scenarios that include more than one actor this may not be the case. Take a look at the scenario below:

[25] https://www.merriam-webster.com/words-at-play/singular-nonbinary-they

Listing 24 – Ambiguous usage of *they*

```
Scenario: Manager cancels customer restriction
  Given a restricted customer
  And a manager processing a valid request to cancel their restriction
  When they override the restriction
  Then they should no longer be restricted
```

When usage of "they" does not diminish clarity, the choice between repetition and using a pronoun is stylistic and has to be made by the team. When making this decision, remember that readability is the prime goal when formulating scenarios.

4.7 – Readability trumps ease of automation

In the previous section we replaced the repeated use of "customer" with the pronoun "they". A common concern is that this makes it harder to automate the scenario.

The wording of your specification should *not* be unduly influenced by the needs of the automation framework that you are using. There are many automation approaches that can handle linguistic shortcuts (such as pronouns), and your engineers should use them as appropriate. Remember that the primary goal of Gherkin is to document the specification in a business readable format. All barriers to readability should be avoided, even if this adds some complexity to the automation code.

Once a formulated scenario has been reviewed and approved by the team it should not be modified unless the system's behaviour itself changes. After all, any change to a scenario is effectively a change of the specification.

4.8 – Background

> "Let's formulate the next scenario for an authenticated customer," says Tracey. They quickly get this scenario:

Listing 25 – Pay-on-collection scenario for a restricted customer

```
Scenario: Restricted authenticated customer not offered pay-on-collection
                                                                    option
  Given a restricted customer
  And they are authenticated
  When they place an order for customer-collection
  Then pay-on-collection should not be offered as a payment option
```

> "Both these scenarios start with the same statement, *'**Given** a restricted customer'*," says Dishita. "Maybe we could use a *Background* to simplify the feature file."

Background is a way that Gherkin allows you to specify that a number of steps should be run before every scenario in a feature file. It is intended to be used when each scenario in a feature file shares the same context.

The two scenarios that we have discussed so far are shown below using the *Background* keyword. Their behaviour is unchanged.

Listing 26 – Scenarios sharing context using *Background*

```
Background:
  Given a restricted customer

Scenario: Restricted unauthenticated customer not offered
                                                pay-on-collection option
  Given they are unauthenticated
  When they place an order for customer-collection
  Then pay-on-collection should not be offered as a payment option

Scenario: Restricted authenticated customer not offered pay-on-collection
                                                                    option
  Given they are authenticated
  When they place an order for customer-collection
  Then pay-on-collection should not be offered as a payment option
```

Gáspár's story: Clean, but not so dry

A proper implementation of the agile principles on a project requires us to think differently about testing and test automation code. Quality (and all related testing and coding activities) are the responsibility of the whole team. The results should be treated as important and valuable artifacts. "Tests should be treated as first-class citizens," is a commonly expressed attitude, and it follows that test automation code should adhere to local coding standards, such as the "clean code" principles.

There are discussions about how these principles can and should be applied to test code, especially on the "Don't repeat yourself" (DRY) principle. While the DRY principle for production code is important to avoid duplication and redundancy, for test code, keeping the structure simple and readable (sometimes mentioned as DAMP – Descriptive And Meaningful Phrases) seems to be more important than the DRY principle.

The BDD scenarios will be used as automated tests, you might want to consider the "clean code" principles. Your conclusion should be the same: simplicity and readability is more important than rigorously eliminating all duplications. Consider this when using *Background*.

We believe that the *Background* keyword should not be used, because it has a negative effect on the readability of the scenarios. When reading a scenario, it is necessary to bear in mind what steps (if any) are included in the *Background*. In even moderately sized feature files, this requires scrolling to the top of the file, which is tedious and error-prone.

A *Background* also makes it harder to maintain the feature file. Anyone editing existing scenarios (or adding new ones) needs to be aware of the steps in the *Background*, since they will set the context for every scenario in the feature file. More dangerous still is the possibility of editing the *Background* without understanding the impact that will have on every scenario.

Seb's story: Background compatibility

> I have suggested to my colleagues maintaining Cucumber and Gherkin that the *Background* keyword should be removed from Gherkin. Even though many practitioners agree with me, this has not been approved due to a desire to preserve backwards compatibility.
>
> At some time in the future we may deprecate this dangerous feature. If this happens, we will certainly protect historic users by providing options that preserve the existing behaviour.

"That doesn't look as readable to me," says Tracey. "Let's leave it the way it was, without the *Background*."

4.9 – Unformulated examples

"These two scenarios do look very similar, though. The only difference is that in one the customer is authenticated and in the other they aren't," observes Tracey. "Is there really a significant difference between them?"

"No, I don't think so," says Dishita. "Let's combine them into one"

Listing 27 – Combined restricted scenario

```
Scenario: Restricted customer is not offered pay-on-collection option
  Given a restricted customer
  When they place an order for customer-collection
  Then pay-on-collection should not be offered as a payment option
```

"This is another example where we were tempted to make a scenario dependent on incidental details," says Tracey. "Do you think it's alright to change the concrete examples during formulation?"

This is a common question. Concrete examples generated during example mapping are used to discover how imperfect our understanding is. We don't want to slow the process down to make sure that all examples are perfectly formed. Instead, during formulation we make judgements about whether a scenario, once formulated, would become a valuable part of the specification. In the process we may merge examples, or decide not to formulate an example at all.

Even if a concrete example ends up not being formulated it has already served a useful purpose – clarifying the expected behaviour of the system for the person that created the example. Furthermore, unformulated examples may still make great programmer tests.

Programmer tests

In addition to the scenarios, programmers will create additional automated technical tests to guide their development. Programmers talk about many different types of tests, including "unit tests", "integration tests", "component tests". In these books we generically refer to tests written by programmers for programmers as *programmer tests*.

"Yes," says Dishita, "I believe that the single scenario accurately illustrates the expected behaviour of the system. Other scenarios should show how we determine if a customer is restricted or not. And if the product owner is not happy with our formulation, then she will let us know when she reviews our scenarios."

4.10 – Commenting in feature files

"Let's leave a comment for Patricia about why we decided that one scenario was enough. If she agrees, we can delete it," says Tracey.

They modify the scenario to explain what they have done:

Listing 28 – Restriction scenario with comment

```
# Two scenarios merged, because authentication is incidental
Scenario: Restricted customer is not offered pay-on-collection option
  Given a restricted customer
  When they place an order for customer-collection
  Then pay-on-collection should not be offered as a payment option
```

The # sign at the beginning of a line indicates that the line is a comment and has no effect on the living documentation. This is known as a *line comment*. There is no way in Gherkin to create a *block comment* (that comments out multiple contiguous lines), except by placing a # at the start of every line (although many IDEs will do this for you automatically).

As well as the comment character #, Gherkin provides designated areas in a feature file where you can leave plain *text descriptions*. After any line that starts with a block keyword (see Section 3.3, "Gherkin basics"), automation tools will ignore all text until they find the next line that begins with a keyword. So, Dishita and Tracey could have written the scenario like this:

Listing 29 – Restriction scenario with text description

```
Scenario: Restricted customer is not offered pay-on-collection option
  Two scenarios merged, because authentication is incidental

  Given a restricted customer
  When they place an order for customer-collection
  Then pay-on-collection should not be offered as a payment option
```

In this case, since the comment is not relevant to the understanding of the scenario, but more for the formulation task itself, the comment is a more appropriate solution. Text descriptions should only be used when they contribute to the documentation of the system's behaviour.

Use comments and descriptions wisely

Feature files are intended to be readable by all interested stakeholders. They should be written in business language, that is intended to be unambiguous and self-explanatory, so there should rarely be any need for comments or descriptions.

Whenever you see a comment or description in a feature file, ask yourself how you could change the feature file to make it unnecessary.

4.11 – Setting the context

"Should we have a scenario that shows that pay-on-collection is available if you're not restricted?" asks Dishita.

"We already do, in the *'Customer chooses to pay on collection,'* scenario we reverse engineered earlier," says Tracey (discussed in Chapter 2, "Cleaning up an old scenario").

Listing 30 – *Customer chooses to pay on collection* scenario

```
Scenario: Customer chooses to pay on collection
  Given the customer has chosen to collect their order
  When they choose to pay on collection
  Then they should be provided with an order confirmation
```

"That scenario doesn't mention restrictions at all," says Dishita.

"No," agrees Tracey, "it's implicit that most customers are not restricted."

Scenarios need to be unambiguous and precise, but they also need to be brief. If we stated all the system preconditions in every scenario, nobody would ever be able to read them, let alone review them. Their value would disappear.

The goal of a scenario is to illustrate a rule. To do that it needs to make clear why it exists – why *this* scenario is interesting. If we keep reiterating the default situation, scenarios quickly become boring.

When you head to an ATM to withdraw cash, how often do you say to yourself, "I hope the ATM has some cash in it today?" The same is true in this case: most customers that order a pay-on-collection pizza do actually collect and pay for their order, so, we don't have to restate the obvious. When the unusual situation of a restricted customer occurs, we draw attention to this by stating that the customer *is* restricted.

In the extreme, the context may be entirely implicit, which means that there is no need for any *Given* statements to set the context. Consider the following scenarios describing the behaviour of an electric kettle.

Listing 31 – Electric kettle – context steps not essential

```
Scenario: Kettle boils water
  Given the electricity supply is live
  And the fuse is not blown
  And the kettle has water
  When I turn the kettle on
  Then the water should boil

Scenario: Kettle is protected from burn-out
  Given the electricity supply is live
  And the fuse is not blown
  And the kettle is empty
  When I turn the kettle on
  Then the burn-out protection should engage
```

A typical reader of these scenarios will consider that most of the context statements are *implicit*, so we could shorten them to:

Listing 32 – Electric kettle – electricity context implicit

```
Scenario: Kettle boils water
  Given the kettle has water
  When I turn the kettle on
  Then the water should boil

Scenario: Kettle is protected from burn-out
  Given the kettle is empty
  When I turn the kettle on
  Then the burn-out protection should engage
```

The shortened scenarios still have very similar contexts and the reader may wonder why the outcomes are so different. It takes time and effort to notice the change from, *"the kettle has water,"* to, *"the kettle is empty"*. Additionally, there will be many scenarios that cover the common context that *"the kettle has water"*. However, there will be very few scenarios covering the exceptional context that, *"the kettle is empty"*, so this must be stated explicitly.

Forcing a reader to parse a common or implicit context many times is wasteful. Instead, treat the common context as *implicit* and write the scenarios as:

Listing 33 – Electric kettle – water context implicit

```
Scenario: Kettle boils water
  When I turn the kettle on
  Then the water should boil

Scenario: Kettle is protected from burn-out
  Given the kettle is empty
  When I turn the kettle on
  Then the burn-out protection should engage
```

We find that omitting the context when it can reasonably be considered *implicit* leads to more readable, maintainable specifications.

Gáspár's story: Decide by seeing

Applying the six BRIEF principles we listed in Chapter 2, "Cleaning up an old scenario", is straightforward in many cases but sometimes it needs more consideration. If you are not careful and people are not in the right mood, you can quickly get into a heated discussion about whether we should formulate something in one or two steps, whether a particular bit is essential or not. Once the discussion gets harsh and personal, it is much harder to get back to constructive work. I have seen this happen.

Stopping the discussion and asking the participants to write down the scenario that expresses their own viewpoint is often a good solution. Being able to see how the various scenarios differ can help decide which version supports the team better – like you saw in the discussion between Tracey and Dishita or with the variations of the kettle scenarios.

By applying Aslak's advice to, "write it upwards" (Section 4.2, "Write it upwards"), we can easily spot what context is essential to ensure that the system behaves as expected.

4.12 – Staying focused

"Now let's move on to another scenario," says Dishita. "Which one shall we choose?"

"Let's try, *'Customer's email OR phone must match those on the blocklist'*," says Tracey.

Figure 6 – E-mail or phone match examples

They work on the first example and write the following scenario:

Listing 34 – Restricted customer – exact match

```
Scenario: Email and phone both match
  Given the blocklist contains:
    | email             | phone number |
    | peter@example.com | +123456789   |
  And the customer's contact details are:
    | email             | phone number |
    | peter@example.com | +123456789   |
  When the blocklist is checked
  Then they should be treated as restricted
```

"I think this is good summary," says Dishita, contentedly looking at the results. "I like it. It's concise and clear."

"Yep, but won't we need more information to actually implement the functionality?" asks Tracey. "We will have to handle alternative phone number and email formats," and she modifies the scenario:

Listing 35 – Restricted customer – alternate match

```
Scenario: Email and phone both match
  Given the blocklist contains:
    | email             | phone number |
    | peter@example.com | +123456789   |
  And the customer's contact details are:
    | email             | phone number |
    | PETER@EXAMPLE.COM | 00123456789  |
  When the blocklist is checked
  Then they should be treated as restricted
```

"That's a good point," says Dishita, "but now this example is illustrating more than one rule. These examples are supposed to focus on the rule that decides if a customer should be restricted, but now it also includes the rules that define how emails and phone numbers should be matched."

"We could write the scenario like this instead," says Tracey.

Listing 36 – Restricted customer – data matching removed

```
Scenario: Email and phone both on the blocklist
  Given the customer's contact details are on the blocklist:
    | email | phone number |
    | yes   | yes          |
  When the blocklist is checked
  Then the customer should be treated as restricted
```

"But now there's no real data in the scenario," says Dishita. "And we'll need to agree with the team what the matching rules should be."

"You're right," says Tracey. "We can assume really simple rules for now – like exact, character-for-character matching."

"OK," says Dishita, "I'll note them down and ask Patricia to bring them to the next requirement workshop," and she scribbles on two red question cards:

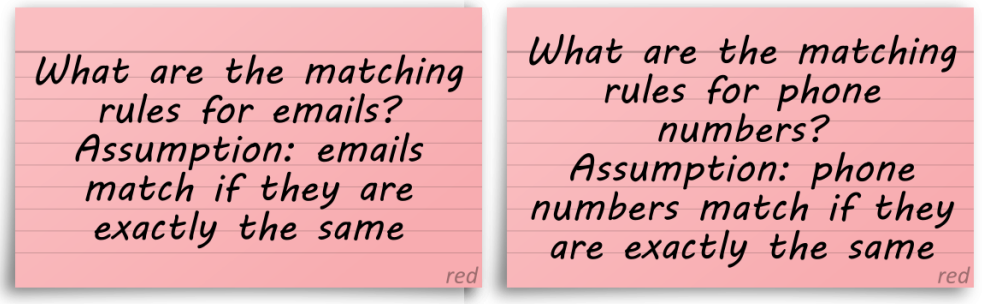


Figure 7 – Question cards

"During that requirement workshop we will create real examples using actual phone numbers and emails," says Dishita. "Now, let's get back to the blocklist and try the example where neither email nor phone number match".

Listing 37 – Unrestricted customer

```
Scenario: Neither email nor phone on the blocklist
  Given the customer's contact details are on the blocklist:
    | email | phone number |
    | no    | no           |
  When the blocklist is checked
  Then the customer should be treated as unrestricted
```

"These scenarios look very similar," says Tracey. "This would make a good scenario outline. We could even include the other two examples from the example map as well."

Listing 38 – Restriction – Scenario Outline

```
Scenario Outline: Decide if customer is restricted
  Given the customer's contact details are on the blocklist:
    | email           | phone number   |
    | <email listed> | <phone listed> |
  When the blocklist is checked
  Then the customer should be treated as <status>

  Examples:
    | description    | email listed | phone listed | status       |
    | Both listed    | yes          | yes          | restricted   |
    | Email listed   | yes          | no           | restricted   |
    | Phone listed   | no           | yes          | restricted   |
    | Neither listed | no           | no           | unrestricted |
```

4.13 – Formulation gets faster

"Let's move on to the remaining examples. Shall we start with the two examples of the, *'Restrictions are independent of authentication status,'* rule?" asks Tracey, as she moves the example cards closer.

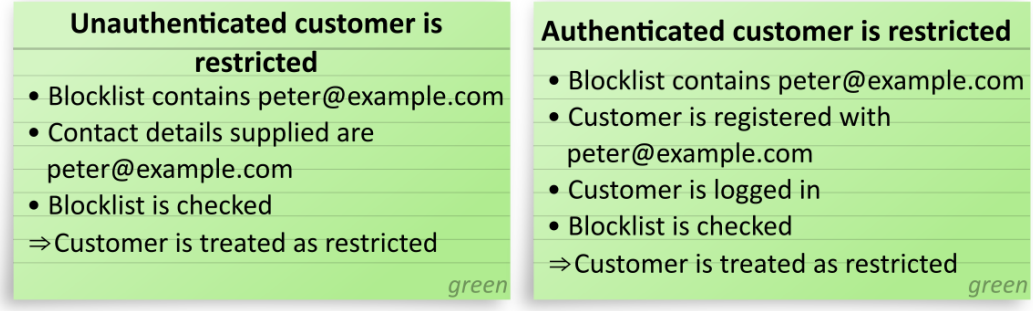


Figure 8 – Restrictions are independent of authentication status examples

"I think we have already formulated all the necessary steps for these in previous scenarios. We just need to put them together in the right order," says Dishita.

"OK, let's reuse the formulated steps, but we should make sure that they are still readable as a whole scenario," agrees Tracey and they compose the following scenarios:

Listing 39 – Scenarios reusing existing step

```
Scenario: Unauthenticated customer is restricted
  Given a customer is unauthenticated
  And the order's contact details are on the blocklist:
    | email | phone number |
    | yes   | yes          |
  When the blocklist is checked
  Then they should be treated as restricted

Scenario: Authenticated customer is restricted
  Given a customer is authenticated
  And the account's contact details are on the blocklist:
    | email | phone number |
    | yes   | yes          |
  When the blocklist is checked
  Then they should be treated as restricted
```

"That was easy," says Dishita. "We only needed to create steps to make it explicit, whether the contact details to check came from the order or the customer's account."

"These may not be the most interesting scenarios in this feature," says Tracey, "But they demonstrate that the system can recognize a restricted customer whether they are authenticated or not. It gives me confidence that the previous scenario (Listing 38) doesn't need to specify whether the customer is authenticated or not."

"Fewer scenarios is better," says Dishita, "Because it's less to read for Patricia and less maintenance for us."

"How is it going?" asks Dave, who peeks into the meeting room. "It looks like you're almost done!"

"Yes, we just have one scenario left," answers Tracey. "We were struggling a bit with the first scenarios, but now that we've got into it, I think that we've produced a consistent result."

"I agree," continues Dishita. "It is more fluent now. The last few scenarios only took a few minutes."

"That's great! This means we can start working on them this afternoon. See you later, then," says Dave and leaves.

Formulating the first few scenarios is always hard. You have to figure out how to phrase the steps in a way that reflects the intention of the example while remaining readable to all team members, and precise enough to be able to build an automation solution for them later. Once you start feeling the rhythm of the ubiquitous language, formulating similar scenarios gets faster and faster. In many cases, you will be able to reuse steps from the previous scenarios.

Gáspár's story: Don't overrate reusability

One of the great productivity features of Cucumber-like tools is that they allow you to define automation code for a step that can be reused in many scenarios.

When I formulate scenarios with a team, there is always someone who suggests using more granular action-like steps, so that they are going to be more reusable in later scenarios. It is hard to argue about what is going to happen in the future, so I usually just ask for their patience until we have automated the first few of them. The result is pretty convincing: usually it turns out that the more readable steps can also be reused and they also provide more flexibility when they come to be automated.

While step reusability is important, don't let the readability of your scenarios suffer from focusing on writing super-reusable steps. Don't be tempted to reuse an existing step unless it really fits well in the scenario you're working on. My advice is to focus on consistency instead of reusability. Once you find a consistent way of phrasing your steps, you will benefit from step reusability anyway.

4.14 – Incremental specification

Dishita and Tracey have almost finished formulating the scenarios that define this story.

> "We only have *'Blocklist is manually configurable'*, left to formulate," says Tracey. "When we discussed it in example mapping, we agreed that it would be a temporary, manual process, only used during development and testing. Adding contact details to the blocklist will eventually happen automatically when a customer fails to pick up an order."

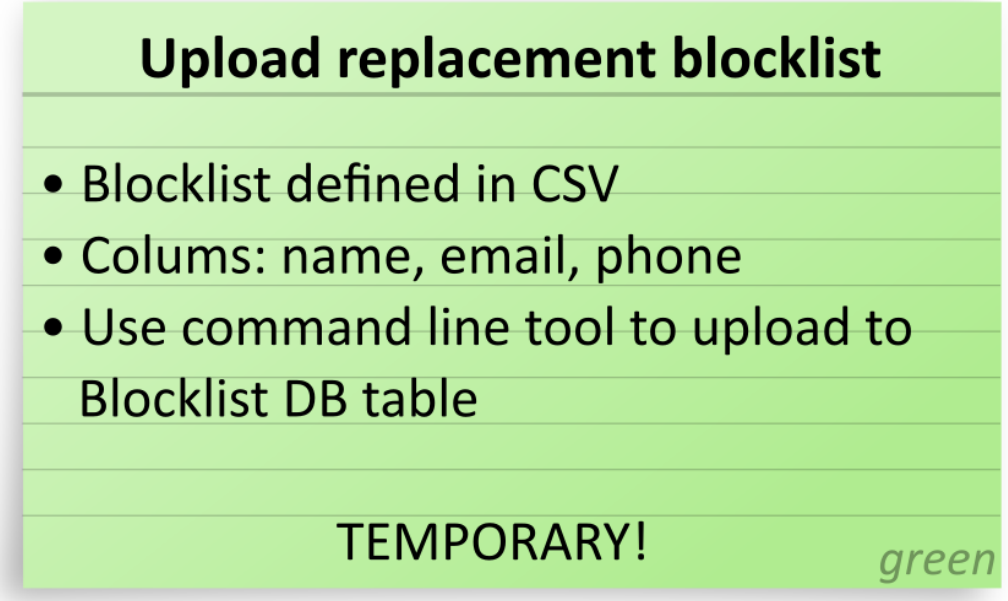


Figure 9 – Upload replacement blocklist

"Yes," agrees Dishita, "but there will always be a way for restaurant managers to manually edit the blocklist."

"That's true. Let's create a high level scenario that describes the intent," says Tracey.

Listing 40 – Upload replacement blocklist

```
Scenario: Upload replacement blocklist
  Given a manually created blocklist defined using names, emails and
                                                              phone numbers
  When the blocklist is uploaded
  Then the upload should replace the existing blocklist
```

> "That's really high level, but it does reflect the behaviour we need. Is that OK?" asks Tracey.
>
> "I think so," says Dishita. "We can let the developer decide on the technical details."

When we deliver functionality in small increments, we often start by implementing a *low fidelity*[26] version. It's called *low fidelity* because it doesn't truly represent the requested functionality yet. Subsequent increments will build on the low fidelity foundations that we've built, gradually satisfying more and more of the functionality needed, eventually producing a high fidelity version that is suitable for release.

By delivering small stories, we're making a decision to defer some of the work so that we can get valuable feedback earlier. Specifying the behaviour in a way that allows the implementation to evolve is valuable – it describes to the product owner exactly what we are trying to achieve while leaving the implementation details open.

As the implementation becomes more finessed we will create more rules and scenarios, but the original, high level scenario(s) may not need to change at all. What will need to change is the automation code, since it's the underlying implementation that will be evolving.

4.15 – Manual scenarios

> "Is it worth automating this scenario?" asks Dishita. "The implementation is definitely going to change, so any automation code that we write will have to change too."

[26] https://availagility.co.uk/2009/12/22/fidelity-the-lost-dimension-of-the-iron-triangle/

> "Yes, the implementation will change, and we're not even sure whether the blocklist will eventually be configured by uploading a file. I guess we could keep this scenario *manual* until we've made that decision," says Tracey.

They add a tag to the scenario:

Listing 41 – Tagging a scenario

```
@manual
Scenario: Upload replacement blocklist
  Given a blocklist is defined using email and phone numbers
  When the blocklist is uploaded
  Then the upload should replace the existing blocklist
```

Gherkin tags

Gherkin tags have a number of uses that will be described in detail in Chapter 5, "Organizing the documentation". For now, what you need to know is that a tag is a word prefixed by the `@` character.

Remember, the feature file is intended to be business readable documentation, so tags should make sense to your PO.

Tracey and Dishita's team have internally agreed that any scenario that will not be automated should be tagged as *"@manual"*. There are no pre-defined tags in Gherkin, so they could have chosen any tag, such as *"@by_hand"*, *"@not_automated"*, or *"@xyz"*. Tags can then be used to tell automation tools (like Cucumber and SpecFlow) which scenarios should be executed (see Section 5.4, "Documenting the domain").

Scenarios that are not automated can be useful, but they are not as valuable as automated scenarios. The risk with a manual scenario is that when the system changes there is no automatic notification that a manual scenario is out-of-date. You rely on people accurately reviewing manual scenarios on a regular basis and in our experience this often does not happen.

Unfortunately, there are some situations where manual scenarios are inevitable, for example when there is no programmable interface available. It should be stressed, however, that there are very few situations where there is *no* way to control your system programmatically – sometimes it's just unexpectedly hard.

4.16 – Who does what and when

Dishita and Tracey have formulated the concrete examples generated in the example mapping session, but their work needs to be reviewed. It is ideal if scenarios are reviewed as soon as they are formulated. After all, it is preferable to ensure that they are correct *before* the team starts implementing any automation or production code. In some organizations this is not possible, so implementation starts before the review has been completed.

We recommend that all team members review the scenarios, especially after formulating the first scenarios of a new functional area. Keep in mind that formulation is intended to capture the shared understanding generated by example mapping using business readable terminology. The whole team needs to agree that the scenarios unambiguously illustrate their understanding of the system's behaviour.

As a minimum, the PO must review the formulated scenarios to ensure that what has been documented accurately reflects their expectations. Ultimately it is the PO who is accountable for the product's specification, but they may delegate their responsibility to another team member or stakeholder.

In *Discovery* [Nagy2018, Chapter 4] we described an idealized BDD approach. In this chapter, we have been digging into nodes #3 (Formulate) and #4 (Review) from the diagram below.

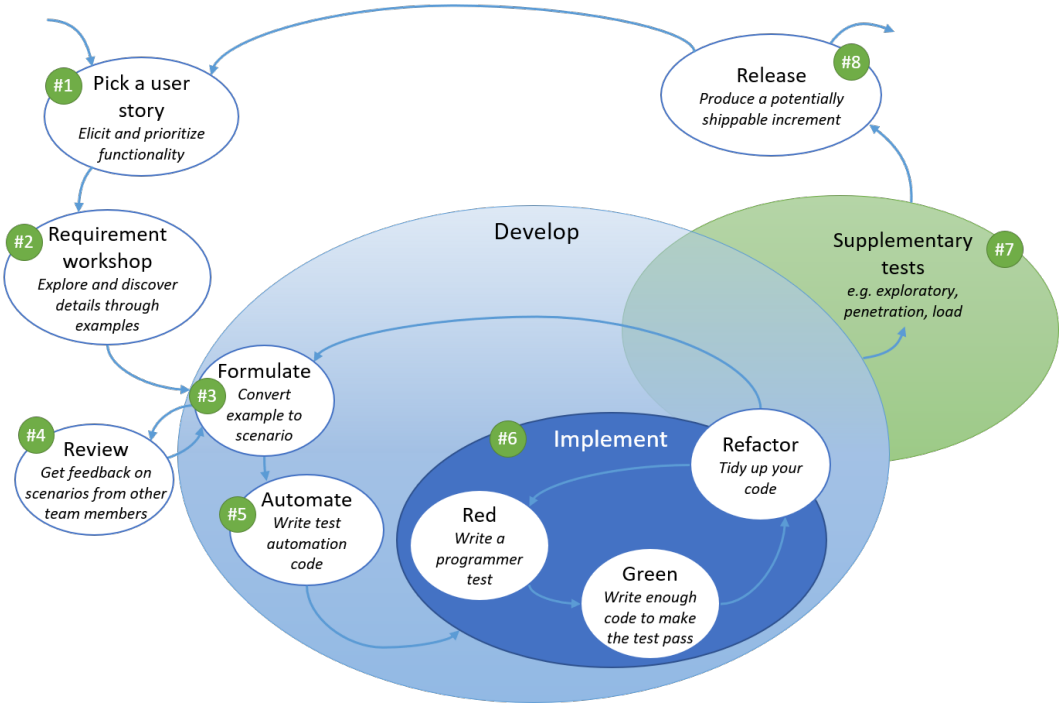


Figure 10 – Tasks and activities in the BDD approach

4.17 – What we just learned

In this chapter we joined the WIMP team while they formulated the scenarios for a new user story. The rules and the examples describing the story had been prepared by the team in an example mapping session earlier. We saw how effective and efficient a developer/tester pair was at formulating these examples into scenarios.

The complete feature file (`blocklisting.feature`) can be found at https://github.com/bddbooks/bddbooks-formulation-wimp/tree/main/ch4-end.

Although the prepared examples are very helpful for writing the scenarios, formulation is usually not a trivial one-to-one translation of the "green cards" into scenarios in a feature file. Dishita and Tracey had to deal with:

- ensuring consistent phrasing and grammar structures,
- reviewing whether all details are actually essential,
- applying the Gherkin structuring features like scenario outline or background whenever they result in better documentation,
- merging, splitting or skipping examples depending on how they contribute to the documentation,
- capturing information for further tasks (e.g. review or automation) as comments or tags,
- reusing existing steps.

Good scenarios sometimes come easy, but in some cases you need to explore different options and review them to be able to make the best decision. Be patient. Even if you spend more time on the first scenarios, it will get faster and easier.

In this chapter the team was focusing on scenarios illustrating the behaviour of a single feature. In the next chapter we will see how to manage scenarios that span multiple features.

Chapter 5 – Organizing the documentation

In the last chapter we saw Tracey and Dishita take an example map and formulate a feature file. In this chapter we'll follow the team as it reviews this feature file and tackles the challenges of evolving the documentation as the product grows.

5.1 – User stories are not the same as features

Patricia is out of the office the next day, but she reads through the feature file as soon as she gets a chance.

Listing 42 – Structure of the blocklisting feature file (scenario steps omitted for brevity)

```
Feature: Blocklisting

Rule: Don't offer pay-on-collection for restricted customers
  # Two scenarios merged, because authentication is incidental
  Scenario: Restricted customer is not offered pay-on-collection option

Rule: Customer's email OR phone must match those on blocklist
  Scenario Outline: Decide if customer is restricted

Rule: Restrictions are independent of authentication status
  Scenario: Unauthenticated customer is restricted
  Scenario: Authenticated customer is restricted

Rule: Blocklist is manually configurable
  @manual
  Scenario: Upload replacement blocklist
```

She doesn't see anything that causes major concern, so she posts a couple of brief messages in the team's online chat room:

> **Patricia**: **Tracey** and **Dishita** – this looks great. I can see why you merged the two examples that illustrated the blocklisting rule. When I'm back in the office, I'd like to really understand what the full implications of the *"@manual"* tag are.
>
> **Patricia**: **Team** – please review the feature file and post your comments here.

A short time later, having reviewed the feature file, Dave makes the following post:

> **Dave**: The scenarios all look fine, but should they all be in the same feature file? I wonder if we could think about how we want the documentation to evolve.

Early the next day the whole team get together in a meeting room, and Dave outlines his concerns.

> "Most of the rules specify how customer blocklisting works, but there's one rule that's very specific about whether a restricted customer is offered the choice of pay-on-collection. Clearly these are related, but aren't they actually different features? What do you think, Patricia?"
>
> "Yes, I suppose we could think of them as different features," agrees Patricia, "But they're part of the same story. Doesn't that imply we should keep them together in the same feature file?"

Patricia is voicing a common concern that leads many teams to create one feature file for each user story they work on. This is a mistake.

Stories are not "agile requirements"

Many teams have forgotten that user stories are not requirements.[27] When first created by the original XP practitioners, stories were described as "placeholders for a conversation". They were a way of deferring detailed requirements analysis until the story had been prioritized for delivery.

During analysis (and discovery) we usually find that each story we discuss generates many smaller stories. Having small stories helps us plan the incremental delivery of the required functionality. Having small stories allows the PO to prioritize at a fine granularity. Having small stories ensures regular delivery of functionality throughout the iteration.

The *rules* are *requirements* and the *scenarios* illustrate the expected behaviour of the system once those requirements are implemented. Once a story is finished, all the important information should be captured in the rules and scenarios that it caused us to create. The story itself has no further use and (if possible) should be deleted.

User stories, once refined in a requirement workshop (see *Discovery* [Nagy2018, Section 4.1]), are used to plan and track the delivery of a small functional increment. If you organize your feature files by story, then your living documentation will be structured according to the order in which you delivered the functionality. This is not useful to either the business or the delivery team.

If, on the other hand, you organize our feature files by coherent functionality, you will provide documentation that exhibits both stability and navigability. Stability because a product's functionality doesn't change frequently. Navigability because a reader will be able to reason about the product's functionality. Working in this way you create living documentation that delivers persistent value to all those that use, maintain, and enhance your product.

Consequently, implementation of a user story might cause new feature files to be created, as well as modifications to existing feature files – either by creating new scenarios or by modifying existing scenarios. Any attempt to create a feature file for every story will lead to unusable documentation.

[27] https://cucumber.io/blog/user-stories-are-not-the-same-as-features/

Gáspár's story: Have you ever been to the top of the Eiffel Tower?

One of the most remarkable places I have ever been was the Eiffel Tower. For me, it is a wonderful combination of beauty and engineering challenge. When I met my wife, I talked to her about my Eiffel Tower enthusiasm, but I also had to confess that I did not climb the 669 steps but used the elevator. "Never mind," she said, "You have been there and enjoyed it. From the photos no one can tell how you got to the top."

When you make your software product better, every bit of the implementation is an achievement – like climbing the 328 steps to the first floor of the Tower. Acknowledge the effort (and take a rest), but remember that the capabilities of your product are the more important achievement. It is these capabilities that a feature file should describe.

"I can see how organizing feature files by functionality will make it easier to use them as documentation," says Tracey. "What feature files shall we create?"

"I think that blocklisting and pay-on-collection are separate features, so each should be documented in their own feature file," says Dave.

"Now I understand why each story doesn't have its own feature file," says Patricia, "But I'm still not sure why the *'Don't offer pay-on-collection for restricted customers'* rule isn't part of the blocklisting feature."

5.2 – Separation of concerns

The team discuss which feature the *"Don't offer pay-on-collection for restricted customers"* rule belongs to. When they reach agreement, Patricia summarizes their decision:

"Blocklisting is a generic way of identifying unreliable customers in the system. Preventing an unreliable customer from choosing to pay-on-collection is one strategy for managing that risk. We may decide to implement other strategies in the future. And we may develop other ways of identifying unreliable customers. It makes sense to treat these concerns separately."

"So, we'll create a new feature file and move the *'Don't offer pay-on-collection for restricted customers'* rule into it," says Dave.

"Do you remember the *'Customers can choose for customer-collection orders to be pay-on-collection'* rule that we discovered when we reverse engineered that legacy scenario?" asks Dishita. "I think that it is really a pay-on-collection rule, not an order collection rule. Maybe we should move it into the same feature file as *'Don't offer pay-on-collection for restricted customers'*."

"That's a good idea," says Tracey. "But what should we call this feature?"

"The two rules both relate to pay-on-collection, so we could call it `pay_on_collection.feature`," suggests Dave.

"That makes sense to me," agrees Patricia.

The team restructures their existing feature files to create:

Listing 43 – Restructured customer-collection feature file (`customer_collection.feature`)

```
Feature: Customers can collect their orders

Rule: Any visitor to the website can place a customer-collection order
  Scenario: Unauthenticated customer chooses to collect order
  Scenario: Authenticated customer chooses to collect order

Rule: Unauthenticated customers must supply acceptable contact details
                             when placing an order for customer-collection
  Scenario Outline: Contact details supplied ARE acceptable
  Scenario Outline: Contact details supplied are NOT acceptable

Rule: Customer should be informed when their order will be ready for
                                                        customer-collection
  Scenario: Estimated time of order completion is displayed
```

Listing 44 – Restructured blocklisting feature file (`blocklisting.feature`)

```
Feature: Blocklisting

Rule: Customer's email OR phone must match those on blocklist
  Scenario Outline: Decide if customer is restricted

Rule: Restrictions are independent of authentication status
  Scenario: Unauthenticated customer is restricted
  Scenario: Authenticated customer is restricted

Rule: Blocklist is manually configurable
  @manual
  Scenario: Upload replacement blocklist
```

Listing 45 – New pay-on-collection feature file (`pay_on_collection.feature`)

```
Feature: Customers can choose to pay on collection

Rule: Customers can choose for customer-collection orders to be
                                                        pay-on-collection
  Scenario: Customer chooses to pay on collection

Rule: Don't offer pay-on-collection for restricted customers
  # Two scenarios merged, because authentication is incidental
  Scenario: Restricted customer is not offered pay-on-collection option
```

The documentation should reflect the team's current understanding of the business problems that they are solving. Be prepared to rename, split or delete feature files over time. Likewise, expect to move rules and scenarios to ensure that the documentation remains easy to use.

5.3 – Documentation evolves

> "Now that we've separated the identification of unreliable customers from the strategy of not offering them a pay-on-collection option, it doesn't feel like *blocklisting* is the right word to use any more," says Tracey. "Can anyone think of a better one?"

It is essential that the terminology you use is as correct and unambiguous as you can make it. However, as you develop your product and the team's understanding of the domain deepens, it's likely that you'll refine the terms that you use and the way in which you use them. Arlo Belshee wrote an influential series of articles[28] in which he characterized naming as, "a process, not a single step". The article encapsulates the challenges that naming presents and an approach to coping with those challenges.

The team discusses alternatives to the term *"blocklist"*, but they aren't able to come up with a suggestion that they prefer.

[28] http://arlobelshee.com/good-naming-is-a-process-not-a-single-step/

> "I'm going to go back to the stakeholders and ask them for suggestions," says Patricia. "I'm sure we will be able to find a better term to describe this feature, but not today!"

Terminology isn't the only thing that will get refined as the product grows. There are ways of structuring information that make it easier to interact with – and there are ways that prevent anyone from actually finding the information that they're looking for.

Rules are the building blocks of your specification. They capture the business requirements that need to be delivered and, when combined with illustrative scenarios, become valuable documentation. Feature files are the containers that group rules and make the documentation easy to navigate.

When you first formulate scenarios to illustrate a rule, don't worry too much which feature file you put them in. However, as you continue your project you should consider how a reader of the documentation will expect the information to be structured. It's the same process you would go through if you were writing a report or a user manual. Expect to create new feature files and move rules and scenarios from one feature file to another.

Tool support

As you've learned in earlier chapters, our feature files are written using a language called Gherkin. Many word processors and text editors simply treat the feature file as a block of text.

Some text editors recognize Gherkin. They are able to help you read and write feature files by coloring keywords or formatting data tables. The most advanced editors automatically build a library of all steps that you have already defined and offer auto-completion when you start a new line of a feature file.

There are editors that can support you with basic rewording, but at the time of writing there are no tools that provide any support for the sort of Gherkin restructuring that the team has just undertaken. Software developers have got used to tools that help them *refactor* their code without resorting to error-prone, manual activities (such as find-and-replace or copy-and-paste). We hope that similar tool support for Gherkin becomes available soon.

5.4 – Documenting the domain

"One thing that I would like to understand today is the meaning of *'@manual'* in the feature file," says Patricia.

"That's a Gherkin tag," says Dishita. "We're using it to indicate that it wasn't worth automating that particular scenario."

"So, is *'@manual'* a special tag defined by Cucumber/SpecFlow that indicates a scenario shouldn't be automated?"

"Good question," answers Dishita. "The answer is 'No'. The tags are defined by the team, not Gherkin or Cucumber/SpecFlow. A tag is simply an `@` character followed by a label. In this case, we have decided to use the label 'manual' to indicate that our build process will not try to run the scenario as an automated test."

"Where is that documented?" asks Patricia. "What other purposes could tags be used for? Have we decided to use any other tags for specific purposes?"

Dishita and Tracey look at each other. "That's at least three good questions," says Dishita. "Let's talk about them one at a time."

"We haven't documented the *'@manual'* tag yet," admits Dishita, "but we should. I'm not sure where, though."

"We haven't documented what we mean by any domain terms yet either," says Patricia. "We've written rules and scenarios, but to understand a specific term someone would have to read through all the relevant feature files."

"In the old days we used to have something called a data dictionary," says Dave. "It was where you would look to find the meaning of a piece of information. That feels similar to what we need here."

"Yes," says Ian. "What we need is a glossary: *'an alphabetical list of words relating to a specific subject... with explanations'*. It would include brief definitions of every piece of domain terminology that we use in the feature files."

"Are tags part of the domain?" asks Tracey.

"Well, the feature file has to be business readable, so everything in it should be rooted in the business domain," says Dishita. "That means that tags should also be part of the domain, so they should be documented in the same way as any other concept is."

"Our glossary should include *'@manual'*, along with blocklist, customer-collection, pay-on-collection, and the others," says Ian.

Seb's story: Living glossaries

A glossary is an extremely useful resource, but when manually compiled, the glossary can easily become out-of-date.

In *Living Documentation* [Martraire2019], Cyrille Martaire explains that, "it's important to keep the knowledge in one single source of truth and publish from there when needed". Although there is currently no existing tooling, it is relatively simple to develop a small script that compiles a list of terms that could be included in the glossary by looking through all feature files. The list tells us which terms used in the documentation aren't explained in the glossary, but needs to be manually reviewed to remove irrelevant or duplicate entries. The script could also check that all terms defined in the glossary appear somewhere in the text.

We used a similar script to compile the index for this book. It has saved us many hours of thankless cross-referencing and has caught many errors and omissions.

"Let's compile a glossary of domain terms in a markdown file," says Dave. "I'll create a simple script to compare it to terms in the feature files. We can add more functionality once we've used it for a bit."

"I can help by writing the explanations for the terms," says Tracey.

"And I'll review the explanations to make sure they match with my understanding," says Patricia. "In fact, we should all review them. It's a shared understanding we're aiming for, after all."

Glossary

Please check Appendices, Listing 54 for the final MarkDown glossary file the team created.

- **Authenticated [customer]** – Customer who has registered in the application earlier and logged in with the registered credentials (e.g. email and password).
- **Blocklist** – List of contact details to identify unreliable customers. (Currently we store email addresses and phone numbers on the blocklist.)
- **Contact details** – Details that allow the customer to be contacted in relation to a particular order. For authenticated customers, the contact details registered to their account are used. For unauthenticated customers the contact details must be provided with the order.
- **Customer** – Person who visits the *Where Is My Pizza* site to order pizza and other items sold. Unauthenticated (anonymous) users are also treated as customers.
- **Customer-collection** – A special delivery method where the customer picks up the ordered items themselves.
- **Delivery method** – The selected method for delivering the ordered items.
- **Order** – Food and beverage items the customer has ordered.
- **Order completion** – Event (time) when order processing has been finished by WIMP, usually when the items have been delivered. In some cases it can be earlier (e.g. *customer-collection*).
- **Pay-on-collection** – Payment method where the customer pays (with cash or card) when they pick up the order. This method can only be used for *customer-collection*.
- **Payment details** – Details required to fulfil the payment. The exact details depend on the *payment method*, e.g. for card payment we need the card details, but for *pay-on-collection* there are no extra details needed.
- **Payment method** – Selected method of payment for the order.

- **Restricted [customer]** – Customer whose contact details are on the blocklist. They can only use a reduced set of features the site offers (e.g. they cannot use pay-on-collection).
- **Unauthenticated [customer]** – Customer who has not logged in.
- **Unrestricted [customer]** – Customer that can use the full set of features of the application.
- **Visitor** – See *Customer*.

5.5 – Tags are documentation too

"Now for your other questions," says Dishita. "Tags have multiple uses. They are annotations that tell us something about a scenario and mostly we use them for grouping scenarios. We use *'@manual'* to indicate which scenarios will be tested manually, but there's no limit to the number of ways that an organization may want to subdivide their scenarios. So far we've only used *'@manual'*."

A little bit more about tags

- Tags are case sensitive – *"@manual"* is not the same as *"@Manual"*.
- There is no limit to the number of tags that can be applied to a block.
- Multiple tags can be applied on the same line, or on consecutive lines.
- Tags can be applied to a *Feature*, *Rule*, *Scenario*, *Scenario Outline*, or *Examples* block.
- Tags applied to a *Feature* are also applied to all *Scenario*s and *Scenario Outline*s within the feature file.
- Tags applied to a *Rule* are also applied to all *Scenario*s and *Scenario Outline*s within the rule.
- Tags applied to an *Examples* block are applied to all scenarios generated from that block.

Common ways that tags are used include:

- selecting a subset of related scenarios (e.g. *"@release:42"* ⇒ changes in a release, *"@smoke"* ⇒ smoke test, *"@fast"* ⇒ fast feedback),
- creating targeted reports for different areas of the business (e.g. *"@pricing"* ⇒ all pricing related scenarios),
- indicating scenarios that are reliant on external systems (e.g. *"@twitter"* ⇒ related to Twitter integration).

Using tags for the purposes mentioned above allows us to group related scenarios based on some domain detail. A tag can then be used to easily find related scenarios or to execute a specific group of automated scenarios (e.g. get quick feedback about the health of the system by running only the scenarios that are tagged with *"@fast"*). If you think of the feature files as structured documentation, then tags are a way of creating entries in the documentation's index.

The primary purpose of the Gherkin feature files is to provide a business readable living documentation. Because of that the readability of the tags should be an important concern. However, automated scenarios also need to be integrated into the existing tool-chain that is used to manage the project: application lifecycle management (ALM) tools, test execution, continuous integration (CI) and reporting frameworks. Integration with these tools might require the team to define additional tags, but even in this case we suggest trying to create tags that express the business needs or the stakeholder value behind them. We should avoid over-decorating the scenarios with tags that only tools or the delivery team can understand.

There are three common usages of tags where you should be even more careful, outlined in the following sections.

Tracking scenario automation lifecycle with tags

We described the BDD process in *Discovery* [Nagy2018, Section 4.1]. As the scenarios represent high level specification elements of the system, the completion of a formulation–automation–implementation cycle for a scenario might take several days. A scenario that has just been formulated will fail if you attempt to execute it as an automated test. The same is true while the team is developing the automation and production code to fulfil the expectations described in the scenario.

These failures do not indicate a problem, therefore they should not be mixed with results from the scenarios that are already "done". We like to keep the results of the "done" scenarios "always green".

Test execution frameworks do not track or detect the different states of the scenario automation lifecycle, but this can be represented by tags. You can define your own set of tags for this. For example, we have used *"@formulated"* to mark the scenarios where the automation has not even started and *"@inprogress"* for scenarios where automation or production code is known to be incomplete. Scenarios that have neither of these tags are considered "done" and should pass when executed.

We prefer not to use an explicit *"@done"* tag, because that generates noise in the living documentation and impacts readability.

Tagging scenarios with external identifiers

As we discussed in Section 5.1, "User stories are not the same as features", although the scenarios are usually created within the scope of a user story, this relationship is not important after the story has been delivered. However, the scenarios might be related to other work items tracked in external ALM tools, such as Jira or Azure DevOps. Collaborative scenario authoring tools might be able to track these references for you, but if the scenarios are edited as plain text, tags can be used to record the connections. For example, a tag *"@feature:1234"* might indicate that the scenario is related to the "feature" work item tracked in the ALM tool with ID 1234.

Obviously such tags are not very "readable" (although the team might need to use these IDs anyway), therefore the team should carefully decide which relationships they really need to record. Although recording the reference to the originating user story work item might seem useful (to "achieve" traceability), consider whether this information will really be used. Your goal is to minimize unnecessary distractions for the readers.

Tagging scenarios with automation details

As most of the scenarios are going to be automated, there will be a set of technical automation details that are going to be attached to the scenario, such as the automation strategy, the interface of the application that is going to be targeted by the automation, or even some specific technical details.

Turning the scenario into a technical asset by representing these as tags is dangerous, because it might prevent stakeholders from collaborating effectively. For example, the tag *"@use_repository_mock"* might not be understandable to all readers, but

annotating scenarios with automation-related tags is sometimes necessary. For example, if the automation of the scenario requires a browser to be ready in order to execute the scenario, you might need to indicate that with a tag (e.g. *"@browser"*).

The *"@manual"* tag that was used by the WIMP team is also related to automation, but it is less distracting, because its meaning is clear to the entire team. At the end of the day, stakeholders are not only interested in the fulfillment of business expectations, but also in the quality of the result in general.

If these tags are kept at a high strategic level, they can contribute to a shared understanding of our quality assurance process and the related risks (e.g. *"@manual"* scenarios are not guaranteed to be tested after a successful CI build; *"@browser"* scenarios are for higher level checks, etc.). When defining automation-related tags always make sure that they are understandable by all stakeholders.

Gáspár's story: Tags to indicate semi-implicit context

I have been working on a project where the majority of the features were only accessible for authenticated users, but not all. We considered having an authenticated user as part of the implicit context (see Section 4.11, "Setting the context"), but we found that people naturally assumed that a user was unauthenticated unless we explicitly mentioned that they had logged in. We first defined a *"Given an authenticated user"* step and used it extensively in the scenarios, but it sounded too verbose.

After considering a few alternatives, we decided to use an *"@authenticated"* tag to annotate those scenarios that need a logged in user. This supported the readability and understandability of the scenarios without the verbosity of the repeated step.

Later we could also use this tag to have an overview of what features are only available to authenticated users.

Tags are a powerful tool to establish arbitrary groupings of scenarios, but if and only if they do not distract the readability and understandability for all stakeholders. You should always discuss new tags with the entire team and, if adopted, their use should be documented.

> "Since the meaning of tags may not be obvious to all readers of the feature files, we should document them in the glossary," says Patricia.
>
> "That's a good idea," agrees Tracey, and she adds an entry to the glossary.

- **@manual** – The scenario is verified by manual checks and has to be excluded from automated test execution

You can apply as many tags to a scenario as you want but this can quickly harm readability, so, ensure that there is value to every tag that you create, and don't be afraid to remove tags when they are no longer useful.

5.6 – Journey scenarios

> "I'm concerned about the customer experience of blocklisting," says Patricia. "Our scenarios are really useful, but I'm struggling to see the bigger picture."

The scenarios that the team have been writing are essential for capturing a persistent understanding of specific behaviours that the system implements. Since they have been following the BRIEF principles, each scenario is focused on illustrating a single business rule. This is incredibly important, but it misses another critical perspective – how can a user interact with the system to achieve their high level goals? This is the bigger picture and, without it, it's hard to get an overview of the functionality that the system delivers.

> "Why don't we talk this over with Ulisses, the new UX (user experience) guy?" suggests Tracey. "I bet he'll have an opinion."

The team contact Ulisses and explain their concerns to him.

> "The concept of a *user journey* has been around for a while," explains Ulisses. "They're used in the UX community to document the step-by-step journey that a user takes to reach their goal. I love the scenarios that you've written so far, but they document individual steps, not the journey."

"How do you design and document user journeys?" asks Patricia.

"There are plenty of different ways to document user journeys, but they're mainly quite visual because there are so many different paths through modern software applications," replies Ulisses. "I don't think the focused scenarios that you're using could be a practical replacement. On the other hand, our user journey maps aren't suitable for business people, so there's definitely a gap that needs to be bridged."

User story mapping

Teams that use *user story mapping* (as described in Jeff Patton's book of the same name [Patton2014]) could use their user story maps to identify the user journeys that could be documented with journey scenarios. If you don't have a user story map, then you'll need to work with business stakeholders to identify the most important user journeys.

"What journeys would you like to see documented, Patricia?" asks Tracey.

"I'd like us to document a typical pay-on-collection journey," says Patricia. "It would also be helpful to see how a restricted customer's journey is different."

"That sounds a bit like *use cases* I've read about in *Writing Effective Use Cases* [Cockburn2016]," says Dave. "We could write a scenario that describes the typical journey and then write others that describe exceptions and alternative paths."

"This will be a different sort of scenario to the ones we have been writing so far," says Tracey. "Shall we call it a *journey scenario*?"

"Good idea," says Dave.

The team agrees to give it a try. They quickly write the following journey scenario:

Listing 46 – Pay-on-collection journey

```
@journey
Scenario: Pay-on-collection journey
  Given Reggie visits our website
  When he adds a pizza to his basket
  And he chooses to checkout as a guest
  And he enters valid contact details
  And he chooses pay-on-collection
  Then he receives an order confirmation
```

"I don't like all those *And*s," says Dishita. "Is there another way to do this?"

"We could replace all the *And*s with *When*," says Dave, "But that wouldn't read like English. There is an alternative, though, using asterisks."

Listing 47 – Pay-on-collection journey using asterisks

```
@journey
Scenario: Pay-on-collection journey
  * Reggie visits our website
  * he adds a pizza to his basket
  * he chooses to checkout as a guest
  * he enters valid contact details
  * he chooses pay-on-collection
  * he receives an order confirmation
```

"Does that behave in exactly the same way?" asks Patricia, "Because, if it does, I like it."

Asterisks in Gherkin

Using an asterisk in place of *Given*, *When*, or *Then* removes the context, action, outcome semantics that help give readers a full understanding of the system's expected behaviour. We never use asterisks when writing illustrative scenarios, but journey scenarios benefit from their conciseness.

The asterisk is available for use in Gherkin, no matter what natural language the feature is written in.

"Yes, the scenarios behave identically whether you use one of the keywords or an asterisk," says Dave. "I wouldn't want to use asterisks when writing focused illustrative scenarios though."

"I agree," says Patricia. "Using the Gherkin keywords for illustrative scenarios and asterisks for journey scenarios will also emphasize the difference."

"Shall we try writing a journey scenario for a restricted customer?" asks Dave. "What would the differences be?"

"The only difference is that restricted customers can't choose to pay-on-collection," says Tracey. "How would we show that?"

The team sketch out another journey scenario:

Listing 48 – Restricted customer's pay-on-collection journey

```
@journey
Scenario: Restricted customer's pay-on-collection journey
  * Brian visits our website
  * he adds a pizza to his basket
  * he chooses to checkout as a guest
  * he enters valid contact details
  * the contact details are on the blocklist
  * he can't choose pay-on-collection
```

"This doesn't feel right," says Dave. "Firstly, it's not a complete journey, because Brian hasn't completed checkout. Secondly, the important difference is already clearly captured in the scenario that illustrates the *'Don't offer pay-on-collection for restricted customers'* rule."

"You're right," agrees Patricia. "The focused illustrative scenario already documents how a restricted customer's experience is different."

> "Right!" says Tracey. "*Illustrative scenarios* are focused examples that illustrate a single rule, while *journey scenarios* give an overview of a user journey and will exercise many rules. This journey scenario is just a longer version of an existing illustrative scenario. Let's delete it."

Journey scenarios are extremely useful when trying to document the big picture, but quickly get repetitive and costly to maintain if there are too many of them. The intent of the journey and illustrative scenarios is to collectively document the behaviour of the system. It is inefficient and ineffective to attempt to use journey scenarios to exhaustively test your system. On the other hand, a carefully curated, small collection of journey scenarios can be invaluable:

- to give an overview of the core business processes
- as a set of smoke tests to confirm successful deployment

Obviously, journey scenarios don't conform to all the BRIEF principles. Specifically, they will be neither *brief* nor *focused*. You should continue to ensure that journey scenarios conform to the other BRIEF principles. However, since the intent of a journey scenario is to document the journey, the need for concrete *real data* is much reduced. For example, the Listing 47 has very few concrete details (such as what pizzas are ordered or the precise address for delivery), because it is intended to act as high level documentation of a user journey, not as detailed illustration of business rules.

Risks associated with journey scenarios

While there is value in journey scenarios, many BDD teams find that illustrative scenarios are all they need. If your team decides to invest in journey scenarios, don't allow your focus to be distracted from illustrative scenarios. *Never* automate a journey scenario until all of the illustrative scenarios for the features that it makes use of have been automated.

- Journey scenarios are *slower to execute* than illustrative scenarios.
- Journey scenarios are *harder to debug* than illustrative scenarios.
- Journey scenarios are *more costly to maintain* as the product evolves.

Journey scenarios live at the top of the test automation pyramid[29] and consequently should be used sparingly!

Journey and illustrative scenarios have different intents. The team has signalled a journey scenario by using the *"@journey"* tag and by using asterisks instead of keywords. It's advisable to keep illustrative and journey scenarios in separate feature files as well.

5.7 – Structuring the living documentation

"Which feature file does this journey scenario belong in?" asks Dishita. "Is it even a feature?"

"The ability to order pizzas online is certainly a feature from a customer's perspective, but it's a much larger feature than we're used to working with," says Patricia.

There is no defined size for a feature. Developers and testers might think of the smallest piece of behaviour as a feature, while customers are normally interested in the ability to complete much larger activities. Feature files are intended to be business-facing, so the team needs to identify features that deliver distinct pieces of business functionality.

[29] https://cucumber.io/blog/bdd/eviscerating-the-test-automation-pyramid/

The illustrative scenarios that we have been looking at for most of this book illustrate fine grained rules, which in turn specify the behaviour of components within the complete application.

The journey scenarios that the team is currently discussing are very different. They don't illustrate the application of a single behaviour, they document the interaction of a large number of behaviours that, together, deliver valuable outcomes for some stakeholders. Accordingly, journey scenarios should not be written in the same feature files that we write illustrative scenarios in.

> "It feels like journey scenarios should have their own feature files," says Patricia.

Navigation

> "We need to organize our feature files in a logical way," says Tracey. "There's nothing worse than trying to find a specific piece of information in poorly organized documentation."

> "A typical way to structure documentation is to use chapters, sections, sub-sections and so on," says Dave.

Each scenario documents an example of the system's behaviour. Scenarios live in feature files, which in turn live in source control. The feature files collectively document the behaviour of the system, but are only useful when people can easily navigate to the part they are interested in. A hierarchical tree of folders in source control provides a simple way to structure our documentation and is identical to the traditional documentation (Figure 11) that Dave is thinking about.

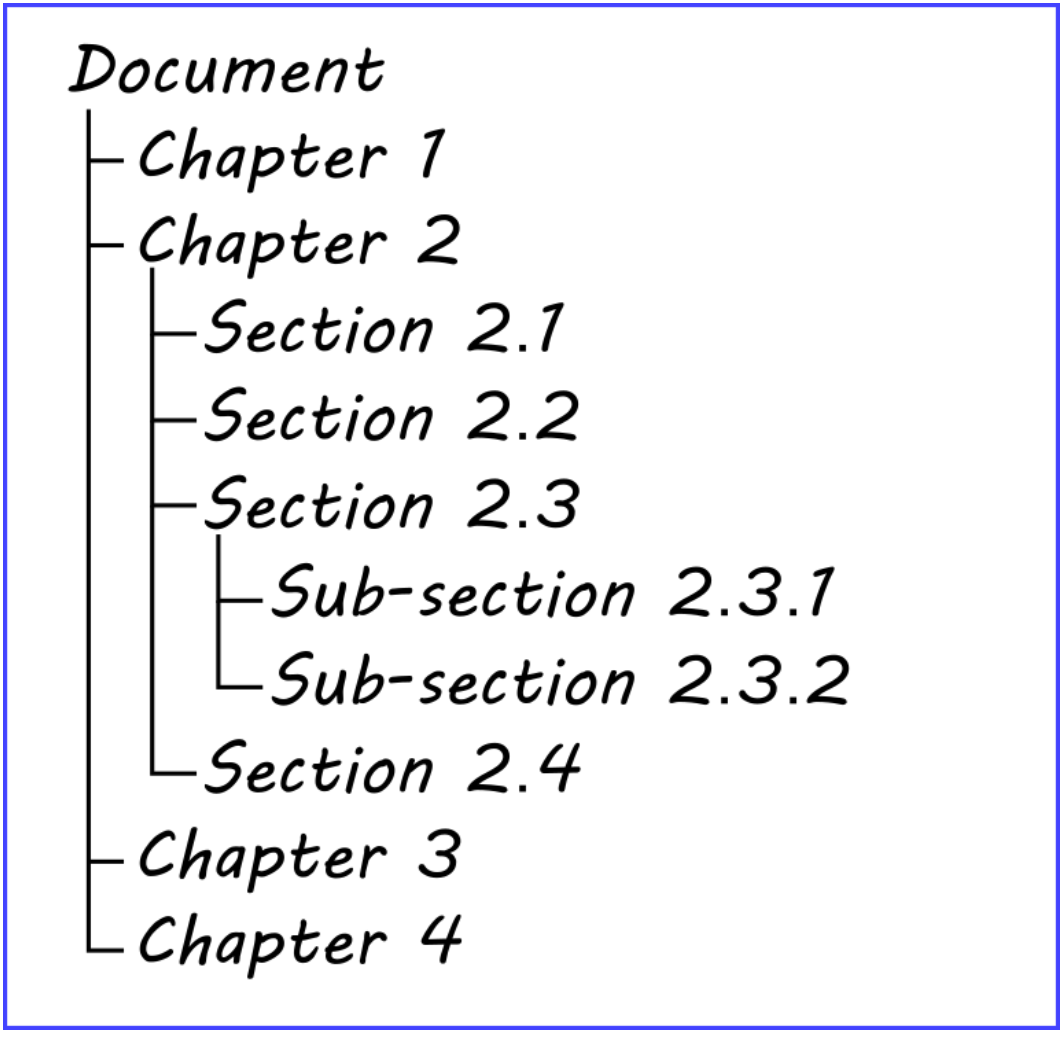


Figure 11 – Traditional documentation

"Can you show me how we might use folders to structure the feature files that we have at the moment?" asks Patricia.

"Sure," says Dave and he grabs a pen and starts to sketch:

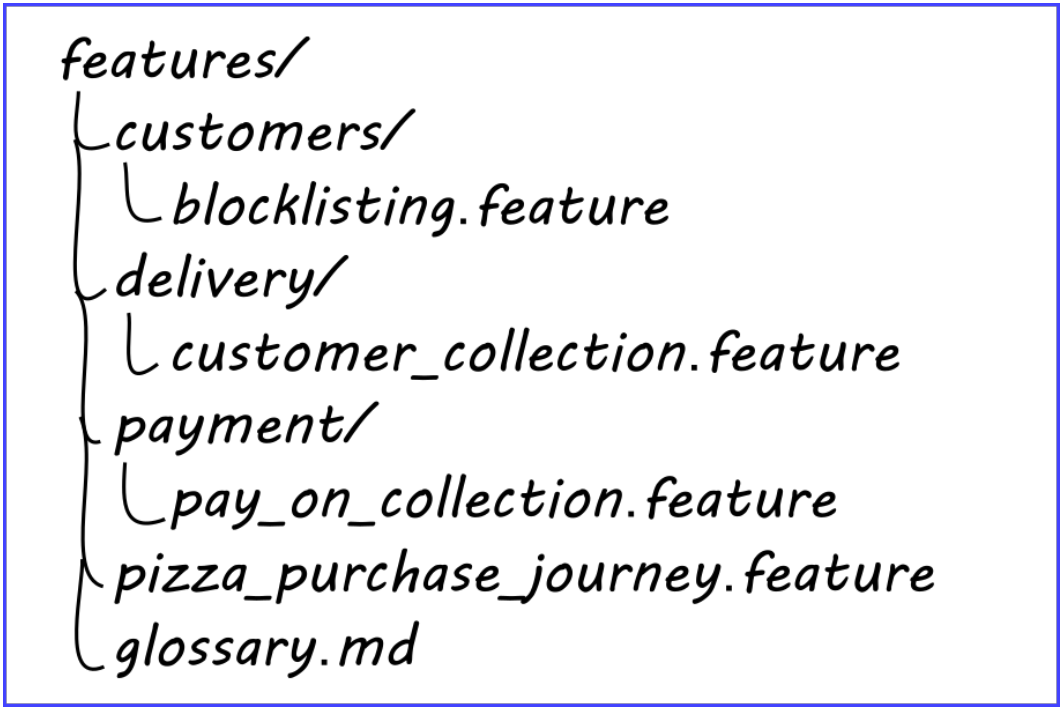


Figure 12 – WIMP documentation

> "There's only one feature file in each folder at the moment, because we've only just started using BDD," says Dave, "but it will fill up as we discover more behaviour to implement."
>
> "There's also all the existing behaviour that we could document," says Tracey.

Tracey is right – the team implemented a lot of functionality in the WIMP product before they started using BDD. All that *legacy code* could be documented as well. We'll cover that in Chapter 6, "Coping with legacy".

Zoom-in/out

> "Will the documentation always be only one level deep?" asks Ian, "Or are there cases where folders will be deeply nested?"

"I can imagine we'll make use of nested folders quite soon," says Dave. "There are lots of behaviours that relate to customers, and we can use the folder structure to communicate the level of detail – the deeper the folder, the more fine grained the rules are. Here's how the Customers folder might look."

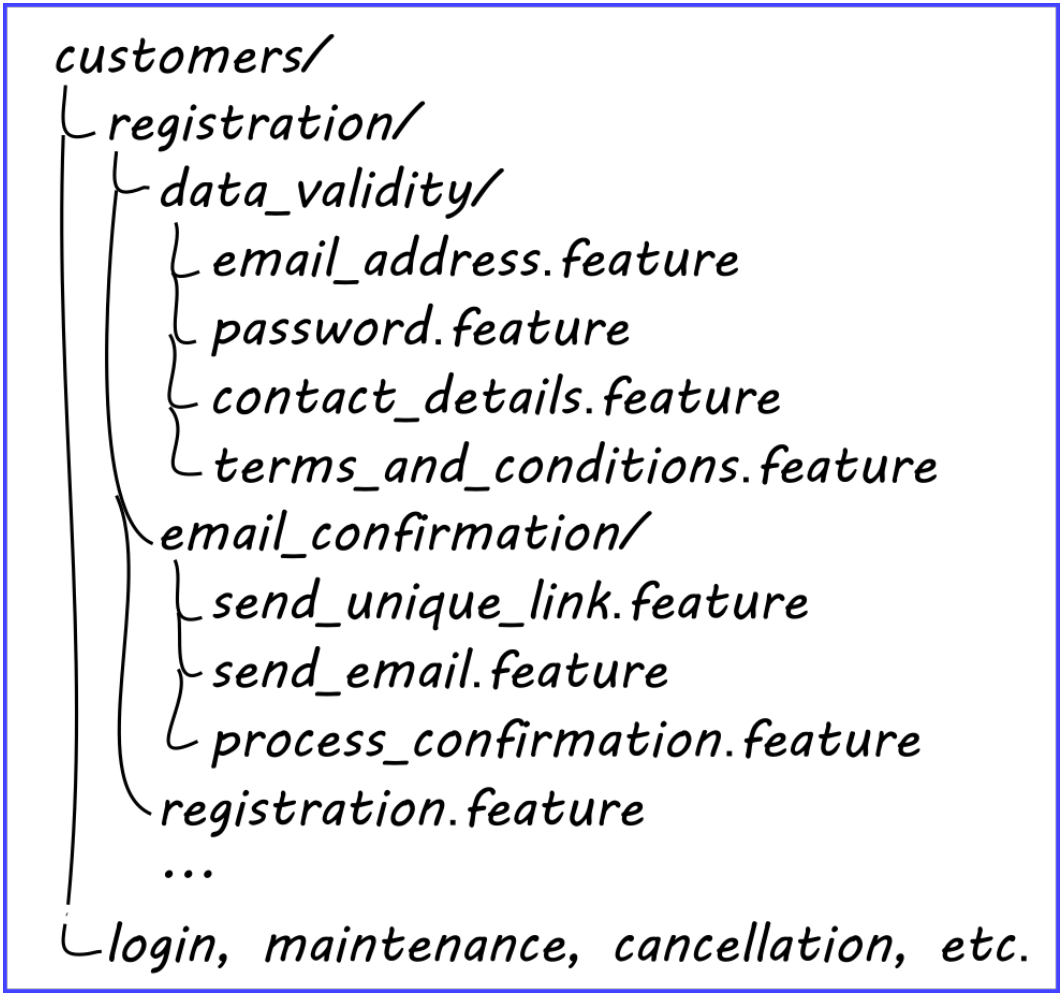


Figure 13 – Customer domain documentation

What Dave is demonstrating is that, even in a simple system, there are many levels of detail. Some readers will be looking for the big picture – they should not need to

travel far from the root of the documentation. Others will be interested in specific details – they should be able to navigate to the relevant section easily.

This is similar to how we use online maps. We can get a good idea of where New York is relative to Washington without needing much detail. If we actually want to drive from one to the other, we will need to zoom in and get details about which route to take.

5.8 – Documenting shared features

> "I see there is a `send_email.feature` inside *'email confirmation'*. Won't there be other features that make use of email sending?" asks Dishita. "Will we create a copy of that feature file wherever that functionality is used?"
>
> "That would be really hard to maintain," says Tracey. "There must be a better way."
>
> "I agree," says Dave. "When we find a feature that is used by several parts of our system, we should ensure that a single feature file describes the behaviour. Then, if it needs to change, we only have to change the documentation in one place."

It's really common to use shared features from many parts of the system. Sending an email is one example, but just in Figure 13 there are other features that will be reused. For example:

- *"Change password"* will depend on rules for *"Password"* validity,
- *"Reset password"* will probably depend on *"Generate unique link"*.

It's easy to move shared features into a separate part of the tree, but at the time of writing there's no special way to link from one feature file to another in Gherkin. For now we recommend that you document the dependencies of a feature in the feature file description (see Section 3.4, "The feature file"). The best information to include is the name and relative file path of the dependency (see Listing 49).

"We could extract *'Send email'* into a communications section," says Dave, and he sketches it out for the team to see.

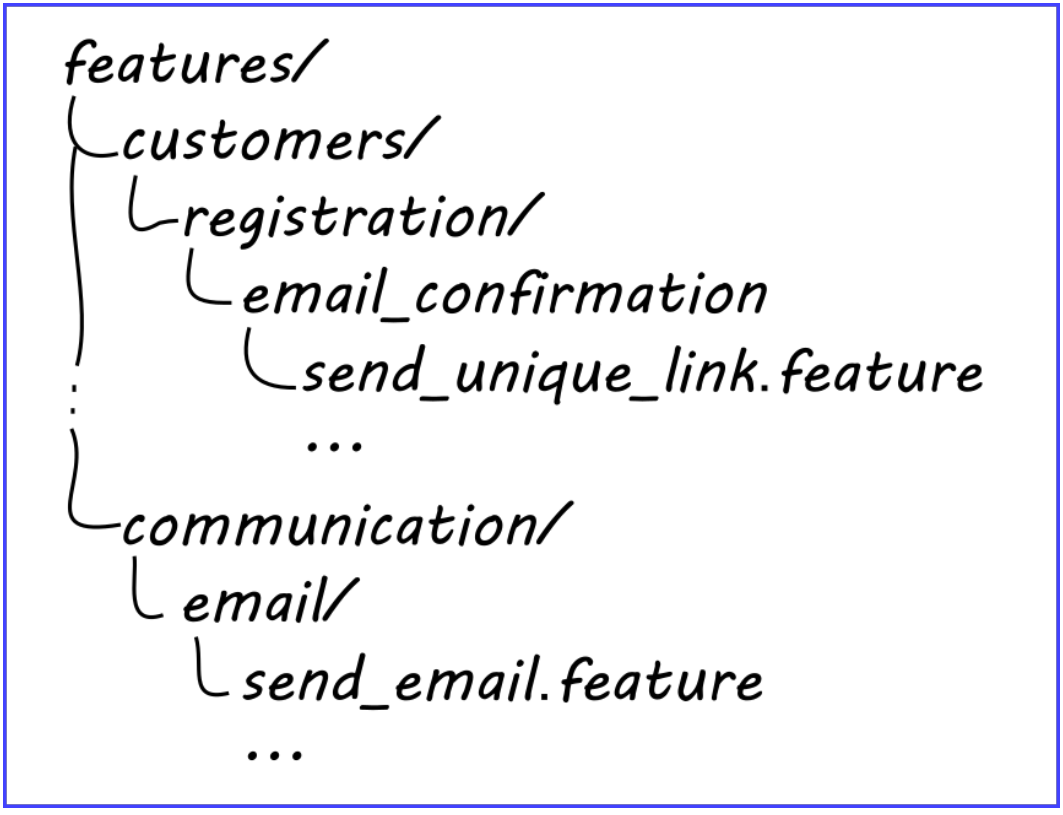


Figure 14 – Links across folder hierarchy

"Then in `send_unique_link.feature` we could document the dependency like this:"

Listing 49 – Documenting dependencies

```
Feature: Send unique link

This feature is dependent on:
- Send Email - ../../../communication/email/send_email.feature
```

Markdown is coming

The Cucumber open source team have an extensive roadmap which includes many improvements. One is that Cucumber will support Markdown in a feature file's textual documentation. This will allow links to be displayed in a much more readable way.

"I see how this could work," says Patricia. "Let's agree to structure our feature files like this from now on."

5.9 – Targeted documentation

"Are we going to use feature files to document our APIs for third party integrators?" asks Dishita.

"That's a really great question," says Patricia. "We would certainly want to provide them with readable, reliable documentation."

Dishita has uncovered another important aspect of documentation – different consumers require different information. To keep the documentation readable, we need to ensure that sales executives don't get forced to read API documentation, while integrators may not be interested in all the details of our application.

"In the old days we'd either print the API documentation separately or put it in an appendix," says Dave. "We could do the same here too and add a new top level folder called `api`."

Seb's story: Actuaries are not the same as POs

I worked for an insurance company that was using Cucumber. We found that the product owners and actuaries were interested in completely different aspects of the system's behaviour. Not only that, but they each used different terms from the business language. To satisfy the needs of both, we created feature files specifically tailored to each audience.

However, we didn't create separate top level folders (like Dave has suggested for APIs). Instead, we organized all the feature files within the same hierarchy, using specific subfolders to signal the intended audience.

Feature files are primarily documentation of a shared understanding between the business and delivery teams. However, we need to appreciate that the *business* includes multiple stakeholders with differing documentation needs. With this in mind, we can make use of the file system or tagging to structure the documentation in a way that facilitates each stakeholder being able to easily find the documentation that is relevant for them.

You can read more about how to document APIs with scenarios in *How to test APIs with Gherkin?*[30].

5.10 – What we just learned

In this chapter the WIMP team focused on the ongoing challenges they needed to address to create usable living documentation. Usable living documentation allows all stakeholders to easily find the information they are looking for. To achieve that we have seen several practices.

First we learned that the requirements (the core building blocks of our living documentation) are not the user stories, but the rules and their related scenarios. The WIMP team reviewed the scenarios they formulated for the user story and rearranged them according to the functionality of the system they contribute to. The team realized that some of the scenarios of the recent stories belong to a separate

[30] https://specflow.org/blog/how-to-test-apis-with-gherkin-givenwhenthenwithstyle/

feature, so they extracted them to a new feature file. Similar review and refactoring activities are common among successful BDD teams and can be used to keep the documentation relevant.

The team then discussed the content of the feature files. Even if some domain terminology was used in earlier discovery workshops, it is during formulation that they write documentation using only unambiguous, business readable terms. The team must agree what each term and tag they use means. Building up a glossary is a useful way to document this agreement.

Illustrative scenarios are useful to document our understanding of a particular business rule. The system is composed of many rules, so there is a need to document the "big picture" using examples. The Gherkin syntax and the scenario steps that were defined for the illustrative scenarios can be used to describe higher level scenarios called journey scenarios.

Journey scenarios are similar to illustrative scenarios, but not all of the BRIEF principles are applicable for them. Journey scenarios are less maintainable and cannot be used to "drive" the development process. Defining a few journeys as scenarios serves the understandability of the big picture and might work well as "smoke tests".

As a system grows, a flat structure of feature files is not enough. Related feature files can be grouped together into higher level features or system areas – represented by the folders of the file system where the feature files are stored. The resulting hierarchical structure is similar to the hierarchical structure of traditional documentation – they both support readers being able to easily find the information they are looking for.

Although the hierarchical structure works well as a core structure, the information we need to document is not strictly hierarchical. Tags are powerful tools to achieve arbitrary groupings of scenarios, but the team should also be prepared to use some common structuring patterns. In this chapter we detailed how shared features and targeted features can be integrated to the structure.

Practicing the concept of evolving documentation is easier when the behaviour of the system has been documented using BDD scenarios right from the beginning, but it is never too late to introduce business readable living documentation of the system. In the next chapter we discuss how formulation can be practiced for legacy applications.

Chapter 6 – Coping with legacy

The WIMP team is adopting BDD partway through delivery of their product. The code that has already been written can be thought of as *legacy code*. If we use Michael Feather's definition of legacy code, "legacy code is code without tests" (*Working Effectively With Legacy Code* [Feathers2005]), then the majority of code that exists is legacy code. Therefore, most software professionals spend much of their career working with legacy code. Teams that find a way to adopt BDD practices while working with legacy code will find that the benefits are significant.

In this chapter we look at how BDD practices can help when working with legacy code. There's a lot more about legacy code that remains out of scope for this book.

6.1 – BDD on legacy projects

In Chapter 2, "Cleaning up an old scenario" we saw the team reverse engineer the behaviour of the code using example mapping, which is a technique that we typically use during discovery. They went on to formulate illustrative scenarios from the concrete examples in the example map. The knowledge recovered during this process will help both now and in the future. For now, it helps developers verify that changes they make to the code have not broken the system. In the future, it will enable any stakeholder to confidently determine the current behaviour of the system.

By definition, however, this process is not Behaviour Driven Development. Since those parts of the system had already been implemented, discovery and formulation were being used to recover and document knowledge, not drive development of the system.

Seb's story: BDD is not about "the BDDs"

At one client, I was confused by hearing their teams talk about "the BDDs". At retrospectives one of the subjects brought up was, "the BDDs broke three times this

> iteration". At standup, a tester said, "the BDDs aren't ready yet". It turned out that whenever they mentioned "the BDDs" what they really meant was, "the Gherkin scenarios". What made this doubly confusing was that the team hadn't adopted a BDD approach – they were simply using Cucumber for test automation.
>
> Cucumber and Gherkin were created to support a behaviour-driven approach to software specification and delivery. However, many teams that use Cucumber use Gherkin scenarios to test functionality that has already been implemented, rather than to drive its development. While Cucumber can be used in this way, it has nothing to do with BDD. BDD expects discovery and formulation to take place *before* writing the implementation code needed to satisfy the requirements being specified.

Recovering knowledge after functionality has been implemented is far less effective than documenting the required behaviour before implementation. As well as the unnecessary cost of rediscovering knowledge that was already known at the time of implementation, it has a negative effect on the architecture of the implementation.

Systems that are not implemented in response to failing scenarios and tests tend to be more *tightly coupled* than those that are implemented by following a behaviour driven or test driven approach. Without significant *refactoring*, tightly coupled architectures are often only testable by using end-to-end automation. As we discussed in Section 5.6, "Journey scenarios", we should minimize the amount of end-to-end automation.

6.2 – Incremental documentation

Very few teams should attempt to apply discovery and formulation to the entire legacy codebase. Instead they will need to adopt a risk-based, incremental strategy. The strategies described below work well for many teams, but you may find other approaches more suitable to your context.

New behaviours

The WIMP team started by applying discovery and formulation to an area of the codebase that they were about to modify. This is a common approach which has the benefit of increasing the team's confidence an area of functionality that they will soon be changing.

Once knowledge of the existing behaviour has been recovered and documented, the team can then implement the behaviours following a behaviour-driven approach. Discovery will generate an example map, formulation will create business readable documentation, and automation will guide the implementation.

80/20 rule

For teams that prefer documentation that gives broad coverage of the system, the approach described above is not suitable. Instead they can utilize the 80/20 rule (also known as the Pareto Principle[31]) that says, "80% of the benefit can be realized from 20% of the work". With this approach, domain experts would identify a few core behaviours across the application to document.

Resist the temptation to document journeys rather than focused behaviours. The need to minimize the number of journey scenarios should continue to be respected, otherwise there will be a significant maintenance burden to pay in the future.

Probability of change

There are often significant parts of a system that rarely change. There is far less value documenting these behaviours than there is in documenting areas of the system that change frequently. Consider using the team's source control system to systematically identify the parts of the system that often change and document them first (see Thornhill's *Code as a crime scene* [Thornhill2015], for example).

[31] https://en.wikipedia.org/wiki/Pareto_principle

Gáspár's story: The second best time

Once I was involved in a project where we needed to maintain a legacy system. Not a dream job, but luckily we only needed to do a few small fixes and feature requests. I already had a couple of years of BDD experience, but I thought we could just "hack in" what was needed without following a proper quality strategy. However, the domain was complex and mainly unknown to most of us, so we often got problems even though the changes had seemed small.

Facing these challenges we decided to give BDD a try. I was still unconvinced, so it was hard to get started, but the result was a positive surprise. The changes got fixed much quicker and with just a few dozen well selected scenarios we even managed to stabilize our releases.

There is a Chinese proverb, "The best time to plant a tree was 20 years ago. The second best time is now."

This is true for BDD as well! The best time to start with BDD is at the beginning. The second best time is now.

6.3 – Making use of manual test scripts

> "What are we going to do with all the manual test scripts that we've been using until now?" asks Tracey.

The team members look at each other. It's a very good question.

Conflicting goals

The illustrative scenarios (which adhere to BRIEF) are optimized for readability and feedback. The assumption is that they will run fast, so we can run as many of them as we need. That, in turn, allows each scenario to be focused on a single behaviour, which means that when the scenario fails, we know exactly what caused the failure.

Manual test scenarios, on the other hand, are optimized for efficient use of a tester's time. The assumption is that we should focus on minimizing repetitive, manual tasks

(to save time and money). That leads us to design test scripts which exercise many behaviours in fine detail, which means that understanding and maintaining the script is more challenging.

Because of these conflicting goals, it is almost never a good idea to naively formulate manual test scripts simply to automate them. Teams that have tried this approach have found that the formulated scripts have little value as documentation, but they do impose a significant maintenance burden. We saw an example of this sort of scenario in Chapter 2, "Cleaning up an old scenario", where it took the team significant effort to understand what the expected behaviours were and then extract BRIEF illustrative scenarios.

Not all journeys are informative

Initially, it may seem attractive to formulate manual test scripts with the intention of using them as journey scenarios. Each manual test script does take the user on a journey through many behaviours, but there is an important distinction. A journey scenario should focus on the overall flow of the journey leaving the detailed exploration of each behaviour to illustrative scenarios. However, manual test scripts usually focus on the fine details of each behaviour that they exercise in the course of the journey.

Manual test scripts may capture valuable information, both about the journey and about the detailed behaviours that they exercise. To realize that value, the team would need to assess and analyze each script in turn. Once they understand the value of a particular manual test script, they can formulate relevant illustrative scenarios to replace it. A journey scenario should only be created from a manual test script if it will contribute to the big picture.

Short-cuts don't always save time

When a manual test script's purpose is clear, it can be used as the basis for an automated illustrative scenario. If, on the other hand, its purpose isn't obvious, then you might be tempted to spend time in detailed analysis in the hope of recovering some of the knowledge that went into its creation. Unfortunately, it's often impossible to work out why a test script was created or what it was intended to test. When that happens, much of your analysis effort has effectively been wasted.

In our experience, teams get better outcomes by using old manual test scripts as inspiration to help identify areas of the product that would benefit from a thorough understanding (like the WIMP team did in Chapter 2, "Cleaning up an old scenario"). Example mapping can then quickly focus on behaviours that need illustrative scenarios or, when the current team are unsure of expected behaviours, guide them through knowledge recovery.

Ask the tester, not the test script

Test scripts are the result of the investment of time and effort, as well as being concrete evidence of the value we have built into the application. It is understandable that teams will not want to waste the knowledge that these scripts contain.

Reading test scripts can be valuable, but it can also be frustrating, which is why we have already suggested using example mapping. An alternative approach is to get a structured view of the requirements that have been implemented so far by talking to people that have worked on the project. Example mapping is a great way to clarify a team's understanding of uncertain requirements, but if you have access to people who were involved in the legacy implementation you should not waste their expertise.

Never underestimate the domain knowledge of the people who worked on the project. Testers in particular will have acquired a deep understanding of the product through their time spent testing it.

6.4 – What we just learned

While introducing Behaviour Driven Development at the beginning of the project is ideal, the reality for many teams is to introduce BDD to an existing software product. It is never too late to think about getting a better understanding of the problem domain or improving the quality of the solution. Therefore, any practices that are useful in such situation are welcomed by the community.

The good news is that this is possible and we have seen successful projects where BDD has been introduced to a legacy project. There is no magic solution though. Start with a plan that the team applies in a sustained and systematic way. Regular reflection and revision of the plan are essential.

In this chapter we have shown the usual strategies that teams use to incrementally introduce BDD. You can focus on covering new behaviours, the commonly used areas or frequently changing areas. Obviously these strategies are not exclusive – some teams make their strategies based on a mix of these.

Many legacy projects use manual test cases and test scripts to check the quality of the release. It seems to be a natural choice to take these test scripts and formulate them as BDD scenarios. While these test cases are helpful to identify areas of the product that are especially important for quality, one-to-one transformation of them produces BDD scenarios that are hard to maintain and understand. As we described in this chapter, only a careful re-discovery of the rules and examples involved in the test case leads to a long-term usable result.

Finally, we highlighted that there may be other sources of value beyond the existing test scripts. The domain knowledge and experience of testers who worked with those test cases is extremely valuable. Turning a legacy project into a BDD project does not mean that the testers are not needed or that they must only focus on automation. Their expertise is valuable in the discovery and the formulation phases as well.

What's next

Thanks for reading this book – we'd love to hear what you thought about it. If you have any suggestions, comments or a good story about your experiences, please write to us at feedback@bddbooks.com.

Where we've got to

In this book we've covered the second part of the BDD approach: formulation. We've followed along with the WIMP team as they practiced their Gherkin writing craft, creating new scenarios and improving existing ones. Throughout, we've illustrated this with our own experience in the software industry.

Our goal in writing this book was that it would be useful to everyone involved in the specification and delivery of software, so please encourage the rest of your team to read it – it won't take long.

What's left to cover

We hope that you have already explored the practice of discovery with us by reading our previous book, *Discovery* [Nagy2018].

This book covers the formulation part of the BDD approach.

We still haven't discussed test automation, which is often a major motivation for organizations looking to adopt BDD. We'll cover automating scenarios using SpecFlow in *Automation with SpecFlow* [NagyInPrep].

How else we can help

Both authors develop and deliver training and coaching for organizations worldwide. If you would like to talk about the services we can provide, please get in touch at services@bddbooks.com.

Appendices

Gherkin cheat-sheet

The following illustration contains an example of the most important language elements of Gherkin. Use the *Gherkin jump-list* below to find the sections in the book where we discuss them in detail.

Figure 15 – Gherkin cheat-sheet

Gherkin jump-list

- Background: Section 4.8, “Background”
- Block keywords (Feature, Background, Rule, Scenario, Scenario Outline, Examples): Section 3.3, “Gherkin basics”
- Block keyword description: Section 4.10, “Commenting in feature files”
- Comment: Section 4.10, “Commenting in feature files”
- Data table: Section 3.12, “Data tables”; Section 3.14, “Keep tables readable”
- Doc string: Section 3.15, “Readable blocks of text”
- Feature file: Section 3.1, “Feature files”
- Feature name, description: Section 3.4, “The feature file”
- Rule: Section 3.5, “Rules”
- Scenario, Scenario name: Section 3.6, “Scenario structure”
- Scenario step: Section 3.6, “Scenario structure”
- Scenario outline, Examples table: Section 3.13, “Scenario outlines”; Section 3.14, “Keep tables readable”
- Step keywords (*Given*, *When*, *Then*, *And*, *But*): Section 3.3, “Gherkin basics”; Section 3.6, “Scenario structure”
- Tags: Section 4.15, “Manual scenarios”; Section 5.5, “Tags are documentation too”

Formulation "smells" jump-list

Formulation smells are patterns that negatively impact readability and maintainability. The following list contains the most important smells that are covered in this book.

- Changing scenarios to make them easier to automate: Section 4.7, "Readability trumps ease of automation"; Section 3.8, "Keeping context essential"
- Conjunctions (and/or) in step text: Section 3.7, "Multiple contexts"; Section 3.11, "Conjunctions always need consideration"
- Creating one feature file for each user story: Section 5.1, "User stories are not the same as features"
- Direct formulation of manual test scripts: Chapter 2, "Cleaning up an old scenario"; Section 6.3, "Making use of manual test scripts"
- Duplicated scenarios describing common/shared behaviour: Section 5.8, "Documenting shared features"
- Examples table rows that are not essential: Section 3.13, "Scenario outlines"
- Excessive details in feature description: Section 3.4, "The feature file"
- Formulation is not collaborative: Section 4.3, "Too many cooks"
- Formulation takes too long: Section 4.3, "Too many cooks"
- *Given* steps that describe actions: Section 3.9, "Is it a Given or a When?"
- Incidental data in scenario: Section 2.6, "Use real data when it provides clarity"; Section 4.12, "Staying focused"
- Incidental steps in scenario: Section 2.4, "Document the essence of the behaviour"; Section 4.11, "Setting the context"
- Ineffective formulation workshop: Section 4.3, "Too many cooks"
- Large examples and data tables: Section 3.13, "Scenario outlines"; Section 3.14, "Keep tables readable"
- Misaligned Gherkin table columns: Section 3.14, "Keep tables readable"
- Multiple *Then* steps: Section 3.10, "Multiple outcomes"
- Multiple *When* steps: Section 3.9, "Is it a Given or a When?"; Section 3.11, "Conjunctions always need consideration"
- Non-descriptive scenario name: Section 2.3, "Using example maps to provide focus"; Section 3.6, "Scenario structure"
- Overuse of scenario outlines: Section 3.13, "Scenario outlines"

- PO writes the scenarios: Section 4.3, “Too many cooks”
- Regular use of comments: Section 4.10, “Commenting in feature files”
- Regular use of descriptions except feature description: Section 4.10, “Commenting in feature files”
- Reusable steps harming readability: Section 4.13, “Formulation gets faster”
- Scenario used only for testing: Chapter 2, “Cleaning up an old scenario”
- Too many context steps: Section 2.4, “Document the essence of the behaviour”; Section 3.7, “Multiple contexts”; Section 3.8, “Keeping context essential”; Section 4.11, “Setting the context”
- Too many journey scenarios: Section 5.6, “Journey scenarios”
- Too many tags: Section 5.5, “Tags are documentation too”
- Use of “I”: Section 4.5, “There’s no “I” in “Persona””
- Using *Background* as a programming construct: Section 4.8, “Background”
- Using terms that aren’t business-readable: Section 2.5, “Scenarios should read like a specification”; Section 5.5, “Tags are documentation too”

Formulated feature files

The feature files are also available at https://github.com/bddbooks/bddbooks-formulation-wimp/tree/main/final.

`blocklisting.feature`

The scenarios of this feature file are discussed in detail in Chapter 4, "A new user story".

Listing 50 – `customers/blocklisting.feature`

```
Feature: Blocklisting

Rule: Customer's email OR phone must match those on blocklist

  Scenario Outline: Decide if customer is restricted
    Given the customer's contact details are on the blocklist:
      | email          | phone number   |
      | <email listed> | <phone listed> |
    When the blocklist is checked
    Then the customer should be treated as <status>

    Examples:
      | description    | email listed | phone listed | status       |
      | Both listed    | yes          | yes          | restricted   |
      | Email listed   | yes          | no           | restricted   |
      | Phone listed   | no           | yes          | restricted   |
      | Neither listed | no           | no           | unrestricted |

Rule: Restrictions are independent of authentication status

  Scenario: Unauthenticated customer is restricted
    Given a customer is unauthenticated
    And the order's contact details are on the blocklist:
      | email | phone number |
      | yes   | yes          |
    When the blocklist is checked
    Then they should be treated as restricted
```

```
28
29    Scenario: Authenticated customer is restricted
30      Given a customer is authenticated
31      And the account's contact details are on the blocklist:
32        | email | phone number |
33        | yes   | yes          |
34      When the blocklist is checked
35      Then they should be treated as restricted
36
37  Rule: Blocklist is manually configurable
38
39    @manual
40    Scenario: Upload replacement blocklist
41      Given a blocklist is defined using email and phone numbers
42      When the blocklist is uploaded
43      Then the upload should replace the existing blocklist
```

customer_collection.feature

The scenarios of this feature file are discussed in detail in Chapter 3, "Our first feature".

Listing 51 – delivery/customer_collection.feature

```
Feature: Customers can collect their orders

Customers can choose to collect their order from the restaurant:
- whether they are authenticated or not
- whether they pay on order or pay on collection

Rule: Any visitor to the website can place a customer-collection order

  Scenario: Unauthenticated customer chooses to collect order
    Given the customer is unauthenticated
    When they choose to collect their order
    Then they should be asked to supply contact details

  Scenario: Authenticated customer chooses to collect order
    Given the customer is authenticated
    When they choose to collect their order
    Then they should be asked to confirm contact details

Rule: Unauthenticated customers must supply acceptable contact details
                            when placing an order for customer-collection

  Scenario Outline: Contact details supplied ARE acceptable
    Given the customer is unauthenticated
    And they've chosen to collect their order
    When they provide <acceptable contact details>
    Then they should be asked to supply payment details

    Examples:
      | description           | acceptable contact details |
      | Name and Phone        | Name, Phone                |
      | Name and Email        | Name, Email                |
      | Everything provided   | Name, Phone, Email         |

```

```
  Scenario Outline: Contact details supplied are NOT acceptable
    Given the customer is unauthenticated
    And they've chosen to collect their order
    When they provide <unacceptable contact details>
    Then they should be prevented from progressing
    And they should be informed of what made the contact details
                                                                unacceptable

    Examples:
      | description | unacceptable contact details |
      | No Name     | Phone, Email                 |
      | Only Name   | Name                         |
      | Only Email  | Email                        |
      | Only Phone  | Phone                        |

Rule: Customer should be informed when their order will be ready for
                                                       customer-collection

  Scenario: Estimated time of order completion is displayed
    Given the customer placed an order
      | time of order | preparation time |
      | 18:00         | 0:30             |
    When they choose to collect their order
    Then the estimated time of order completion should be displayed as
                                                                      18:30
```

pay_on_collection.feature

The scenarios of this feature file are discussed in detail in Chapter 2, "Cleaning up an old scenario", Chapter 3, "Our first feature" and Chapter 4, "A new user story".

Listing 52 – payment/pay_on_collection.feature

```
Feature: Customers can choose to pay on collection

Rule: Customers can choose for customer-collection orders to be
                                                          pay-on-collection

  Scenario: Customer chooses to pay on collection
    Given the customer has chosen to collect their order
    When they choose to pay on collection
    Then they should be provided with an order confirmation


Rule: Don't offer pay-on-collection for restricted customers

  # Two scenarios merged, because authentication is incidental
  Scenario: Restricted customer is not offered pay-on-collection option
    Given a restricted customer
    When they place an order for customer-collection
    Then pay-on-collection should not be offered as a payment option
```

`pizza_purchase_journey.feature`

The scenarios of this feature file are discussed in detail in Section 5.6, "Journey scenarios".

Listing 53 – `pizza_purchase_journey.feature`

```
Feature: Pizza purchase journey

@journey
Scenario: Pay-on-collection journey
  * Reggie visits our website
  * he adds a pizza to his basket
  * he chooses to checkout as a guest
  * he enters valid contact details
  * he chooses pay-on-collection
  * he receives an order confirmation
```

glossary.md

The glossary file is discussed in detail in Chapter 5, "Organizing the documentation". In Section 5.4, "Documenting the domain", we show how the glossary is displayed in Markdown editors, Listing 54 contains the Markdown source of it.

Listing 54 – glossary.md

```
# Glossary

## Domain terms

* **Authenticated [customer]** -- Customer who has registered in the
          application earlier and logged in with the registered
                                    credentials (e.g. email and password).
* **Blocklist** -- List of contact details to identify unreliable
          customers. (Currently we store email addresses and phone
                                                numbers on the blocklist.)
* **Contact details** -- Details that allow the customer to be contacted
          in relation to a particular order. For authenticated customers,
          the contact details registered to their account are used. For
          unauthenticated customers the contact details must be provided
                                                        with the order.
* **Customer** -- Person who visits the *Where Is My Pizza* site to order
          pizza and other items sold. Unauthenticated (anonymous) users
                                             are also treated as customers.
* **Customer-collection** -- A special delivery method where the customer
                                    picks up the ordered items themselves.
* **Delivery method** -- The selected method for delivering the ordered
                                                                    items.
* **Order** -- Food and beverage items the customer has ordered.
* **Order completion** -- Event (time) when order processing has been
          finished by WIMP, usually when the items have been delivered.
            In some cases it can be earlier (e.g. *customer-collection*).
* **Pay-on-collection** -- Payment method where the customer pays (with
          cash or card) when they pick up the order. This method can only
                                        be used for *customer-collection*.
* **Payment details** -- Details required to fulfil the payment. The
          exact details depend on the *payment method*, e.g. for card
          payment we need the card details, but for *pay-on-collection*
```

```
                                        there are no extra details needed.
* **Payment method** -- Selected method of payment for the order.
* **Restricted [customer]** -- Customer whose contact details are on the
          blocklist. They can only use a reduced set of features the site
                          offers (e.g. they cannot use pay-on-collection).
* **Unauthenticated [customer]** -- Customer who has not logged in.
* **Unrestricted [customer]** -- Customer that can use the full set of
                                              features of the application.
* **Visitor** -- See *Customer*.

## Tags

* **@manual** -- The scenario is verified by manual checks and has to be
                                     excluded from automated test execution
* **@journey** -- The scenario does not illustrate a particular rule but
                                            describes a longer user journey
```

Bibliography

[Cockburn2016] Cockburn, Alistair. *Writing Effective Use Cases*. Addison-Wesley, 2016. Print.

[Covey2005] Covey, S. R. *The 7 Habits of Highly Effective People*. Simon & Schuster, 2005. Print.

[Evans2004] Evans, Eric. *Domain-driven Design: Tackling Complexity in the Heart of Software*. Boston: Addison-Wesley, 2004. Print.

[Feathers2005] Feathers, Michael C. *Working Effectively With Legacy Code*. Prentice Hall, 2005. Print.

[Martraire2019] Martraire, Cyrille. *Living Documentation*. Boston: Addison Wesley, 2019. Print.

[Nagy2018] Nagy, Gáspár, and Seb Rose. *Discovery: Explore behaviour using examples (BDD Books 1)*. Gáspár Nagy and Seb Rose, 2018. Print. http://bddbooks.com/discovery.

[NagyInPrep] Nagy, Gáspár, and Seb Rose. *Automation with SpecFlow (BDD Books 3)*. In preparation. http://bddbooks.com/specflow.

[Patton2014] Patton, Jeff. *User Story Mapping: Discover the Whole Story, Build the Right Product*. Beijing: O'Reilly, 2014. Print.

[Thornhill2015] Thornhill, Adam. *Your Code as a Crime Scene*. Pragmatic Bookshelf, 2015. Print.