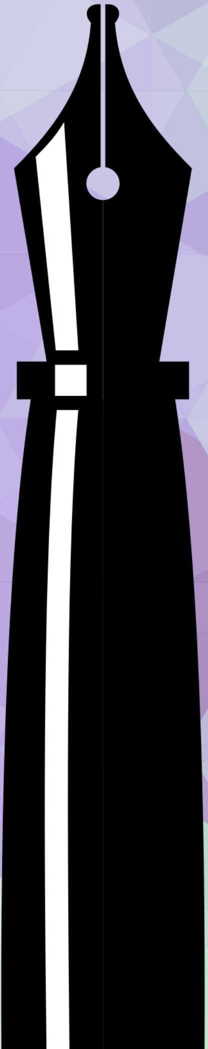
THE BDD BOOKS

Formulation

Express examples using Given/When/Then



Seb Rose
and Gáspár Nagy

Foreword by Angie Jones

The BDD Books - Formulation

Express examples using Given/When/Then

Seb Rose and Gáspár Nagy

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

This version was published on 2020-03-12

Contents

Foreword

by Angie Jones

Coming soon ...

Preface

This book presents a deep dive into Gherkin scenarios and discusses many techniques for writing good scenarios. It defines the Gherkin syntax and connects the Gherkin scenarios to the ubiquitous language (problem domain vs. solution domain). It discusses the most important guidelines (abstraction level, amount of data included, phrasing styles) and provides a collection of useful patterns.

What this book is for

The team may understand the requirements when they come out of a requirements workshop, but they need to document it for people that weren't in the meeting – and for their future selves. In this book we're going to explore Gherkin, using 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.

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). 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. So, whether you come up with the words, do the typing, or provide constructive feedback, you will find this book indispensable.

It's also worth stressing that, while our previous book was completely tool agnostic, this book is focused on tools that understand the Gherkin syntax. That still encompasses a large number of tools, including Cucumber, SpecFlow, JBehave, Behave, Behat.

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 the team is trying to solve. This grouping defines the chapter structure of the book:

- 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 6 core principles of good scenarios (the BRIEF principles).
- Chapter 3, *Our first feature* guides us through a complete feature file that the 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 us through the team's important discussions and decisions as they formulate a new scenario. We see how the resulting agreements lead to faster progress for subsequent scenarios.
- Chapter 5, *Organizing the documentation* focuses on the challenges that you have once you have many feature files. As the team builds up their feature file structure we learn about the difference between story and feature-base structuring, about how the scenarios can be efficiently used as documentation, and about formulating shared features and journey scenarios.
- Chapter 6, *Coping with legacy* discusses how BDD can be introduced into a legacy project. We can see what incremental strategies might work, how to deal with existing manual test scrips, and how testers are essential to the BDD process.

In each chapter, our imaginary 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. We believe that an open discussion of alternatives is a necessary approach in all areas of software development.

In addition to the six chapters, the Appendices contain a complete Gherkin reference, a list of anti-patterns mentioned in the book, and the final feature files that the WIMP team created.

Why you should listen to us

Gáspár is the creator of SpecFlow, the most widely used ATDD/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, called 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 (http://gasparnagy.com) and on Twitter (@gasparnagy). He edits a monthly newsletter (http://bddaddict.com) 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 (https://github.com/specsolutions/deveroom-visualstudio).

Seb has been a consultant, coach, designer, analyst and developer for over 30 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 a BDD Advocate with SmartBear, helping people integrate all three practices of BDD into their development process and ensuring that appropriate tool support is available.

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 and tweets as @sebrose.

Together Seb and Gáspár have over 50 years of software experience.

Acknowledgments

Coming soon …

How this book series is organized

This is the second of the BDD Books series, that will guide you through the end-to-end adoption of BDD, including specific practices needed to successfully drive development using collaboratively authored specifications and living documentation.

Before reading this book, we recommend you read Discovery (Book 1) [1], to understand how to ensure effective team collaboration. Then, once you've practiced formulation using the principles we outline in this book, you can read Automation with SpecFlow (Book 3)[2]

What is not in this book

- Structured conversations (e.g. Example Mapping)
- BDD process overview
- Tools
- Automation
- Code

Online resources

http://bddbooks.com

Seb Rose and Gáspár Nagy, 2019

[1]Nagy, Gáspár, and Seb Rose. *The BDD Books: Discovery - Explore behaviour using examples.* http://bddbooks.com/discovery.

[2]Nagy, Gáspár, and Seb Rose. *The BDD Books: Automation with SpecFlow.* In preparation. http://bddbooks.com/specflow.

Chapter 1 – What is Formulation?

Formulation is the process of turning a shared understanding of how the 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 (see *The BDD Books: Discovery*[3] for more details) 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 the following diagram:

[3]Nagy, Gáspár, and Seb Rose. *The BDD Books: Discovery - Explore behaviour using examples*. http://bddbooks.com/discovery.

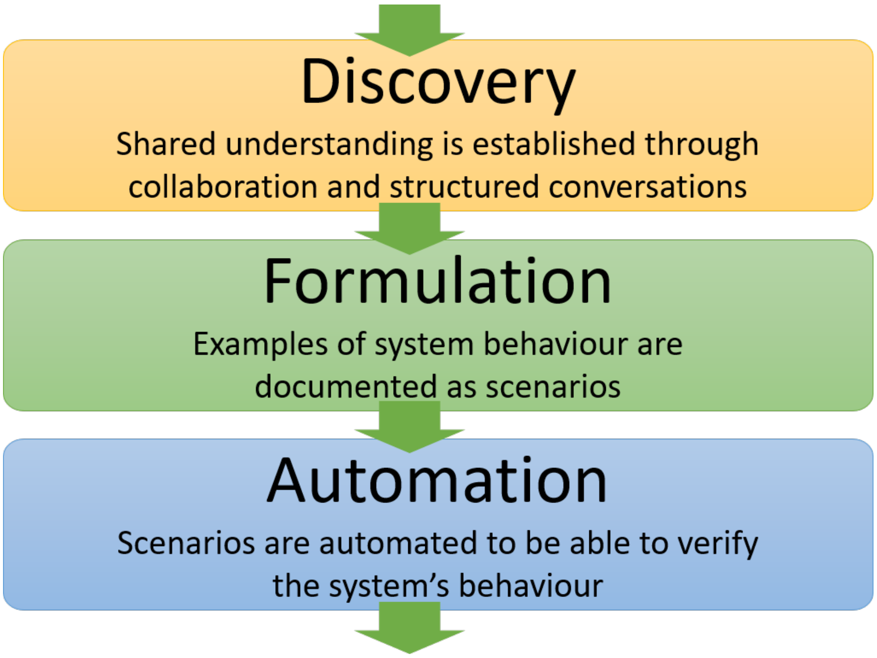

Discovery
Shared understanding is established through collaboration and structured conversations
Formulation
Examples of system behaviour are documented as scenarios
Automation
Scenarios are automated to be able to verify the system's behaviour

Figure 1 – BDD practices

This diagram illustrates the order that the BDD practices should be applied to *each small increment* of functionality in your application. So, by the time the 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.

A more detailed description of the full BDD process is presented in *The BDD Books: Discovery*[4] and has also been published online[5].

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

[4]Nagy, Gáspár, and Seb Rose. *The BDD Books: Discovery - Explore behaviour using examples.* http://bddbooks.com/discovery.

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

input from representatives from business and delivery. The shape of the collaboration varies between organizations (and sometimes between teams), but without it we 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 your are creating a library that performs complex mathematical transformations, you should use terms from the mathematical domain, such as “Matrix”, “Transpose” and “Normalization”.

1.3 – Two types of scenarios

In *The BDD Books: Discovery*[6] 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, 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***.

Illustrative scenarios are excellent for communicating the detailed behaviour of a system, but they don’t give a broad, end-to-end view of how the system works. For this we need longer scenarios that describe a complete user journey, which we call ***journey scenarios***. There should only be a few of this type of scenario – just enough to ensure that a reader can understand how the system delivers useful output.

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.

[6]Nagy, Gáspár, and Seb Rose. *The BDD Books: Discovery - Explore behaviour using examples.* http://bddbooks.com/discovery.

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 be able to use the scenarios to automatically verify the system's behaviour, 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[7], created by Ward Cunningham which allows users to formulate scenarios as fixtures. A Wiki-based UI was grafted onto it by Robert Martin to create Fitnesse[8]. These tools are driven by tabular specifications.

The Robot Framework[9] supports a keyword driven test approach and has been designed to facilitate acceptance test driven development.

Additionally, there are teams that use developer level tools (RSpec[10], Jasmine[11], Cypress[12]) that 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 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 syntax 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 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 will deal exclusively with Gherkin although many of the principles described here are probably applicable to other tools as well.

[7]http://fit.c2.com/
[8]http://www.fitnesse.org/
[9]https://robotframework.org/
[10]http://rspec.info/
[11]https://jasmine.github.io/
[12]https://www.cypress.io/

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[13]. One of the features that make Gherkin attractive internationally is its extensible support for multiple spoken languages, including those that utilize non-Latin alphabets (such as Greek, Chinese, and Russian).

Gaspar'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 Gherkin is the de-facto BDD specification language nowadays.

At the time of writing, the latest version of Gherkin is Gherkin 6. This version introduced the first new keyword (`Rule`) for many years. Not all Cucumber implementations have adopted Gherkin 6 yet, but it is backwards compatible with previous versions.

Although we will use the new ***Rule*** keyword in this book, we will also provide alternatives that will work with earlier versions of Gherkin.

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

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 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 behaviour documented has not yet been implemented
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. Now 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 that is the main topic of our book.

Formulation 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 that we outlined in *The BDD Books: Discovery*[14]: Discovery, Formulation, 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 we are planning 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 us when the implementation diverges from our expectations – so it becomes living documentation that the whole team can rely on.

Formulation is the second book in our BDD Books series.

The first book, *The BDD Books: Discovery*[15], discussed how business requirements can be explored using examples.

This series will be completed by books focusing on automation. The first of these, the *The BDD Books: Automation with SpecFlow*[16] book, will go into detail about the way an application can be *automated* with SpecFlow, including in-depth examples of how to design maintainable automation code.

[14]Nagy, Gáspár, and Seb Rose. *The BDD Books: Discovery - Explore behaviour using examples.* http://bddbooks.com/discovery.

[15]Nagy, Gáspár, and Seb Rose. *The BDD Books: Discovery - Explore behaviour using examples.* http://bddbooks.com/discovery.

[16]Nagy, Gáspár, and Seb Rose. *The BDD Books: Automation with SpecFlow.* In preparation. http://bddbooks.com/specflow.

Chapter 2 – Cleaning up an old scenario

Gherkin is used by thousands of organizations all over the world, and it is not uncommon to find long, complex and unreadable scenarios in their projects. A bad scenario is a legacy that causes continuous maintenance efforts and frustration for the team. Cleaning up a bad scenario not only alleviates these problems, but also provides a good opportunity to learn more about our domain.

In this chapter we're going to learn the basics of formulation by following a team as they take a poorly formulated scenario and make it better.

2.1 – The old scenario

The Where Is My Pizza (WIMP) application (introduced in *The BDD Books: Discovery*[17]) 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, 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 requirements 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.

[17]Nagy, Gáspár, and Seb Rose. *The BDD Books: Discovery - Explore behaviour using examples.* http://bddbooks.com/discovery.

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 "Marghertheyta" within "SearchResults"
  And they select "Medium" from "Size"
  When they press "Add to basket"
  Then they should see "1 item" in "BasketItemCount"
  And 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 "Than 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 organisations 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 *The BDD Books: Discovery*[18], *Chapter 3*). ***And***, ***But*** are conjunctions that continue the preceding section. We cover the majority of Gherkin syntax in {#sec-featurefiles} and {#sec-gherkinbasics}. See {#appendix-gherkin} 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 didn't come from a concrete ***example***. 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:

[18]Nagy, Gáspár, and Seb Rose. *The BDD Books: Discovery - Explore behaviour using examples*. http://bddbooks.com/discovery.

- Scenarios should be thought of as documentation, rather than tests.
- Scenarios should enable collaboration between business and delivery, not prevent it.
- Scenarios should support the evolution of the product, rather than obstruct it.

The following six principles work together to support these goals. 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.

Business language: The words used in a scenario should be drawn from the business domain, otherwise you will not be able to engage your business colleagues painlessly. The language you use should use terms that business team members understand.

Real data: In *The BDD Books: Discovery*[19]*, Section 3.1* we explained that examples should use concrete, real data. This helps expose boundary conditions 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, then 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 an ***Example Mapping*** session.

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

[19]Nagy, Gáspár, and Seb Rose. *The BDD Books: Discovery - Explore behaviour using examples.* http://bddbooks.com/discovery.

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.

Seb's story: Acronyms

Gaspar and I have been explaining the concepts behind BRIEF for many years, but we've 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 Bill Wake and Mike Kohn with INVEST."

We took his advice, and BRIEF is the result. Thanks for the encouragement, Gojko!

2.3 – Focus and Intent

"Well, to start with the name of this scenario 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 it into several scenarios."

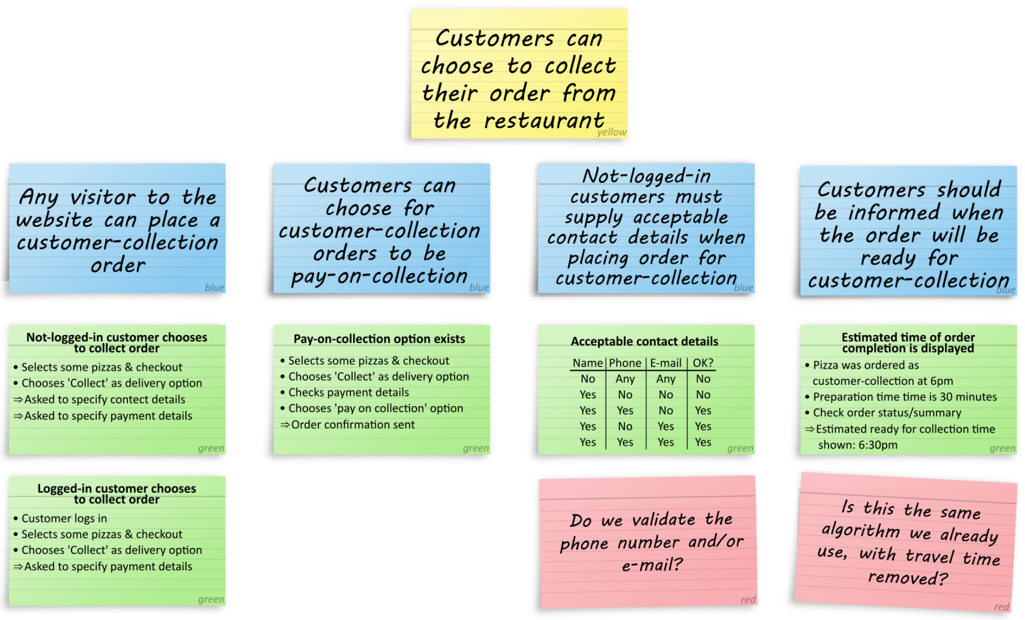
"I can't even remember what the exact requirements were for customer collections," says Tracey. "Maybe we should do a quick example map to make sure we understand how the system works right now."

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 requirements 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 this example map:



Customers can choose to collect their order from the restaurant

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

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

Not-logged-in customers must supply acceptable contact details when placing order for customer-collection

Customers should be informed when the order will be ready for customer-collection

Not-logged-in customer chooses to collect order

- Selects some pizzas & checkout
- Chooses 'Collect' as delivery option

⇒Asked to specify contect details
⇒Asked to specify payment details

Pay-on-collection option exists

- Selects some pizzas & checkout
- Chooses 'Collect' as delivery option
- Checks payment details
- Chooses 'pay on collection' option

⇒Order confirmation sent

Acceptable contact details

Name	Phone	E-mail	OK?
No	Any	Any	No
Yes	No	No	No
Yes	Yes	No	Yes
Yes	No	Yes	Yes
Yes	Yes	Yes	Yes

Estimated time of order completion is displayed

- Pizza was ordered as customer-collection at 6pm
- Preparation time time is 30 minutes
- Check order status/summary

⇒Estimated ready for collection time shown: 6:30pm

Logged-in customer chooses to collect order

- Customer logs in
- Selects some pizzas & checkout
- Chooses 'Collect' as delivery option

⇒Asked to specify payment details

Do we validate the phone number and/or e-mail?

Is this the same algorithm we already use, with travel time removed?

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 had the map on the desk in front of them, so they could turn back to the scenario projected on the wall and continue with their analysis.

"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.

"That will do to start with," agrees Patricia and she types in the change.

Listing 2 – An intention revealing name

```
Scenario: Not-logged-in 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?"

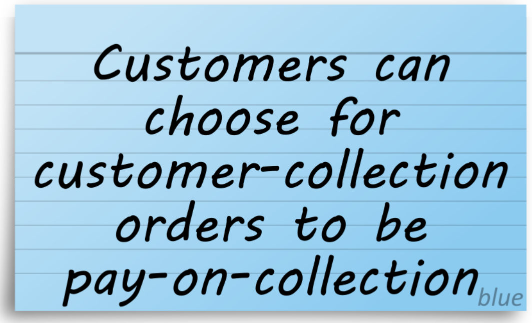

Customers can choose for customer-collection orders to be pay-on-collection
blue

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 and 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: Not-logged-in 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"
  And they click "Checkout"
  When they select "Collect" from "DeliveryInstructions"
  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:

- Focused on a single rule
- Started by identifying the expected outcome
- Deleted all outcomes that don’t directly illustrate the rule
- Gave the scenario an intention revealing name

We will talk more about these in the later chapters of the book.

2.4 – Essential, not incidental

“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 doesn’t reveal *intent*,” 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: Not-logged-in customer chooses to pay on collection
  Given the customer has "1 item" in "BasketItemCount"
  And they click "Checkout"
  When 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 are useful specifications of 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

2.5 – Business language

"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 Tracey, who fixes the scenario:

Listing 5 – Third edit

```
Scenario: Not-logged-in customer chooses to pay on collection
  Given the customer has 1 item in their basket
  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
```

"That's really succinct. It doesn't depend on the UI, or even on the detail of the checkout flow. It expresses *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***

2.6 – Real data and intent

“… 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: Not-logged-in customer chooses to pay on collection
  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.

Tracey deletes the first line of the scenario:

Listing 7 – Fifth edit

```
Scenario: Not-logged-in customer chooses to pay on collection
  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

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

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

Listing 8 – Focus on communication

```
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 logged in 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, smiling. "I think this documents the behaviour!"

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 the business have asked for.

What the team just did

The team has:

- Renamed the scenario to ensure that it accurately describes the behaviour that is 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. In *The BDD Books: Discovery*[20], we 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 Dan North[21]. 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.

[20]Nagy, Gáspár, and Seb Rose. *The BDD Books: Discovery - Explore behaviour using examples.* http://bddbooks.com/discovery.

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

Gaspar’s story: Learning with usable outcomes

I have often participated in meetings like this. 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 Dan 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/

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 we 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 become much simpler, but it was still using technical solution details to describe the different steps. Although probably everyone in the meeting could make a good guess what those technical details mean, 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 learnt 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 our 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, so for this chapter, we're going to limit discussion to a single feature file.

The feature file that the team have 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 have written scenarios for all the examples that they discovered in the last chapter (see Chapter 2, Figure 2) and replaced the old scenario completely. The scenarios have been compiled into a ***Feature File***, which is reproduced in full below.

Have a look through the feature file. Can you see how it relates to the example map? 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 (`CustomerCollection.feature`)

```gherkin
Feature: Customers can collect their orders

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

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

  Scenario: Not-logged-in customer chooses to collect order
    Given the customer is not logged in
    When they choose to collect their order
    Then they should be asked to supply contact and payment details

  Scenario: Logged-in customer chooses to collect order
    Given the customer is logged-in
    When they choose to collect their order
    Then they should be asked to supply payment 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: Not-logged-in customers must supply acceptable contact details wh\
en placing an order for customer-collection

  Scenario Outline: Contact details supplied ARE acceptable

    Given the customer is not logged in
    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 not logged in
    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 unacce\
ptable

    Examples:
      | description          | unacceptable contact details |
      | Name must be provided | Phone, Email                |
      | Name is not enough   | Name                         |
      | Only Email           | Email                        |
      | Only Phone           | Phone                        |

Rule: Customer should be informed when their order will be ready for cu\
stomer-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
```

3.3 – Gherkin basics

Gherkin is an extremely simple language. It's motivation is to create documentation that's close enough to natural language to allow everyone involved in the product can contribute to and understand, while remaining structured enough to enable automation tools (such as Cucumber and SpecFlow) to process it as well.

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 - a template for generating several, similar scenarios
- Examples - 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 in this section. The English variants and other languages supported can be found in the Gherkin project[22].

Strict reading of the text

Since scenarios are used for automation, you have to be aware that consistent usage of spaces, punctuation, and capitalisation is even more important than in normal written text. The keywords are case-sensitive, so inconsistent formatting of the text within a scenario will cause issues during the automation phase. This will be covered in detail in *The BDD Books: Automation with SpecFlow*[23].

3.4 – Diving into 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

[22]https://github.com/cucumber/cucumber/blob/master/gherkin/gherkin-languages.json

[23]Nagy, Gáspár, and Seb Rose. *The BDD Books: Automation with SpecFlow*. In preparation. http://bddbooks.com/specflow.

online documentation[24].

The feature description

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 logged in to WIMP or not
- whether they pre-pay or pay on collection
```

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 feature file shows that textual descriptions can be useful. Here the team has used it to provide 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 use the section after the *Feature* keyword to list the rules that were being illustrated in the feature file. Gherkin 6 introduced the *Rule* keyword which allows for much clearer association between the rule and the scenarios that illustrate it.

To see a version of the feature file written without using the *Rule* keyword download them from the book website[25].

Adding this sort of information does come with a risk – Gherkin completely ignores it when processing your feature file. That 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.

[24]https://cucumber.io/docs/gherkin/reference/
[25]http://speclink.me/bookfeaturefilesworule

Rules

The *Rule* keyword was created to allow feature files to more closely reflect Example Maps as introduced by Matt Wynne[26] and described in *The BDD Books: Discovery*[27].

```
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 that 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 (available for download from the book website[28]).

As soon as you use the *Rule* keyword in a feature file, all scenarios that follow it are assumed to illustrate that rule – until the next *Rule* keyword is encountered. So, the rules and scenarios in the feature file above are associated in exactly the way you would expect when looking at the example map 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.

3.5 – Scenario structure

Each scenario describes a single concrete example of system behaviour, which can (and should) be followed by a descriptive name. In *The BDD Books: Discovery*[29], *2.5* we suggest using the *Friends* naming scheme that copies the way that episodes of the *Friends* sitcom were named: *"The One Where Rachel Finds Out"*. The words you put after the *"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: ***Given***, ***When***, ***Then***, ***And***, ***But***.

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

[27]Nagy, Gáspár, and Seb Rose. *The BDD Books: Discovery - Explore behaviour using examples.* http://bddbooks.com/discovery.

[28]http://speclink.me/bookfeaturefilesworule

[29]Nagy, Gáspár, and Seb Rose. *The BDD Books: Discovery - Explore behaviour using examples.* http://bddbooks.com/discovery.

Given introduces a step that sets up the ***context*** of a scenario. We usually use the past tense for a ***Given*** step, because it describes the state the system should already be in before the behaviour being illustrated is exhibited.

When introduces a step that causes an ***action*** to take place. This describes the stimulus or event that causes the system to behave in a particular way. There should normally be only a single ***When*** step in a scenario.

A ***Then*** step describes the expected ***outcome*** of the behaviour. We recommend the use of the word *should* in a ***Then*** step, as described by Liz Keogh[30]. By using the word *should* it leaves the reader open to ask the question "should it really behave like that?". There should usually be only a single ***Then*** step in a scenario.

Seb's Story: When should we not use "should"?

I usually suggest that clients use "should" to describe the outcome, but there are some contexts in which this clashes with their traditional requirement engineering practice. This is especially true in regulated industries.

One of my clients has explained this 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." In this case the expectation of the customer trumps the desire to encourage open questioning of the requirement.

And and ***But*** are conjunctions. When used after ***Given***, ***When***, or ***Then*** they simply act as a continuation. We'll see examples of their use, and give advice about how to use them, throughout the book.

A simple scenario

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 described the action. Finally, theres a single ***Then*** statement that describes the expected outcome.

[30]https://lizkeogh.com/2005/03/14/the-should-vs-will-debate-or-why-doubt-is-good/

Listing 10 – Customer chooses to collect order

```
Scenario: Not-logged-in customer chooses to collect order
  Given the customer is not logged in
  When they choose to collect their order
  Then they should be asked to supply contact and payment details
```

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

Multiple contexts

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

```
  Given the customer is not logged in
  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 *'Not-logged-in 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 not logged in
- the customer has chosen to collect their order

We could have compressed this into a single step:

```
  Given the customer is not logged in and they've chosen to collect my \
order
```

We find that the single step form is harder to read and understand.

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.

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 more than one outcome 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 unaccept\
able
```

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 *'Not-logged-in customers must supply acceptable contact details when placing an order for customer-collection'*.

3.6 – 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 in a 2-dimensional grid. You can think of them as a simplistic equivalent to a spreadsheet.

When an order is placed using our application, the customer will be told when it will be ready for collection. This is calculated by summing the order preparation time and 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 tak\
e 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.

3.7 – 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 that you might find yourself using them.

How scenario outlines work

When Gherkin encounters a scenario outline it expands it into multiple scenarios, based on the number of rows in the ***Examples table***. The *Examples* table below has three rows – one header row and two example rows:

Listing 11 – 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 placeholder***s. 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 Scenario by substituting the parameter placeholder with the text from the Examples table. So, the Scenario Outline and Examples table above is exactly equivalent to:

Listing 12 – Scenario outline equivalent represention

```
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 8 slices
  When they eat 8 slices
  Then they should have no slices left
```

Use of *Scenario Outline* is becoming optional

Recent versions of Gherkin treat *Scenario* and *Scenario Outline* 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.

At time of writing (October 2019) versions of Gherkin that behave in this way have not been integrated with Cucumber and SpecFlow. However, this work is ongoing and will be available in the very near future.

In the Examples table above there's also 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. We'll talk more about this in *The BDD Books: Automation with SpecFlow*[31].

Data variations

Some behaviours are best understood by thinking about the how they vary according to input data. An obvious indication of this is when it's natural to describe the examples using tables of data. Calculations are a common example – and the *'Describe what remains of my pizza'* scenarios above is one form of calculation.

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

```
Scenario Outline: Contact details supplied ARE acceptable
  Given the customer is not logged in
  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
```

[31]Nagy, Gáspár, and Seb Rose. *The BDD Books: Automation with SpecFlow*. In preparation. http://bddbooks.com/specflow.

```
  Given the customer is not logged in
  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 unaccept\
able

  Examples:
    | description           | unacceptable contact details |
    | Name must be provided | Phone, Email                 |
    | Name is not enough    | 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.

Gaspar'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 thirty rows, but also twenty 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 example set we also halved the time it took to run the automation, which 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: they are not 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 13 – Similar scenarios

```
Scenario: Not-logged-in customer chooses to collect order
  Given the customer is not logged in
  When they choose to collect they order
  Then they should be asked to supply contact and payment details

Scenario: Logged-in customer chooses to collect order
  Given the customer is logged-in
  When they choose to collect they order
  Then they should be asked to supply payment details
```

They could be written as a scenario outline:

```
Scenario Outline: Customer chooses to collect order
  Given the customer is <logged-in-or-not>
  When they choose to collect they order
  Then they should be asked to supply <details-needed> details

  Examples:
    | logged-in-or-not | details-needed       |
    | logged in        | payment              |
    | not logged in    | contact and payment |
```

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 the people reading the documentation find it easier to read as multiple Scenarios or a single Scenario Outline?”

Which do you find more readable?

- The team’s choice of two scenarios (“Not-logged-in customer chooses to collect order”, “Logged-in customer chooses to collect order”)
- The single scenario outline (“Customer chooses to collect order”)

3.8 – Keep tables readable

Tables exist to enhance readability for the business user. 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 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.

3.9 – 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.

Both Data tables and Scenario Outlines are useful elements of the Gherkin language, but they can also be abused. A Data table with twenty columns or a Scenario Outline with thirty 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.

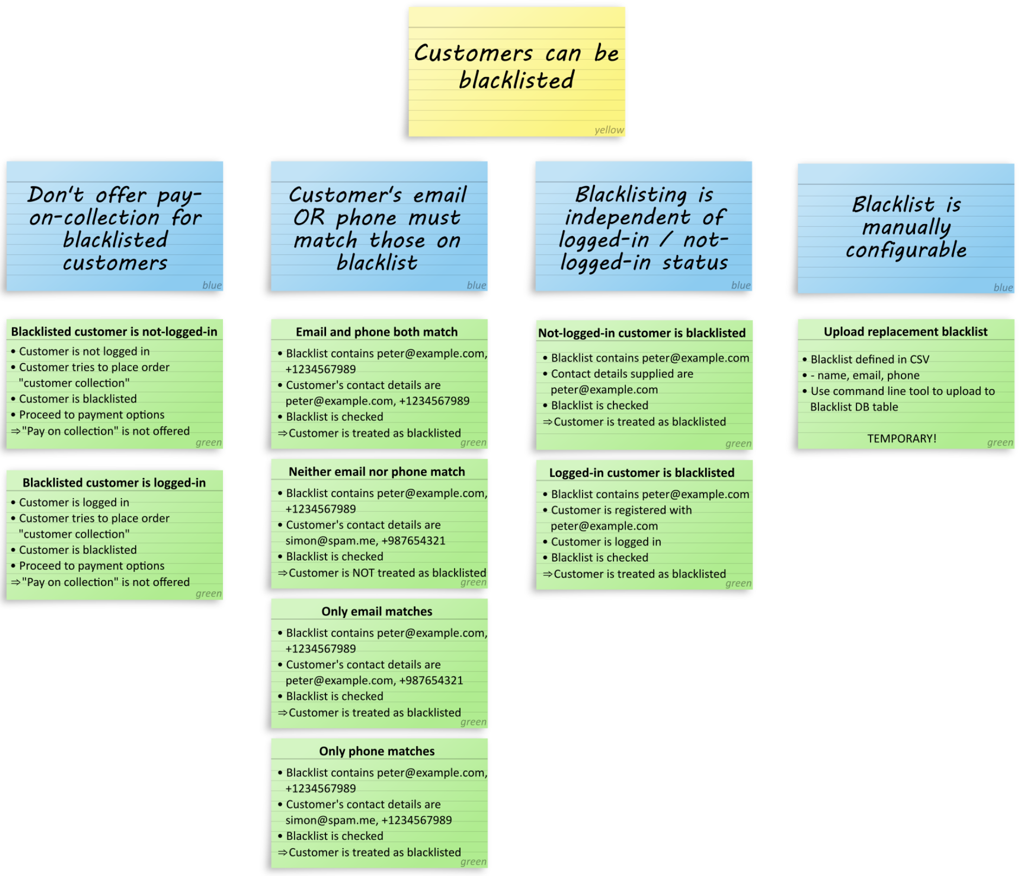
Although we have seen the core elements of the Gherkin language, we haven't seen everything yet. In the following chapters we will see a few more Gherkin constructs and discuss how you can structure and manage the feature files of your entire product.

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 – Blacklist

The business team is concerned by orders placed for *pay-on-collection* that have never been collected (and so, never paid for). In response, Patricia created a story that was explored at several requirement workshops. The team split the story into several smaller stories and Patricia picked the highest priority story to bring into the iteration:


Customers can be blacklisted
yellow
Don't offer pay-on-collection for blacklisted customers
blue
Customer's email OR phone must match those on blacklist
blue
Blacklisting is independent of logged-in / not-logged-in status
blue
Blacklist is manually configurable
blue

Blacklisted customer is not-logged-in
- Customer is not logged in
- Customer tries to place order "customer collection"
- Customer is blacklisted
- Proceed to payment options

⇒"Pay on collection" is not offered

Blacklisted customer is logged-in
- Customer is logged in
- Customer tries to place order "customer collection"
- Customer is blacklisted
- Proceed to payment options

⇒"Pay on collection" is not offered

Email and phone both match
- Blacklist contains peter@example.com, +1234567989
- Customer's contact details are peter@example.com, +1234567989
- Blacklist is checked

⇒Customer is treated as blacklisted

Neither email nor phone match
- Blacklist contains peter@example.com, +1234567989
- Customer's contact details are simon@spam.me, +987654321
- Blacklist is checked

⇒Customer is NOT treated as blacklisted

Only email matches
- Blacklist contains peter@example.com, +1234567989
- Customer's contact details are peter@example.com, +987654321
- Blacklist is checked

⇒Customer is treated as blacklisted

Only phone matches
- Blacklist contains peter@example.com, +1234567989
- Customer's contact details are simon@spam.me, +1234567989
- Blacklist is checked

⇒Customer is treated as blacklisted

Not-logged-in customer is blacklisted
- Blacklist contains peter@example.com
- Contact details supplied are peter@example.com
- Blacklist is checked

⇒Customer is treated as blacklisted

Logged-in customer is blacklisted
- Blacklist contains peter@example.com
- Customer is registered with peter@example.com
- Customer is logged in
- Blacklist is checked

⇒Customer is treated as blacklisted

Upload replacement blacklist
- Blacklist defined in CSV
- - name, email, phone
- Use command line tool to upload to Blacklist DB table

TEMPORARY!

Figure 4 – Blacklist example map

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

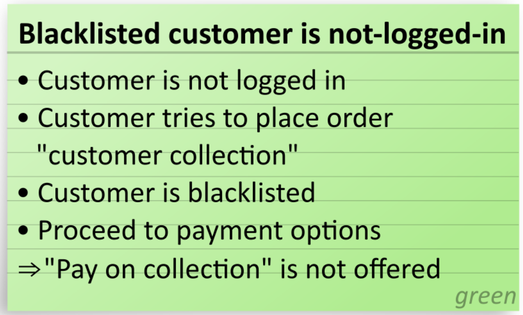
4.2 – Write it upwards

Patricia sets the scene: "We've decided to deliver the blacklisting 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 blacklisted customers'*," says Patricia. "That's the one that I'm most interested in."

The team looks at the first concrete example:



Blacklisted customer is not-logged-in

- Customer is not logged in
- Customer tries to place order "customer collection"
- Customer is blacklisted
- Proceed to payment options

⇒"Pay on collection" is not offered

Figure 5 – Blacklisted customer - not logged in

"How about *'**Given** I am not logged in'*," 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 "7 Habits of Effective People" [a]. 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."

[a] Covey, S. R. The 7 Habits of Highly Effective People. Simon & Schuster, 2005. Print.

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 didn’t we formulate during the Discovery session?
- Shouldn’t the whole team formulate together?
- Don’t the PO/BAs own the specifications?
- Shouldn’t the developers/testers focus on the implementation?
- Isn’t it more efficient for one person do it?

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

Why didn’t we formulate during the Discovery session? During Discovery we want to uncover misunderstandings and hidden assumptions. Our 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, we use concrete examples 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 us 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 benefit from delegating 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?

Secondly, team members that don't participate in the formulation are expected to review the scenarios asynchronously. This gives them the time and space to read through the scenarios at their own pace and offer constructive feedback. Reviewing a scenario asynchronously also gives us confidence that it will be understood by people who weren't involved in the product's development.

Don't the PO/BAs own the specifications? The goal of the product owner (PO) and business analyst (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 the concrete examples, and use appropriate business terminology, then we can be confident that the business and delivery sides of the team really do have a shared understanding.

The PO/BA is still accountable 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 demonstrate that they have a clear understanding of the business requirements and the business terminology.

Why can't just one person do it? If formulation is more effective when done by a subset of the team, there's an argument that it might be most effective when done by just one person, rather than a pair. In our experience, all activities are improved when undertaken by a pair.

In the case of formulation, the tester/developer pair is ideal, because the developer is considering the automation aspect of the scenario, while the business-focus of the tester ensures the scenarios don't become technical.

Your context may be different. Our suggestion of a developer/tester pair might not be acceptable in your organization. Or your product owner might prefer to participate in formulation rather than review scenarios. We have seen examples where other approaches to formulation work just fine, but in the vast majority of cases it's the developer/tester pair that works best.

"Maybe this is an example of 'too many cooks spoils the broth'[32]" 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.

4.4 – Quotation marks

Dishita and Tracey soon have a scenario:

Listing 14 – Pay-on-collection blacklist scenario using quotes

```
Scenario: Blacklisted "not logged in" customer not offered "pay on coll\
ection" option
  Given I have been blacklisted
  And I am not logged in
  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 the quotes can vary," replies Tracey. "In this scenario, I intended the quotes

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

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 that will be discussed in *The BDD Books: Automation with SpecFlow*[33] 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 15 – Pay-on-collection blacklist scenario without quotes

```
Scenario: Blacklisted not logged in customer not offered pay on collect\
ion option
  Given I have been blacklisted
  And I am not logged in
  When I place an order for customer collection
  Then pay on collection should not be offered as a payment option
```

"I'm worried that this makes it harder for the reader to identify what's important about the scenario," says Tracey.

"I see what you mean," replies Dishita. "Can we try hyphenating the terms instead?"

[33]Nagy, Gáspár, and Seb Rose. *The BDD Books: Automation with SpecFlow*. In preparation. http://bddbooks.com/specflow.

Listing 16 – Pay-on-collection blacklist scenario using hyphens

```
Scenario: Blacklisted not-logged-in customer not offered pay-on-collect\
ion option
  Given I have been blacklisted
  And I am not logged in
  When I place an order for customer-collection
  Then pay-on-collection should not be offered as a payment option
```

"I like the hyphens," says Tracey. "They're easy to see, but are a lot less distracting than the quotation marks. Let's use them and see what the rest of the team think later."

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 blacklisted customer," says Tracey. "Why do you ask?"

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

"We could create a blacklisted customer persona called Bruno," suggests Tracey. "Then we could say *'**Given** Bruno is not logged in'*. Would that be better?"

"It would, but it's quite an investment creating a persona. Maybe we could just use a more descriptive noun phrase – like 'customer'," says Dishita.

Listing 17 – Pay-on-collection blacklist scenario using *Customer*

```
  Scenario: Blacklisted not-logged-in customer not offered pay-on-colle\
ction option
    Given a blacklisted customer
    And the customer is not logged in
    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, we decided right at the beginning to not use it," 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, we should 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. Persona 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 persona.

"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."

Does repetition matter?

In the scenario Listing 17 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 18 – Pay-on-collection blacklist scenario using *they*

```
Scenario: Blacklisted not-logged-in customer not offered pay-on-collect\
ion option
  Given a blacklisted customer
  And they are not logged in
  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 the plural pronoun "they". As the Merriam-Webster website[34] 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:

Listing 19 – Ambiguous usage of *they*

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

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.

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

Ease of automation should not influence the specification

A common concern when using pronouns, such as "they", is that this makes it harder to automate the scenario. We will look at this in more detail in *The BDD Books: Automation with SpecFlow*[35], but want to make it clear here that readability of your specification should *not* be influenced by the needs of any automation framework that you are using. There are many automation approaches that can handle *they* (or other common linguistic shortcuts) and your engineers should use them as appropriate.

4.6 – Background

"Let's formulate the next scenario for a logged in customer," says Tracey. They quickly get this scenario:

Listing 20 – Pay-on-collection scenario for a blacklisted customer

```
Scenario: Blacklisted logged-in customer not offered pay-on-collection \
option
  Given a customer has been blacklisted
  And they are logged in
  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 customer has been blacklisted'*," says Dishita. "Maybe we could use a *Background* to simplify the feature file."

Background[36] 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.

[35]Nagy, Gáspár, and Seb Rose. *The BDD Books: Automation with SpecFlow*. In preparation. http://bddbooks.com/specflow.

[36]https://cucumber.io/docs/gherkin/reference/#background

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

Listing 21 – Scenarios sharing context using *Background*

```
Background:
  Given a customer has been blacklisted

Scenario: Blacklisted not-logged-in customer not offered pay-on-collect\
ion option
  Given they are not logged in
  When they place an order for customer-collection
  Then pay-on-collection should not be offered as a payment option

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

Gaspar'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 (based on the book of Robert C. Martin [a]).

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, so it is valid to check them against "Clean code" principles and DRY. The conclusion is the same: simplicity and readability is more important than rigorously eliminating all duplications. Consider this when using *Background*.

[a]Martin, Robert C., et al. Clean Code: a Handbook of Agile Software Craftsmanship. Prentice Hall, 2008. Print.

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 that maintain 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*."

4.7 – Unformulated scenarios

“These two scenarios do look very similar, though. The only difference is that in one the customer is logged in 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 22 – Combined blacklisting scenario

```
Scenario: Blacklisted customer is not offered pay-on-collection option
  Given a customer has been blacklisted
  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 that we create to illustrate the behaviour of the system, the programmers will create additional automated 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 blacklisted or not. And if the product owner is not happy with our formulation, then she will let us know when she reviews our scenarios."

4.8 – 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 23 – Blacklisting scenario with comment

```
# Two scenarios merged, because being logged-in or not is incidental
Scenario: Blacklisted customer is not offered pay-on-collection option
  Given a blacklisted 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 24 – Blacklisting scenario with text description

```
Scenario: Blacklisted customer is not offered pay-on-collection option
  Two scenarios merged, because being logged-in or not is incidental

  Given a blacklisted 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 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 – needing no further explanation – so there should be no 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.

4.9 – Setting the context

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

"We already do, in the *'Customer chooses to pay on collection'* scenario we reverse engineered earlier," says Tracey, shown in Chapter 3, Listing 9.

"That scenario deosn't mention blacklisting at all," says Dishita.

"No," agrees Tracey, "it's implicit that most customer's are not blacklisted."

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 the 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 the pizza. So, we don't have to restate the obvious. When the unusual situation of a blacklisted customer occurs, we draw attention to this by stating that the customer *is* blacklisted.

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 an electric kettle. The default assumption is that there is an electricity supply, the fuse is in the plug, the kettle is functionally sound, and it contains water. In this situation, the following scenario might be perfectly adequate:

```
When I turn the kettle on
Then the water should boil
```

Now when an unexpected situation happens:

```
Given there is no water in the kettle
When I turn the kettle on
Then the burn-out protection should engage
```

These two scenarios are clearly different. Consider if the first scenario had been written:

```
Given there is water in the kettle
When I turn the kettle on
Then the water should boil
```

Now the contexts of the two scenarios are very similar and the reader may wonder why such similar scenarios have outcomes that are so different. It takes time and effort to notice the change from “water in the kettle” to “no water in the kettle.” We find that omitting the context when it can reasonably be considered *implicit* leads to more readable, maintainable specifications.

Gaspar’s Story: Decide by seeing

Applying the the 6 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 the 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.

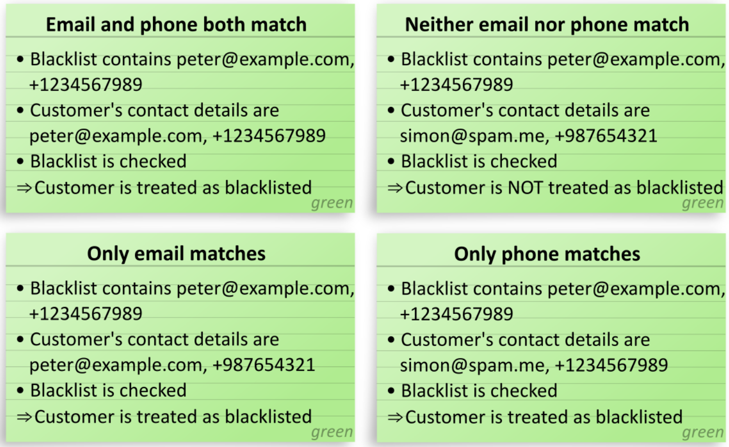
Stopping the discussion and asking the participants to write down the scenario that expresses their own viewpoint is often a good solution. Being able to see how the various scenarios differ can help decide which version supports the team better – like you saw in the discussion between Tracey and Dishita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

4.10 – Formulation gets faster

“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 blacklist’*,” says Tracey.


Email and phone both match
- Blacklist contains peter@example.com, +1234567989
- Customer's contact details are peter@example.com, +1234567989
- Blacklist is checked

⇒Customer is treated as blacklisted

Neither email nor phone match
- Blacklist contains peter@example.com, +1234567989
- Customer's contact details are simon@spam.me, +987654321
- Blacklist is checked

⇒Customer is NOT treated as blacklisted

Only email matches
- Blacklist contains peter@example.com, +1234567989
- Customer's contact details are peter@example.com, +987654321
- Blacklist is checked

⇒Customer is treated as blacklisted

Only phone matches
- Blacklist contains peter@example.com, +1234567989
- Customer's contact details are simon@spam.me, +1234567989
- Blacklist is checked

⇒Customer is treated as blacklisted

Figure 6 – E-mail or phone match examples

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

Listing 25 – Email and phone both match

```
Scenario: Email and phone both match
  Given the blacklist contains peter@example.com, +123456789
  And the customer's contact details are peter@example.com, +123456789
  When the blacklist is checked
  Then customer should be treated as blacklisted
```

"That seemed easy. I think this might make a good scenario outline. Let's try the example where neither match," suggests Dishita.

Listing 26 – Neither email nor phone both match

```
Scenario: Neither email nor phone match
  Given the blacklist contains peter@example.com, +123456789
  And the customer's contact details are simon@spam.me, +987654321
  When the blacklist is checked
  Then customer should be treated as not blacklisted
```

"You're right," agrees Tracey "this would make a good scenario outline."

Listing 27 – Scenario outline for e-mail or phone match rule (emails and phone numbers are shortened for readability)

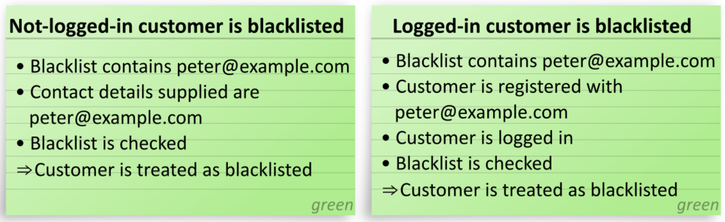
```
Scenario Outline: Matching contact details with blacklist
  Given the blacklist contains <details>
  And the customer's contact details are <contact details>
  When the blacklist is checked
  Then customer should be treated as <blacklisted?>

Examples:
| description    | details          | contact details | blacklisted?    |
| Both match     | p...com, +1...9 | p...com, +1...9 | blacklisted     |
| Neither match  | p...com, +1...9 | s...me, +9...1  | not blacklisted |
| Email matches  | p...com, +1...9 | p...com, +9...1 | blacklisted     |
| Phone matches  | p...com, +1...9 | s...me, +1...9  | blacklisted     |
```

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

"Yep, it contains all the details we need for the implementation." acknowledges Dishita. "And it did not take long to formulate."

"Let's see how quickly we can do the remaining examples. Shall we start with the two examples of the *'Blacklisting is independent of logged-in/not-logged-in status'* rule?" asks Tracey, as moves the example cards closer.



Not-logged-in customer is blacklisted

- Blacklist contains peter@example.com
- Contact details supplied are peter@example.com
- Blacklist is checked

⇒Customer is treated as blacklisted

green

Logged-in customer is blacklisted

- Blacklist contains peter@example.com
- Customer is registered with peter@example.com
- Customer is logged in
- Blacklist is checked

⇒Customer is treated as blacklisted

green

Figure 7 – Blacklisting is independent of login 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 28 – Scenarios reusing existing step

```
Scenario: Not-logged-in customer is blacklisted
  Given a customer is not logged in
  And the contact details the customer supplied for the order are peter\
@example.com, +123456789
  And the blacklist contains peter@example.com, +123456789
  When the blacklist is checked
  Then the customer should be treated as blacklisted

Scenario: Logged-in customer is blacklisted
  Given a customer is logged in
  And the customer's contact details are peter@example.com, +123456789
  And the blacklist contains peter@example.com, +123456789
  When the blacklist is checked
  Then the customer should be treated as blacklisted
```

“That was easy.” says Dishita. “We only needed one new step to express that contact details can be supplied to a customer-collection order of a not-logged-in customer.”

“These may not be the most interesting scenarios in this feature,” says Tracey, “but they demonstrate that the system can recognise a blacklisted customer whether they are logged in or not. It gives me confidence that the previous scenario (Listing 27) doesn’t need to specify whether the customer is logged in 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?” ask Dave, who peeks into the meeting room. “As far as I can see, 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 we did in 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 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 build an automation solution for them later. Once you start feeling the rhythm of the ubiquitous language, formulating similar scenarios just gets faster and faster. In many cases, you will be able to reuse steps from the previous scenarios.

Gaspar's Story: Don't overrate reusability

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

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.

4.11 – Incremental specification

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

"We only have *'Blacklist 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. Blacklisting will eventually happen automatically when a customer fails to pick up an order."

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

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

Listing 29 – Upload replacement blacklist

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

"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***[37] 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.

4.12 – 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."

[37] 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 blacklist 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:

Tagging a scenario

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

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.

Tracey and Dishita's team have internally agreed that any scenario that will not be automated should be tagged as `@manual`. Remember, the feature file is intended to be business readable documentation, so tags should make sense to your product owner.

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.

The complete feature file can be found in Appendices, Listing 39.

4.13 – Review is not optional

Dishita and Tracey have formulated the concrete examples generated in the example mapping session , but their work needs to be reviewed. At the very least, the product owner must review the formulated scenarios to ensure that what has been documented accurately reflects their expectations. This is not an optional exercise, because ultimately it is the PO who is accountable for the product's specification.

It is ideal if the PO reviews scenarios 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 while the PO is reviewing the scenario.

We also recommend that other 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.

4.14 – 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.

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. The pair had to deal with:

- consistent phrasing and grammar structures
- reviewing whether all details are really 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 the scenarios of a single user story. 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 as the team reviews this feature file and tackles the challenges of evolving the documentation as the product grows.

5.1 – User stories are not 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 30 – Structure of the feature file created for the Blacklisting user story (scenario steps omitted for better readability)

```
Feature: Blacklisting

Rule: Don't offer pay-on-collection for blacklisted customers
  # Two scenarios merged, because being logged-in or not is incidental
  Scenario: Blacklisted customer is not offered pay-on-collection option

Rule: Customer's email OR phone must match those on blacklist
  Scenario Outline: Matching contact details with blacklist

Rule: Blacklisting is independent of logged-in / not-logged-in status
  Scenario: Not-logged-in customer is blacklisted
  Scenario: Logged-in customer is blacklisted

Rule: Blacklist is manually configurable
  @manual
  Scenario: Upload replacement blacklist
```

She doesn't see anything that causes major concern, so she posts a couple of brief message 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 blacklisting 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 too 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 blacklisting works, but there's one rule that's very specific about whether a blacklisted 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.[38] When first created by the original XP practitioners, stories were defined 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 Product Owner 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 *The BDD Books: Discovery*[39]*, 4.1*), are used to plan and track the delivery of a small functional increment. If we organize our feature files by story, then our living documentation will be structured according to the order in which we delivered the functionality. This is not useful to either the business or the delivery team.

If, on the other hand, we organize our feature files by coherent functionality, we 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 we create living documentation that delivers persistent value to all those that use, maintain, and enhance our 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.

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

[39]Nagy, Gáspár, and Seb Rose. *The BDD Books: Discovery - Explore behaviour using examples*. http://bddbooks.com/discovery.

Gaspar’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 oganizing feature files by functionality will make it easier to use them as documentation,” says Tracey. “What feature files shall we create?”

“I think that blacklisting 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 blacklisted customers’* rule isn’t part of the blacklisting feature.”

5.2 – Separation of concerns

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

“Blacklisting is a generic way of identifying unreliable customers in the system. Preventing a blacklisted 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 blacklisted 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 blacklisted 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 31 – 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: Not-logged-in customer chooses to collect order
    Scenario: Logged-in customer chooses to collect order

  Rule: Not-logged-in customers must supply acceptable contact details \
when placing an order for 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 \
collection
    Scenario: Estimated time of order completion is displayed
```

Listing 32 – Restructured blacklist feature file (`blacklisting.feature`)

```
Feature: Customer blacklisting

  Rule: Customer's email OR phone must match those on blacklist
    Scenario Outline: Matching contact details with blacklist

  Rule: Blacklisting is independent of logged-in / not-logged-in status
    Scenario: Not-logged-in customer is blacklisted
    Scenario: Logged-in customer is blacklisted

  Rule: Blacklist is manually configurable
    @manual
    Scenario: Upload replacement blacklist
```

Listing 33 – 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 blacklisted customers
  # Two scenarios merged, because being logged-in or not is incidental
  Scenario: Blacklisted customer is not offered pay-on-collection option
```

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 *blacklisting* 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[40] 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 discuss alternatives to the term "blacklist", but aren't able to come up with a suggestion that they prefer.

"I'm going to go back to the customers 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 practically prevent anyone from 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.

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

Tool support

As you've learnt 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 have just undertaken. Software developers have got used over 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 blacklist, customer-collection, pay-on-collection, logged-in and not-logged-in,” 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 [^MartaireLivingDoc], 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 use 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 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 41 for the final MarkDown gloassary file the team created.

- **Authenticated [customer]** – See *Logged in [customer]*.
- **Blacklist** – A list of contact details to identify customers that can only use a reduced set of features the site offers (e.g. they cannot use pay-on-collection). Currently we store email addresses and phone numbers on the blacklist.
- **Contact details** – The details that allow the customer to be contacted related to a particular order. For logged-in customers, the contact details registered to their account is used, for not-logged-in customers the contact details must be provided for the order.
- **Customer** – A 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.
- **Logged in [customer]** – A customer who has registered in the application earlier and authenticated with the registered credentials (e.g. email and password).

- **Order** The collection of food and beverage items that the customer has ordered at once.
- **Order completion** – The event (time) when the order processing has been finished by WIMP. This is usually when the items have been delivered but in some cases it can be earlier (e.g. *customer-collection*).
- **Pay on collection** – A 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** – The 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** – The selected method payment for the order.
- **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 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'*.
- Tags can be applied to a *Feature*, *Scenario*, *Scenario Outline*, or *Examples* table.
- Tags applied to a *Feature* are also applied to all *Scenario*s and *Scenario Outline*s within the feature file.
- There is no limit to the number of tags that can be applied to a *Feature*, *Scenario*, *Scenario Outline*, or *Examples* table.
- Multiple tags can be applied on the same line, or on consecutive lines

Common ways that tags are used include:

- selecting a subset of related scenarios (e.g. *'@release42'* – changes in the current 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`).

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 lifetime management (ALM) tools, test execution, 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:

- **Tracking scenario automation life-cycle with tags** We have described the BDD process in *The BDD Books: Discovery*[41]*, 4.1*. As the scenarios represent higher level specification elements of the system, the completion of a formulation-automation-implementation cycle 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 or 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 would 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 life-cycle, so this is usually

[41]Nagy, Gáspár, and Seb Rose. *The BDD Books: Discovery - Explore behaviour using examples*. http://bddbooks.com/discovery.

represented by tags. You can define your own set of tags for this. For example, we have used `@formulated` for marking 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 features*, although the scenarios are usually created within the scope of a user story, this relationship is not important after the story has been finished. 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. Our goal should be 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, which interface of the application that is going to be automated, 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. However, 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 not distracting, because its meaning is clear to the entire team. At the end of the day, our stakeholders are not only interested in the fulfilment of business

expectations, but also in the quality of the result in general. If these tags are kept to 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.). So, when defining automation-related tags always make sure that they are understandable by all stakeholders.

Gaspar'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 4.9, *Setting the context*), but we found that people naturally assumed that if we don't mention that the user logged in then they are not. 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. Always discuss and document all introduced tags with the entire team.

5.6 – Journey scenarios

"I'm concerned about the customer experience of blacklisting," 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 demand – 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 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."

"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 blacklisted customer's journey is different."

"That sounds a bit like a ***usecase***," 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 34 – 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 35 – 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."

"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 emphasise the difference."

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

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

The team sketch out another journey scenario:

Listing 36 – Blacklisted pay-on-collection journey

```
@journey
Scenario: Blacklisted pay-on-collection journey
  * Brian visits our website
  * he adds a pizza to his basket
  * he chooses to checkout as a guest
  * he enters blacklisted contact details
  * he can't choose to 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 blacklisted customers*'' rule."

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

"Right!" says Tracey. "There really are two different types of scenarios. ***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."

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 system behaviour
- 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 34 has very few concrete details (such as what pizzas are ordered or the precise address for delivery).

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.

Automate journey scenarios after the behaviour they utilize has been implemented

While there is value in journey scenarios, many BDD teams find that illustrative scenarios are all they need. Even 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.

There are other risks associated with journey scenarios that you should be aware of. They are slower to execute, difficult to debug when they fail, and costly to maintain as the product evolves. They live at the top of the test automation pyramid[42] and consequently should be used sparingly!

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.

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

There is no defined size for a feature. The smallest piece of validation is a feature as far as a developer or tester is concerned, while customers are normally interested in the ability to complete much larger activities. 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 are 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.

"I'd expect to find journey scenarios in their own feature files"

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 on the file system. The feature files collectively document the behaviour of the system, but are only useful when people can easily navigate to the part that they are interested in. A hierarchical tree of folders in a file system provides a simple way to structure our documentation that is identical to the traditional documentation that Dave is thinking about.

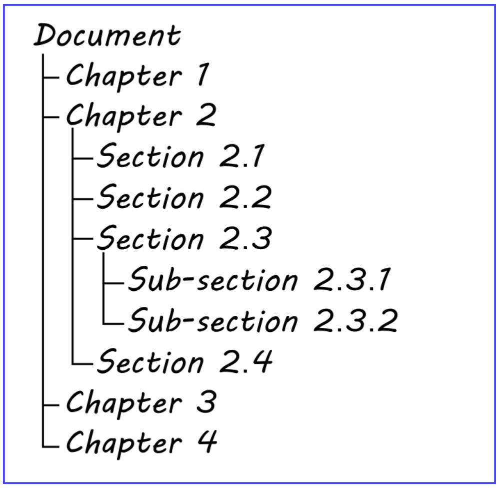


Figure 8 – Traditional documentation

“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:

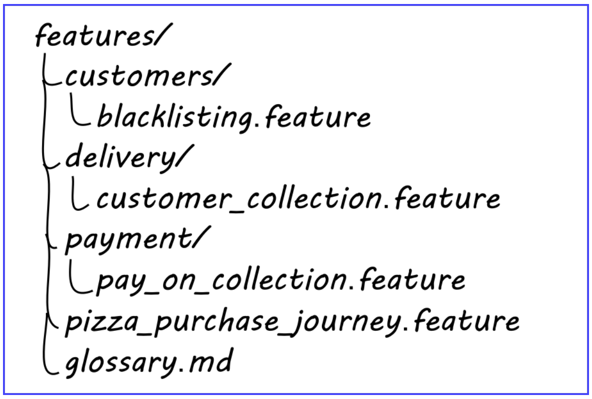

features/
customers/
blacklisting.feature
delivery/
customer_collection.feature
payment/
pay_on_collection.feature
pizza_purchase_journey.feature
glossary.md

Figure 9 – 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.”

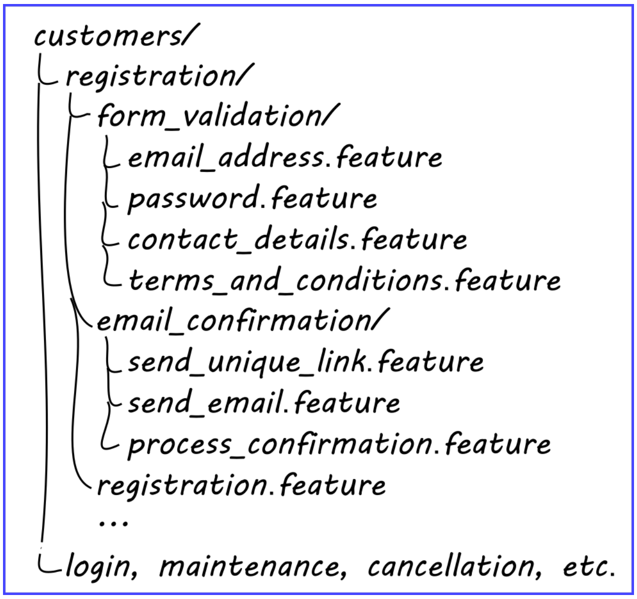
“There’s also all the existing behaviour that we could document,” says Tracey.

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."



```
customers/
└registration/
  ├form_validation/
  │ └email_address.feature
  │ └password.feature
  │ └contact_details.feature
  │ └terms_and_conditions.feature
  └email_confirmation/
  │ └send_unique_link.feature
  │ └send_email.feature
  │ └process_confirmation.feature
  └registration.feature
   ...
└login, maintenance, cancellation, etc.
```

Figure 10 – Customer 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 10 there are other features that will be reused. For example:

- *‘Change password’* will depend on *‘Password’* validation
- *‘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 (Section ?, *The feature description*). The best information to include is the name and relative file path of the dependency (see Listing 37).

“We could extract *‘Send email’* into a communications section” says Dave and he sketches it out for the team to see.

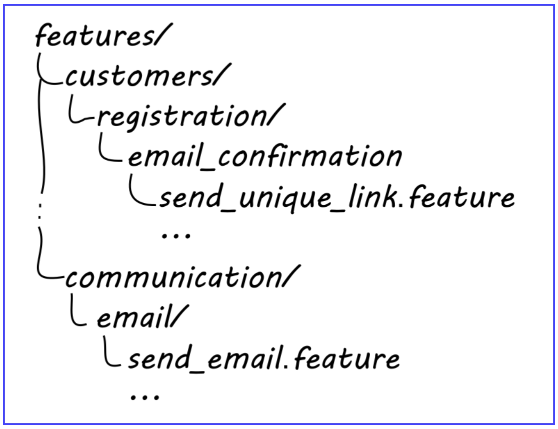


Figure 11 – Links across folder hierarchy

“Then in `send_unique_link.feature` we could document the dependency like this”

Listing 37 – 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 Product Owners

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 the 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.

5.10 – What we just learned

In this chapter the WIMP team focused on the ongoing challenges the teams need to address to build up 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 team reviewed the scenarios they formulated for the user story and rearranged them according the functionality of the system they contribute to. The WIMP team realized that some of the scenarios of the recent stories belong to a separate feature, so they extracted them to a new feature file. Similar review and refactoring activities are common among successful BDD teams and they 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 a classic 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 part way 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"[43]), then the majority of code that exists is legacy code. Therefore, most software professionals spend some 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.

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 (see *The BDD Books: Discovery*[44]). 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. 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 the 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. However, 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, this approach has a negative effect on the architecture of the implementation. Systems that are not implemented in response

[43]Feathers, Michael C., Working Effectively With Legacy Code. Prentice Hall, 2005. Print.

[44]Nagy, Gáspár, and Seb Rose. *The BDD Books: Discovery - Explore behaviour using examples*. http://bddbooks.com/discovery.

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 {#sec-journey-scenarios}, we should minimize the amount of end-to-end automation. There will be much more discussion of this in the next book, *The BDD Books: Automation with SpecFlow*[45].

6.2 – Incremental documentation

Very few teams will be able 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 of 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 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.

[45]Nagy, Gáspár, and Seb Rose. *The BDD Books: Automation with SpecFlow*. In preparation. http://bddbooks.com/specflow.

There will be a great temptation to document journeys rather than focused behaviours. This temptation should be resisted. 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

Often, there are 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. With this approach you would use the team's source control system to systematically identify the parts of the system that often change and document them first.

Gaspar'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. But 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.

The Chinese proverb[a] is true for BDD as well: The best time to start with BDD is at the beginning. The second best time is now.

[a]The best time to plant a tree was 20 years ago. The second best time is now.

6.3 – Manual test scripts

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

The team 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.

These conflicting goals mean that it is almost never appropriate to naively formulate manual test scripts, simply to automate them. Teams that have tried that approach have found that the formulated scripts have little value as documentation and yet 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 will 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 needs to assess and analyze each script in turn, perhaps by reverse engineering example maps like the WIMP team did in {#ch-cleaningup}. 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.

6.4 – Teams need skilled testers

There's a common Internet meme that says "automate all the things." There may come a time when artificial intelligence makes that possible, but for the time being many aspects of software development rely on skilled, human professionals. Skilled testers are critical to delivering quality products. Despite the huge benefit of test automation, organizations should do everything they can to retain skilled testers, especially those with extensive domain knowledge, because:

- 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
- 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.

Exploratory testing

Elizabeth Hendrickson gave a very influential CAST keynote in which she explained that to effectively test a system the team would need to undertake both ***checking*** and ***exploration***. In her opinion, the scenarios and automated tests constitute checks that exercise parts of the system and assert that it behaves in the way that we expect. Exploration is a creative activity undertaken by professionals who have testing skills and an understanding of the technical architecture of the system.

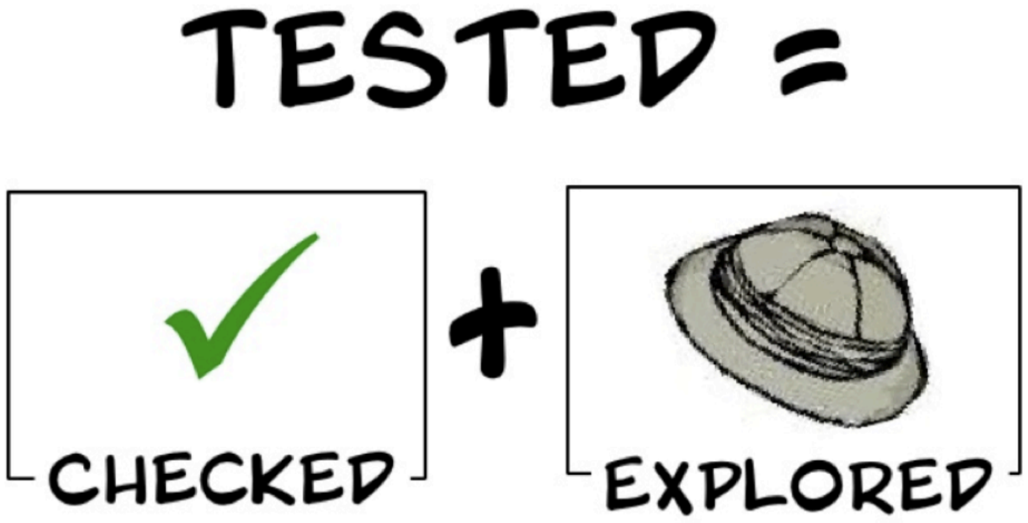


Figure 12 – Tested = Checked + Explored (with thanks to Elizabeth Hendrickson)

Exploratory testing[46] is a large topic, but the headline message is clear: *manual exploratory testing is essential*

Shift left

The realization that we could catch many of the defects that lead to rework *before* development was even started has been around for decades. This realization was given a name in the 21st century - ***Shift Left***. The name derives from the diagram below, where the X-axis represents time. By moving some activity left along the X-axis, that means that it is being performed earlier. In the case of Shift Left, the activity being performed earlier is testing.

Since testing is frequently seen as an activity that can only be performed after the code has been written, people have struggled with realizing the benefits. The issue is that Hendrickson formula of Tested = Checked + Explored is too narrow. Jerry Weinberg defined testing as "the process of gathering information with the intent that the information could be used for some purpose." When seen in this light, we can extend Hendrickson's formula: Tested = Discovered + Checked + Explored

[46]Hendrickson, Elisabeth, Ward Cunningham, and Jacquelyn Carter. *Explore It! Reduce Risk and Increase Confidence with Exploratory Testing*. Dallas: Pragmatic helf, 2013. Print.

So, even if exploration could be performed automatically (which is not possible at present), our teams will still need skilled testers during Discovery. It's at this point that the team first *tests* the specifications of a story by challenging their understanding using concrete examples. If there were no tester participating in the Discovery workshop, then many potential failure conditions would be ignored. The *checks* would therefore not check for them and they would only be discovered after development had been completed, necessitating wasteful rework.

6.5 – 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 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 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 probably even more 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. There are also other testing techniques (we mentioned exploratory testing as a prominent example) that complement the benefits provided by BDD.

What's next

You've made it to the end. Congratulations!

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

Where are we now?

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 more is there?

We hope that you have already explored the practice of discovery with us by reading our previous book, *The BDD Books: Discovery*[47].

This book covers the Formulation part of the BDD approach, but there's still more to BDD than we have covered.

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 *The BDD Books: Automation with SpecFlow*[48].

[47]Nagy, Gáspár, and Seb Rose. *The BDD Books: Discovery - Explore behaviour using examples.* http://bddbooks.com/discovery.

[48]Nagy, Gáspár, and Seb Rose. *The BDD Books: Automation with SpecFlow.* In preparation. http://bddbooks.com/specflow.

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 reference

Coming soon...

- Feature file: Section 3.1, *Feature files*
- Block keywords (Feature, Background, Rule, Scenario, Scenario Outline, Examples): Section 3.3, *Gherkin basics*
- Step keywords (Given, When, Then, And, But): Section 3.3, *Gherkin basics*, Section 3.5, *Scenario structure*
- Feature name, description: Section 3.4, *Diving into the feature file*
- Rule: Section 3.4, *Diving into the feature file*
- Scenario, Scenario name: Section 3.5, *Scenario structure*
- Scenario step: Section 3.5, *Scenario structure*
- Data table: Section 3.6, *Data tables*, Section 3.8, *Keep tables readable*
- Scenario Outline, Examples table: Section 3.7, *Scenario outlines*, Section 3.8, *Keep tables readable*
- Then steps, should style: Section 4.5, *There's no "I" in "Persona"*
- Comment: Section 4.8, *Commenting in feature files*
- Block keyword description: Section 4.8, *Commenting in feature files*
- Background: Section 4.6, *Background*

Anti-patterns

- Too many context steps: Section 3.5, *Scenario structure*
- Create one feature file for each user story: Section 5.1, *User stories are not features*
- Extensive usage of feature description can lead to outdated documentation Section 3.4, *Diving into the feature file*

- Non-descriptive scenario name: Section 3.5, *Scenario structure*
- Describing actions in *Given* steps (instead of describing the intended context): Section 3.5, *Scenario structure*
- Multiple *When* steps: Section 3.5, *Scenario structure*
- Multiple outcomes (*Then* steps): Section 3.5, *Scenario structure*
- Examples table rows without purpose: Section 3.7, *Scenario outlines*
- Large Examples and Data tables: Section 3.8, *Keep tables readable*, Section 3.7, *Scenario outlines*
- Overuse of Scenario Outlines, use them as programming construct: Section 3.7, *Scenario outlines*
- Misaligned Gherkin table columns: Section 3.8, *Keep tables readable*
- Ineffective formulation workshop: Section 4.3, *Too many cooks*
- Formulated scenarios hard to automate: Section 4.3, *Too many cooks*
- Formulated scenarios too technical: Section 4.3, *Too many cooks*
- Use of "I": Section 4.5, *There's no "I" in "Persona"*
- Extensive use of comments and descriptions: Section 4.8, *Commenting in feature files*
- Use Background as a programming construct: Section 4.6, *Background*

Formulated feature files

The feature files are also availabe for download from the book website[49].

`customer_collection.feature`

The scenarios of this feature file are discussed in detail in Chapter 3, *Our first feature*.

[49] http://speclink.me/bookfeaturefiles

Listing 38 – `delivery/customer_collection.feature`

```
Feature: Customers can collect their orders

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

  Scenario: Not-logged-in customer chooses to collect order
    Given the customer is not logged in
    When they choose to collect they order
    Then they should be asked to supply contact and payment details

  Scenario: Logged-in customer chooses to collect order
    Given the customer is logged-in
    When they choose to collect they order
    Then they should be asked to supply payment details


Rule: Not-logged-in customers must supply acceptable contact details wh\
en placing an order for collection

  Scenario Outline: Contact details supplied ARE acceptable
    Given the customer is not logged in
    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 not logged in
    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 unacce\
ptable

    Examples:
      | description            | unacceptable contact details |
      | Name must be provided | Phone, Email                  |
      | Name is not enough    | Name                          |
      | Only Email            | Email                         |
      | Only Phone            | Phone                         |


Rule: Customer should be informed when their order will be ready for co\
llection

  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
```

blacklisting.feature

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

Listing 39 – `customers/blacklisting.feature`

```
Feature: Customer blacklisting

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

  Scenario Outline: Matching contact details with blacklist
    Given the blacklist contains <details>
    And the customer□s contact details are <contact details>
    When the blacklist is checked
    Then customer should be treated as <blacklisted?>

    Examples:
      | description   | details                       | contact details\
         | blacklisted?    |
      | Both match    | peter@example.com, +123456789 | peter@example.c\
om, +123456789 | blacklisted     |
      | Neither match | peter@example.com, +123456789 | simon@spam.me, \
+987654321     | not blacklisted |
      | Email matches | peter@example.com, +123456789 | peter@example.c\
om, +987654321 | blacklisted     |
      | Phone matches | peter@example.com, +123456789 | simon@spam.me, \
+123456789     | blacklisted     |


Rule: Blacklisting is independent of logged-in / not-logged-in status

  Scenario: Not-logged-in customer is blacklisted
    Given a customer is not logged in
    And the contact details the customer supplied for the order are pet\
er@example.com, +123456789
    And the blacklist contains peter@example.com, +123456789
    When the blacklist is checked
    Then the customer should be treated as blacklisted

  Scenario: Logged-in customer is blacklisted
    Given a customer is logged in
    And the customer's contact details are peter@example.com, +123456789
    And the blacklist contains peter@example.com, +123456789
    When the blacklist is checked
    Then the customer should be treated as blacklisted


Rule: Blacklist is manually configurable

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

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 40 – `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 blacklisted customers

  # Two scenarios merged, because being logged-in or not is incidental
  Scenario: Blacklisted customer is not offered pay-on-collection option
```

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

```

glossary.md

The glossary file is discussed in detail in Chapter 5, *Organizing the documentation*.

Listing 41 – `glossary.md`

```
# Glossary

## Domain terms

* **Authenticated [customer]** -- See *Logged in [customer]*.
* **Blacklist** -- A list of contact details to identify customers that\
 can only use a reduced set of features the site offers (e.g. they cann\
ot use pay-on-collection). Currently we store email addresses and phone\
 numbers on the blacklist.
* **Contact details** -- The details that allow the customer to be cont\
acted related to a particular order. For logged-in customers, the conta\
ct details registered to their account is used, for not-logged-in custo\
mers the contact details must be provided for the order.
* **Customer** -- A person who visits the *Where Is My Pizza* site to o\
rder pizza and other items sold. Unauthenticated (anonymous) users are \
also treated as customers.
* **Customer collection** -- A special delivery method, where the custo\
mer picks up the ordered items themselves.
* **Delivery method** -- The selected method for delivering the ordered\
 items.
* **Logged in [customer]** -- A customer who has registered in the appl\
ication earlier and authenticated with the registered credentials (e.g.\
 email and password).
* **Order** The collection of food and beverage items that the customer\
 has ordered at once.
```

```
* **Order completion** -- The event (time) when the order processing ha\
s been finished by WIMP. This is usually when the items have been deliv\
ered but in some cases it can be earlier (e.g. *customer-collection*).
* **Pay on collection** -- A payment method where the customer pays (wi\
th cash or card) when they pick up the order. This method can only be u\
sed for *customer-collection*.
* **Payment details** -- The details required to fulfil the payment. Th\
e exact details depend on the *payment method*, e.g. for card payment w\
e need the card details, but for *pay-on-collection* there are no extra\
 details needed.
* **Payment method** -- The selected method payment for the order.
* **Visitor** -- See *Customer*.

## Tags

* **@manual** -- The scenario is verified by manual checks, has to be e\
xcluded for automated test execution
* **@journey** -- The scenario does not illustrate a particular rule bu\
t describes a longer user journey
```

Index