Extracted from:

The Cucumber Book

Behaviour-Driven Development for Testers and Developers

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

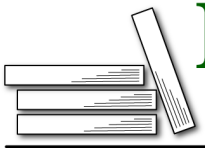
The Pragmatic Programmers

The Cucumber Book

Behaviour-Driven Development for Testers and Developers

Matt Wynne
and Aslak Hellesøy

edited by Jacquelyn Carter



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

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

Printed in the United States of America.
ISBN-13: 978-1-934356-80-7
Printed on acid-free paper.
Book version: P2.0—August 2012

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

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

3.1 What's Gherkin For?

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

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

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

Concrete Examples

By using real-world examples to describe the desired behavior of the system we want to build, we stay grounded in language and terminology that makes sense to our stakeholders: *we're speaking their language*. When we talk in terms of these examples, they can really imagine themselves using the system, and that means they can start to give us useful feedback and ideas before a line of code has been written.

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

> *Customers should be prevented from entering invalid credit card details.*

This is an example of what agile teams often call *acceptance criteria* or *conditions of satisfaction*.[1] We use the word *acceptance* because they tell us what

1. *Agile Estimating and Planning* [Coh05]

the system must be able to do in order for our stakeholders to find it acceptable.

The previous requirements statement is useful, but it leaves far too much room for ambiguity and misunderstanding. It lacks *precision*. What exactly makes a set of details invalid? How exactly should the user be prevented from entering them? We've seen too many projects get dragged into the tar pit[2] by these kind of worthy but vague statements. Let's try illustrating this requirement with a concrete example:

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

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

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

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

Try This

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

Executable Specifications

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

---

2. The tar pit metaphor comes from the seminal book by Fred Brooks, *The Mythical Man Month: Essays on Software Engineering* [Bro95].
3. *Extreme Programming Explained: Embrace Change* [Bec00]

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

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

Download gherkin_basics/sample.feature

```
Feature: Feedback when entering invalid credit card details

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

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

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

  Scenario: Expiry date invalid
    When I enter a card expiry date that's in the past
    And all the other details are correct
    And I submit the form
    Then the form should be redisplayed
    And I should see a message telling me the expiry date must be wrong
```

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

4. A note for the pedantic reader: the Gherkin language does have a grammar enforced by a parser, but the language is not Turing Complete.

de vil forstå. (Gherkin lets you to write your features in a language they will understand.) Tocino grueso! (Chunky Bacon!) More on that later.

3.2 Format and Syntax

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

Keywords

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

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

We'll spend the rest of this chapter exploring how to use the most common of these keywords, which will be enough to get you started writing your own Cucumber features. We'll come back to look at the remaining keywords later in Chapter 5, *Expressive Scenarios*, on page ?.

Dry Run

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

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

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

3.3 Feature

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

Here's an example:

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

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

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

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

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

- Scenario
- Background
- Scenario Outline

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

3.4 Scenario

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

A Template for Describing a Feature

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

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

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

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

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

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

Scenarios all follow the same pattern:

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

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

Given, When, Then

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

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

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

And, But

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

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

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

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

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

Replacing Given/When/Then with Bullets

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

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

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

Stateless

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

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

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

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

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

Name and Description

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

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

Try This

- Now that you understand how to write Gherkin scenarios, try converting some of the concrete examples you wrote down earlier for your own project into Gherkin.
- Show them to someone who knows nothing about your project, and ask them what they think your application does.

Matt says:
Take Care with Your Naming Scenarios

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

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

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

- Practice describing what you're doing with `Given/When/Then` while you're doing everyday things such as starting your car, cooking breakfast, or switching channels on the TV. You'll be surprised how well it fits.