Extracted from:

# The Cucumber Book, Second Edition

## Behaviour-Driven Development for Testers and Developers



The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Cucumber 2.4

# The Cucumber Book

Second Edition

## Behaviour-Driven Development for Testers and Developers

Matt Wynne
Aslak Hellesøy
with Steve Tooke

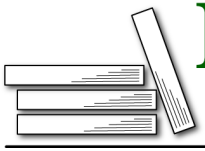
*edited by Jacquelyn Carter*

# The Cucumber Book, Second Edition

## Behaviour-Driven Development for Testers and Developers

Matt Wynne
Aslak Hellesøy
with Steve Tooke

The Pragmatic Bookshelf
Raleigh, North Carolina





Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Jacquelyn Carter (editor)
Gilson Graphics (layout)
Janet Furlow (producer)

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.



Printed in the United States of America.
ISBN-13: 978-1-68050-238-1
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—February 2017

In the real world, you don't always get the luxury of working on shiny new code. Reading a book like this can be a little bit frustrating because the examples all deal with nice, simple problems, usually in a new codebase.

We all know that software development isn't really like that.

Even in the best teams we've worked on, there have always been a few dark and ugly corners of the codebase where nobody really wanted to go. People would sometimes disappear into there for days at a time and emerge exhausted and confused, blinking in the bright sunlight.

Like an old abandoned mine, those areas of the codebase are dangerous. The code is fragile, and the slightest change can cause other parts of the code to collapse and stop working. Everyone knows this, and that's why people are reluctant to go in there: it's stressful work.

If you had to go down an old mine tunnel, what would you do to make it safer? You would probably build some scaffolding to hold the roof of the tunnel up so that it did not collapse—just enough to help you get in and get out again safely.

You can think of automated tests like this. When you have to make a change to an area of the code that you know is brittle, it's scary. What is the thing that you're most afraid of?

*Breaking something.*

Automated tests can help you keep this fear at bay. If you make a change and other parts of the code start to collapse, the tests will give you a warning while you still have a chance to do something to correct it. But what if you don't have any tests?

Even if you have recognized that automated testing can help your team write better code, the prospect of adding tests to a large legacy codebase can seem overwhelming. In this chapter, we'll show you some simple techniques you can use to help to solve the problem gradually, giving you added benefit each step of the way.

This isn't a technical recipe, but it will give you some useful techniques and encouragement if you're facing this situation.

Let's start with the first tool you'll need when you enter the mine: a flashlight.

## Characterization Tests

In his excellent book *Working Effectively with Legacy Code [Fea04]*, Michael Feathers talks about two different types of tests, which he calls *specification tests* and *characterization tests*.

Specification tests are the ones we've been talking about in the rest of this book. They check that the code does what it's supposed to. Ideally you write them before you write the code itself and use them as a guide to help you get the code into the right shape.

Characterization tests are different. You can think of them more like a science experiment, where you test the properties of a mysterious substance by boiling it or mixing it with other substances to see how it reacts. With characterization tests, the aim is just to understand what the system currently does.

Characterizations tests apply perfectly to legacy code that has limited or no tests and where the code is hard to read and understand. Here is our recipe, adapted from Michael Feathers's, for creating a Cucumber characterization test:

1. Write a scenario that exercises some interesting—but mysterious—behavior of your system.
2. Write a `Then` step that you know will fail.
3. Wire up the step definitions and run the scenario. Let the failure in the `Then` step tell you what the actual behavior is.
4. Change the failing `Then` step so that it describes the actual behavior of the system.
5. Repeat.

As an example, let's suppose we're making some changes to the checkout system for a supermarket. We've been asked to add a new special offer to the system, which was developed a couple of years ago by a firm of expensive management consultants, who are sadly no longer with us. We've grabbed the source code, managed to get the system to spin up on our development machine, and have been poking around through the user interface. The code is pretty gnarly and it's hard to tell exactly what's going on, but we noticed something about shampoo. It looks like there might be an existing special offer on shampoo bottles already baked into the code, but it's hard to tell exactly what the rules are. Let's write a test:

```
Scenario: Buy Two Bottles of Shampoo
  Given the price of a bottle of shampoo is $1.99
  When I scan 2 bottles of shampoo
```

```
  Then the price should be $0
```

We know that's going to fail: they're not going to give us the shampoo for free, are they? (Steps 1 and 2.)

So, we wire up the step definitions to the system and run the scenario. Here's what happens:

```
Feature:

  Scenario: Buy Two Bottles of Shampoo
    Given the price of a bottle of shampoo is $1.99
    When I scan 2 bottles of shampoo
    Then the price should be $0

      expected: 0
           got: 1.99

      (compared using ==)
       (RSpec::Expectations::ExpectationNotMetError)
      ./features/step_definitions/steps.rb:8
      features/bogof.feature:5

Failing Scenarios:
cucumber features/bogof.feature:2

1 scenario (1 failed)
3 steps (1 failed, 2 passed)
```

A-ha! Now we know what price the system is charging for two $1.99 bottles of shampoo: $1.99. (Step 3.)

Now we update the `Then` step of our scenario to reflect our new understanding of what the system does (step 4):

```
Scenario: Buy Two Bottles of Shampoo
  Given the price of a bottle of shampoo is $1.99
  When I scan 2 bottles of shampoo
  Then the price should be $1.99
```

We run the scenario again, and this time it passes. Great! We've added our first characterization test. It's not much, but it's a start, and we know that whatever we do to the code from now on, we'll always have this scenario to tell us whether we break this particular aspect of its behavior.

It looks like there's a buy-one-get-one-free offer on shampoo, but we can't be sure from this single example. Following the recipe, we now need to repeat steps 1 to 4 and add some more scenarios to give us some more clues as to what the system is doing.

We can refactor the scenario into a scenario outline (see Chapter 5, *Expressive Scenarios*, on page ?) to allow us to try different amounts and prices:

```
Feature: Special Offers

  Scenario Outline: Shampoo
    Given the price of a bottle of shampoo is $1.99
    When I scan <number> bottles of shampoo
    Then the price should be <total>

    Examples:
      | number | total |
      | 1      | $0    |
      | 2      | $1.99 |
```

We can now add examples into the table, one at a time. Each time we start with a silly value for the total price and then run the scenario and let it tell us what the real total is. Then we update the scenario to document that behavior.

## Squashing Bugs

Our legacy application is big and mysterious, and we could spend an awfully long time writing characterization tests if we just started adding them aimlessly. So, assuming that we do want to grow our suite of automated tests, where should we start?

One of the best ways to start practicing with Cucumber is when you have a bug to fix. Bug reports generally come to you in the form of an example, so they're nice and easy to translate into Cucumber scenarios.

1. Translate the bug report into a Cucumber scenario. Show the scenario to the person who reported the bug, and ask them whether it accurately describes what they were doing.
2. Wire up the step definitions and run the scenario. It should fail in the same way as the real system did when the bug was first discovered. The bug is trapped!
3. Examine the defective code, and think about what you'll need to change. If you're unsure or worried about breaking some existing behavior, write a characterization test scenario for it.
4. Fix the code so that the bug report scenario passes.
5. Run the characterization tests to check you didn't break anything.

You'll find that the bug report scenario acts as a driving force, helping you focus on the code instead of having to keep running a manual test to see whether the fix has worked.

Despite your new characterization tests, you may still find that you missed something and introduced a new bug. That happens sometimes, and it would have happened just the same if you hadn't used any automated tests. If you use the same recipe to fix that new bug and each new bug that comes along, then gradually, over time, you'll build up a solid suite of Cucumber scenarios. Not only will those scenarios prevent any of these bugs from recurring, but they'll start to document the behavior of the system for anyone doing maintenance on it in the future.

**Matt says:**

### Features for Bug Fixes

We've just told you to use Cucumber to help you reproduce and trap bugs, so it's quite likely that you might end up writing a new feature and calling it something like features/verify_bugfix_52553.feature. We've done this ourselves, and trust us—it doesn't make for great documentation!

There are two ways around this problem. If you judge that the scenario is relevant enough to keep as business-facing documentation, then just talk to your team about where to file it away tidily in your features. If you're adding features to a legacy system, that might will mean creating a new empty feature file about a whole big area of the system and then just adding a single scenario to describe one aspect of its behavior. That's OK; you've created a space where other people can add more scenarios as they come up. Try not to mention the bug itself as you write the scenario—just describe the behavior you want as though it's always been there.

On the other hand, if you decide that it's such an obscure edge case that it isn't interesting enough to remain as business-facing documentation, you can just delete the scenario once you've fixed the bug. That's assuming you've used a unit test to cover the changes you've made to make the fix, so you can be safe in the knowledge that the bug won't reappear.

## Adding New Behavior

When you think about it, the process of adding new behavior to a system isn't so different from fixing a bug. The overall goal is to change some aspect of how the system behaves, without breaking anything else.

Just as with bug fixing, we can use characterization tests to pin down the surrounding behavior of the system to be sure it isn't dislodged by our work. Characterization tests can also be useful before that, while the new feature is first being considered. You can use them to understand how much work is involved in the new feature by clarifying exactly what the system currently does.

Here's our recipe for adding new behavior to a legacy system:

1. Examine the new feature. If necessary, write a few characterization scenarios to examine and clarify the current behavior of the system in that area.
2. Now, with the new feature in mind, modify those scenarios or write new ones to specify the desired new behavior.
3. Run through the scenarios with your team's stakeholder representative to check that you're about to build the right thing. Correct the scenarios with them if necessary.
4. Run the scenarios. If more than one fails, pick one, and examine the code you'll need to change to make it pass.
5. Write any extra characterization tests you need to give you the confidence to change that code.
6. Change the code to make the scenario pass.
7. Repeat from step 4 until all the scenarios pass.

Sometimes we find that when we're about to implement the change (step 6), we see that the code we're about to change is responsible for doing things that we hadn't anticipated when we wrote the original characterization tests in step 1. At this point, we'll stop and add some new characterization tests (step 5) first if we think there's a significant risk we could break something.

Just as with bug fixing, following this process means that you quickly get the benefits of automated testing in the areas of the system where you most need them: the ones that are most prone to change and instability.

As you gradually build up your suite of Cucumber scenarios for your legacy application, you'll find you have more confidence to refactor and clean up the code. This becomes a virtuous cycle, with cleaner code causing less bugs.

## Code Coverage

Code coverage tools allow us to discover which specific lines of code in the system were executed during a test. When you are starting to add tests to a legacy application, it can be really useful to know your code coverage:

- If you know that a line of code isn't covered by your tests yet, you can see more clearly what kind of a characterization test you need to write.
- If you know that a line of code is covered by your tests, you can be more confident in refactoring or changing it.

There is no built-in support in Cucumber for collecting code coverage statistics. Instead, you need to install the SimpleCov[1] gem and add a couple of lines to your env.rb:

```
require 'simplecov'
SimpleCov.start
```

When your features finish running, you should find a code coverage report in coverage/index.html.

### Track Your Code Coverage for Team Encouragement

Code coverage is a much maligned metric of code quality. We've heard of managers who've demanded that their teams achieve 100 percent code coverage. We've also heard of teams that, under those circumstances, simply wrote a load of tests with no assertions in them. The tests were exercising all the code, but they weren't testing anything!

As a source of internal feedback for the team, though, code coverage can be useful, especially when you're embarking on a long-term project to bring a legacy system under the control of automated tests. Start measuring your code coverage today, and come back every few weeks to measure it again. You should be pleased to see the number is climbing all the time.

Another great related metric to track, which your manager might be more impressed with, is the defect rate (number of new bugs discovered per week). If your Cucumber tests are going to add any value to your customers, this is the most likely place it will surface. As you add more test coverage, your team will make fewer and fewer mistakes, and you should see your defect rate drop.

## What We Just Learned

Working with legacy code is always a challenge, but you can use Cucumber tests to help make it a much more enjoyable challenge.

- Be pragmatic! Don't exhaust yourself trying to retrofit a complete set of Cucumber features for everything the system already does. Instead, add them gradually, one at a time, as you need them.
- Use characterization tests to help you understand what the system is already doing and to give you some security before you make a change.
- Every time you discover a bug, trap it with a new Cucumber scenario. Every time you add new behavior to the system, start by describing it with a Cucumber feature.

1. http://rubygems.org/gems/simplecov

- Use code coverage to give you and your team feedback and encouragement about your progress in getting the system under test.
- Enjoy the newfound confidence with which you can refactor and clean up the code that's covered by the tests.