Iron Chef Cucumber: Cooking Up Software Requirements That Get Great Results

Chris Cowell

Christopher Cowell, LLC

christopher.w.cowell@gmail.com

Abstract

Gathering requirements with behavior-driven development (BDD) tools like Cucumber seems simple at first, but at Cambia Health Solutions we discovered how hard this tool can be to use correctly and how quickly teams can sour on it when they don't understand its nuances. After two years of experimenting with Cucumber and learning how to harness its power, Cambia has successfully made it an essential part of the company's software development process. Based on this experience, I'll share best practices, anti-patterns, and real-world examples to help you use Cucumber and other requirements-gathering tools more artfully and effectively.

This discussion is aimed at QA people, developers, UX designers, product owners, and development managers who use Cucumber or other BDD tools to gather software requirements. You'll learn how you might be using Cucumber in unhelpful ways, how to write crisper and more understandable requirements, and how to make it a pleasure for everyone on your development team to write, read, and implement those requirements. Although the talk looks at requirements through the lens of Cucumber, most of the content is relevant to anyone who reads or writes software requirements no matter what tools or processes they use.

Biography

Chris Cowell is an independent technical trainer, writer, and coach based in Portland, Oregon. He developed and tested software for two decades at Cambia, Puppet, Jive, Oracle, and Accenture, and in a previous life wrote for the Let's Go travel guidebook series. Since realizing that he likes people more than computers, he now focuses on technical training. His specialties are Cucumber, behavior-driven development, and helping less-technical folks conquer impostor syndrome and feel comfortable and confident as they onboard at software companies. Chris studied computer science at Harvard and philosophy at Berkeley, but really learned to teach by showing his aging parents how to use email.

1 Introduction

If you grew up in the ’70s or ’80s, you might remember the classic board game Othello (also known as Reversi). My childhood Othello box had the slogan “seconds to learn, a lifetime to master,” which, I discovered as my brother beat me again and again, was a good description of the game’s deceptive depth.

Cucumber, a tool for supporting behavior-driven development (BDD), can be described the same way. Because you can learn the basics in 60 seconds, it’s easy to be lulled into thinking you fully understand it. But after working with Cucumber for a year as a QA person at Cambia Health Solutions, and then for another year as a Cucumber consultant, I find myself constantly learning more about it and changing my mind about how best to use it. I’m not alone in this regard: even the people who designed Cucumber are evolving their understanding of how to use it most effectively.

In this paper I’ll give a lightning-fast introduction to Cucumber for people who have never used it, and then walk through some of the best practices that I’ve developed. Most of these best practices are relevant to anyone who gathers business requirements regardless of whether they use Cucumber to do so. But I’ll explain these strategies through the lens of Cucumber, and all examples will use Cucumber’s syntax (which you may see referred to elsewhere as “Gherkin”). Off we go!

2 What is Cucumber?

I said that you could learn the basics of Cucumber in 60 seconds, so start your stopwatch now.

Cucumber supports BDD in two ways. First, it gives a formal structure for writing and organizing software requirements. Second, it allows development teams to write code to turn those requirements into executable tests. The end result is magical. You get an organized list of requirements that all the software’s stakeholders can understand, and you have a constantly updated view of how many of those requirements have been delivered by the development team. It’s a dream tool for anyone who wants to write software using BDD, but is also fantastic if you use a more traditional write-code-first, write-tests-later approach.

Cucumber calls its requirements “scenarios.” Each scenario has a title and a set of **Given**, **When**, and **Then** statements: given some state of the system, when the user does something, then something else happens.

For example, here’s a typical scenario for a web app. It specifies that users must log in before they can use the app:

```
Scenario: Require login
      Given Anne is not logged in
      When she navigates to any page
      Then it redirects her to the login screen
```

Scenarios can get more complicated than this, but some really are just this straightforward.

One additional complexity: you can use **And** or **But** keywords to allow multiple **Given** or **Then** statements in a single scenario:

```
Scenario: Require login
      Given Anne is not logged in
      And the system does not recognize her IP address
      When she navigates to any page in the app
```

```
            Then it redirects her to a login screen
            And it gives her the option to create a new account
```

Believe it or not, you now know enough to use Cucumber! But because of its Othello-like hidden complexity, you don't know enough to use Cucumber effectively. Let's see what we can do to fix that.

3 Best practices

I mentioned that Cucumber has two parts: the requirements-gathering part and the executable test part. The best practices I'm presenting here all have to do with the first of these: they are about how to write good requirements. A discussion of the best practices for turning those requirements into tests deserves a whole separate paper.

Out of all the techniques I've learned from using Cucumber, I'll present the seven that have had the biggest impact on me and my teams. These tips will help you write thorough requirements that are understandable by all the software's stakeholders and will help your team deliver high quality software quickly.

For each of these best practices I'll explain a technique and include a made-up but realistic example. The best practices are presented in no particular order.

3.1 Context

Imagine that we're writing a web app for ordering high-priced, unnecessarily fancy custom-made ice through a startup called Rose City Artisanal Ice.

We want to use Cucumber to write high-level, feature-focused requirements for this web app, and we need to share those requirements with both technical and non-technical stakeholders. We intend to have each requirement understood and approved before developers write a single line of product code to satisfy that requirement. In other words, we're following strict BDD procedures.

3.2 Write like you talk

The first best practice is to write requirements using a natural, normal tone and everyday vocabulary. In other words, write as if you were explaining the requirement to a friend in a casual conversation.

This might sound obvious, but it's a common problem. People often write specifications using a style they'd never use when talking with a colleague. Sometimes they write in a compressed, almost cryptic way, probably thinking that this seems more "high tech" or is more efficient. At other times they write too much, using convoluted, unnecessarily complicated prose. This probably comes from a belief that more language is better language, or that specifying every last detail is a necessary condition of good requirements. But when requirements are either too compressed or too verbose, they're hard to read and understand.

Let's illustrate this principle with an example. Say we want to let a web app user filter their ice options based on the source of the water it's made from. Here's a scenario that captures that requirement but is too compressed:

```
# bad example: too compressed
Scenario: filter source
```

```
            Given Barbara searches ice
            When filter set to spring water
            Then only spring water ice
```

Here's a scenario that goes too far in the other direction, expressing the requirement with unnecessarily complex language and too many words:

```
      # bad example: too complex
      Scenario: filter the results of a search for ice, using the source of the
                 water from which it is made
            Given Barbara is searching for ice but would like that search to exclude
                   ice made from any water source other than her preferred source of
                   spring water
            When Barbara activates the filter functionality within the list of
                  results returned by the search feature
            Then the only ice that is presented to Barbara within the search results
                  is ice made from spring water, which is her preferred water type
```

Finally, here's a well-written scenario that falls somewhere between the two previous examples. It uses a conversational style and proper but succinct English:

```
      Scenario: filter ice by water source
            Given Barbara is searching for ice
            When she filters out ice not made from spring water
            Then the results show only spring water ice
```

Now that sounds like something you might actually say out loud in a conversation.

3.3 Be careful with your logic

The next best practice is to double- or even triple-check any requirements that use logical terms like "and," "or," "not,", "all," and "only". Even though programmers are used to thinking carefully about logic as they code, it's easy for them—and anyone else who writes Cucumber scenarios—to overlook logical nuances in ways that break their requirements.

Consider this scenario about selecting the region that the ice comes from:

```
      # bad example: incorrect logic
      Scenario: filter ice by region of origin
            Given Claire is searching for ice
            When she requests only Alaskan ice
            Then the results show all the Alaskan ice options
```

Do you see the problem in this requirement? There's a logical fault that could potentially cause the scenario to fail (if run as a test) when it should pass, and pass when it should fail. And even if it's not turned into an executable test, it might lead a programmer to implement the feature in a way that's not what the authors of the requirement intended.

The problem lies with the word "all" in the **Then** step. Say there are only two ice options made in Alaska. If the app returned both of the two Alaskan-made ice options but also returned 98 ice options that are made elsewhere, this scenario would be satisfied even though only 2% of the results are relevant to Claire. She asked for *only* Alaskan options, after all. The person writing the requirements probably intended the **Then** statement to look like this instead (notice the single-word change):

```
# correct logic
Then the results show only Alaskan ice
```

With this rewritten statement, the product code is only correct if it returns Alaskan ice and nothing else. That's almost certainly what users expect to see when they try to filter the results in this way.

Logical mistakes like this are easy to make and commonly found in the wild. They make it impossible to know whether your software does what you want it to. So always double-check your usage of logical terms and make sure the scenario accurately captures your intention.

3.4 General scenario titles and concrete steps

Scenarios are most easily understood when their titles are general and their steps are concrete. This means that titles should steer clear of including specific details, whereas steps should include lots of those sorts of details.

For example, say our web app must validate the ZIP code of the customer's shipping address. Here's a good way to capture that requirement:

```
# good example of general title and detailed steps
Scenario: validate well-formed shipping ZIP
      Given Diana is ready to check out
      When she enters "97201" for her ZIP
      Then the system accepts her ZIP

# good example of general title and detailed steps
Scenario: validate malformed shipping ZIP
      Given Diana is ready to check out
      When she enters "ABCDE" for her ZIP
      Then she gets an error asking her to correct the ZIP
```

The scenario titles describe the feature without getting into specifics of how the feature might be tested. These titles are all you need if you want a quick glimpse of the system's intended behavior. It's easiest to understand the behavior if you're given a high-level overview of what it should do instead of concrete examples.

The scenario steps, on the other hand, serve as examples of how the feature actually operates. If you enter a specific ZIP code, specific behavior should result. Including concrete details in the steps provides extra color to scenarios, improving them in several ways: they become more vivid, more powerful, easier to read, and easier to remember. This is important when a stakeholder is trying to read and understand 200 scenarios without falling asleep. You won't *break* your scenarios if you omit specific details and instead use general descriptions in your steps, but your scenarios won't be as effective as they could be.

For example, this **When** step is not as good as the **When** step in the previous scenario:

```
# bad example: no concrete details
When she enters a malformed ZIP
```

Just referring to a "malformed ZIP" makes this a less powerful example of the ZIP-validation requirement than referring to a *specific* malformed ZIP.

Note that the concrete ZIP values included in the "good" scenarios above are relevant to the main point of those scenarios. Including concrete details about what kind of ice Diana is buying, explaining that she has logged in three times today, or including her login credentials in the steps would add harmful clutter to these scenarios since those details have nothing to do with the scenarios' main focus, or what aspect of the requirements they are trying to illustrate. So while details are important in to include in steps, only include *relevant* details.

3.5 Don’t overuse Cucumber

It’s tempting to write a Cucumber scenario for everything a user might do with your software: describe every error condition, every odd workflow, and every possible operation and permutation. But well-written scenarios are expensive to write, share, edit, and maintain, and creating a mountain of scenarios that cover every conceivable situation is almost always more expensive than it’s worth.

Just as the “runnable tests” feature of Cucumber isn’t meant to provide your only layer of testing (you still need unit and integration tests!), the requirements-capturing feature of Cucumber isn’t meant to be your only means of writing formal requirements. To avoid moving beyond the point of diminishing returns with Cucumber, it’s generally best to write a scenario for each happy path (*i.e.*, the most common sequences of clicks and data entry that result in the software being used the way it was intended) and only a handful of unhappy paths (user actions that produce warnings or errors). Include just the unhappy paths that are likely to happen, that are catastrophic when they occur, or that exercise code that developers are nervous about. While the other, less-critical unhappy paths should of course also be captured in requirements, it’s more sensible to put them in plain English paragraphs or whatever non-Cucumber format your team agrees on. In short: save Cucumber for the most important stuff.

Besides the expense involved with writing and maintaining a a large number of scenarios, there’s another reason to limit the scope of your Cucumber usage. Even well-written scenarios can be hard to read, understand, and compare. The whole point of Cucumber is to make software requirements as easy as possible for all stakeholders—technical and non-technical alike—to understand, discuss, and approve. It does that well, but it’s exhausting to read hundreds of scenarios and it’s hard to notice the details and logical subtleties that distinguish one scenario from other similar scenarios. After a while, they all blur together and the usefulness of Cucumber is lost. Keeping your scenarios short and the number of scenarios small is the best way to retain their impact and usefulness.

Imagine that we want to capture requirements around changing a user’s password. A happy path scenario might look like this:

```
Scenario: change password
      Given Elizabeth is logged in
      When she changes her password to “p@ssw0rd$”
      Then she’s redirected to the login page
      And she can log in with the new password
```

We might decide that the only unhappy path that’s likely to occur around password changes is when a user enters an invalid new password. There are many ways the new password could be invalid, but in the interest of keeping our collection of scenarios to a manageable size, it’s reasonable to capture just one type of invalid password in a Cucumber scenario:

```
Scenario: change password to invalid new password
      Given Elizabeth is logged in
      When she changes her password to “ABC”
      Then she sees a warning that it’s too short
      And she is prompted to enter a new password
```

Again, we still need to establish requirements around all the other ways a user could enter an invalid new password, but it makes more sense to write these specs in a non-Cucumber format (and to test them at the unit test level instead of with Cucumber scenarios). We’ve captured one example of invalid password behavior in Cucumber, and that’s enough to give stakeholders an idea of how that feature should generally work. Describing all the ways a password could be invalid–and testing that those invalid passwords are flagged properly by the system–are jobs best left to other tools.

Note that the first scenario above has a short, general title, whereas the second scenario has a longer title that includes more details. This is a recommended approach: happy path scenarios should have titles that assume that everything works as expected, while unhappy path scenarios should have titles that include details about why they trigger an error condition.

3.6 Focus on features, not controls

BDD works best when it describes how people use your software's features, not how they interact with its GUI controls. So Cucumber scenarios should rarely, if ever, refer to buttons, text fields, drop-down menus, or other GUI elements.

For example, here's a badly written scenario that captures the check-out process by referring to what the user clicks and types rather than what she is trying to do:

```
# bad example: refers to GUI elements
Scenario: check out
      Given Francis has finished adding items to her cart
      When she clicks “checkout”
      And she enters her address in the “shipping address” fields
      And she enters her billing info in the “billing info” fields
      And she clicks “pay now”
      And she clicks the “confirm payment” button
      Then she is shown the “order complete” page
      And she sees “thank you for shopping with Rose City Artisanal Ice”
      And she sees “your FedEx tracking number is <TRACKING NUMBER>”
      And the “continue shopping” button is highlighted
```

One problem with this is that referring to all the GUI elements makes the scenario way too long to read comfortably. Another problem is that it might capture Francis's specific actions, but it doesn't capture the *spirit* of her actions. If you asked her what she was doing, she wouldn't say "I'm clicking on this button and entering text in that field." She'd say "I'm checking out" or "I'm paying for the items in my shopping cart." And that *user intention* is what the scenario should capture.

This better-written scenario explains what she's trying to do, not how she's trying to do it:

```
Scenario: check out
      Given Francis has finished adding items to her cart
      When she checks out
      Then she an order confirmation message
      And she sees a shipment tracking number
```

This second scenario is shorter, easier to understand, and more relevant to stakeholders because it captures the behavior of the software's *feature* rather than its *controls*.

If you're using Cucumber scenarios as executable tests, don't worry: the scenarios will still test most of the GUI controls in the normal course of using the feature. Any controls that Cucumber misses can be covered with JavaScript unit tests.

3.7 Use personas

A persona is a fictitious person who serves as the user in one or more of your Cucumber scenarios. It's easy to think of a persona as nothing more than a gussied-up test user, and in a sense that's true. But a well-crafted persona is defined using a formal syntax (which I'll explain below, but maybe you can already guess what that syntax is?) and is shared with all the stakeholders so everyone has a common understanding of who the user is and how they're likely to use the software.

For example, we might need two personas to populate the Rose City Artisanal Ice scenarios. Let's call them *Gretchen* and *Hannah*. Gretchen is security-conscious: she pays only with Bitcoin, changes her password daily, and opts not to store shipping information in her user account. Hannah, on the other hand, is a typical user with the one quirk of frequently changing her email subscription settings.

Instead of referring generically to "the user" or "you" in our scenarios, we can refer specifically to Gretchen and Hannah:

```
Given Gretchen is logged in
```

or

```
When Hannah sorts ice by cost, ascending
```

It's important to keep personas streamlined, and not burdened with confusing, extraneous details. A great way to ensure that they are as minimal as possible is to define the personas only after you start using them in scenarios. Figure out what traits a persona needs in order to participate in a scenario, and then add those traits to the persona's definition as you write the scenario. By doing this, you make sure that every part of a persona's description is relevant to at least one Cucumber scenario.

Are you ready to learn about the mind-bending, *Inception*-like way that we define personas? It's best to define them in... more Cucumber scenarios. Granted, these are weird scenarios because they only have **Given** statements (no **When** or **Then** statements), but they're still scenarios.

```
Scenario: define persona of Gretchen
      Given Gretchen has a Rose City Artisanal Ice account
      And she has the password "Swizzle$"
      And she has stored Bitcoin info
      And she has not stored shipping info


Scenario: define persona of Hannah
      Given Hannah has a Rose City Artisanal Ice account
      And she subscribes to the email newsletter
```

Why would you go to the bother of defining personas in Cucumber scenarios instead of using plain English paragraphs or something less formally structured? Because if you use Cucumber's optional feature of treating the requirements as executable tests, these scenario-defined personas can themselves be run as tests, and will give you a warning (by failing) whenever the configuration information for the personas drifts away from what you and your tests expect.

For instance, if you execute the scenario that defines Gretchen's persona, it will make sure that she still has Bitcoin info in her account, that she doesn't have any shipping info stored, and that her password is set as expected. If someone accidentally changes any of these pieces of information in the database that stores her user account info, the scenario that defines Gretchen's persona will fail and you will know that scenarios further down the line that rely on Gretchen's Bitcoin info might also fail. So you should ignore the other failing tests until you fix Gretchen's user info in the database, at which point her persona definition scenario will pass again.

Alternatively, you could write your test code so that instead of validating Gretchen's user account info, it instead is responsible for *configuring* her account information to conform to the persona definition. This is an effective brute-force way to avoid configuration drift problems for test users.

There's a nice fringe benefit to using formal personas: they make it simple to follow the best practice of using the active voice in each step. If you think back to high school English, using the active voice means that you have a clear subject performing some action. For example, without personas you might write a scenario using the less effective passive voice, like this:

```
            # bad example: passive voice
            When the email newsletter is received
```

As your high school English teacher probably told you, switching to the active voice makes your writing more direct and powerful. Fortunately, using the active voice is almost effortless when you include a persona in your steps:

```
            When Hannah receives the email newsletter
```

3.8 Use present tense

Let's end with a best practice that's easy to understand and easy to implement: use present tense in all of your steps. If your brain works like mine does, it's almost impossible *not* to write **Given** steps in past tense, **When** steps in present tense, and **Then** steps in future tense:

```
      # bad example: inconsistent tenses
      Scenario: view cart
            Given Isabelle added ten items to her cart
            When she views her cart
            Then she will see those items alphabetical order, ascending
```

Fight the urge to do so! Or go back afterwards and change all tenses to present tense:

```
      Scenario: view cart
            Given Isabelle adds ten items to her cart
            When she views her cart
            Then she sees those items in alphabetical order, ascending
```

There are two reasons to keep everything in the present tense. First, using a consistent tense within and across scenarios makes those scenarios easier to read and understand, especially for a bleary-eyed reader who is looking at the 50th scenario in a row. Second, by writing everything in the present tense you simplify the process of re-using any code you've written to turn these steps into executable tests. For example, in the future you might write a scenario that includes this step:

```
            When Jessica adds ten items to her cart
```

Even though the persona has changed and this step is a **When** instead of a **Given**, the "adds ten items to her cart" text is the same, which means that Cucumber can execute the same underlying code for either of those steps.

3.9 The most important best practice: use your best judgment

There are very few black-and-white issues in life, and that includes Cucumber usage. The most important best practice is: *it's OK to break any rule that doesn't make sense in a particular context*. If one of the rules I've described makes your scenario clunky, complicated, or difficult to understand, ignore that rule! These recommendations are sensible starting places, but you shouldn't hold yourself strictly to them. Writing Cucumber scenarios is as much art as science, so trust your instincts about what makes a scenario clear, succinct, and unambiguous. Consistency is more important than any of the other rules we've discussed, so if your company or team prefers a way of writing scenarios that bends or breaks any of these rules, follow that direction.

4 Conclusion

Cucumber is a transformative technology for software development teams. It allows all stakeholders, regardless of their level of technical sophistication, to understand, share, and discuss software requirements. It provides a constantly updated view of how many of those requirements have been satisfied. To top it off, Cucumber makes behavior-driven development not only possible, but a pleasure.

It's not cheap to use. Writing, revising, organizing, and sharing scenarios takes a lot of time. It can be hard to keep a development team focused on gathering requirements instead of jumping ahead to writing product code. But the payoff is worth it both in terms of development speed and software quality.

It can be tough to convince management to try Cucumber, given the potential disruption to their teams' established workflow and the amount of up-front time that it demands before actual product code starts showing up. But if you think Cucumber might work for your team, these best practices might help you convince your management that Cucumber is worth a try. It's a mature technology with a supportive ecosystem, active practitioners, and a proven track record. I hope these ideas will help you use Cucumber so effectively and efficiently that you'll become as enthusiastic an evangelist for the technology as I am.