Getting Started with Cucumber

Paul Rayner

paul@virtual-genius.com

www.virtual-genius.com

@virtualgenius

VIRTUAL GENIUS

AGILITY. CRAFT. ARCHITECTURE.

Copyright ©2010 Virtual Genius, LLC.

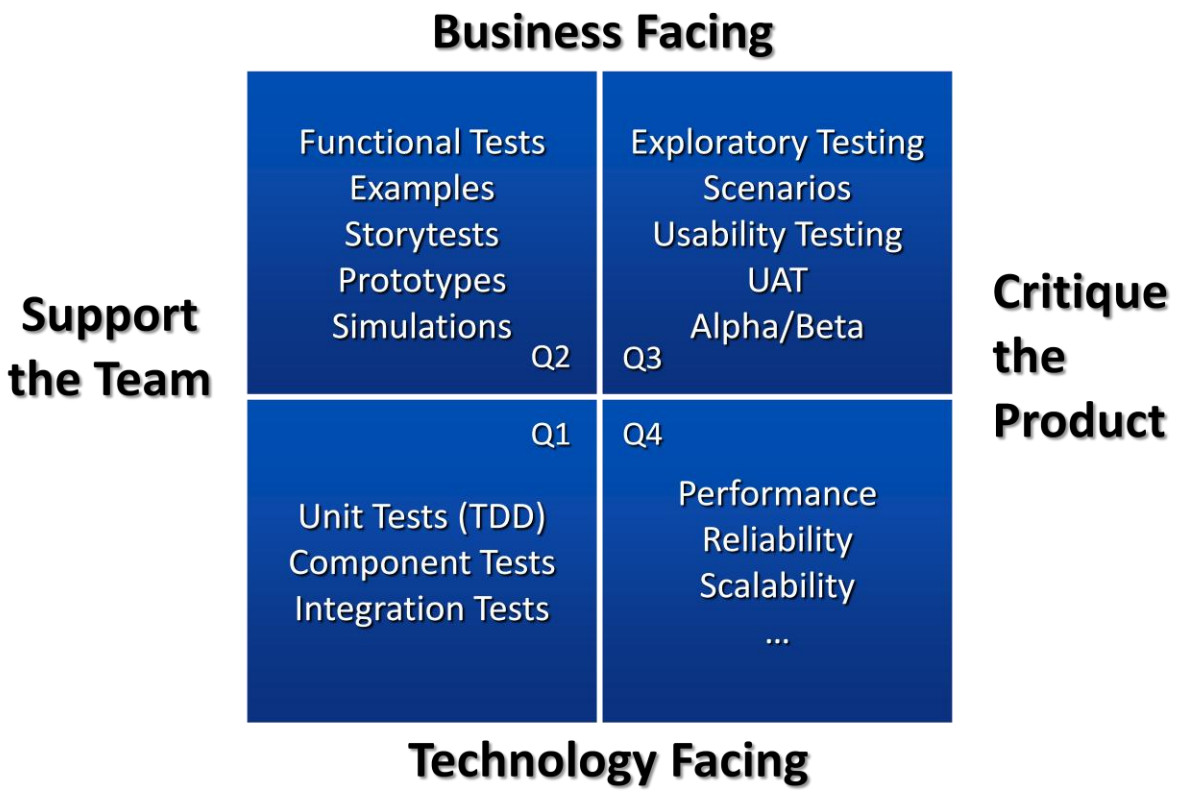
Build the Right Product

"The hardest part of building a software system is deciding precisely what to build" - Brooks[1]

- Unless everyone involved understands the business goals, there is a high risk of delivering the wrong product.
- Requirements do not explain *why* something is required, and assume a solution.
- Specifications do not contain enough information for effective development or testing. They are prone to ambiguity and they only use the knowledge of a selected few individuals.
- Gaps and inconsistencies in specifications get discovered only when developers start writing and testing code.
- Matching what customers expect is essentially a communication problem, not a technical one.

Agile Testing

The agile testing matrix[2] provides a helpful overview of the various agile testing quadrants and where common tools and testing approaches fit into the overall picture. Standard regression testing tools such as TestComplete are properly categorized as quadrant 3 tools.



[1] Fred Brooks, *The Mythical Man Month: Essays on Software Engineering,* 2nd ed. (New York: Addison Wesley Professional, 1995).
[2] Lisa Crispin and Janet Gregory, *Agile Testing: A Practical Guide for Testers and Agile Teams.* (Boston: Addison Wesley, 2009).

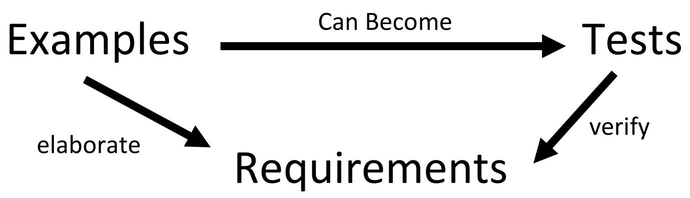
We need to *build quality in*, and not just inspect for it afterwards.

CUCUMBER

WHAT IS CUCUMBER?

Cucumber is a quadrant 2 collaboration and communication tool for executing plain-text functional descriptions as automated tests[3]. The purpose of Cucumber is to define and to verify the acceptance criteria for each feature a team develops. The team specifies as concretely as possible what the acceptance criteria for the feature are, and then codes enough of the system to make the test pass to satisfy the acceptance criteria for that specification. This approach is known as Acceptance Test-Driven Development (ATDD), Storytesting or Behavior-Driven Development (BDD). "An acceptance test investigates a system to determine whether it correctly implements a given responsibility. The essence of an acceptance test is the responsibility that it investigates, regardless of the technology used to implement the test."[4]

Cucumber employs the approach of specification by example. Instead of talking in abstract terms about what the system will do, the team collaborates to create specific examples that specify what the system should do from the user's perspective. This means that the "tests" (plain text feature descriptions with scenarios) are typically written before anything else and verified by business analysts, domain experts, etc. non technical stakeholders. The production code is then written outside-in, to make the stories pass. The relationship between examples, tests and requirements can be shown as follows:[5]



Cucumber divides this testing activity into two parts, the outward facing feature *steps* and the inward facing *step definitions*. Features are descriptions of desired outcomes (**Then**) following upon specific events (**When**) under predefined conditions (**Given**). They are typically used in conjunction with end-user input and, in some cases, may be entirely under end-user (in the form of a domain expert) control. Feature files are given the extension `.feature`.

[3] Much of this section is adapted from material in the Wiki documentation at http://cukes.info.
[4] Dale H. Emery, *Writing Maintainable Automated Acceptance Tests*. (http://dhemery.com), 2.
[5] Gojko Adzic, *Bridging the Communication Gap: Specification by Example and Agile Acceptance Testing* (London: Neuri Limited, 2009), 27.

GHERKIN

The language that Cucumber understands for these functional descriptions is called *Gherkin*. It is a Business Readable, Domain Specific Language (DSL) that lets you describe software's behavior without detailing how that behavior is implemented.

Gherkin serves two purposes – documentation and automated tests. The third is a bonus feature – when it yells in red it's talking to you, telling you what code you should write. Line endings terminate statements (eg, steps). Either spaces or tabs may be used for indentation (but spaces are more portable). Most lines start with a keyword.

Comment lines are allowed anywhere in the file. They begin with zero or more spaces, followed by a sharp sign (#) and some amount of text.

Parser divides the input into features, scenarios and steps. When you run the feature the trailing portion (after the keyword) of each step is matched to a code block called a step definition.

A Gherkin source file usually looks like this

```
 1: Feature: Some terse yet descriptive text of what is desired
 2:   In order to realize a named business value
 3:   As an explicit system actor
 4:   I want to gain some beneficial outcome which furthers the goal
 5:
 6:   Scenario: Some determinable business situation
 7:     Given some precondition
 8:       And some other precondition
 9:      When some action by the actor
10:       And some other action
11:       And yet another action
12:      Then some testable outcome is achieved
13:       And something else we can check happens too
14:
15:   Scenario: A different situation
16:       ...
```

First line starts the feature. Lines 2-4 are unparsed text, which is expected to describe the business value of this feature. Line 6 starts a scenario. Lines 7-13 are the steps for the scenario. Line 15 starts next scenario and so on.

GIVEN-WHEN-THEN

GIVEN

The purpose of *Given* steps is to ***put the system in a known state*** before the user (or external system) starts interacting with the system (which happens in the *When* steps). If you had worked with use cases, you would call this preconditions.

Examples:

- Create records (model instances) / set up the database state.

- It's ok to call into the layer "inside" the UI layer here (in Rails: talk to the models).
- Log in a user[6].

When

The purpose of *When* steps is to ***describe the key action*** the user performs.

Examples:

- Interact with a web page (Webrat/Watir/Selenium ***interaction*** etc should mostly go into *When* steps).
- Interact with some other user interface element.
- Developing a library? Kicking off some kind of action that has an observable effect somewhere else.

Then

The purpose of *Then* steps is to ***observe outcomes***. The observations should be related to the business value/benefit in your feature description. The observations should also be on some kind of ***output*** – that is something that comes ***out*** of the system (report, user interface, message) and not something that is deeply buried inside it (that has no business value).[7]

Examples:

- Verify that something related to the *Given*+*When* is (or is not) in the output
- Check that some external system has received the expected message (was an email with specific content sent?)

And, But

If you have several *Given's*, *When's* or *Then's* you can write

```
Scenario: Multiple Givens
  Given one thing
  Given another thing
  Given yet another thing
  When I do an action
  Then I see something
  Then I see a second thing
  Then I don't see something else
```

Or you can make it read more fluently by writing

```
Scenario: Multiple Givens
  Given one thing
    And another thing
    And yet another thing
  When I do an action
  Then I see something
```

[6] An exception to the no-interaction recommendation. Things that "happened earlier" are ok.

[7] While it might be tempting to implement *Then* steps to just look in the database – resist the temptation. You should only verify outcome that is observable for the user (or external system) and databases usually are not.

```
    And I see a second thing
    But I don't see something else
```

Note that, as far as Cucumber is concerned, steps beginning with *And* or *But* are exactly the same kind of steps as all the others.

STEP DEFINITIONS

Step Definitions start with an adjective or an adverb[8], and can be expressed in any of Cucumber's supported spoken languages such as Ruby, C#, Java and Groovy.[9] All Step definitions are loaded (and defined) before Cucumber starts to execute the plain text. When Cucumber executes the plain text, it will look for a registered Step Definition (via a matching regular expression) for each step. If it finds a Step Definition it will execute its code, passing all groups from the Regexp matching as arguments to the code.

For example, the following scenario:

```
Scenario: Adding cucumbers
  Given I have 3 cucumbers
  When I add 4 cucumbers
  Then I should have 7 cucumbers
```

Can be related to the following Java Step Definition methods:

```
 @Given("I have (\\d+) cucumbers")
public void iHaveNCucumbers(int numberOfCucumbers) {
    /// Initialize cucumber count
     cukeCount = numberOfCucumbers;
}

@When("I add (\\d+) cucumbers")
public void iAddNCucumbers(int numberOfCucumbers) {
    /// Update cucumber count
     cukeCount += numberOfCucumbers;
}

@Then("I should have (\\d+) cucumbers")
public void iShouldHaveNCucumbers(int expectedNumberOfCucumbers) {
    /// Verify cucumber count
     assertEquals(numberOfCucumbers, number);
}
```

SUCCESSFUL STEPS

When Cucumber finds a matching Step Definition it will execute it. If the block in the step definition doesn't raise an Exception, the step is marked as successful (green). What you return from a Step Definition has no significance whatsoever.

UNDEFINED STEPS

[8] Note that the adjective or adverb have **no** significance when Cucumber is registering or looking for Step Definitions.
[9] See www.cukes.info for a comprehensive, up-to-date list of supported languages.

When Cucumber can't find a matching Step Definition the step gets marked as yellow, and all subsequent steps in the scenario are skipped. If you use --strict this will cause Cucumber to exit with 1.

Pending steps

When a Step Definition's Proc invokes the #pending method, the step is marked as yellow (as with undefined ones), reminding you that you have work to do. If you use --strict this will cause Cucumber to exit with 1.

Failed steps

When a Step Definition's Proc is executed and raises an error, the step is marked as red. What you return from a Step Definition has no significance what so ever. Returning nil or false will **not** cause a step definition to fail.

Skipped steps

Steps that follow undefined, pending or failed steps are never executed (even if there is a matching Step Definition), and are marked cyan.

Multi-Line Step Argument

The regular expression matching in Step Definitions lets you capture small strings from your steps and receive them in your step definitions. Multi-line step arguments enable you to pass a richer data structure from a step to a step definition. They are written on the lines right underneath a step, and will be yielded as the last argument to the matching step definition's block.

Tables

Tables as arguments to steps are handy for specifying a larger data set – usually as input to a Given or as expected output from a Then. (This is not to be confused with tables in Scenario outlines, which are syntactically identical but serve a different purpose).

Multiline Strings

Multiline Strings are handy for specifying a larger piece of text. The text should be offset by delimiters consisting of three double-quote marks on lines of their own[10]:

```
Given a blog post named "Random" with Markdown body
  """
  Some Title, Eh?
  ==============
  Here is the first paragraph of my blog post. Lorem ipsum dolor sit amet,
  consectetur adipiscing elit.
  """
```

Scenario Outlines

[10] This is done using the PyString syntax. (The inspiration for PyString comes from Python where """ is used to delineate docstrings.)

Copying and pasting scenarios to use different values quickly becomes tedious and repetitive:

```
Scenario: eat 5 out of 12
  Given there are 12 cucumbers
  When I eat 5 cucumbers
  Then I should have 7 cucumbers

Scenario: eat 5 out of 20
  Given there are 20 cucumbers
  When I eat 5 cucumbers
  Then I should have 15 cucumbers
```

Scenario outlines allow us to more concisely express these examples through the use of a template with placeholders:

```
Scenario Outline: Search for matching sessions by time and day
    When I search for sessions at <Time> on <Day>
    Then I should see <Session> in the search results

        Examples:
            | Day   | Time      | Session                          |
            | Tue   | 8:30pm    | Visualizations for Code Metrics  |
            | Wed   | 11:00am   | Agile Engineering Practices      |
            | Wed   | 1:30pm    | Emergent Design                  |
            | Thu   | 1:30pm    | Workshop: ATDD/BDD with Cucumber |
```

The Scenario outline steps provide a template which is never directly run. A Scenario Outline is run once for each row in the `Examples` section beneath it (not counting the first row). The way this works is via placeholders. Placeholders must be contained within < > in the Scenario Outline's steps.

The placeholders indicate that when the Examples row is run they should be substituted with real values from the Examples table. If a placeholder name is the same as a column title in the Examples table then this is the value that will replace it.

So when running the first row of our example:

```
        Examples:
            | Day   | Time      | Session                          |
            | Tue   | 8:30pm    | Visualizations for Code Metrics  |
```

The scenario that is actually run is:

```
Scenario Outline: Search for matching sessions by time and day
    When I search for sessions at 8:30pm on Tue
    Then I should see Visualizations for Code Metrics in the search results
```

TAGS

Tags are a great way to organize features and scenarios. Consider this example:

```
@billing
Feature: Verify billing

  @important
  Scenario: Missing product description
  Scenario: Several products
```

A Scenario or feature can have as many tags as you like. Just separate them with spaces:

```
@billing @bicker @annoy
Feature: Verify billing
```

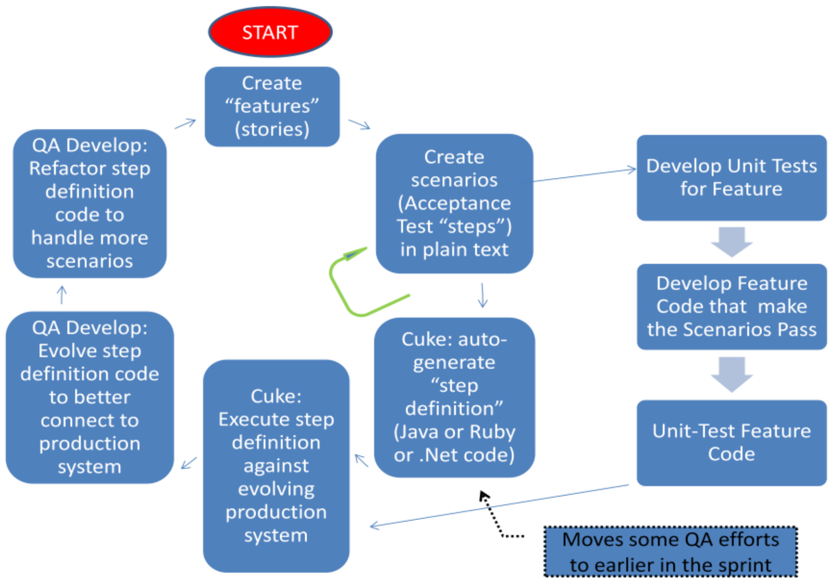
BACKGROUND

Background allows you to add some context to the scenarios in a single feature. A Background is much like a scenario containing a number of steps. The difference is when it is run. The background is run before each of your scenarios but after any of your Before Hooks. See later section for tips on when to use backgrounds.

THE ATDD PROCESS WITH CUCUMBER

The process can be summarized in the following steps (see diagram[11]):

1. Discuss the story
2. Work with domain experts, customers and delivery team (coders and testers) to generate examples
3. Formulate examples with Given-When-Then to create acceptance criteria
4. Automate acceptance criteria
5. Build *just enough product* to satisfy the acceptance criteria
6. Get feedback and repeat


START
Create "features" (stories)
Create scenarios (Acceptance Test "steps") in plain text
Develop Unit Tests for Feature
Develop Feature Code that make the Scenarios Pass
Unit-Test Feature Code
Cuke: auto-generate "step definition" (Java or Ruby or .Net code)
Cuke: Execute step definition against evolving production system
QA Develop: Evolve step definition code to better connect to production system
QA Develop: Refactor step definition code to handle more scenarios
Moves some QA efforts to earlier in the sprint

[11] Thanks to Al Segal for creating (and kindly sharing) this diagram.

BUILD JUST-ENOUGH PRODUCT

It is important to write each test/feature step ***one at a time***. After each new feature step is added then you should immediately create the corresponding step definition method. After creating your new test/step definition you must prove to yourself that it fails by running it against the, as yet, non-existent application code. Then, and only then, should you write the least application code that gets your test/step definition to pass. Once this cycle is complete then move on to the next feature step.

It is often tempting to just ***skip ahead*** with the analysis and complete as many scenarios and feature steps as one can imagine. In some cases limited access to domain experts and end users may require that many feature steps be completed long before coding the associated step definitions is undertaken. Nonetheless, in the long run, it has proven best practice to write feature steps and step definitions incrementally, using the absolute minimum of code; and then immediately implement the new feature step requirement in application code, also using the minimum code to satisfy it.

This is a hard discipline to accept but the value with this approach is that you will rarely (never) have untested code anywhere in your application and your application will only have code that satisfies required features. Consequently, you can face design changes with complete equanimity, secure in the knowledge that if unanticipated changes break anything else then you will know of the side-effect immediately upon running your test suite.

Cucumber functions as an *integration* test harness. It is designed to exercise the entire application stack from view down to the database. It is certainly possible to write features and construct step definitions in such a fashion as to conduct unit tests and no great harm will result. However, this is not considered best practice by many and may prove insupportable for very large projects due to the system overhead it causes and the commensurate delay in obtaining test results.

UBIQUITOUS LANGUAGE

A ubiquitous language[12] is a language structured around the domain model and used by all team members to connect all the activities of the team with the software. It applies within a particular business context, and should be based on a creative collaboration with the domain experts in the business. The development team works with the domain experts to clarify and to refine the domain language into something that can be shared between everyone involved in developing the solution.

The team should commit the team to exercising that language relentlessly in all communication within the team and in the code. Use the same language in diagrams, writing and especially speech. Domain experts should object to

[12] http://domaindrivendesign.org/node/132.

terms or structures that are awkward or inadequate to convey domain understanding; developers should watch for ambiguity or inconsistency that will trip up design.

The language used in the Cucumber feature files is yet another place where the use of the ubiquitous language is critical. Cucumber can be a great tool for clarifying and refining the language.

KEEPING AUTOMATED TESTS MAINTAINABLE

"***Test automation is software development.*** This principle implies that much of what we know about writing software also applies to test automation…much of the cost of software development is maintenance – changing the software after it is written. This single fact accounts for much of the difference between successful and unsuccessful test automation efforts…successful organizations understand that test automation is software development, that it involves significant maintenance costs, and that they can and must make deliberate, vigilant effort to keep maintenance costs low."[13]

AVOID INCIDENTAL DETAILS AND UNNECESSARY DUPLICATION

"*The need to change comes from two directions: changes in requirements and changes in the system's implementation…the only way to keep the maintenance costs of tests low is to make the tests adaptable to those kinds of changes.* Developers have learned – often through painful experience – that ***two key factors make code difficult to change: Incidental details and duplication.***"[14]

- "When we cannot see the essence of a test, it's more difficult and costly to understand how to change the test when the system's responsibilities change. Incidental details increase maintenance costs…hide the incidental details…to more easily see the essence of the test."[15] "Express system responsibilities clearly and directly."[16]
- "One useful approach is to ask yourself: How would I write that first step if I knew nothing about the system's implementation?"[17]
- "Duplication often signals that some important concept lurks unexpressed in the tests. That's especially true when we duplicate not just single steps, but *sequences* of steps...make the concepts explicit."[18]
- "*Any* time spent puzzling out the meaning and significance of a test is maintenance cost…these 'trivial' maintenance costs add up, and they kill test automation efforts."[19]

[13]Emery, 1. Emphasis mine.
[14] Ibid.
[15] Ibid, 3-4.
[16] Ibid, 10.
[17] Ibid, 4.
[18] Ibid, 6.
[19] Ibid, 8.

Expert Tips for Maintainability

1. Feature Files Should Actually be *Features*, Not Entire Portions of an App[20]

One feature per well named file, please, and keep the features focused.

2. Avoid Inconsistencies with Domain Language

You'll get the most benefit out of using Cucumber when your customers are involved. To that end, make sure you use their domain language when you write stories. The best course of action is to have them involved in writing the stories.

3. Organize Your Features and Scenarios with the Same Thought You Give to Organizing Your Code

One useful way to organize things is by how fast they run. Use 2-3 levels of granularity for this:

- **Fast**: scenarios that run very fast, e.g. under 1/10 of a second
- **Slow**: scenarios that are slower but not painfully so, maybe under one second each
- **Glacial**: scenarios that take a really long time to run

You can do this separation several different ways (and even some combination):

- Put them in separate features
- Put them in separate subdirectories
- Tag them

4. Use Tags

Tags are a great way to organize your features and scenarios in non functional ways. You could use @small, @medium and @large, or maybe @hare, @tortoise, and @sloth. Using tags let you keep a feature's scenarios together structurally, but run them separately. It also makes it easy to move features/scenarios between groups, and to have a given feature's scenarios split between groups.

The advantage of separating them this way is that you can selectively run scenarios at different times and/or frequencies, i.e. run faster scenarios more often, or run really big/slow scenarios overnight on a schedule.

Tagging has uses beyond separating scenarios into groups based on how fast they are:

- When they should be run: on @checkin, @hourly, @daily
- What external dependencies they have: @local, @database, @network
- Level: @functional, @system, @smoke
- Etc.

[20] Most of these expert tips are taken from Dave Astels blog entry at http://www.engineyard.com/blog/2009/15-expert-tips-for-using-cucumber

5. Start with When and Then, Add the Given After

It is usually easier to start by thinking about the action that is occurring, rather than all the important preconditions that need to be in place first, so start with writing the When step first for each scenario. The Given steps can always come later, and are not required. Note that a When can always be changed to a Given, and that a scenario with only a Then is still valid.

6. Don't Get Carried Away with Backgrounds (Stick to Givens)

The larger the background, the greater the load of understanding for each scenario. Scenarios that contain all the details are self-contained and as such, can be more understandable at a glance. If you do use a background, here are some things to keep in mind:

- Don't use "Background" to set up complicated state unless that state is actually something the client needs to know.
    - For example, if the user and site names don't matter to the client, you should use a high-level step such as "Given that I am logged in as a site owner".
- Keep your "Background" section short.
    - You're expecting the user to actually remember this stuff when reading your scenarios.
    - If the background is more than 4 lines long, can you move some of the irrelevant details into high-level steps?
- Make your "Background" section vivid.
    - You should use colorful names and try to tell a story, because the human brain can keep track of stories much better than it can keep track of names like "User A", "User B", "Site 1", and so on.
- Keep your scenarios short, and don't have too many.
    - If the background section has scrolled off the screen, you should think about using higher-level steps, or splitting the *.features file in two.

7. Make Scenarios Independent and Deterministic

There shouldn't be any sort of coupling between scenarios. The main source of such coupling is state that persists between scenarios. This can be accidental, or worse, by design. For example one scenario could step through adding a record to a database, and subsequent scenarios depend on the existence of that record. This may work, but will create a problem if the order in which scenarios run changes, or they are run in parallel. Scenarios need to be completely independent.

Each time a scenario runs, it should run the same, giving identical results. The purpose of a scenario is to describe how your system works. If you don't have confidence that this is always the case, then it isn't doing its job. If you have non-deterministic scenarios, find out why and fix them.

8. Write Scenarios for the Non-Happy-Path Cases As Well

Happy path tests are easy; edge cases and failure scenarios take more thought and work. Here's where having testers on the team with good exploratory testing skills can reap rewards.

9. Be DRY: Refactor and Reuse Step Definitions

Especially look for the opportunity to make reusable step definitions that are not feature specific. As a project proceeds, you should be accumulating a library of step definitions. Ideally, you will end up with step definitions that can be used across projects.

10. Use a Library (Such as Chronic in Ruby) for Parsing Time in Your Step Definitions

This allows you to use time in scenarios in a natural way. This is especially useful for relative times.

```
Background:
  Given a user signs up for a 30 day account

Scenario: access before expiry
  When they login in 29 days
  Then they will be let in

Scenario: access after expiry
  When they login in 31 days
  Then they will be asked to renew
```

11. Revisit, Refactor, and Improve Your Scenarios and Steps

Look for opportunities to generalize your steps and reuse them. You want to accumulate a reusable library of steps so that writing additional features takes less and less effort over time.

12. Refactor Language and Steps to Reflect Better Understanding of Domain

This is an extension of the previous point; as your understanding of the domain and your customer's language/terminology improves, update the language used in your scenarios.

13. Avoid Using Conjunctive Steps

Each step should do ***one*** thing. You should not generally have step patterns containing "and." For example:

```
Given A and B
```

should be split into two steps:

```
Given A
And B
```

14. Step Organization

Technically it doesn't matter how you name your step definition files and what step definitions you put in what file. As a general rule, create a step definition file for each domain concept (eg. one file for each major model/database table).

Tips for Adopting ATDD/BDD with Cucumber in Your Team

1. REMEMBER: "It's not in the tool!" Focus on communication and collaboration gains.
2. Demonstrate *value early and often* rather than imposing a new tool and process. Show others why it is so great.
3. Get an influential, open-to-new-approaches QA/tester on board as soon as possible. Seek their advice and support by walking them through creating test scenarios and asking how they would do it. Show them that ATDD and Cucumber will actually make their job easier and more enjoyable, while increasing quality and collaboration with the rest of the team. Express how much you appreciate what they do (be aware that some testers – like coders – can be very territorial about their area of expertise, and may feel threatened – at first – by coders adopting new and unknown "testing tools").
4. Start using specification by example. Discuss requirements in terms of concrete examples from the business domain. Your customers will love you for it, and the developers will improve their understanding of what the system really needs to do.
5. Choose a feature that would really benefit from driving out some reference scenarios and gaining clarity about the requirements and domain.
6. Socialize the process and tool, and don't let it get in the way of the rest of the team while you experiment (perhaps learn it as a side-project in your own time to start with, so the learning experience doesn't negatively impact your team's current deliverables).
7. Do *not* criticize existing test automation tools (eg. TestComplete etc) used in your company, especially if the organization has spent a lot of money on them. Emphasize that these tools are address a different area by using the testing matrix to show that they solve a different problem (quadrant 3) and address a different need to Cucumber (quadrant 2!). Focus on the value of Cucumber, and apply it from the start where it can deliver the most "bang for the buck."

Resources

Recommended Books

Gojko Adzic, *Bridging the Communication Gap: Specification by Example and Agile Acceptance Testing* (London: Neuri Limited, 2009).

Lisa Crispin and Janet Gregory, *Agile Testing: A Practical Guide for Testers and Agile Teams*. (Boston: Addison Wesley, 2009).

Online

- Main Cucumber page: http://cukes.info/

- Cucumber mailing list: http://groups.google.com/group/cukes?hl=en
- Cuke4Duke API: http://cukes.info/cuke4duke/apidocs/overview-summary.html
- Regex cheat sheet (there are many others): http://www.addedbytes.com/cheat-sheets/regular-expressions-cheat-sheet/
- Java regular expression syntax: http://www.programmersheaven.com/2/RegexQuickRef
- User story quick reference guide (latest copy will always link from here): http://www.virtual-genius.com/resources.html