Chapter 2

First Taste

We figured you'd be eager to start playing with your shiny new toy right away, so we're going to work through a simple example that will give you a feel for what it's like to work with Cucumber. You might not quite understand everything that happens here just yet, but try not to let that worry you. We'll come back and fill in the details over the next few chapters.

We're going to build a simple command-line application from the outside-in, driving our development with Cucumber. Watch how we proceed in very small baby steps, going back to run Cucumber after we make each change. This patient rhythm is an important characteristic of working effectively with Cucumber, but it's much easier to demonstrate than to explain.

Assuming you want to follow along with the action in this chapter (it'll be a lot more fun if you do), you'll need to have Cucumber installed. If you haven't already

done so, see Appendix 2, *Installing Cucumber* for installation instructions.

Right...shall we get started then?

2.1 Understanding Our Goal

Our goal is to write a program that can perform calculations. Some people might call it a calculator.

We have an incredible vision of what this calculator will one day become: a cloud-based service that runs on mobile phones, desktops, and browsers, uniting the world with the opportunity of ubiquitous mathematical operations. We're pragmatic businesspeople, though, so the first release of our program has to be as simple as possible. The first release will be a command-line program, implemented as a Ruby script. It will take input containing the calculation to be done and display the result at the command prompt.

So, for example, if the input file looks like this:

```
2+2
```

then the output would be `4`.

Similarly, if the input file looks like this:

```
100/2
```

then the output would be `50`.

You get the idea.

2.2 Creating a Feature

Cucumber tests are grouped into *features*. We use this name because we want them to describe the features that a user will be able to enjoy when using our program. The first thing we need to do is make a directory where we'll store our new program along with the features we'll be writing for it.

```
$ mkdir calculator
$ cd calculator
```

We're going to let Cucumber guide us through the development of our calculator program, so let's start right away by running cucumber in this empty folder:

```
$ cucumber

You don't have a 'features' directory.  Please create
one to get started.
See http://cukes.info/ for more information.
```

Since we didn't specify any command-line arguments, Cucumber has assumed that we're using the conventional features folder to store our tests. That would be fine, except that we don't have one yet. Let's follow the convention and create a features directory:

```
$ mkdir features
```

Now run Cucumber again.


```
$ cucumber

0 scenarios
0 steps
0m0.000s
```

Each Cucumber test is called a *scenario*, and each scenario contains *steps* that tell Cucumber what to do. This output means that Cucumber is happily scanning the features directory now, but it didn't find any scenarios to run. Let's create one.

Our user research has shown us 67 percent of all mathematical operations are additions, so that's the operation we want to support first. In your favorite editor, create a plain-text file called features/adding.feature:

first_taste/01/features/adding.feature

```
Feature: Adding

  Scenario: Add two numbers
    Given the input "2+2"

    When the calculator is run
    Then the output should be "4"
```

This feature file contains the first scenario for our calculator program. We've translated one of the examples we were given in the previous section into a Cucumber scenario that we can ask the computer to run

over and over again. You can probably see that Cucumber expects a little bit of structure in this file. The keywords Feature, Scenario, Given, When, and Then are the structure, and everything else is documentation. Although some of the keywords are highlighted here in the book—and they may be in your editor too—it's just a plain-text file. The structure is called *Gherkin*.

When you save this file and run cucumber, you should see a great deal more output than the last time:

```
$ cucumber

Feature: Adding

  Scenario: Add two numbers
    Given the input "2+2"
    When the calculator is run
    Then the output should be "4"

1 scenario (1 undefined)
3 steps (3 undefined)
0m0.003s

You can implement step definitions for undefined steps
with these snippets:

Given /^the input "([^"]*)"$/ do |arg1|
  pending # express the regexp above with the code you
wish you had
end

When /^the calculator is run$/ do
  pending # express the regexp above with the code you
wish you had
```



```
end

Then /^the output should be "([^"]*)"$/ do |arg1|
  pending # express the regexp above with the code you
wish you had
end

If you want snippets in a different programming
language,
just make sure a file with the appropriate file
extension
exists where cucumber looks for step definitions.
```

Wow, that's a lot of output all of a sudden! Let's take a look at what's going on here. First, we can see that Cucumber has found our feature and is trying to run it. We can tell this because Cucumber has repeated the content of the feature back to us. You might also have noticed that the summary output 0 scenarios has changed to 1 scenario (undefined). This means that Cucumber has read the scenario in our feature but doesn't know how to run it yet.

Second, Cucumber has printed out three code snippets. These are sample code for *step definitions*, written in Ruby, which tell Cucumber how to translate the plain English steps in the feature into actions against our application. Our next step will be to put these snippets into a Ruby file where we can start to flesh them out.

Before we explore beneath the layer of business-facing

Gherkin features into step definitions, it's worth taking a quick look at the map in case anyone is feeling lost. Figure 2, *The main layers of a Cucumber test suite*, reminds us how things fit together. We start with features, which contain our scenarios and steps. The steps of our scenarios call step definitions that provide the link between the Gherkin features and the application being built.

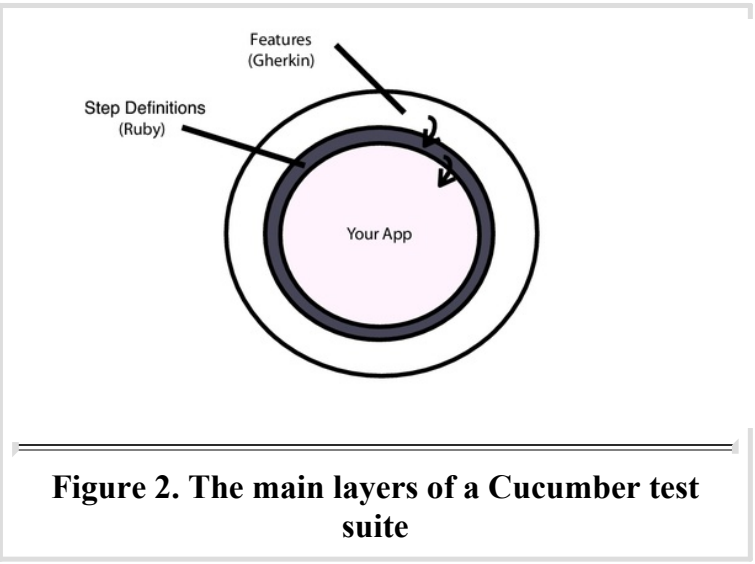

Figure 2. The main layers of a Cucumber test suite

Now we'll implement some step definitions so that our scenario is no longer undefined.

2.3 Creating Step Definitions

Without thinking too much about what they mean, let's just copy and paste the snippets from Cucumber's last output into a Ruby file. Just like the features, Cucumber is expecting the step definitions to be found in a conventional place:

```
$ mkdir features/step_definitions
```

Now create a Ruby file in features/step_definitions, called calculator_steps.rb. Cucumber won't mind what you call it as long as it's a Ruby file, but this is a good name to use. Open it in your text editor and paste in those snippets:

first_taste/02/features/step_definitions/calculator_steps.rb

```
Given /^the input "([^"]*)"$/ do |arg1|
  pending # express the regexp above with the code you
wish you had
end

When /^the calculator is run$/ do
  pending # express the regexp above with the code you
wish you had
end

Then /^the output should be "([^"]*)"$/ do |arg1|
  pending # express the regexp above with the code you
wish you had
end
```

Running cucumber will tell us what to do next:


```
Feature: Adding

  Scenario: Add two numbers
    Given the input "2+2"
      TODO (Cucumber::Pending)
      ./features/step_definitions/calculator_steps.rb:2
      features/adding.feature:4
    When the calculator is run
    Then the output should be "4"

1 scenario (1 pending)
3 steps (2 skipped, 1 pending)
0m0.002s
```

The scenario has graduated from undefined to pending. This is good news, because it means Cucumber is now running the first step, but as it did so, it hit the call to pending inside our copied-and-pasted step definition code, which tells Cucumber that this scenario is still a work in progress. We need to replace that pending marker with a real implementation.

Notice that Cucumber reports the two other steps as skipped. As soon as it encounters a failed or pending step, Cucumber will stop running the scenario and skip the remaining steps.

Let's implement the first step definition.

2.4 Implementing Our First Step Definition

We've decided this first release of our calculator is going to take its input from the user as a command-line argument, so our job in the step definition for `Given the input "2+2"` is just to remember the input so that we know what to pass on the command line when we run the calculator in the next step. In the `features/step_definitions` folder, edit the `calculator_steps.rb` file so that the first step definition looks like this:

first_taste/03/features/step_definitions/calculator_steps.rb

```
Given /^the input "([^"]*)"$/ do |input|
  @input = input
end
```

All we've done here is store the input from the feature in a Ruby instance variable. That instance variable will be around for as long as this particular scenario is running, so we can use it again in the next step when we actually run the calculator.

Great, that was easy. Now, where were we again? Let's ask `cucumber`:

```
Feature: Adding

  Scenario: Add two numbers
```

```
    Given the input "2+2"
    When the calculator is run
      TODO (Cucumber::Pending)
      ./features/step_definitions/calculator_steps.rb:9
      features/adding.feature:5
    Then the output should be "4"

1 scenario (1 pending)
3 steps (1 skipped, 1 pending, 1 passed)
0m0.002s
```

Yay! Our first step passed! The scenario is still marked as pending, of course, because we still have the other two steps to implement, but we're starting to get somewhere.

2.5 Running Our Program

To implement the next step, edit features/step_definitions/calculator_steps.rb so that the second step definition looks like this:

first_taste/04/features/step_definitions/calculator_steps.rb

```
When /^the calculator is run$/ do
  @output = `ruby calc.rb #{@input}`
  raise('Command failed!') unless $?.success?
end
```

This code attempts to run our calculator program calc.rb, passing it the input we stored in the first step and storing any output in another instance variable. Then it checks a specially—but rather cryptically—named Ruby variable $? to check whether the command succeeded and raises an error if it didn't. Remember that Cucumber fails a step if the step definition raises an error; this is the simplest way to do it.

This time when we run Cucumber, we should see that it has actually tried to run our calculator:

```
Feature: Adding

  Scenario: Add two numbers
    Given the input "2+2"
ruby: No such file or directory -- calc.rb (LoadError)
    When the calculator is run
      Command failed! (RuntimeError)
```

```
./features/step_definitions/calculator_steps.rb:10
      features/adding.feature:5
    Then the output should be "4"

Failing Scenarios:
cucumber features/adding.feature:3

1 scenario (1 failed)
3 steps (1 failed, 1 skipped, 1 passed)
0m0.027s
```

Our step is failing, because we don't have a calc.rb program to run yet. You should see that Cucumber has highlighted the output from our raised error in red just beneath the step, helping you spot the problem.

You might well think it's a bit odd that we've written and run code that tries to run our calc.rb program, knowing perfectly well that the file doesn't even exist yet. We do this deliberately, because we want to make sure we have a fully functioning test in place before we drop down to working on the solution. Having the discipline to do this means we can trust our tests, because we've seen them fail, and this gives us confidence that when the tests pass, we're really done. This gentle rhythm is a big part of what we call *outside-in development*, and while it might seem strange at first, we hope to show you throughout the book that it has some great benefits.

Another benefit of working from the outside-in is that we've had a chance to think about the command-line interface to our calculator from the point of view of a user, without having made any effort to implement it yet. At this stage, if we realize there's something we don't like about the interface, it's very easy for us to change it.

2.6 Changing Formatters

We think it's starting to get distracting looking at the whole content of the feature in Cucumber's output each time we run. Let's switch to the progress *formatter* to get a more focused output. Run this command:

```
$ cucumber --format progress
```

You should see the following output:

```
.ruby: No such file or directory -- calc.rb (LoadError)
F-

(::) failed steps (::)

Command failed! (RuntimeError)
./features/step_definitions/calculator_steps.rb:10
features/adding.feature:5

Failing Scenarios:
cucumber features/adding.feature:3 # Scenario: Add two
numbers

1 scenario (1 failed)
3 steps (1 failed, 1 skipped, 1 passed)
0m0.023s
```

Instead of printing the whole feature, the progress formatter has printed three characters in the output, one for each step. The first `.` character means the step passed. The `F` character means the second step, as we know, has failed. The final `-` character means that the

last step has been skipped. Cucumber has several different formatters that you can use to produce different types of output as your features run; you'll learn more about them through the course of the book.

Formatters

Cucumber formatters allow you to visualize the output from your test run in different ways. There are formatters that produce HTML reports, formatters that produce JUnit XML for continuous integration servers like Jenkins, and many more.

Use `cucumber --help` to see the different formatters that are available and try some out for yourself. We'll explain more about formatters in Chapter 11, *The Cucumber Command-Line Interface*.

That was an interesting little diversion, but let's get back to work. We have a failing test to fix!

2.7 Adding an Assertion

So, following Cucumber's lead, we need to create a Ruby file for our program. Let's just create an empty Ruby file for the time being so that we can stay on the outside and get the test finished before we go on to the solution. Linux/Mac users can type this to create an empty file:

```
$ touch calc.rb
```

If you're using Windows, there is no touch command, so either just create an empty text file named calc.rb in your editor or use this little trick:

```
C:\> echo.>calc.rb
```

When we run cucumber again, we should see that the second step is passing and we're on to the final step:

```
$ cucumber --format progress

..P

(::) pending steps (::)

features/adding.feature:6:in 'Then the output should be
"4"'

1 scenario (1 pending)
3 steps (1 pending, 2 passed)
0m0.460s
```


To get the last step definition working, change the last step definition in features/step_definitions/calculator_steps.rb to look like this:

first_taste/07/features/step_definitions/calculator_steps.rb

```
Then /^the output should be "([^"]*)"$/ do
|expected_output|
  @output.should == expected_output
end
```

We're using an RSpec[6] assertion to check that the expected output specified in the feature matches the output from our program that we stored in @output in the previous step definition. If it doesn't, RSpec will raise an error just like we did using raise in the last step definition.

Joe asks:

I Feel Weird: You're Making Tests Pass but Nothing Works!

We've implemented a step that calls the calculator program and passes, even though the "program" is just an empty file. What's going on here?

Remember that a step isn't a test in itself. The test is the whole scenario, and that isn't going to pass until all of its steps do. By the time we've implemented all of the step definitions, there's only going to be one way to make the whole scenario pass, and that's to build a calculator that can perform additions!

When we work outside-in like this, we often use temporary *stubs* like the empty calculator program as placeholders for details we need to fill in later. We know that we can't get away with leaving that as an empty file forever, because eventually Cucumber is going to tell us to come back and make it do something useful in order to get the whole scenario to pass.

This principle, *deliberately doing the minimum useful work the tests will let us get away*

with, might seem lazy, but in fact it's a discipline. It ensures that we make our tests thorough: if the test doesn't drive us to write the right thing, then we need a better test.

Now when we run cucumber, we have ourselves a genuine failing test:

```
$ cucumber --format progress

..F

(::) failed steps (::)

expected: "4"
     got: "" (using ==)
(RSpec::Expectations::ExpectationNotMetError)
./features/step_definitions/calculator_steps.rb:16
features/adding.feature:6

Failing Scenarios:
cucumber features/adding.feature:3 # Scenario: Add two
numbers

1 scenario (1 failed)
3 steps (1 failed, 2 passed)
0m0.587s
```

Great! Now our test is failing for exactly the right reason: it's running our program, examining the output, and telling us just what the output *should* look like. This is a natural point to pause for a break. We've done the hard work for this release: when we come back to this code, Cucumber will be telling us exactly what we need to do to our program to make it work. If only all our requirements came ready-rolled with a failing test like

this, building software would be easy!

Try This

Can you write an implementation of calc.rb that makes the scenario pass? Remember at this stage, we have only a single scenario to satisfy, so you might be able to get away with quite a simple solution.

We'll show you our solution in the next section.

2.8 Making It Pass

So, now that we have our solid failing Cucumber scenario in place, it's time to let that scenario drive out a solution.

There is a very simple solution that will make the test pass, but it's not going to get us very far. Let's try it anyway, for fun:

first_taste/08/calc.rb

```
print "4"
```

Try it. You should see the scenario pass at last:

```
...

1 scenario (1 passed)
3 steps (3 passed)
0m0.479s
```

Hooray! So, what's wrong with this solution? After all, we already said that we want to do the minimum work that the tests will let us get away with, right?

Actually, that's not quite what we said. We said we want to do the minimum *useful* work that the tests will let us get away with. What we've done here might have made the test pass, but it isn't very useful. Apart from the fact that it certainly isn't going to function as a

calculator yet, let's look at what we've missed out on testing with our smarty-pants one-liner solution:

- We haven't tried to read the input from the command line.
- We haven't tried to add anything up.

In *Crystal Clear: A Human-Powered Methodology for Small Teams* [Coc04], Alistair Cockburn advocates building a *walking skeleton* as early as possible in a project to try to flush out any potential problems with your technology choices. Obviously, our calculator is trivially simple, but it's still worth considering this principle: why don't we build something more useful that passes this scenario *and* helps us learn more about our planned implementation?

If you're unconvinced by that argument, try looking at it as a problem of duplication. We have a hard-coded value of `4` in two places: once in our scenario and once in our calculator program. In a more complex system, this kind of duplication might go unnoticed, and we'd have a brittle scenario.

Let's force ourselves to fix this, using what Kent Beck calls *triangulation* (*Test Driven Development: By*

Example [Bec02]). We'll add another scenario to our feature, using a new keyword called a Scenario Outline:

first_taste/09/features/adding.feature
```
Feature: Adding

  Scenario Outline: Add two numbers
    Given the input "<input>"

    When the calculator is run
    Then the output should be "<output>"


    Examples:
      | input | output |
      | 2+2   | 4      |
      | 98+1  | 99     |
```

We've turned our scenario into a Scenario Outline, which lets us specify multiple scenarios using a table. We could have copied and pasted the whole scenario and just changed the values, but we think this is a more readable way of expressing the examples, and we want to give you a taste of what's possible with Gherkin's syntax. Let's see what the output looks like now:

```
$ cucumber

Feature: Adding

  Scenario Outline: Add two numbers
    Given the input "<input>"
    When the calculator is run
    Then the output should be "<output>"
```



```
    Examples:
      | input | output |
      | 2+2   | 4      |
      | 98+1  | 99     |
      expected: "99"
           got: "4" (using ==)
(RSpec::Expectations::ExpectationNotMetError)

./features/step_definitions/calculator_steps.rb:15
      features/adding.feature:6

Failing Scenarios:
cucumber features/adding.feature:3

2 scenarios (1 failed, 1 passed)
6 steps (1 failed, 5 passed)
0m1.035s
```

We can see from the summary 2 scenarios (1 failed, 1 passed) that Cucumber has run two scenarios. Each row in the Examples table is expanded into a scenario when Cucumber runs the scenario outline. The first example —where the result is 4—still passes, but the second example is failing.

Now it definitely makes sense to reimplement our program with a more realistic solution:

first_taste/10/calc.rb
```
print eval(ARGV[0])
```

First we read the command-line argument ARGV[0] and send it to Ruby's eval method. That should be able to

work out what to do with the calculation input. Finally, we print the result of eval to the console.

Try that. Do both scenarios pass? Great! You've just built your first program with Cucumber.

2.9 What We Just Learned

We've taken a quick skim over a lot of different things in this chapter, all of which will be covered again in more detail later. Let's recap and highlight some of the most important points.

Directory Structure

Cucumber likes you to use a conventional directory structure for storing your features and step definitions:

```
features/
  adding.feature
  ...
  step_definitions/
    calculator_steps.rb
    ...
```

You can override this structure, if you really want to, by passing arguments to Cucumber, but this is the conventional and simplest way to store your files.

Baby Steps

As we progressed through the example, did you notice how often we ran cucumber?

One of the things we love about working outside-in with Cucumber is how it helps us to stay focused. We can let Cucumber guide us through the work to be done,

leaving us free to concentrate on creating an elegant solution. By running Cucumber every time we make a change, any mistakes we make are found and resolved quickly, and we get plenty of feedback and encouragement about our progress.

Gherkin

Cucumber tests are expressed using a syntax called Gherkin. Gherkin files are plain text and have a `feature` extension. We'll talk more about Gherkin in Chapter 3, *Gherkin Basics*.

Step Definitions

Step definitions are the Ruby glue that binds your Cucumber tests to the application you're testing. When the whole thing plays together, it looks a bit like Figure 2, *The main layers of a Cucumber test suite*.

You'll learn more about step definitions in Chapter 4, *Step Definitions: From the Outside*.

After that whistle-stop tour of Cucumber's features, we're going to slow down and get into a bit more depth. We'll work our way in through the layers over the next few chapters, starting with a look at Gherkin, the language we use to write Cucumber features.

Try This

See whether you can add a second feature, `division.feature`, using the other example from the start of this chapter. Do you need to change the solution to make it pass?

Footnotes

[6] *The RSpec Book* [CADH09]