Contents

Chapter 1

Cucumber Backgrounder

1.1 Or: How I Learned to Stop Worrying and - Love Testing - Start Behaving

1.1.1 Introduction

Cucumber is a tool that implements a Behaviour Driven Design (BDD) workflow. This document deals principally with initial set up and first use of the **Cucumber-Rails** and **Cucumber** Rubygems. It takes as its background the Ruby on Rails (RoR) web application framework. Detailed discussion of Behaviour Driven (BDD), Test Driven (TDD), and Panic Driven Development (SNAFU aka Cowboy Coding) can be found elsewhere. Of course, there are still some skeptics on this whole Agile thing but, if you are reading this then you probably are not one of them.

You will find some familiarity with the Ruby language helpful and of the RoR framework somewhat less so. This article is aimed at the near novice and is somewhat long in consequence. If you are familiar with BDD/TDD concepts or are an experienced Rubist you will find some of the contents so obvious as to question its utility. Others, particularly those new to Ruby, may not have your advantages and it is for these readers that this material is provided.

Since the original version of this article appeared, Cucumber has undergone repeated revisions and re-factorings. Among these was the sensible decision to move portions of the implementation specific to particular programming frameworks into their own gems. Consequently, installing Cucumber for a framework now frequently starts with installing the specific framework Cucumber gem, which in turn pulls in the core Cucumber gem as a dependency. Cucumber provides support for a wide range of Ruby VMs such as JRuby, alternative application frameworks such as Sinatra, other programming languages such as Python, test environments such as Capybara, and provides i18n language support for feature and step files. Obtaining some of these features requires the installation of additional gems such as cucumber-sinatra.

Details regarding installing the Cucumber Rubygem and its recommended support tools for RoR are found on this wiki under the heading [[Ruby on Rails]]. To experiment with Cucumber and Cucumber-Rails I recommend that you create a new RoR project and use the default SQLite3 database. The article Getting Started with Rails is a useful introductory tutorial dealing with Rails and Cucumber.

Note that in this document I often use the terms testing and test where BDD practitioners prefer the terms behaviour and expectation. When I use the word test in a BDD context I am in fact discussing expressing and verifying expected behaviour.

Readers should always consider that the information contained herein may be out of date and therefore incomplete or erroneous in some respects. However, any such defects will usually be confined to specific implementation details and should not detract greatly from the validity of the overall presentation.

1.1.2 Where to Start?

```
Feature: Design and Build a Ruby on Rails web app using Behaviour Driven Development (BDD)
In order to reduce rework and produce a web app at low cost and high speed
A developer
Should employ a BDD methodology and agile tools

Scenario: Cucumber should be installed and configured
Given I have installed the gem named "rails"
  And I have installed the gem named "cucumber-rails"
  And I have generated a RoR project named "my_project"
  And the present working directory is the Rails root directory of "my_project"
  And I have the file cucumber.yml in the config directory of "my_project"
  And the file cucumber.yml contains a valid default profile

When I run "rails g cucumber"

Then I should create the directory ./features
  And I should create the file ./features/features.feature
  And I should create the directory ./features/step_definitions
.  .  .
  And I should create the file ./config/environments/environment/cucumber.rb
  And I should create the file ./config/cucumber.yml
  And I should modify ./config/database.yml
  .  .  .
```

The foregoing gives a sample of the form that feature files often take (sadly). The statements, called feature steps or feature statements, make up the user interface to Cucumber testing. Those given above are written in the Imperative Style simply for illustrative purposes. **Never put statements that look anything like these in a feature file** (but you will). In practice, all those Then/And statements should be subsumed into one simple Declarative Style statement. For example: `I should create the Cucumber environment`. The messy details of just what comprises a Cucumber environment are placed in the step definition files. Instead, a feature should look more like this:

```
Feature: Design and Build a Ruby on Rails web app using Behaviour Driven Development (BDD)
In order to produce a web app at low cost and high speed
A developer
Should employ Ruby on Rails with Cucumber BDD tools

Scenario: Cucumber-Rails should be installed and configured
  Given I am in a rails project root
    And I have installed cucumber-rails
    And I do not have a cucumber environment
  When I run the cucumber-rails generator
  Then I should have a cucumber environment
```

We will return to how to write features and steps later. For the moment we deal with the logical arrangement of Cucumber files within the context of an RoR project. The root level of the archetypal RoR project directory tree looks like this:

```
MyProject
.
| app
| config
| config.ru
| db
| doc
| Gemfile
| Gemfile.lock
| lib
| log
| public
| Rakefile
| README.rdoc
| script
| test
| tmp
 vendor
```

Depending upon the version of Rails, running `rails g cucumber:install` or `script/generate cucumber:install` adds this layout to the existing structure:

```
features
| step_definitions
 support
     env.rb
```

If you are not using Rails and Cucumber-Rails in your project then you can accomplish much the same thing by creating the directory tree from the command line (mkdir -p features/step_definitions) adding the support directory under features. People returning to Cucumber who are familiar with its early versions will note the absence of all the support files cucumber-rails used to provide saving only env.rb. Most significantly web_steps.rb is gone. Since version 1.1.0 you are required to provide all of your web steps from your own resources. (see wayback machine archive). But the old, deprecated, steps can still be found if you read the referenced blog entry.

It is a **really** good idea to read the contents of `./features/support/env.rb@` before doing anything else with cucumber and again after every update to cucumber-rails.

Once the features directory structure is in place then we are ready to begin testing with Cucumber.

1.1.3 Where do I put Tests?

Cucumber divides testing 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`.

[[Step definitions]], or stepdefs, are keyed by their snippets of text from the feature scenario statements and invoke blocks of Ruby and Rails code that usually contain api methods and assertion statements from whatever test system you have installed (MiniTest/TestUnit, RSpec, Shoulda, etc.). Given that Cucumber originally evolved out of RSpec stories it is unsurprising that the Cucumber-Rails generator once assumed that RSpec was available. This has long since ceased to be the case. What the generator does now is detect if the RSpec gems are installed. If so

then the rails generator builds the environment files to suit and if not then it ignores RSpec and configures for test-unit instead. In fact, the availability of options is increasing over time. To see what is available in the version of Cucumber-Rails that you have installed use the command: `rails g cucumber:install --help` or `script/generate cucumber --help`.

A particular source of potential confusion is that the term **steps**, when used loosely, has two, closely related but vitally distinct, meanings depending on context.

Inside feature.feature files, steps are the textual descriptions which form the body of a scenario. These are prefaced with the keywords **Given**, **When**, **Then**, **And** or **But** (note as well that the capitalization of these five names is significant). Alternatively, one may simply use the ***** character to stand in place of any of the foregoing.

Inside a step_definitions.rb file, steps (which strictly speaking should always be called step definitions and are now often simply called stepdefs) refers to the matcher methods, which may be given any of the same names (Given, When, Then, And or But), each provided with a matcher "string" or /regexp/ that corresponds to one or more feature steps. Note that the method name does NOT form part of the matcher. A Given feature clause can match a When step definition matcher. Over time, clauses from features have come to be referred to simply as features while steps now refers almost exclusively to step definitions.

As shown above, the generated features directory tree is fairly shallow. One can put every feature into a single file in the features directory and every step in a single file in the step_definitions directory (or even in the features directory itself) if one so chooses. Alternatively, one can choose to have one or more feature files for each feature, together with one or more step files for each feature file, or any combination thereof. However, Cucumber is programmed with the flexibility to support a much more expressive directory structure. For instance:

```
|-- features
|   |-- entities
|   |   |-- entity.feature
|   |   `-- step_definitions
|   |       |-- anything.rb
|   |       `-- entity_steps.rb
|   |-- locations
|   |   |-- location.feature
|   |   `-- step_definitions
|   |       `-- location_steps.rb
|   |-- sites
|   |   `-- step_definitions
|   |-- step_definitions
|   |   |-- local_assert_steps.rb
|   |   |-- local_crud_response_steps.rb
|   |   |-- local_email_steps.rb
|   |   |-- local_file_steps.rb
|   |   |-- local_script_steps.rb
|   |   |-- local_steps.rb
|   |   |-- local_web_steps.rb
`   |   `-- local_xml_file_steps.rb
    `-- support
        |-- env.rb
        |-- local_env.rb
        `-- local_transforms.rb
```

In this case the bland initial set-up has been divided into sub-directories informed by model-centric testing. This could equally well have been broken up in to model/controller/view hierarchies:

```
|-- features
```

```
|   |-- models
|   |   `-- entities
|   |       |-- entity.feature
|   |       `-- step_definitions
|   |           |-- anything.rb
|   |           `-- entity_steps.rb
|   |-- views
|   |   |-- entity_new
|   |   `-- step_definitions
|   |       `-- entity_new_steps.rb
|   |-- step_definitions
`   |   `-- local_steps.rb
    `-- support
        |-- env.rb
        |-- local_env.rb
        `-- local_transforms.rb
```

or this

```
|-- features
|   |-- invoicing.feature
|   |-- product.feature
|   |-- step_definitions
|   |   |-- local_steps.rb
|   |   |-- model_steps.rb
|   |   |-- service_steps.rb
|   |   `-- web_steps.rb
|   |-- user.feature
`   |-- user_auth.feature
    `-- support
        |-- env.rb
        |-- local_env.rb
        `-- local_transforms.rb
```

It is considered an anti-pattern to relate step_definition files to specific feature files. As is the case for many (most? all?) programming suggestions there are exceptions and contrary opinions respecting the orthodox position on the practice. Nonetheless, at the outset it is probably best to follow the recommendation that one avoid feature-specific step-definition files. Thereafter you may depart from it but only when you are experienced enough to evaluate the trade-offs between approaches.

Be aware that, regardless of the directory structure employed, Cucumber effectively flattens the features directory tree when running tests. By this I mean that anything ending in .rb under the start point for a Cucumber feature run (the default case) or specified with the `-r` option is searched for feature matches. Thus, a step contained in `features/models/entities/step_definitions/anything.rb` can be used in a feature file contained in `features/views/entity_new`, providing that cucumber is invoked on a root directory common to both, `./features` in this case; or explicitly required on the command line, `$ cucumber -r ./features features/views/entity_new`. Remember that step definition files can be called anything so long as they end in `.rb` and that anything ending in .rb anywhere under the root library directory for a Cucumber run will be treated as a step definition file.

Note that if the `-r` option is passed then **ONLY** that directory tree will be searched for step definition matches. You may specify the `-r` option multiple times if you need to include step definitions from directories that do not share a convenient root.

1.1.4 How do I Write Tests?

Constructing ones first tests, or features as BDD purists prefer, is often accompanied by what can only be described as writer's block. The question of "Where to begin?" seems to be a particular roadblock for some. If you truly have no idea of where to start then I suggest that you consider what you are writing, presumably a web application, and what the initial point of contact between it and a user is, the home page. You can do worse than simply starting with:

```
Feature: An application to do whatever
  In order to generate revenue
  The users
  Should be able to visit our web site

Scenario: The application has a home page
  Given I do have a web application
  When I visit the home page
  Then I should see the home page
```

Once upon a time the easiest thing to do for the first time tester/behaviourist was to use Cucumber's built-in scaffold generator to create a feature scaffold for each new feature desired and modify the resulting files to suit.

```
script/generate feature Frooble name color description
      exists  features/step_definitions
      create  features/manage_froobles.feature
      create  features/step_definitions/frooble_steps.rb
```

This might have been the easiest thing to do but, it was never the best thing to do. In my experience framework generated scaffolds of all types provide nothing more than a comforting illusion of progress through voluminous production of boilerplate code. And all too frequently said code is, for all intents and purposes, worthless; save only as an example of proper syntax and even then the syntax is oft-times of dubious quality.

In any case the whole point of BDD is that it is vitally important to write each test/feature step **one at a time, with a domain expert, and in plain language**. In the BDD world there is no point to feature scaffolding generators to begin with. This fact eventually led to the feature generator's removal from cucumber-rails. Now, like step definitions, you have to write your own code (or steal somebody else's) from the outset.

The use of plain language in the feature file is crucial to successful use of Cucumber. What does plain language mean? Basically, it comes down to stating the result that you wish to obtain while avoiding specifying how you expect to get it. Detailed discussion of feature writing and step construction are provided elsewhere (see [[Given-When-Then]] and Telling a Good Story).

For example, for an authentication scenario you should write:

```
When "Bob" logs in
```

and not:

```
  Given I visit "/login"
  When I enter "Bob" in the "user name" field
    And I enter "tester" in the "password" field
    And I press the "login" button
  Then I should see the "welcome" page
```

What is important about the difference in styles? The first example, **When "Bob" logs in**, is a functional requirement. The second, much longer, example is a procedural reference. Functional requirements are features but procedures belong in the implementation details. Ironically, given the propensity to use the word **should** in BDD/TDD, the real **Plain English** folks advocate the use of **must** in place of should or shall.

What you and your client should concern yourselves with in your feature files is that which has to happen and not how you expect it to happen. That way when somebody later decides that challenge and response authentication schemes are passé then you simply need change the authentication process steps behind the scenes. Your outward facing feature files, the ones that your clients get to see, need not change at all. In fact, a good question to ask yourself when writing a feature clause is: **Will this wording need to change if the implementation does**? If the answer is yes then the clause is poorly written and you should rework it avoiding implementation specific details. As a side benefit, in consequence your scenarios will be a lot shorter and much easier to follow and understand.

After each new feature statement is added to its scenario then you should immediately create the corresponding step definition method. This is where the implementation details are put because, in the normal course of events, your users will never see them. Once your new step definition is written then 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. Now that you have a passing step, without changing the step definition's logic change the test criteria within it to something that cannot be and prove to yourself that it fails again. Once you have assured yourself that your test is passing for the right reason then reset the criteria so that the test passes again. Once this cycle is complete then move on to the next feature clause. For example:

```
Scenario: Users can enter an invoice item
   .  .  .
   Then I enter a product quantity of 5
```

Now, immediately go to your step_definition file and do this:

```
When /enter a product quantity of (\d+)/ do |quantity|
  pending "TODO: Do we need to have a product code passed as well?"
end
```

Think about how you are going to express this behaviour in your application and how you can detect that it occurs. Go back and rework your feature and step until you are satisfied that it will indeed produce some testable result **and** that the test fails. Now, go write the code to implement this requirement in your application.

It is tempting, sometimes irresistibly so, to just skip ahead with the analysis stage alone and to complete as many features, scenarios and scenario statements as one can imagine. In some cases limited access to domain experts and end users may require that many features have their scenario details completed long before coding the associated step definitions is undertaken. When this is avoidable it should be and when it is not avoidable then every effort should be made to avoid it nonetheless. In the long run, the best results are achieved when you write feature statements and step definitions incrementally, using the absolute minimum of code to express the requirement. Immediately implement the new step requirement in the application using the absolute minimum code that will satisfy it.

You should, in fact, treat this part somewhat as a game. The application code that you write should literally be the minimum that will satisfy the requirement; particularly if this code is totally unsuited for production use. This way you are forced to add additional feature scenarios to drive

out exactly what is acceptable and this pressure forces the application code to evolve strictly to meet those requirements. This might seem foolish and a waste of time but if you pre-empt the design process by writing more sophisticated code than is called for then you will inevitably fail to provide scenario coverage for some of that code. You will also write code that will never, ever, be used. This **will** happen and it **will** bite you at some point. Keep the YAGNI principle in mind at all times.

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. More importantly, if you rigorously adhere to this methodology then your application will contain the minimal code that satisfies required features. This is an often overlooked or undervalued consideration that contributes greatly to the efficiency of coding, to the robustness of the resulting code, and to the performance of the the resulting program. Avoiding diversions into technically interesting but financially pointless coding adventures concentrates your limited resources on the tasks that count and measurably reduces the overall complexity of the project. Whenever you find yourself led down this garden path to the creeping **featuritis** plant ask: **If the user did not ask for it then exactly why are we writing it?**

Strictly following this approach permits you to face significant design changes (and gem updates) with complete equanimity. Having built your code to reproducible performance measurements you may rest secure in the knowledge that if unanticipated changes anywhere in your project break anything then you will know of this immediately upon running your test suite (Which is always running, right? Right??). More importantly, you will know exactly what is broken and where it is broken.

As is the case with most professions, the real value that a skilled programmer provides lies not so much in knowing how to do something as in knowing when and where it must be done. The real challenge with maintaining code is simply discovering which piece of code to change. Finding the exact spots in an application that need attention is usually the biggest maintenance problem. By strictly coding to features backed by suitable step definitions you can simplify that task almost to the point of triviality.

If it happens that, on occasion, you do anticipate feature steps (and we all do this however much we try not to) then omitting any matcher for them in the step definitions files causes those steps to be reported as missing by cucumber. Not only does cucumber report them it helpfully provides a suggested step matcher and argument to implement. If you end up writing stub step matchers prior to full implementation then you have an explicit **pending** method available to designate defined but pending/unspecified/stub step definitions. The `pending` method provides for specifying an optional message. Step definitions containing the `pending` method will display as defined but pending in your cucumber runs and will print any message that you provided it.

```
Given /this step is not implemented yet/ do
  pending "your message goes here"
end
```

By the way, never, ever, write a step definition for a clause that is not already present in one of your features. Most such step definitions will end up as unused cruft that somebody at sometime will end up discarding anyway. And those few that are not discarded will need rework when they are finally employed. It is best to wait until the need for each step definition is both evident and pressing.

1.1.5 What are Features and Scenarios?

A feature can be conceptualized as an indivisible unit of functionality embedded in the project to which it belongs. For example, an authentication challenge and response user interface is usually considered a feature while an entire authentication system necessarily comprises many features. A

single **Feature** is typically contained in its own file (ending in `.feature`). Each Feature is usually elaborated though multiple **Scenarios**.

A **Scenario** is a block of statements inside a feature file that describe some behaviour desired or deprecated in the feature to which it belongs. A scenario might check that the login interface provides input fields for the requisite responses, that failures are logged or otherwise reported, that user ids are locked out after a given number of failed attempts, and so forth. Each scenario exercises the implementation code to prove that for each anticipated condition the expected behaviour is indeed produced. Recall that scenarios specify **What** and should avoid answering the question: **How**?

Each Scenario consists of three classes of statements, **Given**, **When** and **Then** which effectively divide each scenario into three stages. Each stage of a scenario consists of one or more statements that are used to match to test step definitions. The conventional arrangement is:

```
Feature: Some terse yet descriptive text of what is desired
In order that some business value is realized
An actor with some explicit system role
Should obtain some beneficial outcome which furthers the goal
To Increase Revenue | Reduce Costs | Protect Revenue  (pick one)

  Scenario:   Some determinable business situation
      Given some condition to meet
         And some other condition to meet
       When some action by the actor
         And some other action
         And yet another action
       Then some testable outcome is achieved
         And something else we can check happens too

  Scenario:  A different situation
      ...
```

For Cucumber features the key words used here are **Feature**, **Scenario**, **Given**, **When**, **Then**, and **And**. Feature is used to provide identification of the test group when results are reported.

At the present time (2013-Jul-10) the `Feature` statement and its descriptive text block are not used by Cucumber other than as an identifier and documentation. However, the `Feature` statement arguably contains the most important piece of information contained in a feature file. It is here that you answer the question of just why this work is being done. And if you do not have a very good, defensible, reason that can be clearly elucidated in a few sentences then you probably should not be expending any effort on this feature at all. First and foremost, **BDD** absolutely **MUST** have some concrete business value whose realization can be measured before you write a single line of code. (see popping the why? stack)

As with `Feature`, `Scenario` is used only for identification when reporting failures and to document a piece of the work. The clauses (steps) that make up a Scenario each begin with one of: Given, When, Then, And and But (and sometimes *). These are all [[Gherkin]] keywords / Cucumber methods that take as their argument the string that follows. They are the steps that Cucumber will report as passing, failing or pending based on the results of the corresponding step matchers in the step_definitions.rb files. The five keywords (and *) are all equivalent to one another and completely interchangeable.

1.1.6 What are Step Definitions?

The string following the keyword in the feature file is compared against all the matchers contained in all of the loaded step_definitions.rb files. A step definition looks much like this:

```
Given /there are (\d+) froobles/ do |n|
  Frooble.transaction do
    Frooble.destroy_all
    n.to_i.times do |n|
      Frooble.create! :name => "Frooble #{n}"
    end
  end
end
```

The significant things here are that the method (Given) takes as its argument a regexp bounded by `/` (although a quoted "simple string" can be used instead) and that the matcher method is followed by a block. In other words, written differently in Ruby this matcher method could look like this:

```
Given( /there are (\d+) froobles/ ) { |n|
.  .  .
}
```

Among other things, this means that the step definition method blocks receive all of the matcher arguments as string values. Thus `n.to_i.times` and not simply `n.times` (but also look into Cucumber transforms). It also means that step matchers themselves can be followed by the special regexp modifiers, like **i** if you want to avoid issues involving capitalization.

In the feature example provided above we had the scenario statement: `And some other action`. This could be matched by any of the following step definition matchers if present in any step_definitions.rb file found under the features root directory.

```
Given /some other action/ do
When /some other Action/i do
When /some other (Action)/i do |action|
Then /(\w+) other action/i do |prefix_phrase|
Given /(\w+) other (\w+)/i do |first_word,second_word|
But /(\w+) Other (.*)/i do |first_word,second_phrase|
And /(.*) other (.*)/i do |first_phrase,second_phrase|
```

The step definition match depends only upon the pattern given as the argument passed to the Given/When/Then method and not upon the step method name itself. I have therefore adopted the practice of only using `When /I have a match/ do` in my step definitions files as **When** has a more natural appearance, to me, for a matcher. Others find that the word "Given" has a more natural language feel in this context.

If Cucumber finds more than one matcher in all of the step definitions files matches a scenario statement then it complains that it has found multiple step definition matches for that step and forces you to distinguish them. You can instruct Cucumber to just choose one of the candidates instead by passing it the `--guess` option on the command line.

It is considered better form by some to surround with double quotation marks, `" "@`, all of the elements in the feature step clauses that are meant to be taken as values for variables passed to the step definition. This is just a convention. However, if you choose to follow this road then you must adjust your step definition matchers accordingly if the " characters are now considered part of the literal matcher string. For example:

feature statement:

```
Given some determinable "business" situation
```

step definition:

```
When /determinable "(.*)" situation/ do |s|
```

Finally, you can have step definitions call other step definitions, including those contained in other step definitions files. This is where you can specify the procedural details by combining other steps. For example:

```
When /some "(.*)" action/  do |act|
   .  .  .
end

When /in an invoiced non-shipped situation/ do
  step( "some \"invoiced\" action" )
  step( "some \"non-shipped\" action" )
  .  .  .
end
```

1.1.7 Steps within Steps – an anti-pattern

If one step definition calls another step definition then the matcher argument to the called **Given/When/Then** method must be enclosed with string delimiters. Because of this, if you have adopted the practice of demarcating parameter values present in feature steps with double quotation marks, you must escape these quotation marks when calling another definition_steps.rb matcher from inside a step_definitions.rb file. You must take care not to include the quote marks in the step_definitions parameter matchers, for `"(.*)"` is not the same as `(.*)` or `(".*")`. If you use quote delimited values in the .feature file steps and do not account for them in the corresponding step_definition.rb matcher regexp then you will obtain variables that contain leading and trailing quotes as part of their value.

```
Scenario: Quotes surround value elements
  Given some "required" action

# step_definitions
When /some (.*) action/ do |a|
  a => "required"

When /some "(.*)" action/ do |a|
  a => required
```

Once-upon-a-time one could simply nest Given, When, Then matchers in step definition files and call steps from within other steps directly. This practice became more and more frowned upon until at last that ability was removed (well strongly deprecated) and nested steps must now be called using the `step@` method. You may still encounter the following forms in step definition files created before this change:

```
When /my matcher named (.*)/ do |match|
  Then "my other matcher named \"#{match}\""
end

When /my matcher named (.*)/ do |match|
  When %Q(my other matcher named "#{match}")
```

```
end
```

When encountered these nested Then and When keywords should be replaced with the step method:

```
When /my matcher named (.*)/ do |match|
  step( "my other matcher named \"#{match}\"" )
end

When /my matcher named (.*)/ do |match|
  step %Q(my other matcher named "#{match}")
end
```

In Ruby programming usually one may choose to include the method parameter list within parentheses or not. Both forms of calling the step method are shown above.

Using the %Q method (which is usually shortened to just %) within the step method removes the necessity to escape (\) any embedded quotation characters (”). Multiple steps may be called en bloc using the `steps@` (note the plural form) method which itself takes a string argument. However, with the steps method the Gerkin keywords deprecated in simple nested steps are still required:

```
When /my matcher named (.*)/ do |match|
  steps %Q{
    Then step my other matcher named "#{match}"
     And the next matcher with value "{match}"
  }
end
```

Always keep in mind that Cucumber is simply a DSL wrapper around the Ruby language whose full expressiveness remains available to you in the step definition files (but not in the feature files). On the other hand, do not lose sight that every step called as such in a step definition file is first parsed by [[Gherkin]] and therefore must conform to the same syntax as used in feature files.

Returning to our example of “Bob” the user, one could define things in the step_definitions file like this:

```
When /"Bob" logs in/ do |user|
  steps( %Q(
    Then I visit "/login"
      And I enter "#{user}" in the "user name" field
      And I enter "#{user}-test-passwd" in the "password" field
      And I press the "login" button
  ) )
```

That is acceptable (barely) usage in your step_definitions because your users are never going to see how ugly it looks. Instead, given that the necessary classes and methods exist, “Bob” could, and should, be authenticated without recourse to the user interface thus:

```
When /"Bob" logs in/ do |user|
```

```
  @current_user = User.find_by_username!(user) # ! method raises exception on failure
  @current_session = UserSession.create!(@current_user)  # ! method raises exception on failure
  .  .  .
end
```

In fact, this step should be further re-factored into a method. And said method can reside in the same .rb file as the step definition. This both simplifies the step and encourages the reuse of the resulting method. It also adds immeasurably to the ease of comprehension for people unfamiliar with the project history who may later have cause to review your tests. Can you spell M A I N T E N A N C E?

```
When /‘‘Bob’’ logs in/ do |user|
  create\_new\_user\_session\_for( user )
end
```

```
def create_new_user_session_for( user_in )
  # TODO create proper setter and getter methods for instance variables
  @current_user = User.find_by_username!( user_in ) # ! method raises exception on failure
  @current_session = UserSession.create!(@current_user)  # ! method raises exception on failure
  .  .  .
end
```

Of course, when you are testing the login user interface the ugly approach seems unavoidable, but in fact it is not. Providing for the purposes of testing that certain conventions are followed respecting user names and passwords the following works just as well and is much cleaner. Plus you have removed all inter-step dependencies.

feature statement:

```
When "Bob" logs on through the logon page
```

step definition:

```
When /"([\w[\d\w]+)" logs on through the logon page/ do |user_name|
  visit(logon_path)
  fill_in( "User Name", :with => user_name )
  fill_in( "Password", :with => user_name + "-test-passwd" )
  click_button( "Logon" )
end
```

Having just shown you how to do it now take heed that you do not write **any** step definitions that call other steps (you will do this too, but try hard not to). At times this will seem like the quickest solution to a troublesome bit of environment building. However, for anything beyond trivial use it is always better to implement a custom method using the **api** provided by Cucumber (or by any other libraries you have installed) and then call that method directly from your step. You can either keep these custom methods in the same file as the regular steps or stick them in any convenient file ending in `.rb` that is located in the support directory (well, anywhere that cucumber can find it really) in which case you must enclose your methods within the following block:

```
Cucumber::Rails::World.class_eval do
  def your_method(parm)
    .  .  .
  end
end
```

My rule of thumb is that if a step definition is called from another step definition then its contents probably should be extracted out into a custom method. For example a logon step definition is likely to be used repeatedly throughout many features. Turning it into a method is probably called for.

```
def logon_for_user( uname )
  visit(logon_path)
  fill_in( "User Name", :with => uname )
  fill_in( "Password", :with => uname + "-test-passwd" )
  click_button( "Logon" )
end

When /‘‘([\textbackslash{}w[\textbackslash{}d\textbackslash{}w]+)’’ logs on through the logon page/
  logon\_for\_user( user\_name )
end
```

1.1.8 Before, After and Background

If all your feature's scenarios share the same 'setup' feature steps then Cucumber provides the Background section. Steps contained within a Background section are run before each of the scenarios.

```
Feature: .  .  .
  Background: .  .  .
  Scenario: .  .  .
```

Step definition files have a corresponding method available in the `before(condition) do . . .` method, which has however a matching `after(condition) do . . .` method as well. Recall that we are working in Ruby and therefore the condition which enables the before/after block is anything that is not false or nil, like a **tag** for instance. Also be aware that **all** eligible `before` methods are run before any scenario statements are processed, and that they are run in the order encountered. Likewise, every eligible `after` block will run at the completion of every scenario, again in the order that it is encountered. These two methods are powerful tools, but be aware that if you use them excessively then you will hang yourself eventually.

1.1.9 What is a good Step Definition?

Opinions vary of course, but for me a **good** step definition has the following attributes:

- The matcher is short.
- The matcher handles both positive and negative (true and false) conditions.
- The matcher has at most two value parameters

- The parameter variables are clearly named
- The body is less than ten lines of code
- The body does not call other steps

My template for a step definition presently looks like this:

```
When /statement identifier( not)? expectation "([^\"]+)"/i do |boolean, value|
  actual = expectation( value )
  expected = !boolean
  message = "expectation failed for #{value}"
  assert( actual == expected, message )
end
```

For example (admittedly contrived):

```
When /product ([^\"]+) should( not)? belong to category ([^\"]+)/i do |product, boolean, category|
  actual = ( Product.find_by_stock_number!( product )category ) == category
  expected = !boolean
  message = "Product '#{product}' should#{boolean} belong to category '#{category}'"
  assert( actual == expected, message )
end
```

1.1.10 What are "tags"?

Cucumber provides a simple method to organize features and scenarios by user determined classifications. This is implemented using the convention that any space delimited string found in a feature file that is prefaced with the commercial at (`@@`) symbol is considered a tag. As distributed, Cucumber-Rails builds a Rake task that recognizes the `@wip@` tag. However, any string may be used as a tag and any scenario or entire feature can have multiple tags associated with it. For example:

```
  @init
  Feature:  .  .  .
  .  .  .
  @wip @authent
  Scenario:  A user should authenticate before accessing any resource.
     Given I do have a user named "testuser"
      When the user visits the login page
         And the user named "testuser" authenticates successfully
     Then I should see .  .  .
         .  .  .
```

Given that the forgoing is contained in a file called `features/login/login.feature` and that the cucumber-rails gem is installed and configured then you can exercise this scenario, along with any others that are similarly tagged, in any of the following ways:

```
$ rake cucumber:wip
```

```
$ cucumber --profile=my_profile --tags=@wip features
$ cucumber --profile=my_profile --tags=@authent features/login
$ cucumber --profile=my_profile --tags=@init
```

However, the following will not work, unless you [[build a custom rake task—Using-Rake]] for it:

```
$ rake cucumber:authent
```

There is an obscure gotcha with this particular combination of tags. The default profile contained in the distributed `config/cucumber.yml` contains these lines:

```
<%
.  .  .
std_opts = "--format #{ENV['CUCUMBER_FORMAT'] || 'progress'} --strict --tags ~@wip"
%>
default: <%= std_opts %> features
.  .  .
```

Note the trailing option `--tags` ~wip@. Cucumber provides for negating tags by prefacing the `--tags` argument with a tilde character(`@`). This tells Cucumber to not process features and scenarios so tagged. If you do not specify a different profile (`cucumber -p profilename`)then the default profile will be used. If the default profile is used then the `--tags` ~wip@ will cause Cucumber to skip any scenario that is so tagged. This will override the `--tags=`authen@ option passed in the command line and so you will see this:

```
$ cucumber --tags=@authent
Using the default profile...

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

Since version 0.6.0 one can no longer overcome this default setting by adding the `--tags=`wip@ to the Cucumber argument list on the command line because now all `--tags` options are ANDed together. Thus the combination of `--tags` wip@ **AND** `--tags` ~wip@ fails everywhere. You either must create a special profile in `config/cucumber.yml` to deal with this or alter the default profile to suit your needs. Note as well that @wip tags are a special case. If any scenario tagged as @wip passes all of its steps without error and the `--wip` option is also passed then Cucumber reports the run as failing since scenarios that are marked as a work in progress are not supposed to pass. Note as well that the `--strict` and `--wip` options are mutually exclusive.

The number of occurrences of a particular tag in your feature set may be controlled by appending a colon followed by a number to the end of the tag name passed to the tags option, as in `$ cucumber --tags=`wip:3 features/log*@. The existence of more than the specified number of occurrences of that tag in all the features that are exercised during a particular cucumber run will produce a warning message. If the `--strict` option is passed as well, as is the case with the default profile, then instead of a warning the run will fail.

Limiting the number of occurrences is commonly used in conjunction with the wip@ tag to restrict the number of unspecified scenarios to manageable levels. Those following Kanban or Lean Software Development based methodologies will find this facility invaluable.

As outlined above, tags may be negated by prefacing the tag with the tilde (`~`) symbol. In other words, you can exclude all scenarios that have a particular tag proviing that tag is not elsewhere passed to cucumber as a parameter. For example, the following will only exercise all scenarios found in the directory tree rooted at `features/wip` that do not have the tag `@ignore@`:

```
$ cucumber --require=features --tag=~@ignore features/wip
```

A convention that I have adopted is tagging all scenarios created to track down a specific defect with tags of the form @issue_### where ### is the issue number assigned to the defect. This both handles multiple related scenarios and provides a convenient and self-documenting way to verify with cucumber that a specific defect either has been completely resolved or that a regression has occurred.

Be aware that tags are heritable within Feature files. Scenarios inherit tags from the Feature statement.

1.1.11 What Way do I Run the Tests?

Unless you are knowledgeable enough that you can use mocks and stubs with flair then I consider it best to begin with creating a Rails migration file for the models you are testing (or expressing features for) followed by:

```
rake db:migrate
rake db:test:prepare
```

As this is opinionated software my opinion is that, except for the most trivial of cases, you should always use test data obtained from actual production environments. You are, after all, embarked on a real-world adventure; namely to discover how to make something work. However, to discover what actually works requires more than a passing familiarity with what is real. And made-up data is not reality. Since your manufactured data necessarily originates in the same place as most of your errors will come from, your own limited understanding of the problem domain, it is always suspect.

That said, there remains an important environmental consideration to keep in mind when using an actual database for testing: Cucumber, by default, uses database transactions and these transactions are rolled back after each scenario. This makes out-of-process testing problematic (for that see the Cucumber Aruba project) and may result in some unanticipated outcomes under certain scenarios. Transactions can be turned off, but then your features become responsible for ensuring that the database is in a condition suitable for testing. Cucumber provides hooks to accomplish this and the gem Database-Cleaner is configured in support/env.rb to assist (you have read `env.rb@, right?`). In the normal case the end of any scenario results in the database being returned to a nil state.

Cucumber can be run in several ways. Be aware that `rake cucumber`, `cucumber features`, and `autotest` with `ENV AUTOFEATURE=true` do not necessarily produce the same results given the same features and step definitions.

Running `rake cucumber` from the command line provides the simplest, if not the speediest, method to run Cucumber tests. The rake script provided with cucumber performs much of the background magic required to get the test database and requisite libraries properly loaded. In fact, an important habit to acquire is to run cucumber as a rake task immediately after performing a migration. This step ensures that the test database schema is kept in sync with the development database schema. You can achieve the same effect by running `rake db:test:prepare` before your first cucumber run following a migration but developing the habit of just running `rake cucumber` or `rake cucumber:wip` is probably the better course.

As discussed above, the Cucumber Rake task recognises the wip@ tag, so `rake cucumber:wip` will run only those scenarios tagged with **@wip**. For example, given a feature file containing:

```
Feature: . . .

  Scenario: A

  @wip
  Scenario: B

  Scenario: C
```

Then running the command `rake cucumber:wip` will run the steps contained inside Scenario B only, while running `rake cucumber:ok` will run the steps within all Scenarios other than B.

Cucumber-Rails creates a `cucumber.yml` file in the project config directory containing a number of predefined profiles, one of which is the default profile. When Cucumber is run from the command line it is usually necessary to provide both the directory name containing the root directory of the tree containing feature files and the directory name containing references to the necessary library files. In the typical project `cucumber -r features features/some/path` will normally suffice. Repetitious usages can be added to user defined profiles contained in the project's `cucumber.yml` file.

Finally, running `autotest` with the environment variable `AUTOFEATURE=true` will run ALL tests, including those in `/test` and (if present) `/rspec`. As this will load all the TestUnit and RSpec fixtures as well, your test database may be left in an indefinite state when the Cucumber features are run. It is wise, as always, to write Cucumber steps either so that they do not depend upon an empty database or they place the database in the requisite state.

1.1.12 Ruby, Rails, Bundler, RVM and RBenv

Software is frequently designed to run on different environments and new software may need to co-exist with earlier efforts whose dependencies do not match those of current projects. Ruby has the Ruby Version Manager (RVM) and RBenv to manage multiple vm environments on a single host. Recent versions of Ruby on Rails ship with a dependency on Bundler, a utility gem that manages project specific RubyGem dependencies such that each Rails project is unaffected by the gem requirements of another. These are covered elsewhere but you should be aware of them from the outset of your project.

The only consideration relating to these I will mention here is that if you use Bundler to support multiple gem versions in multiple Rails projects on a single development host then you must run cucumber using `bundle exec cucumber [options]`.

1.1.13 Anything Else?

The terminology for elements of Behaviour Driven Development differs somewhat from that employed by Test Driven Development. This article, because of the introductory nature of its contents, tends to blur the semantic distinction between these two divergent philosophies.

Cucumber is still evolving, although the pace seems to have slowed somewhat (finally!). Originally, Cucumber was written for Ruby on Rails but, as discussed above, this has long since ceased to be true. Besides Cucumber-Rails Cucumber now has another supplementary library, Aruba, which permits testing of Command Line Interface processes and shell scripts. This article is revised to Cucumber version 1.2.1 and Cucumber-Rails version 1.3.0 but it does not cover many of the ever expanding attributes of Cucumber and only mentions the JVM version, Cuke4Duke, here.

Because of this consideration it would not be wise to use any of the examples from this article as a recipe. Nonetheless, the essentials of this article remain applicable throughout all recent versions

of Cucumber and Cucumber-Rails even where the implementation details may have changed since this review.

Cucumber supports tables in feature files. These are roughly analogous to Framework for Integrated Test (FIT) tables. You can use these when you are specifying behaviour that changes at some data threshold or as a substitute for data fixtures. I tend to avoid using tables in feature files altogether and use them sparingly in step definition files. I do not have any explicit reason for this avoidance but, tables and feature statements just do not seem to go together in my head. Cucumber also provides for scenario outlines using an Example block to cut down on repetitive scenarios. You should also research the use of transforms in Cucumber.

If you are testing with intent then you should be using the debugger gem. A really neat method to drop into an interactive debugging session inside a Cucumber step definition using debugger's predecessor, ruby-debug, was provided by Scott Taylor on the rspec mailing list (The technique should still work with debugger but I have not yet tested it). Just put these statements inside the step definition at the point that you wish to debug: `require 'rubygems'; require 'debugger'; debugger`. When that code interrupts then type `irb` and you open an interactive debugging session wherein you can step forwards and backwards inside the code under test to determine exactly where the breakage is happening. Alternatively, you can add `require 'rubygems'; require 'debugger';` to your `support/local_env.rb` file (see below) and just put `debugger` wherever you desire it inside any step definition.

Realize that tests/assertions/expectations either pass or fail (raise an error) and that fail is NOT the same as false, whereas anything but fail is a pass. When, in RSpec, `something.should_be 0` and it is not, then what is returned is an error exception and not a Boolean value. In raw Cucumber (pardon the pun) one writes `fail if false` and not simply `false`. A little reflection reveals why this is so, since false might be the expected successful outcome of a test and thus not an error. However, this distinction between fail and false escaped my notice until I tripped over it in an actual test suit.

Sometimes however, we wish to test how our application handles an exception and therefore do not want that exception to be handled by Cucumber. For that situation use the allow-rescue@ tag (and read the contents of `env.rb@`)

Recall that Cucumber is 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.

Cucumber-Rails is pre-configured with support for view integration testing using Capybara (script/generate cucumber —capybara). As of v0.5.0 support for Webrat (script/generate cucumber) was dropped. If you do not know what Capybara or Webrat are or do, and you are doing web application testing, then you should find out more about both. Unless instructed otherwise the Cucumber-Rails install generator will set up the necessary support files for Capybara. After a significant delay in release Webrat now supports Rails-3.x as of v0.7.3. However, since the release of Rails 3.0, Capybara has apparently captured sufficient mind-share in the Cucumber community that generator support for Webrat was dropped and Capybara is now the default for cucumber-Rails.

While Capybara has emerged as the preferred testing method for html views it does not play well with Rails' own built-in MiniTest/Test::Unit. In particular, whenever Capybara is required into a Cucumber World then the response.body method of Rails Test::Unit is removed. This is an annoyance more than anything else but people converting from Webrat need to be aware of it. Capybara depends upon Nokigiri and Nokogiri prefers to use xml rather than css tags. This behaviour is overridden in `./features/support/env.rb` (I did mention that you should really read env.rb did I not?).

Calls for any additional Ruby libraries that your particular testing environment may require are typically placed in the `./features/support/local_env.rb` file. I advise against putting local customization in `support/env.rb` as it is typically overwritten by `script/generate cucumber:install | rails g cucumb`
As a matter of good practice you should always run `script/generate cucumber | rails g cucumber:install`

whenever you install an updated version of cucumber-rails. Unfortunately, there are some configuration options that simply **must** go into env.rb to have effect. So, check in your env.rb along with the rest of your version controlled files and be prepared to merge changes between versions of Cucumber-Rails.

Those of you that have used `growl` or `snarl` to provide desktop notifiers from autotest are advised that, as of this writing, Cucumber did not hook into the :red :green notifier capability of autotest; so, no pop-ups when a step fails. However, there exists a project to add a similar functionality to Cucumber. See Cucumber_Growler.

`autotest` is installed via the ZenTest gem. If you use autotest then take a look at the contents of `example_dot_autotest.rb` in the ZenTest gem root directory.

1.1.14 Note Respecting cucumber-rails v0.5.0. (2011 June 28)

The latest versions of cucumber and cucumber-rails are oriented towards Ruby-1.9.3. If you are using Ruby-1.8.7 and you are employing testunit or Capybara then one or both of the following may need to be added to the `support/env.rb` file produced by the rails generator:

```
# for testunit in Ruby-1.8.7
ENV["RAILS_ENV"] = "test"
if RUBY_VERSION =~ /1.8/
  require 'test/unit/testresult'
  Test::Unit.run = true
end
# for capybara
require "capybara"
```

This requirement may no longer be case following the release of cucumber-rails v0.5.1 but I have not checked this against Ruby-1.8.7 as yet (and may never get around to it). In any case, Rails-4 is dropping support for all versions of MRI Ruby prior to v1.9.3 so do yourself a favour and just use that version or later for anything new.

1.1.15 Need Help?

The best place to go for help, that I know of, is the Google Cucumber Group.

If you find a bug in Cucumber, or wish a new feature added, then you should open a ticket at GitHub for it.

2008 November 28 – J. B. Byrne initial 2010 January 17 – J. B. Byrne revised to 0.6.1 2010 May 28 – J. B. Byrne revised 0.7.3 2010 July 13 – J. B. Byrne revised to 0.8.4 2010 October 11 – J. B. Byrne revised to 0.9.0 2010 November 06 – J. B. Byrne revised to 0.9.4 2011 March 29 – J. B. Byrne revised to 0.10.0 2011 May 13 – J. B. Byrne revised to 0.10.2 2011 June 28 – J. B. Byrne revised to 1.0.0 2012 October 19 – J. B. Byrne partially revised to 1.1.0 2012 November 29 – J. B. Byrne revised to 1.1.1 2013 February 26 – J. B. Byrne revised to 1.2.1 2013 July 10 – J. B. Byrne reviewed with minor corrections.

1.1.16 Postscript.

A caution, Cucumber is meant to facilitate expressing required behaviours. Indirection and excessive adherence to the principle of DRY, particularly in features, is at variance with the intent and defeats the major benefit of the tool. Requirement expression in features should remain as self evident to the non-technical reader and be as self contained as possible. Resist the temptation to program the features themselves using esoteric aspects of the DSL. Features should remain patent statements of intent. If you feel the need to "program" a scenario in order to simplify writing a feature then you are likely doing something considerably at odds with the fundamental intent of

BDD methodology. In such circumstances, mentally step back and reconsider your approach to the problem.

Remind yourself, frequently, that some years hence somebody else is going to have to understand what you write today and that in all likelihood you will not be around to explain it. Do not be overly clever and needlessly terse in Feature files and Step Definitions. Save the coding-fu for the the code and leave the tests plain, unadorned, and easy to understand.

A final word of advice: Get the Cucumber Book from the Pragmatic Programmers and **read** it; carefully. Details are found at http://pragprog.com/book/hwcuc/the-cucumber-book

Chapter 2

Gherkin

Gherkin is the language that Cucumber understands. It is a Business Readable, Domain Specific Language that lets you describe software's behaviour without detailing how that behaviour 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.

Gherkin's grammar is defined in the Treetop grammar that is part of the Cucumber codebase. The grammar exists in different flavours for many [[spoken languages]] (37 at the time of writing), so that your team can use the keywords in your own language.

There are a few conventions.

- Single Gherkin source file contains a description of a single feature.
- Source files have `.feature` extension.

2.1 Gherkin Syntax

Like Python and YAML, Gherkin is a line-oriented language that uses indentation to define structure. 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 hash 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 Ruby code block called [[Step Definitions—step definitions]].

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.

Read more

[Feature Introduction] – general structure of a feature

[Given-When-Then] – steps

Chapter 3

Feature Introduction

Every .feature file conventionally consists of a single feature. A line starting with the keyword **Feature** followed by free indented text starts a feature. A feature usually contains a list of scenarios. You can write whatever you want up until the first scenario, which starts with the word **Scenario** (or localized equivalent; Gherkin is localized for [[dozens of languages—Spoken languages]]) on a new line. You can use [[tagging—Tags]] to group features and scenarios together independent of your file and directory structure.

Every scenario consists of a list of steps, which must start with one of the keywords **Given**, **When**, **Then**, **But** or **And**. Cucumber treats them all the same, but you shouldn't. Here is an example:

```
Feature: Serve coffee
  In order to earn money
  Customers should be able to
  buy coffee at all times

  Scenario: Buy last coffee
    Given there are 1 coffees left in the machine
    And I have deposited 1$
    When I press the coffee button
    Then I should be served a coffee
```

In addition to a scenario, a feature may contain a background, scenario outline and examples. Respective keywords (in English) and places to read more about them are listed below. You can get a list of localized keywords with `cucumber --i18n [LANG]`.

3.1 Step definitions

For each step Cucumber will look for a matching **step definition**. A step definition is written in Ruby. Each step definition consists of a keyword, a string or regular expression, and a block. Example:

```
# features/step_definitions/coffee_steps.rb

Then "I should be served coffee" do
  @machine.dispensed_drink.should == "coffee"
end
```

keyword	localized	more info, see
name	'English'	
native	'English'	
encoding	'UTF-8'	
feature	'Feature'	[[Feature Introduction]]
background	'Background'	[[Background]]
scenario	'Scenario'	[[Feature Introduction]]
scenario_outline	'Scenario Outline'	[[Scenario outlines]]
examples	'Examples' / 'Scenarios'	[[Scenario outlines]]
given	'Given'	[[Given-When-Then]]
when	'When'	[[Given-When-Then]]
then	'Then'	[[Given-When-Then]]
and	'And'	[[Given-When-Then]]
but	'But'	[[Given-When-Then]]

Step definitions can also take parameters if you use regular expressions:

```
# features/step_definitions/coffee_steps.rb

Given /there are (\d+) coffees left in the machine/ do |n|
  @machine = Machine.new(n.to_i)
end
```

This step definition uses a regular expression with one match group – `(\d+)`. (It matches any sequence of digits). Therefore, it matches the first line of the scenario. The value of each matched group gets yielded to the block as a string. You must take care to have the same number of regular expression groups and block arguments. Since block arguments are always strings, you have to do any type conversions inside the block, or use [[Step Argument Transforms]].

When Cucumber prints the results of the running features it will underline all step arguments so that it's easier to see what part of a step was actually recognised as an argument. It will also print the path and line of the matching step definition. This makes it easy to go from a feature file to any step definition.

Take a look at [[Step Definitions]] and the examples directory to see more.

Chapter 4

Given When Then

[[Cucumber scenarios—Feature-Introduction]] consist of steps, also known as Givens, Whens and Thens.

Cucumber doesn't technically distinguish between these three kind of steps. However, we strongly recommend that you do! These words have been carefully selected for their purpose, and you should know what the purpose is to get into the BDD mindset.

Robert C. Martin has written a great post about BDD's Given-When-Then concept where he thinks of them as a finite state machine.

4.1 Given

The purpose of givens is to **put the system in a known state** before the user (or external system) starts interacting with the system (in the When steps). Avoid talking about user interaction in givens. If you had worked with usecases, 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 (An exception to the no-interaction recommendation. Things that "happened earlier" are ok).

And for all the Rails users out there – we recommend using a Given with a multiline table argument to set up records instead of fixtures. This way you can read the scenario and make sense out of it without having to look elsewhere (at the fixtures).

4.2 When

The purpose of When steps is to **describe the key action** the user performs (or, using Robert C. Martin's metaphor, the state transition).

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.

4.3 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).

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?)

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.

4.4 And, But

If you have several givens, whens or thens you can write

```
Scenario: Multiple Givens
  Given one thing
  Given another thing
  Given yet another thing
  When I open my eyes
  Then I see something
  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 open my eyes
  Then I see something
    But I don't see something else
```

To Cucumber steps beginning with And or But are exactly the same kind of steps as all the others.

Chapter 5

Step Definitions

Step definitions are defined in ruby files under `features/step_definitions/*_steps.rb`. Here is a simple example:

```
Given /^I have (\d+) cucumbers in my belly$/ do |cukes|
  # Some Ruby code here
end
```

A step definition is analogous to a method definition / function definition in any kind of OO/procedural programming language. Step definitions can take 0 or more arguments, identified by groups in the Regexp (and an equal number of arguments to the Proc).

Some people are uncomfortable with Regular Expressions. It's also possible to define Step Definitions using strings and $variables like this:

```
Given "I have $n cucumbers in my belly" do |cukes|
  # Some Ruby code here
end
```

In this case the String gets compiled to a Regular Expression behind the scenes: `/^I have (.*) cucumbers in my belly`

Then there are Steps. Steps are declared in your `features/*.feature` files. Here is an example:

```
Given I have 93 cucumbers in my belly
```

A step is analogous to a method or function invocation. In this example, you're "calling" the step definition above with one argument – the string "93". Cucumber matches the Step against the Step Definition's Regexp and takes all of the captures from that match and passes them to the Proc.

Step Definitions start with a preposition or an adverb (**Given**, **When**, **Then**, **And**, **But**), and can be expressed in any of Cucumber's supported [[Spoken languages]]. All Step definitions are loaded (and defined) before Cucumber starts to execute the plain text.

When Cucumber executes the plain text, it will for each step look for a registered Step Definition with a matching Regexp. If it finds one it will execute its Proc, passing all groups from the Regexp match as arguments to the Proc.

The preposition/adverb has **no** significance when Cucumber is registering or looking for Step Definitions.

Also check out [[Multiline Step Arguments]] for more info on how to pass entire tables or bigger strings to your step definitions.

5.1 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 what so ever.

5.2 Undefined steps

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`.

5.3 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`.

5.4 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.

5.5 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.

5.6 String steps

Steps can be defined using strings rather than regular expressions. Instead of writing

```
Given /^I have (.*) cucumbers in my belly$/ do |cukes|
```

You could write

```
Given ‘‘I have \$count cucumbers in my belly’’ do |cukes|
```

Note that a word preceded by a $ sign is taken to be a placeholder, and will be converted to match `.*`. The text matched by the wildcard becomes an argument to the block, and the word that appeared in the step definition is disregarded.

5.7 Ambiguous steps

Consider these step definitions:

```
Given /Three (.*) mice/ do |disability|
end

Given /Three blind (.*)/ do |animal|
end
```

And a plain text step:

```
\verb@Given Three blind mice@
```

Cucumber can't make a decision about what Step Definition to execute, and wil raise a `Cucumber::Ambiguous` error telling you to fix the ambiguity.

Guess mode

Running the plain text step will match the Regexp of both step definitions and raise `Cucumber::Ambiguous`. However, if you run Cucumber with `--guess`, it will guess that you were aiming for the step definition with 2 match groups.

There is ranking logic that gets invoked when the option is turned on:

1. The longest Regexp with 0 capture groups always wins.
2. The Regexp with the most capture groups wins (when there are none with 0 groups)
3. If there are 2+ Regexen with the same number of capture groups, the one with the shortest overall captured string length wins
4. If there are still 2+ options then an Ambiguous error is raised

So if you try `--guess` with the mice above, Cucumber will pick `/Three blind (.*)/`, because "mice" is shorter than "blind".

Consider guess mode a workaround. We still recommend you try to have unambiguous regular expressions. When you have a lot of step definitions you quickly lose track of the situations where cucumber will apply guessing logic, and that can lead to some surprises.

5.8 Redundant Step Definitions

In Cucumber you're not allowed to use a regexp more than once in a Step Definition (even across files, even with different code inside the Proc), so the following would cause a `Cucumber::Redundant` error:

```
Given /Three (.*) mice/ do |disability|
  # some code
end

Given /Three (.*) mice/ do |disability|
  # some other code
end
```