COMP5222 Group Project

Behavior-Driven Development

User Management Rails Application with Cucumber

Qu Xiaofeng

09903198R

# What Is Behavior-Driven Development

According to the Wikipedia, behavior-driven development (BDD) is a software development process developed on test-driven development (TDD) in software engineering field. The traditional techniques and principles of TDD and thoughts from domain-driven design and object-oriented analysis and design are combined together. It provides software developers and business analysts with common tools and a common process to cooperate in software development process.

BDD is principally based on the concept that software development should be performed by both business side and technical side. The practice of BDD relies on the use of customized software tools to support this development process. This development environment is not only specifically developed for use in BDD projects, but also it can be used as a specialized form of the tool-chain that supports test-driven development. The central theme of BDD is the tools who automate the testing from specifications written with ubiquitous language.

The most important issues include:

1. The Acceptance Test is not written only by coder, but by the client. (Features and Scenarios)

2. The Acceptance Test can be written in natural language, but not in programming languages.

3. The definition of the product is consistent with the code, testable and reliable through the whole life cycle of the product.

## Definition of BDD

Behavior-driven development was inspired by the problems encountered by developers practice test-driven development.

* Where to start the whole development process
* What should be tested and what should not to be tested
* How much tests should be include in one test iteration
* What to call the tests
* How to understand why a test fails

The main problem the BDD concerns is the mechanism of unit testing and acceptance testing to solve above problem.

The main results are:

* The unit test name should be sentences starting with the word “should”
* The unit test should be written in order of business value
* Acceptance tests should be written using the standard agile framework of a User story: "As a [role] I want [feature] so that [benefit]"
* Acceptance criteria should be written in terms of scenarios and implemented as classes: Given [initial context], when [event occurs], then [ensure some outcomes]

Based on above principles, the first BDD framework, JBehave, was developed by Dan North in 2003. Then it was ported to Ruby as a story-level BDD framework called RBehave which was integrated into the RSpec project. The story runner of RSpec was later replaced by Cucumber which is developed by Aslak Hellesøy. Cucumber was first developed to be merged into RSpec as a next generation story-runny module, but eventually it found its own reason to live.

## Agile Development, Philosophy behind BDD

Agile Development is “an approach to software development that attempts to reduce the cost of change, and so deliver software that meets business needs in a more predictable, timely manner”.

Though there are a lot of approaches and methods for agile development, they all share common features and are grounded in principles of the Agile Manifesto. This manifesto is hosted in the website: <http://agilemanifesto.org>.

* Individuals and interactions over processes and tools
* Working software over comprehensive documentation
* Customer collaboration over contract negotiation
* Responding to change over following a plan

Agile development practitioners believe that planning too far ahead is doomed to failure, because, however good we are, we are not prescient and we will make mistakes. They further believe that many Traditional Development processes are based on such predictions and that this is a fundamental reason why so many software projects fail.

Agile processes accept that mistakes will happen, that people will change their minds, misunderstand things and generally behave like human beings.

Instead of mitigating risk by trying to anticipate problems, agile projects are designed to avoid the increase in the cost of change completely. By accepting that change will happen and relying on several techniques to reduce the impact of late changes, agile processes alter the cost of change curve.

There are several techniques used on agile projects to effect this reduction in the cost of change. Instead of taking a qualitative approach to development, agile projects try as much as possible to take a quantitative approach.

Estimates for this week are based on what was accomplished last week.

Progress is measured by how much software is actually working.

The usefulness of the software produced is measured by users using it and reporting back if it meets, or fails to meet, their needs.

These are pragmatic, measurable things. Of course, there remain times when guesses must be made based on experience. Agile projects attempt to reduce the impact of getting those guesses wrong by not extrapolating too far ahead, typically only one or two weeks ahead.

Agile projects are characterized by the following:

* Iterative Development
* Small Teams
* Continuous Integration
* Minimal Documentation
* No Big Design Up Front
* Iterative Planning

## The concept of behavior-driven development

Test Driven Development and Acceptance Test Driven Planning is the ground of Behavior-Driven Development (BDD).

Test Driven Development and Domain Driven Design are combined together into an systematic development process.

The main focus of BDD is on the delivery of prioritized, verifiable business value. It provides a common vocabulary (or Ubiquitous Language) which narrows the gap between Business and Technology.

BDD framework of practice is based on three core schemes:

* Business and Technology should refer to the same system in the same way
* Any system should have an identified, verifiable value to the business
* Up-front analysis, design and planning all have a diminishing return

A specialized and small vocabulary is provided by BDD which minimizes misunderstanding and makes sure that all people in the team – Business, Developers, Testers, Analysts and Managers – are using the same language and words.

In the view of Domain Driven Design, BDD can be seen as a Ubiquitous Language for test driven software development.

“Getting the words right” was the original idea of the development of BDD, and it is the core of BDD. Getting the words right provides some insights and extrapolations that can help us to better understand our approach and to improve it.

## Steps and processes of BDD

The main tool of Rails-oriented BDD is RSpec. It is a testing tool for the Ruby programming language. The BDD is its main field. The object of it is to make Test-Driven Development a productive and enjoyable experience.

According to the [rspec.info](http://rspec.info/), the main features are

* A rich command line program (the rspec command)
* Textual descriptions of examples and groups (rspec-core)
* Flexible and customizable reporting
* Extensible expectation language (rspec-expectations)
* Built-in mocking/stubbing framework (rspec-mocks)

Actually Behavior-Driven Development is a improved version of test-driven development which focuses on behavioral specification.

Test-driven development is a software development methodology which asks developers to follow this guideline for each unit of software.

* Define a test set for the unit first
* Then implement the unit
* Finally verify that the implementation of the unit makes the tests succeed

BDD requires that the tests of any unit should be stated in terms of the desired behavior of the unit. The Desired behavior represents the business value of the unit under construction.

### Behavioral Specifications

The desired behavior is specified in a user story format in BDD. BDD specifies that Business and Developers should write the behavior user stories in co-operate. The user story should be like this:

Title: The story should have a clear, explicit title.

Narrative: A short, introductory section that specifies

**WHO** (which business or project role) is the driver or primary stakeholder of the story (the actor who derives business benefit from the story)

**WHICH** effect the stakeholder wants the story to have

**WHAT** business value the stakeholder will derive from this effect

Acceptance criteria or **Scenarios**

Scenarios: A description of each specific case of the narrative. Such a scenario has the following structure:

* Start by specifying the initial condition that is assumed to be true at the beginning of the scenario. This may consist of a single clause, or several.
* State which event triggers the start of the scenario.
* State the expected outcome, in one or more clauses.

BDD does not have any formal requirements for exactly how these user stories must be written down, but it does insist that each team using BDD come up with a simple, standardized format for writing down the user stories which includes the elements listed above.

Story: Returns go to stock

In order to keep track of stock

As a store owner

I want to add items back to stock when they're returned

Scenario 1: Refunded items should be returned to stock

Given a customer previously bought a black sweater from me

And I currently have three black sweaters left in stock

When he returns the sweater for a refund

Then I should have four black sweaters in stock

Scenario 2: Replaced items should be returned to stock

Given that a customer buys a blue garment

And I have two blue garments in stock

And three black garments in stock.

When he returns the garment for a replacement in black,

Then I should have three blue garments in stock

And two black garments in stock

The scenarios should be depicted declaratively rather than imperatively - in the business language, with no reference to elements of the UI through which the interactions take place.

This format is sometimes (incorrectly) referred to as the Gherkin language, which is very close to the syntax shown above. Gherkin is however specific to the Cucumber software tool.

### Specification as a ubiquitous language

Behavior-driven development borrows the concept of the ubiquitous language from domain driven design. A ubiquitous language is a (semi-)formal language that is shared by all members of a software development team — both software developers and non-technical personnel. The language in question is both used and developed by all team members as a common means of discussing the domain of the software in question. In this way BDD becomes a vehicle for communication between all the different roles in a software project.

BDD uses the specification of desired behavior as a ubiquitous language for the project team members. This is the reason that BDD insists on a semi-formal language for behavioral specification: some formality is a requirement for being a ubiquitous language. In addition, having such a ubiquitous language creates a domain model of specifications, so that specifications may be reasoned about formally. This model is also the basis for the different BDD-supporting software tools that are available.

The example given above establishes a user story for a software system under development. This user story identifies a stakeholder, a business effect and a business value. It also describes several scenarios, each with a precondition, trigger and expected outcome. Each of these parts is exactly identified by the more formal part of the language (the term Given might be considered a keyword, for example) and may therefore be processed in some way by a tool that understands the formal parts of the ubiquitous language.

### Story versus Specification

A separate subcategory of behavior-driven development is formed by tools that use specifications as an input language rather than user stories. An example of this style is the RSpec tool that was also developed by Dan North. Specification tools don't use user stories as an input format for test scenarios but rather use functional specifications for units that are being tested. These specification often have a more technical nature than user stories and are usually less convenient for communication with business personnel than are user stories. An example of a specification for a stack might look like this:

Specification: Stack

When a new stack is created

Then it is empty

When an element is added to the stack

Then that element is at the top of the stack

When a stack has N elements

And element E is on top of the stack

Then a pop operation returns E

And the new size of the stack is N-1

Such a specification may exactly specify the behavior of the component being tested, but is less meaningful to a business user. As a result, specification-based testing is seen in BDD practice as a complement to story-based testing and operates at a lower level. Specification testing is often seen as a replacement for free-format unit testing.

Specification testing tools like RSpec and JDave are somewhat different in nature than tools like JBehave. Since they are seen as alternatives to basic unit testing tools like JUnit, these tools tend to favor forgoing the separation of story and testing code and prefer embedding the specification directly in the test code instead. For example, an RSpec test for a hashtable might look like this:

describe Hash do

before do

@hash = Hash.new({:hello => 'world'})

end

it "should return a blank instance" do

Hash.new.should == {}

end

it "should hash the correct information in a key" do

@hash[:hello].should == 'world'

end

end

This example shows a specification in readable language embedded in executable code. In this case a choice of the tool is to formalize the specification language into the language of the test code by adding methods named it and should. Also there is the concept of a specification precondition – the before section establishes the preconditions that the specification is based on.

**BDD process**

The BDD process looks like this:

A SubjectMatterExpert (typically a business user) works with a BusinessAnalyst to identify a business requirement. This is expressed as a story using the following template:

* As a Role
* I request a Feature
* To gain a Benefit

The speaker, who holds the Role, is the person who will gain the Benefit from the requested Feature.

## BDD software in different languages

See references from here [http://behaviordrivendevelopment.wikispaces.com/MoreTools](http://behaviordrivendevelopment.wikispaces.com/MoreTools).

* ASSpec - ActionScript 3
* Aero - PHP 5
* Aubergine - .NET
* BDoc - Extracting documentation from unit tests, supporting behavior driven development
* BDD in Python - is core module doctest
* Bumblebee - Extract documentation from JUnit tests with support for adding text, code-snippets, screenshots and more. Puts focus on the end-user.
* beanSpec - Java
* Behat - PHP implementation of the Gherkin Domain-specific language
* Cedar - Objective C
* CppSpec - C++
* cfSpec - ColdFusion
* CSpec - C
* dSpec - Delphi
* Concordion - a Java automated testing tool for BDD that uses plain English to describe behaviors.
* Cucumber - Plain text + Ruby. Works against Java, .NET, Ruby, Flex or any web application via Watir or Selenium.
* easyb - Groovy/Java
* EasySpec - Groovy, usable in Java. Developer also working on Perception a tool for doing Context/Specification reporting for many different tools.
* EXTasy - Behavior-driven framework for ExtJS interfaces. Written in python.
* FitNesse - Java, .NET, C++, Delphi, Python, Ruby, Smalltalk, Perl. Now supports BDD directly with plain text tables and scenarios.
* Freshen - Python - clone of the Cucumber BDD framework
* GivWenZen - Java and FitNesse
* GivWenZen for Flex and ActionScript3 - Flex cousin of Java GivWenZen
* GSpec - Groovy
* Igloo - C++
* Instinct - Java
* Jasmine - JavaScript - framework-independent BDD with easy CI integration
* JavaStubs - Java - BDD framework supporting partial-mocking/method stubbing
* JBee - Java
* JBehave - Java - The first BDD framework, now at version 3.x
* JDave - Java
* JFXtras Test - JavaFX
* JSpec - JavaScript - BDD framework independent, async support, multiple reporters (terminal, dom, server, console, etc.), Rhino support, over 50 matchers and much more
* JSSpec - JavaScript
* Kiwi - RSpec like BDD library for iOS
* Lettuce - a Cucumber-like BDD tool for Python
* Morelia viridis - Cucumber clone for Python
* MSpec - .NET
* NBehave - .NET
* NSpec - .NET
* NUnit - A TDD framework in .NET which can be used for BDD examples and scenarios
* ObjectiveMatchy - iPhone - A Matcher System for iPhone development.
* Pyccuracy - Behavior-driven framework in Python.
* Pyhistorian - General purpose BDD Story Runner in Python (internal DSL, not plain-text)
* PyCukes - Cucumber-like BDD tool built on top of Pyhistorian
* Robot Framework - Generic keyword-driven test automation framework for acceptance level testing and acceptance test-driven development (ATDD) written in Python
* RSpec - Ruby
* Spock - Spock is a testing and specification framework for Java and Groovy
* SSpec - SSpec is the BDD framework for Smalltalk (multiple dialects) created by Dave Astels
* SpecFlow - SpecFlow is inspired by Cucumber and the community around it. Binding business requirements to .NET code
* screw-unit - JavaScript
* ScalaTest - Scala
* specs - Scala
* spec-cpp - C++
* Spectacular - Open source BDD and ATDD tool incorporating several types of tests in a single document and introduces Executable Use Cases
* Specter - Another implementation of BDD framework in .NET with focus on specification readability
* StoryQ - .NET 3.5, can be integrated with NUnit to provide both specification readability and testing
* TickSpec - Gherkin based framework supporting F# and C#
* tspec - Groovy/Java (Thai syntax)
* Tumbler - Java. Integrated with JUnit
* Twist - Commercial Eclipse-based tool for creating executable specifications
* Vows - JavaScript
* XSpec - XPath, XSLT and XQuery

# Cucumber - A Ruby Based BDD Software

Cucumber lets software development teams describe how software should behave in plain text. The text is written in a business-readable domain-specific language and serves as documentation, automated tests and development-aid - all rolled into one format.

Cucumber works with Ruby, Java, .NET, Flex or web applications written in any language. It has been translated to over 40 spoken languages.

Cucumber also supports more succinct tests in tables - similar to what FIT does. Dig around in the examples and documentation to learn more about Cucumber tables.

Cucumber is Aslak Hellesøy’s rewrite of RSpec’s “Story runner”, which was originally written by Dan North. (Which again was a rewrite of his first implementation - RBehave. RBehave was his Ruby port of JBehave). Early versions of the RSpec “Story Runner” required that stories be written in Ruby. Seeing how much this sucked David Chelimsky added plain text support with contributions from half a dozen other people.

Cucumber is a tool that executes plain-text functional descriptions as automated tests. The language that Cucumber understands is called Gherkin. Here is an example:

Feature: Search courses

In order to ensure better utilization of courses

Potential students should be able to search for courses

Scenario: Search by topic

Given there are 53 courses which do not have the topic "computer"

And there are 2 courses COMP425p, COMP5522, ENG2003 that each have "computer" as one of the topics

When I search for "biology"

Then I should see the following courses:

| Course code |

| COMP425p |

| COMP5522 |

| ENG2003 |

While Cucumber can be thought of as a “testing” tool, the intent of the tool is to support BDD. This means that the “tests” (plain text feature descriptions with scenarios) are typically written before anything else and verified by business analysts, domain experts, etc. non-technical stakeholders. The production code is then written outside-in, to make the stories pass.

Cucumber itself is written in Ruby, but it can be used to “test” code written in Ruby or other languages including but not limited to Java, C# and Python. Cucumber only requires minimal use of Ruby programming and Ruby is easy, so don’t be afraid even if the code you’re developing in is not Ruby.

## Specification based acceptance testing

The acceptance specification should be structured like this:

Application

has\_many Features

Feature

has\_a comment

has\_many Scenarios

Scenario

Given …

When …

Then …

When you decide you want to add a new feature or fix a bug, start by writing a new feature or scenario that describes how the feature should work. Don’t write any code (yet).

Now run the features again. The one you wrote should have yellow, pending steps – or failing, red ones. (If you don’t get that you’re doing something wrong, or the feature is already implemented).

This is when you start writing code. Start by writing a couple of lines of code to address the failure you got from Cucumber. Run cucumber again. Repeat and rinse until you’re happy with your feature. When you get down to nitty gritty details, drop down one abstraction level and use RSpec, or any Ruby testing framework, to write some specs/tests for your classes. Write the specs first! If you follow this process you have a good guard against brittle, unmaintainable, undocumented code that nobody understands. (Yes, features and specs are documentation too).

If you think this sounds annoying, try it out anyway. You’ll end up writing better, lesser coupled (and less) code this way. Trust me. Work outside-in (the outside being the feature, the inside being the low level code). Do it the BDD way.

### Business value and MMF

You should discuss the “In order to” part of the feature and pop the why stack max 5 times (ask why recursively) until you end up with one of the following business values:

* Protect revenue
* Increase revenue
* Manage cost
* Increase brand value
* Make the product remarkable
* Provide more value to your customers

If you’re about to implement a feature that doesn’t support one of those values, chances are you’re about to implement a non-valuable feature. Consider tossing it altogether or pushing it down in your backlog. Focus on implementing the MMFs (Minimum Marketable Features) that will yield the most value.

### Outcomes and bottom-up scenarios

The value provided by a system is what you can get out of it – not what you put into it (Chris Matts said that). Just like the value is expressed at the top of a feature (In order to…), the value should be in the steps of a scenarios too, more precisely in the Then steps.

When you’re writing a new scenario, I recommend you start with the formulation of the desired outcome. Write the Then steps first. Then write the When step to discover the action/operation and finally write the Given steps that need to be in place in order for the When/Then to make sense.

## Scenarios and Features

### Structure of scenarios

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) on a new line. You can use tagging 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.

### 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

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.

## Gherkin, a language describing the specification

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

Gherkin serves two purposes – documentation and automated tests. The third is a bonus feature – when it yells in red it’s talking to you, telling you what code you should write.

Gherkin’s grammar is defined in the Treetop grammar that is part of the Cucumber codebase. The grammar exists in different flavors for many spoken languages, 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.

### 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.

A Gherkin source file usually looks like this

1: Feature: Some terse yet descriptive text of what is desired

2: In order to realize a named business value

3: As an explicit system actor

4: I want to gain some beneficial outcome which furthers the goal

5:

6: Scenario: Some determinable business situation

7: Given some precondition

8: And some other precondition

9: When some action by the actor

10: And some other action

11: And yet another action

12: Then some testable outcome is achieved

13: And something else we can check happens too

14:

15: Scenario: A different situation

16: ...

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

**Given When Then**

Cucumber scenarios 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.

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).

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.

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.

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.

**Step definition**

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.

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.

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.

Pending steps

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

Failed steps

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

Skipped steps

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

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.

Ambiguous steps

Consider these step definitions:

Given /Three (.\*) mice/ do |disability|

end

Given /Three blind (.\*)/ do |animal|

end

And a plain text step:

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:

The longest Regexp with 0 capture groups always wins.

The Regexp with the most capture groups wins (when there are none with 0 groups)

If there are 2+ Regexen with the same number of capture groups, the one with the shortest overall captured string length wins

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.

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

**Scenario Outline**

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

Scenario: eat 5 out of 12

Given there are 12 cucumbers

When I eat 5 cucumbers

Then I should have 7 cucumbers

Scenario: eat 5 out of 20

Given there are 20 cucumbers

When I eat 5 cucumbers

Then I should have 15 cucumbers

Scenario outlines allow us to more concisely express these examples through the use of a template with placeholders, using Scenario Outline, Examples with tables and < > delimited parameters:

Scenario Outline: eating

Given there are <start> cucumbers

When I eat <eat> cucumbers

Then I should have <left> cucumbers

Examples:

| start | eat | left |

| 12 | 5 | 7 |

| 20 | 5 | 15 |

The Scenario Outline steps provide a template which is never directly run. A Scenario Outline is run once for each row in the Examples section beneath it (not counting the first row).

The way this works is via placeholders. Placeholders must be contained within < > in the Scenario Outline's steps. For example:

Given <I'm a placeholder and I'm ok>

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

You can also use placeholders in Multiline Step Arguments.

IMPORTANT: Your step definitions will never have to match a placeholder. They will need to match the values that will replace the placeholder

So when running the first row of our example:

Examples:

| start | eat | left |

| 12 | 5 | 7 |

The scenario that is actually run is:

Scenario: controlling order

Given there are 12 cucumbers # <start> replaced with 12

When I eat 5 cucumbers # <eat> replaced with 5

Then I should have 7 cucumbers # <left> replaced with 7

## Organizing and tagging of features

# An Example Project Using Cucumber

User Management is a fundamental function for any website.

## Requirements

## The first prototype, image uploading and showing

## The second iteration, core image processing

## The third iteration, security and style issue

## The fourth iteration, refactoring

# Defects And Pitfalls of BDD

## How to write great features

## How to organize features

## BDD using different languages