CSCI 3308
Fall 2013
Lab October 23, 2013

Lab 3: Tools for Behavior Driven Design

GOALS:

After doing this lab you will be able to:

1. Create a project in Pivotal Tracker and understand how to use it to track team progress.
2. Create a Lo-Fi User Interface sketch.
3. You will also know how to set up and use cucumber test-case generation tool to create test cases for Ruby on Rails projects.

PREPARATION FOR LAB (Complete before arriving at Lab)

1. Create an account on http://www.pivotaltracker.com/ (a 60 day free trial will work for this course)
2. Once the email arrives complete the registration process.
3. Create a new project when prompted calling it <your last name>-Lab3.
4. Watch the video about Pivotal Tracker when prompted.

HOW TO PROCEED:

1) Creating Lo-Fi UI Sketch

a) Create a Lo-Fi Sketch similar to the ones that appear on Pages 228-229 of ESaaS (at 52% of kindle version) that you could show to the customer who said they want to add an option to add user ratings of the movie where the rating is from 1-5 rotten potatoes and 5 is highest rating.
b) Take a picture of your sketch and upload it to create a submittable file and submit to Lab3 moodle assignment.

2) Pivotal Tracker

a) Log into Pivotal Tracker and open your Lab3 project.
b) In the textbox available for submission of Lab 3, tell what your initial velocity is and what this number means.
c) Using the user stories provided at the end of this lab, follow workflow from http://www.pivotaltracker.com/features#workflow to add user stories to Icebox, assign story points, add your Lo-Fi UI sketch to the appropriate user story, label, and prioritize (do steps from "First Step" to "Prioritize User Stories"
d) In the submission textbox describe what happens and why it does, when you drag your first user stories from the Icebox to the Backlog.
e) Take a screen shot of your Pivotal Tracker Window that shows the uploaded image in the user story and upload to moodle as part of your submission for Lab 3. If the expanded user story is too big to fit on the screen then you can hoover your mouse over the attachment icon to the left of the user story name and it should appear so that you can take the screenshot.

3) Using Cucumber

You can read a brief overview of cucumber here: http://www.rubyinside.com/cucumber-the-latest-in-ruby-testing-1342.html

The Following Tutorial is a slightly adapted version of Bastion Krol's online tutorial "Cucumber: An Introduction for Non-Rubyists – Setup and the Basics". When you have finished with it you will have a good idea of how cucumber functions, what step-definitions are, and how they are written.

This tutorial takes you through the installation and configuration of Cucumber, shows you how to write your first Cucumber feature and how to use Cucumber and Capybara to write beautiful acceptance tests for a web application. The examples should be quite readable.

Setting up Ruby, Cucumber and PhantomJS

Clone the example project (which contains the Gemfile you will need during the setup) into your rails_projects directory by doing

```
git clone -b 00_setup https://github.com/basti1302/audiobook-collection-manager-acceptance.git
```

Then, run bundle install

When you have completed the setup, typing `cucumber` in the top level directory in the example project should give you the following output:

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

Note: *Since we are using bundler, we should actually type `bundle exec cucumber` instead of simply typing `cucumber`. This ensures that the cucumber command is executed in the context of the bundle defined in the Gemfile. As long as you do not have multiple versions of ruby gems installed on your system, there should be no difference. But if Cucumber behaves unexpectedly, try your luck with `bundle exec cucumber` – or train yourself to always use `bundle exec cucumber` right from the start, because it is a good habit anyway.*

Cucumber Basics

Features and Scenarios

The top most artifacts of Cucumber are called features. They are defined in a DSL called Gherkin that more or less resembles natural language. (How well the features resemble natural language of course depends on the author of the feature.) Feature files have the suffix `.feature` and live in a directory named `features`. When the Cucumber executable is started without arguments, it always looks for a subdirectory of the current working directory that has this name – so it's a good idea to start Cucumber in the parent directory of `features`, otherwise it will complain.

A feature can contain a description (which is unparsed text) and one or more scenarios (think scenario ~ test). A scenario describes one behavioral aspect of the system under test. It makes assertions that – given that certain preconditions hold – the system behaves or responds in an expected way when a specified action is executed.

A scenario is comprised of steps. Each step is a short sentence that falls into one of the three familiar BDD categories:

Given (setting up a precondition for the scenario),
When (execute an action in the application under test) and
Then (assert the desired outcome)

On a technical level, it does not matter if you prefix a step with Given, When, Then or And (which can also be used for all three categories). You could set up preconditions in a Then step and make assertions in a Given step – but using the conventions correctly makes scenarios much more readable.

Further down the chain, there needs to be a *step definition* for each step. These live in the directory `features/step_definitions`. We'll take a closer look at them later.

Your First Feature

Put the following code into a file named `features/first.feature`.

```
#encoding: utf-8
Feature: Showcase the simplest possible Cucumber scenario
  In order to verify that cucumber is installed and configured correctly
  As an aspiring BDD fanatic
  I should be able to run this scenario and see that the steps pass (green like a cuke)

  Scenario: Cutting vegetables
    Given a cucumber that is 30 cm long
    When I cut it in halves
    Then I have two cucumbers
    And both are 15 cm long
```

Now run the feature. Go to the root directory of the project (the directory directly above `features`) and type `cucumber` (or better: `bundle exec cucumber`). The output should be something like this:

```
#encoding: utf-8
Feature: Showcase the simplest possible cucumber scenario
  In order to verify that cucumber is installed and configured
correctly
  As an aspiring BDD fanatic
  I should be able to run this scenario and see that the steps pass
(green like a cuke)

  Scenario: Cutting vegetables          # features/first.feature:8
    Given a cucumber that is 30 cm long # features/first.feature:9
    When I cut it in halves             # features/first.feature:10
    Then I have two cucumbers           # features/first.feature:11
    And both are 15 cm long             # features/first.feature:12

1 scenario (1 undefined)
4 steps (4 undefined)
0m0.003s

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

Given(/^a cucumber that is (\d+) cm long$/) do |arg1|
  pending # express the regexp above with the code you wish you had
end

When(/^I cut it in halves$/) do
  pending # express the regexp above with the code you wish you had
end
```

```
Then(/^I have two cucumbers$/) do
  pending # express the regexp above with the code you wish you had
end

Then(/^both are (\d+) cm long$/) do |arg1|
  pending # express the regexp above with the code you wish you had
end

If you want snippets in a different programming language,
just make sure a file with the appropriate file extension

exists where cucumber looks for step definitions.
```

After echoing the feature the result is listed. It complains that we did not yet define the steps that we have used in this feature. Quite true. But Cucumber goes a step further – it even says how we can solve this and implement the steps (as stubs). How exceptionally nice and polite. It also guesses (correctly) that the numbers used in the steps might vary and replaces the literal numbers with capturing groups. So lets just paste Cucumber's suggestions into a step definition file. (In fact I often run Cucumber intentionally with undefined steps and use the suggestions as a template for the step implementations.)

Matching Steps With Step Definitions
Create the directory `features/step_definitions if it doesn't already exist` and in this directory the file `first_steps.rb`. Step definitions are written in ruby when you are writing ruby code.

The following code will go into `features/step_definitions/first_steps.rb` (the name does not matter, all `.rb` files in `features/step_definitions` will be loaded when running the features).

```ruby
#encoding: utf-8
Given /^a cucumber that is (\d+) cm long$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end

When /^I cut it in havles$/ do
  pending # express the regexp above with the code you wish you had
end

Then /^I have two Cucumbers$/ do
  pending # express the regexp above with the code you wish you had
end

Then /^both are (\d+) cm long$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end
```

Now just run Cucumber again (remember, from the parent directory of features). The output only looks a little bit better:

```
#encoding: utf-8
Feature: Showcase the simplest possible cucumber scenario
  In order to verify that cucumber is installed and configured
correctly
  As an aspiring BDD fanatic
  I should be able to run this scenario and see that the steps pass
(green like a cuke)

  Scenario: Cutting vegetables          # features/first.feature:8
    Given a cucumber that is 30 cm long #
features/step_definitions/first_steps.rb:1
      TODO (Cucumber::Pending)
      ./features/step_definitions/first_steps.rb:2:in `/^a cucumber
that is (\d+) cm long$/'
      features/first.feature:9:in `Given a cucumber that is 30 cm long'
    When I cut it in halves             #
features/step_definitions/first_steps.rb:5
    Then I have two cucumbers           #
features/step_definitions/first_steps.rb:9
    And both are 15 cm long             #
features/step_definitions/first_steps.rb:13

1 scenario (1 pending)
4 steps (3 skipped, 1 pending)
0m0.005s
```

The summary tells us that at least one step is pending and so the whole scenario is marked as pending. A scenario is skipped entirely when the first step fails (or is pending), so Cucumber does not even try to execute the other pending steps – that's why it reports 3 steps as skipped.

Before we implement the steps, let's take a close look on `first_steps.rb`. If you have never seen a Cucumber step file before, it might look a little weird. The individual step implementations look a little bit like methods – in fact, they are ruby methods and the code inside is just plain ruby code. But the method header looks a bit queer – it contains a regular expression. That is a very neat feature: When Cucumber needs to find the step definition for a step in a scenario, it checks all step definition files (all `.rb` files in `features/step_definitions`, including subdirectories). If it finds one with a matching regular expression, this implementation is called. (By the way, if it finds more than one match, it complains about an ambiguous match, instead of executing the first or an arbitrary step.)

This mechanism has several advantages. The first is flexible and well-readable parameterization of step definitions. We have already seen that in

```
Given /^a cucumber that is (\d+) cm long$/ do |arg1|
```

Here the match of the first capturing group (enclosed in parentheses) is passed into the step implementation as the first argument (called `arg1` – we should probably rename it to `length`). The `\d+` matches one or more digits, so it is clear that we need to pass a number here.

The regular expression matching also gives you the ability to write scenarios that are nice to read, without duplication in the step file. Here's a simple example on how to exploit this: We could rewrite the second step as

```
When /^I (?:cut|chop) (?:it|the cucumber) in (?:halves|half|two)$/ do
  pending # express the regexp above with the code you wish you had
end
```

With this change all of the following steps would match:

```
When I cut the cucumber in halves
When I chop the cucumber in half
When I cut it in two
```

We have used non-capturing groups (starting with "`(?:`") to cater for expressive variety. This is quite common in Cucumber steps.

Once you start using this pattern, you should be careful to not take it too far. Only provide for the step variations that you really need right now. In the end you do not want to spend more time fighting with regular expressions than testing your system. It might even make sense to write two step definitions if it is too difficult to put all variations into one regex. You also have two other options at your disposal to reduce duplication: First, you can put plain ruby methods into the step file and call them from your steps or you can `require` other ruby files and reuse their code. Finally, you can even call other steps from within a step.

Implementing Steps

Now for what's inside a step implementation: Currently, all our steps only contain a call to `pending`. This is like a todo marker. With `pending`, you can first write you scenario, stub all steps with `pending` and then implement the steps one after another until the scenario passes. This fits nicely into an outside-in ATDD approach where you drive the implementation of the system under test with acceptance tests:

1. Write a Cucumber feature first (with pending steps),
2. implement the first step in Cucumber,
3. implement the required funcionality in the system under test to make just this step pass,
4. repeat.

All this said, let's put some real implementation into the step methods.

```
Given /^a cucumber that is (\d+) cm long$/ do |length|
  @cucumber = {:color => 'green', :length => length.to_i}
end

When /^I (?:cut|chop) (?:it|the cucumber) in (?:halves|half|two)$/ do
  @choppedCucumbers = [
    {:color => @cucumber[:color], :length => @cucumber[:length] / 2},
    {:color => @cucumber[:color], :length => @cucumber[:length] / 2}
  ]
end

Then /^I have two cucumbers$/ do
  @choppedCucumbers.length.should == 2
end
```

```
Then /^both are (\d+) cm long$/ do |length|
  @choppedCucumbers.each do |cuke|
    cuke[:length].should == length.to_i
  end

end
```

If you run Cucumber again, you should have the following output:

```
#encoding: utf-8
Feature: Showcase the simplest possible cucumber scenario
  In order to verify that cucumber is installed and configured
correctly
  As an aspiring BDD fanatic
  I should be able to run this scenario and see that the steps pass
(green like a cuke)

  Scenario: Cutting vegetables          # features/first.feature:8
    Given a cucumber that is 30 cm long #
features/step_definitions/first_steps.rb:3
    When I cut it in halves             #
features/step_definitions/first_steps.rb:7
    Then I have two cucumbers           #
features/step_definitions/first_steps.rb:14
    And both are 15 cm long             #
features/step_definitions/first_steps.rb:18

1 scenario (1 passed)
4 steps (4 passed)
0m0.005s
```

Yay! Now all steps are printed in green - like a cucumber (that's why the tool has been named after the fruit).

Let's review the implementation: In ruby, an identifier that begins with `@` is an instance variable. You do not need to declare them up front, if you assign a value to it at runtime, the instance variable is defined on the fly. That's why `@cucumber` is visible in all steps and can be used to keep some state across step boundaries. The expressions enclosed in curly braces are hashes (aka dictionaries or associative arrays, think java.util.Map if you're coming from the JVM) and the identifiers beginning with a colon are the keys (the colon makes them a symbol, but you don't need to worry about that for now).

The expressions `@choppedCucumbers.length.should == 2` and `cuke[:length].should == length.to_i` might require a bit of explanation. What we are using here is the RSpec module Spec::Expectations. In `features/support/env.rb` we have the line `require 'rspec/expectations'`, thus we have the module at our disposal in all our step files. This module gives us the ability to express expectations on any object by adding methods to *all* objects (by monkey patching). We are using object expectations here, which provides methods like `should ==` and `should_not ==` (and some more) which are nothing more than syntactic sugar for saying that you expect one value to equal another value. Therefore, because this methods have been added to all objects and also because in Ruby really everything is an object - even a numeric value like the length of an array - we can call the method `should ==` on the length of the array `@choppedCucumbers`.

The step definitions are quite silly (actually, the whole scenario is silly) and in this example we did not even test any production code - all the code is in the step file. But some of the basic concepts behind Cucumber should have become clear now.

Upload your first.feature and first.steps files to moodle.

4) Capybara

Look up information about Capybara and write a few lines into the text box in moodle submission.

5) Tryout Writing a Step Definition for for Rotten Potatoes

Write a step definitions that will handle the following feature:

Given I am on the RottenPotatoes homepage, when I follow Add New Movie,

Then I should be on the Create Movie page.

When I fill in Title with Men in Black, And I select PG-13 from Rating,

and I press Save Changes, Then I should be in the RottenPotatoes home

page, And I should see Men in Black.

HINT: This example is done by Dave in Lecture 8 (V3) on EdX. You can complete this part of the lab by going through that example with him.

Submit your step definitions in the text box on the moodle submission page.

SUBMISSION

You should have already uploaded the

- picture of your Lo-Fi sketch
- your screenshot of pivotal tracker
- your first.feature file
- your first.steps file
- your text describing what Pivotal Tracker did when you dragged your user story out of the Icebox from Part 2
- your text describing what capybara does for you from Part 4
- your text describing the step definitions from Part 5

GRADING

Total possible points for this lab is 100 up to # of points for each of the following:

- 15 pts for Lo-Fi Sketch submitted that is reasonable sketch of what the window could look like and is legible enough to use as a basis for discussion with stakeholder.
- 5 pts for descriptions about velocity and of what happens when you drag user story out of Icebox.
- 20 pts for screenshot of Pivotal Tracker that shows final status of the user stories and the details for the user story with the Lo-Fi sketch attached.
- 30 pts for completing Cucumber tutorial
- 15 pts for Capybara description
- 25 pts for Step Descriptions

USER STORIES:

US1
As a user
so that I can see information about the movie
I want to be able to click on a movieand have a link to wikipedia page.

US2
As a RottenPotatoes User
So that I can preview the movie's plot
I want to see a trailer of the movie.

US3
As a user, in order to find the movies I am searching for, I would like to be able to sort movies by title in both alphabetical and reverse alphabetical orders.

US4
As a moviegoer with children of multiple ages, so that I can view movies by age of appropriate viewer, I want to add a feature that allows me to sort movies by age appropriateness.