More on Cucumber: Steps, Scenarios, and Troubleshooting

CSCI 5828: Foundations of Software Engineering
Lecture 13 — 02/28/2012
Goals

• Review material from chapters 4-6 of our testing textbook
• Learn more about Cucumber and how it supports behavior-driven design
  • More about Steps and Step Definitions
  • More about scenarios
  • Examining typical problems encountered with Cucumber
• Review additional examples throughout
Perspective (I)

- Cucumber is aimed at **integration and acceptance** testing
  - It is a testing and communication tool for expressing
    - **end-to-end** tests that cover the major capabilities of your system
      - view to controller to model and back
      - UI to database and back
      - tests that touch all of the subsystems of your system; these subsystems might exist on multiple nodes
  - As such, you will still write unit tests for your system, using some other testing framework, and run them alongside cucumber-based tests
    - Your customer will only be involved with the latter
Perspective (II)

• To make this work, cucumber requires discipline to ensure that the right people work together to create the integration tests
  • customers must work with developers
    • to prevent the developers from writing tests that are too low level
    • to ensure that tests are written using customer terminology
    • to ensure that what is being tested is important (to the customer)
  • developers must work with testers
    • testers will be looking for corner cases and good coverage
    • developers can use their expertise to ensure that the test cases are properly decomposed and help with refactoring duplicate behavior
Perspective (III)

- All three stakeholder roles are needed to balance each other
  - customers ensure that tests are “in scope” and important
  - developers ensure that tests are well maintained
    - also help to ensure that all the information needed to run a test is present
  - testers ensure that the test set is comprehensive
    - and that we are not ignoring certain tests because they are hard
    - “The first principle is that you must not fool yourself and you are the easiest person to fool” — Richard Feynman
• From Lecture 9

  • A Cucumber test suite involves
    • Features that consist of scenarios
    • Scenarios consist of steps
      • Each step must have a step definition
      • enables leap from spec to code
Review (and two new tidbits)

- Cucumber conventions lead to the above folder/file hierarchy
  - There can be **multiple directories** under “features” to help organization
  - There can be **multiple files** in each step_definitions directory to help organize the code that implements the step definitions
Matching Steps (I)

• In lecture 9, we encountered steps that looked like this

  • **Scenario**: Attempt withdrawal using stolen card

    • **Given** I have $100 in my account
    • **But** my card is invalid
    • **When** I request $50
    • **Then** my card should not be returned
    • **And** I should be told to contact the bank

  • It turns out the step keywords (**Given**, **When**, **Then**, **And**, **But**) are for humans only; cucumber **doesn’t care what you use**
Matching Steps (II)

• Indeed, this is an equivalent scenario

• **Scenario**: Attempt withdrawal using stolen card
  
  * I have $100 in my account
  * my card is invalid
  * I request $50
  * my card should not be returned
  * I should be told to contact the bank

• The “*” is simply used to denote a new step
Matching Steps (III)

• The reason this is equivalent is that cucumber uses just the highlighted text...

• **Scenario**: Attempt withdrawal using stolen card

  * I have $100 in my account
  * my card is invalid
  * I request $50
  * my card should not be returned
  * I should be told to contact the bank

• ... to match a step to its step definition

• All step definitions are read in at run-time and then regular expressions are used to find a match
Matching Steps (IV)

• As a result, it does not matter how you organize your step definitions

• Here is a version of calculator that splits its step defs across three files

The behavior of cucumber is identical to the previous config.

Three files with one step def. each

2 directories, 5 files
Matching Steps (V)

• This also means that you have to be careful with how you write your steps

• Scenario 1:
  • Given I have $100 in my Account
  • When I request $20
  • Then $20 should be dispensed
  • And my balance is $80

• Scenario 2:
  • Given a starting balance of $20 in my Account
  • When I deposit $80
  • Then I have $100 in my Account

What’s the problem?
Matching Steps (V)

• This also means that you have to be careful with how you write your steps.

• Scenario 1:
  
  • Given I have $100 in my Account
  
  • When I request $20
  
  • Then $20 should be dispensed
  
  • And my balance is $80

• Scenario 2:
  
  • Given a starting balance of $20 in my Account
  
  • When I deposit $80
  
  • Then I have $100 in my Account

These two statements are treated as equivalent by Cucumber; but in one case, it is being used to initialize a scenario; in the second case, it is being used to assert that something is true of the scenario.
Matching Steps (VI)

- When you have two equivalent steps
  - Given I have $100 in my Account *(First Step; used to initialize)*
  - Then I have $100 in my Account *(Last Step; used to assert)*
- they will both cause the same step definition to be invoked
  - Given /I have \$100 in my Account/ do
    - <code goes here>
    - end
- The problem is that <code goes here> will do the same thing each time, it will not be able to customize what it does based on the two different contexts
- How to fix?
Matching Steps (VII)

• To fix this problem, you need to rewrite the steps

  • Given I have deposited $100 in my Account

  • Then the balance of my Account should be $100

• Now, it will be clear that

  • the first is used to initialize the Account

  • and, the latter is used to verify the account’s balance
The Matching Process

• When invoked, Cucumber reads in all of the step definitions that it can find
  • Each step definition causes a pattern to be registered with Cucumber

• It then starts to process each feature file, looking for scenarios
  • For each step in a scenario, it checks to see that it matches one of the registered regular expressions
  • If a match is found, the code associated with the step definition is executed and a result is recorded
    • If a match is not found, the step and scenario is considered undefined
  • The next step is then processed (assuming the previous step passed)
    • Otherwise, the scenario either failed due to an exception in the step or the scenario is marked pending because the step itself was marked pending
Regular Expressions (I)

• Regular expressions are a mechanism for specifying patterns that can appear inside text documents

  • Each expression can consist of

    • regular characters
    • metacharacters
    • groups
    • anchors
Regular Expressions (II)

- A regular expression (in ruby) begins with a slash (/) and ends with a slash
  - /Ken/

- All regular expressions start and end with the “/” metacharacter. Metacharacters have special meaning; in this case, slash acts as a delimiter

- The above regular expression consists of three regular characters “K”, “e”, and “n”, in that order
  - It would match these sentences
    - Ken is a faculty member
    - Pete called Ken on Tuesday
  - but not this sentence
    - Dirk is a faculty member

  Here the regular expression is “unanchored” and so it will match any sentence that contains the string “Ken”
Regular Expressions (III)

• If you want a metacharacter to act like a regular character, you must escape the metacharacter using a backslash

  • /Ken\VPete/

• This expression would match the sentence

  • The meeting will be led by a faculty member (Ken/Pete)

• But not

  • Ken will lead the meeting

• Since backslash is a metacharacter, if you want to match it, you need to escape it with, you guessed it, another backslash character

  • /In LaTeX, use \cite to reference a journal or conference paper\./
Regular Expressions (IV)

• A period (.) is a metacharacter that will match any character in the text file
  • /I ate .... slices of pizza\./

• will match
  • I ate four slices of pizza.
  • I ate five slices of pizza.
  • I ate nine slices of pizza.

• Note: since we wanted to make sure that a period appeared at the end of a matched sentence, we explicitly matched the period by escaping the last period in the regular expression with a backslash
  • Otherwise, the expression would match “I ate nine slices of pizza!”
Regular Expressions (V)

• If you need to specify that any one of a particular set of characters might appear in a particular spot in a regular expression, you use a character class

• A character class is specified using square brackets and then can list one or more ranges of characters assuming ASCII ordering

  • /There are [23456789] cows in the field\./

• Matches “There are 3 cows in the field.” but not “There are cows in the field.”

• If characters appear in sequence, you can use a hyphen to express a range

  • /Your id number is [A-Z][A-Z][0-9][0-9][0-9]\./

  • “Your id number is BZ232.”
Regular Expressions (VI)

- Beware unintended inclusions of characters when using the hyphen
  - `[a-Z]` is an empty range and `[A-z]` includes “[”, “\”, “]”, “^”, “–”, and “” (!!!)
  - Instead, you need to do `[a-zA-Z]`
Regular Expressions (VII)

• More on character classes
  • If you want to match all characters BUT the ones listed, start the class with the “^” character
    • [^0-9] == match any character that is not a digit
  • If you need to match a hyphen, list it first
    • [-A-Za-z] == match any letter (upper case or lower case) or a hyphen
  • If you need to match a “^” character, list it in any position but the first
    • [A-Z^a-z]
• Some character classes are predefined: \s (whitespace), \d (digit), etc.
  • See page 49 of the testing textbook for examples (not required)
Regular Expressions (VIII)

• Patterns can be tagged with repetition modifiers
  • * — the preceding pattern can appear zero or more times
  • + — the preceding pattern can appear one or more times
  • ? — the preceding pattern can appear zero or one times

• Alternative choices for a pattern can be separated by the pipe character “|”
  • Parenols “(“ and “)” can be used to group patterns for alternation

• /There (is|are) [0-9]+ cows? in the field\./

• Matches
  • “There are 2 cows in the field.” and “There is 1 cow in the field.” but also
    “There is 5 cow in the field.” and “There are 9999999993421 cows in the field.”
Regular Expressions (IX)

• Patterns can be anchored
  
  • \(^\) at the beginning of a regular expression anchors it at the beginning of a line of text
  
  • \(\) at the end of a regular expression anchors it at the end of a line of text
  
  • /\(^Ken\slikes\sto\splay\ssoccer\).\$/
    
    • This regular expression matches only the string “Ken likes to play soccer.” and nothing else.
Regular Expressions (X)

• Parens are also used to specify “capture groups”
  • That is they “capture” what was matched inside of them and make the captured pattern available for later processing
  • /There are ([1-9][0-9]*) cows in the field./

• The above expression matches sentences like
  • “There are 10 cows in the field.” or “There are 19920 cows in the field.”
  • Also (unfortunately) “There are 1 cows in the field.”

• AND makes the actual number available
  • In a step definition, a captured pattern is passed as an argument to the step definition’s method body;
Example from Lecture 9

- Given `/^the input "([^\"]*)"$/ do |arg1|
  - @input = arg1
- end

- We now should understand the regular expression better
  - “the input ” appears at the start of the step, followed by a quotation mark
  - `[^"]*` match any character that is not a quotation mark, zero or more times
  - the parens around the above pattern captures the result as arg1
  - the step must end with a quotation mark
More about Steps (I)

• Any step can be augmented by a data table

• These are not the same as the table that appeared when using a “Scenario Outline” within a feature

• Instead, it is a table that appears immediately after a step, like this

<table>
<thead>
<tr>
<th>Onions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Potatoes</td>
</tr>
<tr>
<td>Sausages</td>
</tr>
<tr>
<td>Apples</td>
</tr>
<tr>
<td>Relish</td>
</tr>
</tbody>
</table>

Then my shopping list should contain:

• The step definition will then contain an argument where this table is passed as a first-class object
More about Steps (II)

• The details of what you can do with the table is provided in the cucumber documentation
  • <http://cukes.info/cucumber/api/ruby/latest/Cucumber/Ast/Table.html>
• The book provides a basic example using Tic-Tac-Toe
  • DEMO
• It also hints at what can be accomplished
  • Given these Users:
    • | name               | date of birth |
    | Michael Jackson     | August 29, 1958 |
    | Elvis               | January 8, 1935 |
    | John Lennon         | October 9, 1940 |
• A step definition could process this table at run-time and create 3 instance of the User class configured as shown and stored in a collection @users
More about Steps (III)

• Any step can also be augmented with a doc string
  • Then I should receive an email containing:
    • ""
      Dear Sir,
      You are no longer subscribed to our mailing list.
      Sincerely,
      SpamIsUs
    ""
  • The entire contents of the doc string will be passed to the step definition
    • Your code can then store the string or manipulate/parse it using any of ruby’s string manipulation capabilities
    • We’ll see examples later this semester
Nesting Steps

• You can have a step definition that turns around and invokes other step definitions

  • This is called “nested steps”

• This is touted initially as a way to create more abstract steps

  • A step that says “Given the account is activated for Ken” might delegate to

    • “Given the account is created”

    • “Given the account has a balance of $50”

    • “Given the account has an owner named Ken”

    • “Then Ken activates the account”

• But, the book ends up strongly warning you away from this feature
More on Scenario Outlines

• A scenario outline can have more than one table of examples

• Scenario Outline: Withdraw fixed amount
  • Given I have <Balance> in my account
  • When I choose to withdraw the fixed amount of <Withdrawal>
  • Then I should <Outcome>
  • And the balance of my account should be <Remaining>

• Examples: Successful withdrawal
  
<table>
<thead>
<tr>
<th>Balance</th>
<th>Withdrawal</th>
<th>Outcome</th>
<th>Remaining</th>
</tr>
</thead>
<tbody>
<tr>
<td>$500</td>
<td>$50</td>
<td>receive $50 cash</td>
<td>$450</td>
</tr>
<tr>
<td>$500</td>
<td>$100</td>
<td>receive $100 cash</td>
<td>$400</td>
</tr>
</tbody>
</table>

• Examples: Attempt to withdraw too much
  
<table>
<thead>
<tr>
<th>Balance</th>
<th>Withdrawal</th>
<th>Outcome</th>
<th>Remaining</th>
</tr>
</thead>
<tbody>
<tr>
<td>$100</td>
<td>$200</td>
<td>see an error message</td>
<td>$100</td>
</tr>
<tr>
<td>$0</td>
<td>$50</td>
<td>see an error message</td>
<td>$0</td>
</tr>
</tbody>
</table>
Staying Organized

• When creating features and scenarios, cucumber offers two mechanisms to help organize them
  • First, as already mentioned, you can have as many subdirectories under the features directory of a cucumber project as you want
    • features/
      • sorting/
      • adding_employees/
      • calculating_payroll/
  • Second, you can classify both features and scenarios with tags
Tags (I)

• A tag is a word prefixed by the @ character that can appear on the line before either the keyword Feature or the keyword Scenario
  
  • A tag on Feature will be inherited by all of that feature’s scenarios

• Example
  
  • @employee

  • Scenario: Add an employee
    
    • Given Ken is a Person
    
    • And Ken is accepted for a job at our company
    
    • Then Ken is added as an employee

• This scenario is now tagged with the keyword “employee”
Tags (II)

• You can have multiple tags, separated by spaces

• Example

  • @slow @widgets @nightly

  • Scenario: Generate overnight report

    • Given I am logged in

    • And there is a report "Total widget sales history"

    • ...

• This scenario has three tags: slow, widgets, and nightly
Tags (III)

• Now that you have tagged scenarios, they become useful because you can ask cucumber to run just the scenarios with a particular tag
  
  • cucumber --tags @nightly,@slow

• Cucumber will now run only those features and scenarios that have been tagged with the “nightly” tag
  
  • This enables you to raise the quality of your development process by configuring a continuous build system to invoke this command at night and log the output for review the next day
Troubleshooting Cucumber

• The authors of the cucumber book identify
  • four types of problems that can be encountered
  • when trying to incorporate behavior-driven design into a life cycle

• Those problems are
  • **Flickering Scenarios**: Tests are unstable; some randomly fail
  • **Brittle features**: Changes to the system cause existing features to break
  • **Slow features**: The test suite takes too long to run
  • **Bored stakeholders**: Our customer is no longer creating/reading features
Flickering Scenarios

• A flickering scenario is one that every now and then fails randomly
  • The unpredictable nature of the failure reduces team confidence
  • This uncertain situation, in turn, reduces the desire to run the test suite
• The biggest problem with this situation is that
  • you cannot fix the situation if you cannot get the bug to be reproducible

• Contributing Factors
  • Shared Environments
  • Leaky Scenarios
  • Race Conditions and Sleepy Steps
Shared Environments

- Shared Environments
  - Multiple people use the same machine to test in parallel
    - The tests of the two users have the potential of clobbering each other
      - Creating/editing the same database at the same time, writing to a shared XML file, etc. **Boom!**

- Solution
  - Use techniques that isolate one instance of a test from another instance of that same test
    - For instance, have the test create a tmp directory, unique to it, where it stores all of its data
  - Multiple instances of that same test can now be run in parallel
Leaky Scenarios

• Leaky Scenarios
  • One test creates an environment that another test depends on
    • The tests have different tags and cucumber gets invoked on just the tag of the second test: **Boom!**
    • Someone changes the first test, not realizing that a dependency exists: **Boom!**
  • Solution: design tests to create everything they need from scratch
    • Have a really complex system? Use mock objects to simulate non-essential parts (with respect to the test) of the system
Race Conditions and Sleepy Steps

• Race Conditions and Sleepy Steps
  • You have a complex system and your integration test causes two parts of the system to run in parallel
    • or the system to run in parallel with cucumber
  • The test will pass when only the “right” component finishes first
    • If the “race” is close, you end up with a flickering scenario
  • Developers combat this by causing certain steps to “sleep” to wait for the concurrent operation to end; hence “sleepy steps”
• Solution: You need to engineer synchronization points for cucumber that ensures it waits for a system component to finish its work before testing it
Brittle Features

• Brittle features are ones that break at the slightest change to other parts of the system
  • The design of the underlying system may be too tightly coupled and will need to be refactored

• Contributing Factors
  • Fixture Data
  • Duplication
  • Leaky Scenarios (see above: related to dependencies between tests)
  • Tester Apartheid
Fixture Data

• Fixture data refers to having a large amount of data stored somewhere in your test environment that all tests share and come to depend on
  
  • A change in that data can cause tests to fail because developers fail to realize that lots of tests depend on it
  
  • Large sets of fixture data can slow test suites down if all of the data has to be loaded for each scenario
  
• Solution: The book recommends an approach called test data builders in which all the data for a particular test is created by the test itself
  
  • It points to a ruby-based framework called FactoryGirl as an example of this approach
    
    • https://github.com/thoughtbot/factory_girl
Duplication

• Duplication refers to having multiple features that test the same thing
  • Duplication can
    • make scenarios brittle (one change, breaks multiple scenarios)
    • slow your test suite down (as the same functionality gets tested again)
    • and make your customers bored (can’t attach significance to features)

• Solutions
  • Make use of the Background and Scenario Outline keywords
  • Watch out for steps with low abstraction
    • “User clicks on next button to go to the next page” vs. “Users navigates to Accounts page”
Tester Apartheid

- Testers are often regarded as second-class citizens on a software team.
  - They may not have as much technical or software engineering skills as developers but they are
    - capable (and good at) writing automation scripts
    - good at coming up with corner cases
    - good at coming up with comprehensive test suites
  - However, if they do not work with developers their test code can degrade if it is not properly maintained
  - Solution: Have testers and developers work together and encourage this as part of company culture; developers can refactor the test suite when needed and learn from the testers about how to best test their code
Slow features

• After creating a lot of tests, it takes a long time for the entire test suite to run
  • You accumulate tests because you want to know when a change has broken previously passing tests; You can back out the change and/or figure out how to fix the regression

• When test suites take a long time to run
  • developers shy away from running them and as such, they start to commit their changes without testing them!
  • This leads quickly to a situation where a broken build is the norm

• Contributing Factors
  • Race Conditions and Sleepy Steps (see above)
  • Lots of Scenarios
  • Big Ball of Mud
Lots of Scenarios

- Lots of scenarios will, of course, lead to slow test runs
  - It takes a certain amount of time for each scenario to run and that adds up
- Often, however, this is a symptom of the system architecture
  - For instance, a big, monolithic system might require all the features/scenarios to live in one place and all be tested together

- Solutions
  - Decompose the system architecture and have features that target just individual components and then add features that target inter-component interactions
  - Divide feature folders into hierarchies and tag features so that subsets can be easily run independently from one another
Big Ball of Mud

• No software design has been applied to a system at all
  • My friend once encountered a “system” which implemented “shopping cart” functionality for websites
    • it consisted of a single method that when printed covered 42 pages (!)
  • These systems have low cohesion (one component doing too many tasks) and tight coupling (too many dependencies between components)
    • As a result, its difficult to test “just one thing” and your scenarios will have lots of unintended duplication, slowing things down

• Solution:
  • Refactor, refactor, refactor
  • Have the team focus on the architecture of the system for an iteration or two
Bored Stakeholders

• Stakeholders become disengaged with the process of developing the test suite that is needed to help guide development
  • They no longer read existing features
  • They no contribute to the creation of new features
  • They are unwilling to meet with the development team

• Contributing Factors
  • Incidental Details
  • Imperative Steps
  • Duplication (see above)
  • Ubiquitous What?
  • Siloed Features
Incidental Details

• Scenarios contain a lot of detail that are not relevant to what is being tested

  • The book presents an example that is testing whether an e-mail is received after it has been sent

    • The original example had steps that declared the passwords of the users but these passwords were never used

    • The example was rewritten to be much shorter by abstracting away most of the incidental details and leaving clear what exactly was being tested

• Solution

  • Always ask yourself if you are writing at the right level of abstraction

  • Do not let yourself be influenced by existing step/step definitions
Imperative Steps

• Imperative steps are ones that are written in the style of “do this; do that”
  • The problem is that it is very easy for the steps to be written at too low level of abstraction, containing lots of unnecessary detail

• Declarative steps are written at a higher level of abstraction and allow the programmer leeway in how they are carried out

• Contrast this
  • User is not logged in; He goes to home page; He is redirected to login page

• with
  • User is not authenticated; He tries to view restricted content; System authenticates user
Ubiquitous What?

• The team has failed to incorporate the language of the customer (and their application domain) into the system design and project culture

  • If you are developing a ticketing system, you might have words in your system like concert, performance, artist, venue

  • If you ignore those terms, and use arbitrary or terms so generic that there is no obvious mapping (or the terms could be mapped to anything)

    • then your customer can become discouraged and disengaged

• Instead, encourage your team to develop and use a language which is shared with the customer

  • it will reduce mistakes and misunderstandings, improve team confidence and morale, and foster/strengthen the relationship with the customer
Siloed Features

- Cucumber is a command line tool and the features it processes are text files stored in the file system and checked into configuration management systems.
  - As a result, it is very easy for the features to “hide” from the customer.
  - They might not feel like they can access the features easily.
    - Access might require the use of unfamiliar tools (git, text editors).
- Solution
  - Publish the features in a way that your customer can access them.
    - Use scripts, for instance, to convert them to HTML and share them with the customer via a website.
  - Engage with the customer to ensure they are always reading/writing the features and scenarios with the development team.
Summary

• We learned more about Cucumber
  • Steps and the step matching process
  • Regular expressions and their use in steps
  • Scenario Outlines
  • Tags

• We also learned about some of the problems that can be encountered when executing behavior-driven design
  • and solutions that can be used to address those problems
Coming Up Next

• Lecture 14: Review for Midterm

• Lecture 15: Midterm (!!)