Behavior-Driven Development and Cucumber

CSCI 5828: Foundations of Software Engineering
Lecture 09 — 02/14/2012

💘

Goals

- Introduce concepts and techniques encountered in the first three chapters of the Cucumber textbook (I may also refer to it as the “testing” textbook)
  - Installing Cucumber
  - Behavior-Driven Development
  - Gherkin
  - Cucumber
    - It’s integration with ruby (other languages are also supported)

Side Note: Fred Brooks Appears Again

- I like receiving confirmation that the information I put in front of you is important
  - At the start of Chapter 3 of the Cucumber book, take a look at the first citation:
  - “In his famous essay, No Silver Bullet [Bro95], Fred Brooks says:
    - ‘The hardest single part of building a software system is deciding precisely what to build.’”
- They even reference The Mythical Man-Month on the same page!

😃

Installing Cucumber (I)

- If you would like to run the examples presented in lecture, you will need to install Cucumber
- To do so, you will first need to install ruby
  - If you are on Linux or Mac OS X, the book recommends first installing rvm (Ruby Version Manager)
    - Instructions are here: <http://beginrescueend.com/>
    - Then use it to install ruby version 1.9.3-p0
    - On Lion with XCode 4.2.1: “rvm install ruby-1.9.3-p0 --with-gcc=clang”
  - If you are on Windows, use the Ruby Installer
    - http://rubyinstaller.org/

Installing Cucumber (II)

- Once you have ruby installed, you may need to install a “gem” called Bundler
- At the command line
  - gem install bundler
  - **Note: When I installed ruby using rvm, bundler was installed by default**
- Bundler then functions like a package/dependency manager
  - You create a file called “Gemfile” similar to a Maven file or a Makefile and you specify that you need two additional projects: cucumber and RSpec
- You will place this file in the directory where you would like to use cucumber
  - See the contents for this file on the next slide
- Once you have this file created, type “bundle install” in that directory

Installing Cucumber (III)

- `source :rubygems`
- `group :test do`
  - `gem 'cucumber', '1.1.3'`
  - `gem 'rspec-expectations', '2.7.0'`
- `end`
- These are instructions to Bundler that tell it to test whether you have particular versions of cucumber and rspec installed as gems
  - If you do not, then bundler will download and install those gems for you
    - Plus any gems that they, in turn, depend on
- Once “bundle install” is done, you can test whether cucumber was installed by typing “cucumber --help” at the command line

Behavior-Driven Development (I)

- Cucumber is a software requirements and testing tool that enables a style of development that builds on the principles of test-driven development
  - Test-driven development is a core principle and practice of extreme programming and has since been adopted by many other agile life cycles
    - Test-driven development is supported by a few key ideas
      - No production code is written except to make a failing test pass
      - The acceptance criteria for a user story should be specified by a test case
        - and this test case should be written by the developer and customer together
      - The tests should be automated so we are encouraged to run them all the time; they provide insight into the progress being made

Behavior-Driven Development (II)

- These tests are called acceptance tests because they document what behaviors the customer will find acceptable in terms of the final functionality of the system
    - They are different from unit tests
        - Unit tests are for developers and help you “build the thing right”
        - Acceptance tests are for customers and help you “build the right thing”
    - Acceptance tests are higher-level constructs that can be used to guide developers down the path of writing unit tests that will get them to their goal
        - Cucumber helps you write acceptance tests in the customer’s language
            - this encourages the customer to participate in a task they might otherwise skip

Behavior-Driven Development (III)

- Behavior-driven development expands on test-driven development by
  - formalizing its best practices
    - in particular the perspective of working from the outside-in
      - we start our work with failing customer acceptance tests
    - we write the tests such that they serve as examples that anyone can read
      - both to understand a requirement and to understand **how to generate more requirements**
    - we develop a process to encourage our customers to get involved with writing these requirements and to stay involved
    - we aim to develop a **shared, ubiquitous language** for talking about the system

Ubiquitous Language

- Similar to extreme programming’s practice of “metaphor”
- Make sure everyone (including your customers) speak about the system, its requirements and its implementation, in the same way
  - “The case management system tracks the cases handled by service reps”
    - The whole team will now talk about “service reps” and “cases”
      - Any attempt to change that to, say, “workers” and “jobs” will be rejected
- We want to see the same terms used to discuss the system to be present in the requirements, design documents, code, tests, etc.
  - Cucumber helps with this process since, as we shall see, it ties together the tests with the actual code of the system
    - the terms can easily jump the gap!

Example: Typical Cucumber Acceptance Test

- **Feature**: Sign up
    - Sign up should be quick and friendly
    - **Scenario**: Successful sign up
        - New users should get a confirmation e-mail and be greeted personally
        - **Given** I have chosen to sign up
        - **When** I sign up with valid details
        - **Then** I should receive a confirmation email
        - **And** I should see a personalized greeting message

Discussion

- From this example, we can see that
  - acceptance tests refer to **features**
  - features are explained by **scenarios**
  - scenarios consist of **steps**
- The spec is written in natural language in a plain-text file (low entry barrier)
  - BUT the spec is **executable**!
- Cucumber can guide us into turning the language of each step into an executable test case that calls our systems and can then either pass or fail
  - The way it does this is actually designed to get customers and developers working together

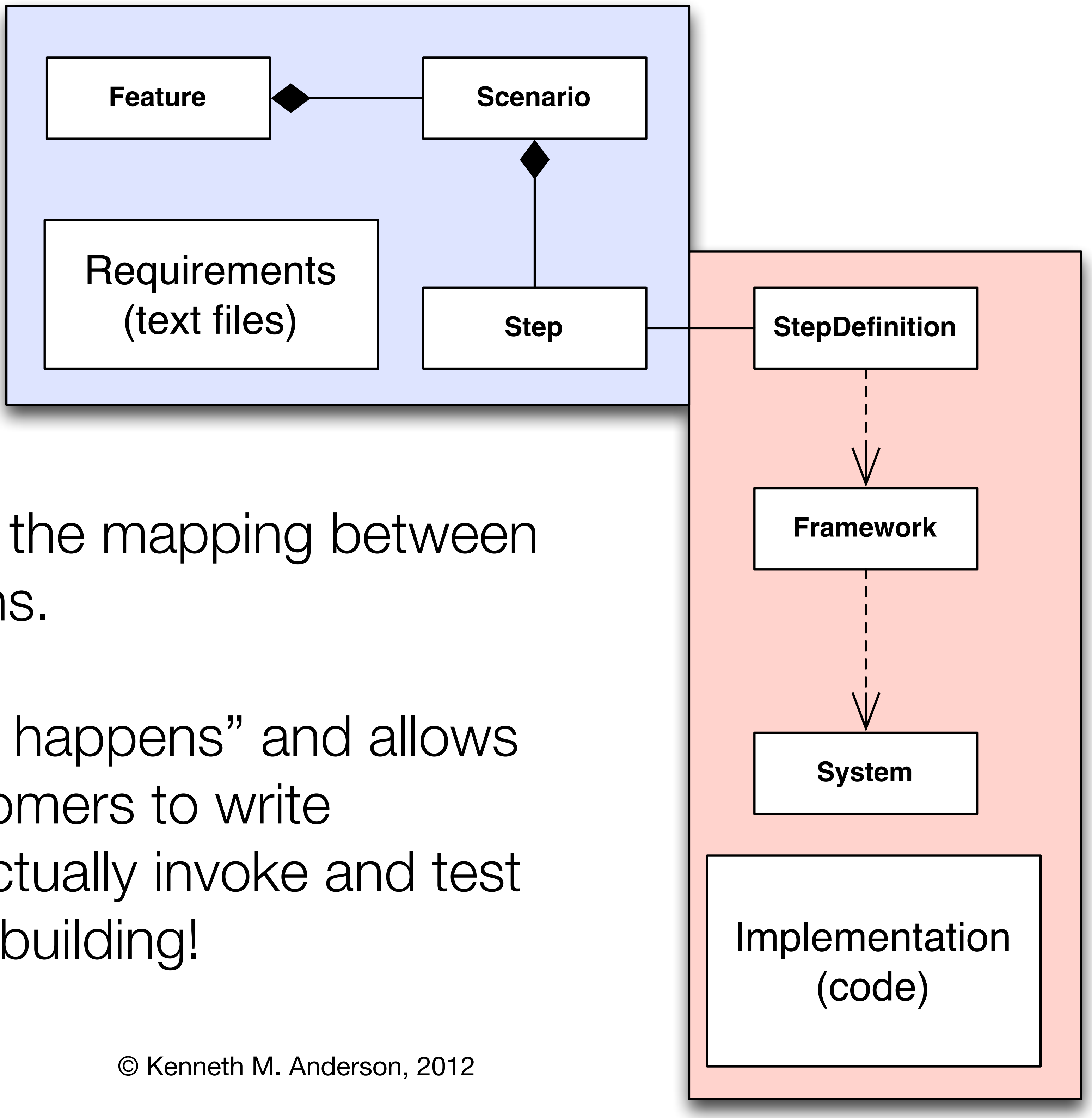
How Cucumber Works (I)

- Cucumber is a command line tool that processes text files that contain features looking for scenarios that can be executed against your system
  - It makes use of a bunch of conventions about how the files are named and where they live to make it easy to get started
    - learn the conventions, follow them, and get most of Cucumber's functionality "for free"
- Each scenario is a list of steps that describe the pre-conditions, actions, and post-conditions of each scenario; if each step executes without error, the scenario is marked as having passed
  - At the end of a run, Cucumber will report how many scenarios passed; if something fails, it provides information about what failed so the developer can make progress
- Features, scenarios, and steps are written in language called Gherkin

How Cucumber Works (II)

- To turn natural language into executable specifications
  - each step must be accompanied by a step definition
  - most steps will gather input and then delegate to a framework that is specific to your application domain in order to make calls on your framework
    - for instance, Capybara is a framework for automating interactions with web browsers
    - if your application is a web app, then your step definitions will most likely delegate to Capybara to make sure that your application is loaded and then stepped through the scenario ("click this link", enter "Ken" in the "first name" field, etc.)
  - the point is that your step definitions will likely be short and easy to maintain

How Cucumber Works (III)



The key to Cucumber is the mapping between steps and step definitions.

That's "where the magic happens" and allows your non-technical customers to write specifications that will actually invoke and test the system that you are building!

Gherkin

- Gherkin is the language used to write features, scenarios, and steps
- The purpose of the language is to help us write concrete requirements
- Consider
    - Customers should be prevented from entering invalid credit card details.
- Versus
    - If a customer enters a credit card number that isn’t exactly 16 digits long, when they try to submit the form, it should be redisplayed with an error message advising them of the correct number of digits.
- The latter is **much more testable**; we want to remove as much ambiguity as possible in our requirements since ambiguity is a proven source of errors
    - Gherkin is designed to create more concrete requirements

Example

- **Feature**: Feedback when entering invalid credit card details **Feature definition**
  - In user testing, we've seen a lot of people who make mistakes... **Documentation**
  - **Background**: **True for all scenarios below**
    - **Given** I have chosen an item to buy
    - **And** I am about to enter my credit card number
  - **Scenario**: Credit card number too short **Scenario definition**
    - **When** I enter a card number that is less than 16 digits long
    - **And** all the other details are correct
    - **And** I submit the form

      Steps
    - **Then** the form should be redisplayed
    - **And** I should see a message advising me of the correct number of digits

Gherkin Format and Syntax

- Gherkin files are plain-text and have the extension .feature
- Keywords:
  - **Feature**
  - Background
  - **Scenario**
  - **Given**
  - **When**
  - **Then**
  - **And**
  - **But**
  - *
  - *Scenario Outline*
  - Examples

Small and Simple!

Feature

- The **feature** keyword is used to group a set of tests (scenarios)
    - The text on the same line as the keyword is considered the name of the feature
    - All text between the initial line and a line that starts with **Scenario**, **Background**, or **Scenario Outline** is considered part of a feature's description
- It is conventional to name a .feature file by taking the name of the feature, converting to lowercase and replacing spaces with underlines
    - feedback_when_entering_invalid_credit_card_details.feature
- We will see **Scenario Outline** in action briefly later in this lecture but we will learn more about it and **Background** in a separate lecture later this semeser

Coming Up With Features: Think of your Users

- In order to identify features in your system, you can use what the authors describe as a “feature injection template”
  - In order to <meet some goal> as a <type of user> I want <a feature>
- This is good advice
  - The functional requirements of a system are determined by asking the question
    - for each type of user
      - what goals are they trying to achieve?
      - what tasks must they perform to achieve those goals
      - how does our system support those tasks?

Scenario (I)

- To express the behavior of our system, we attach one or more scenarios with each feature
    - it is typical to see 5 to 20 scenarios per feature to completely specify all the behaviors we'd like to see around a particular feature
- Scenarios follow a pattern
    - Configure the system
    - Have it perform a specific action
    - Verify that the new state of the system is what we expected
- We start with a **context**, describe an **action**, and check the **outcome**

Scenario (II)

- Gherkin provides three keywords to describe contexts, actions, and outcomes
  - Given: establish context
  - When: perform action
  - Then: check outcome
- Example
  - **Scenario**: Withdraw money from account
    - **Given** I have $100 in my account
    - **When** I request $20
    - **Then** $20 should be dispensed

Scenario (III)

- You can add additional steps to the context, action, and outcome sections using the keywords **and** and **but**
- They allow you to specify scenarios in more detail
  - **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
- These keywords help increase the **expressiveness** of the scenario

Scenario (IV)

- In creating scenarios, a key design goal (indeed requirement) is that they must be stateless; as the book says
  - “Each scenario must make sense and be able to be executed independently of any other scenario”
- You can’t have the success condition of one scenario depend on the fact that some other scenario executed before it
  - Each scenario creates its particular context, executes one thing, and tests the result

Scenario (V)

- Having stateless scenarios provides multiple benefits
    - Tests are simpler and easier to understand
    - You can run just a subset of your scenarios and you don't have to worry about your test set breaking
    - Depending on your system, you might be able to run tests in parallel reducing the amount of time it takes to execute all of your tests

Detailed Example

- Let’s step through a variant of the example from chapter 2
    - We will see that cucumber actually helps you perform test driven design
        - A process which should be
            - iterative
            - incremental (take small steps)
            - starts with a failing test case
            - solves it with the simplest code possible
            - looks for opportunities to refactor the code to something more complex (which can then do more than it could previously)
- In this simple example, we’ll create a calculator with one feature

Step 1: Create project directory

- Steps
    - mkdir calculator
    - cd calculator
    - cucumber
- Output
    - You don’t have a features directory
- Discussion
    - Example of cucumber moving the process forward AND teaching the user about its conventions (what it expects to find)

Step 2: Create features directory

- Steps
  - mkdir features
  - cucumber
- Output
  - Cucumber reports that it found zero scenarios with zero steps; all is fine!
- Discussion
  - Progress! We took a small step and now cucumber reports that (from its standpoint) everything is fine
    - Implicitly it is telling us “Give me some scenarios to execute!”

Step 3: Add a feature

- Steps
  - cd features
  - <create adding.feature>
  - <edit to contain scenario “Add two numbers”>
  - cd ..
  - cucumber
- Output
  - Cucumber complains that our scenario is undefined because it doesn’t know how to execute its steps; it then provides us with information on how to create step definitions
    - indeed, it gives us templates for our step definitions!

Regular Expressions are the Duck Tape of SE

- Let’s look at one of the step definition templates
    - Given /^the input "([^"]*)"$/ do |arg1|
        - pending # express the regexp above with the code you wish you had
    - end
- Everything between the first “/” and the second “/” is a regular expression
- Cucumber analyzed the syntax of our steps and generated a regular expression that can parse each one
    - It is set-up to capture information that appears in quotes as arguments
- The regular expression "([^"]*)" means “match a quotation mark, then match zero or more characters that are not a quotation mark, then match another quotation mark”; the parentheses mean “remember what you matched”

Discussion

- Understanding the rest of the template
  - **do |arg1|** is the beginning of a ruby block; arg1 equals what we matched in the expression; for our specific example it will equal the string “2+2”
  - **pending** is a cucumber-specific statement that tells cucumber that the step is currently undefined
  - **end** is a ruby keyword used to denote (in this case) the end of the block
- Ruby blocks are essentially anonymous functions that can be passed around
  - What we did was define a function that cucumber can invoke to help automate the testing of our calculator program
- Regular expressions are used to quickly parse the information of the step
  - I refer to them as the “duck tape” of software engineering because they are used **ubiquitously** to add power to software engineering tools

Step 4: Create step definitions

- Steps
  - mkdir features/step_definitions; cd features/step_definitions
  - <create file “calculator_steps.rb”> # note: this is a ruby program file
  - cd ../..; cucumber
- Output
  - Scenario is no longer undefined; it is now pending
- The next few steps will focus on replacing the **pending** keyword with code that responds to the step
  - Note: my step definition file diverges from the textbook to keep things interesting!

Step 5: Edit the first step definition

- Steps
  - Change arg1 to “first, second” and replace pending keyword with
    - @first = first
    - @second = second
  - cucumber
- Note: you will need to have two windows open; one in “calculator” directory and one in the “step_definitions” directory
  - Also @first and @second are simply ruby instance variables for a class that is being dynamically defined behind the scenes
- Output
  - First step passed; second step is now pending

Step 6: Edit the second step definition

- Steps
  - Replace pending keyword with
    - @output = `ruby calc.rb add #{@first} #{@second}`
    - raise('Command failed!') unless $?.success?
- Notes:
  - The backquotes are ruby's process execution command
  - This command will run in the "calculator" directory
  - **unless** is a ruby keyword; **$?** is a ruby variable that lets us query the status of the last executed process
- Output
  - The exception is raised because "calc.rb" does not exist

Discussion: Small Steps

- With test-driven design, we would do the following steps next
    - create an empty calc.rb
    - verify that step 2 now “passes”
    - implement the third step definition to check the output
    - create a program that outputs “4” to make sure that “2+2” passes
        - “do the simplest thing to make the test pass”
    - update our scenario to contain two or more test cases; watch it fail
    - change our program to actually do the addition and return the answer
        - this is an example of “look for opportunities to refactor the code” that was discussed back on slide 26
- We’re going to “skip a bit” and do more in each step

Steps 7 and 8: Create calc.rb and define 3rd step

- Steps
  - In “calculator” directory: touch calc.rb
  - In “step_definitions” directory
    - change arg1 to expected_output
    - replace pending with
      - @output.should == expected_output
      - “should” is a mechanism provided by a framework called RSpec
      - It is an assertion that @output should equal our expected output
- Output: Test Fails! Our program is empty and returns nothing not “4”

Step 9: Implement calc.rb

- Steps
    - Put the following code in calc.rb

      ```
      if ARGV[0] == "add"
        first = ARGV[1].to_i
        second = ARGV[2].to_i
        print first + second
      end
      ```

    - cucumber
- Output: Scenario passed!

Step 10: Provide more test cases (I)

- Steps
  - Change adding.feature to execute multiple test cases using “Scenario Outline”; In particular, make it look like this

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

    Examples:
      | input   | output |
      | 2+2     | 4      |
      | 98+1    | 99     |
      | 255+390 | 645    |
```

Step 10: Provide more test cases (II)

- Output
  - All tests passed
    - because we already implemented a fully working calc.rb
  - If we had instead created a version of calc.rb that contained the line
    - print “4”
  - This scenario outline would have forced us to refactor calc.rb to a working version

Discussion

- The example has demonstrated several things
  - The process is highly incremental and iterative
    - and is driven by test cases
  - Cucumber's output usually tells the developer what to do next
    - sometimes the instruction is implied
  - Regular expressions can enable powerful functionality
- Limitations
  - The example is centered around a very simple example
    - We'll see more complicated examples in future lectures

Summary

- In testing your system, it is key that your customer be engaged in the process
    - They are the source of “ground truth”
    - If they can give you concrete examples of what they expect, your job as a developer gets a lot easier
- Cucumber is a test automation tool that
    - allows tests to be specified in natural language using Gherkin
    - but allows those tests to be executable, testing your system
    - supports an iterative and incremental process that actually guides you through the process of test driven design

Coming Up Next

- Lecture 10: Agile Project Inception
- Lecture 11: Taming Shared Mutability, Part 1