Intermediate Cucumber Continued

CSCI 5828: Foundations of Software Engineering
Lecture 22 — 04/05/2012
Goals

• Continue to work through a detailed example of using Cucumber by reviewing the material in chapter 8 of the Cucumber textbook

• Using Cucumber to test a user interface (simple web app)
  • Involves use of three frameworks: Capybara, launchy, and Sinatra

• Cucumber Hooks
Where Were We?

• In the last lecture on Cucumber, we
  • started a detailed example centered around a simple ATM domain model
  • learned about transforms (they help convert captured regex strings)
  • learned about the world object (helps step definitions share state)
  • learned about features/support
    • and how that directory can help us separate
      • our step definitions (app-specific test code)
        • from our “testing harness” (app-independent (ish) test code)
      • our testing harness from the system under test
Next Up: Adding a UI (and testing it)

• We are now going to transition from testing our domain model directly
  • to testing a user interface that will
    • instantiate an instance of the domain model
    • and make calls on it in response to user commands

• We need to find a place in the code where we can insert a handle to the user
  interface and call it instead of the classes in our domain model directly
  • But first, we need to get our project ready to add a UI
    • We’ll make use of Bundler to ensure that we have the right frameworks
      installed
Updating our Gemfile (I)

• Recall back in Lecture 9, we covered steps to install Cucumber
  • It went like this
    • Install ruby version manager
    • Use it to install ruby 1.9.3
    • Install bundler: “gem install bundler”
    • Create a Gemfile and invoke “bundler install”
      • The gemfile contained code that listed the gems we needed
Updating our Gemfile (II)

- Having done all that work, we are now in a much better position to add new frameworks
  - We can now change our Gemfile to look like this and run “bundler install”
    - `source :rubygems`
    - `gem 'sinatra', '1.3.1'`
    - `group :development do`
      - `gem 'rspec', '2.7.0'`
      - `gem 'cucumber', '1.1.3'`
      - `gem 'capybara', '1.1.2'`
      - `gem 'launchy', '2.0.5'`
    - `end`
  - Don’t worry about the “group” statement, this essentially loads all four listed gems
    - and any of the frameworks they depend on
Sinatra

• Sinatra is a ruby framework that makes it easy to generate simple web applications
  
  • If we add this code to our system (nicebank.rb)
    
    • require 'sinatra'
    
    • get '/' do
      
      • 'Welcome to our nice bank.'
    
    • end
  
  • we will have created a web app that returns the above string when accessed
    
    • by default Sinatra runs on port 4567
    
    • To launch Sinatra (in the ATM directory):
      bundle exec ruby lib/nicebank.rb
Connecting our Environment

• Recall that the file env.rb in features/support is used to connect our testing harness to the system under test
  • Since we are developing a web app (using Sinatra)
  • We will use a framework called Capybara to test it
    • Capybara was designed to interact with web apps
• First, we connect our test environment with Sinatra with the following code in env.rb
  • require 'capybara/cucumber'
    Capybara.app = Sinatra::Application
    Sinatra::Application.set :environment, :test
• This tells Capybara to talk to Sinatra and configures Sinatra with a default env
Connecting the Test to the UI (I)

• Now, we need to connect our Cucumber Feature/Scenario to our new UI
  • Our acceptance test should work no matter what it is connected to
    • We want a withdrawal for $20 to work regardless if the transaction is handled
      • in person
      • over the web
      • via e-mail
      • via text message
      • etc.
  • Therefore, we want to keep as much of our tests independent from UI
Connecting the Test to the UI (II)

- To keep most of the existing test infrastructure the same, we must try to avoid changing our
  - features
  - scenarios
  - step definitions
- That doesn’t leave many options; what’s left? Our world object
  - We used our World object to generate instances of our domain model
  - For instance, if a step definition needs access to the “teller” object
    - then we went to the world object and asked it for a teller object
  - We can use this set-up to hide the fact that a UI has now entered the equation
Connecting the Test to the UI (III)

• For instance, our withdraw step definition says

  • When /^I withdraw (#\{CAPTURE_CASH_AMOUNT\})$/ do |
    amount|

    • teller.withdraw_from(my_account, amount)

  • end

• In that single line of code, we have a “contract”

  • I’m going to ask for a teller object

    • but I don’t really care if I get one

    • what I care about is that I get back an object that

      • responds to the message “withdraw_from”
Connecting the Test to the UI (IV)

• This type of contract is also known as “duck typing”
  • “If it looks like a duck, walks like a duck, & quacks like a duck, it’s a duck”
• So, in our world object, we’re going to create a UserInterface class and we’re going to return it rather than an instance of the teller object
• We go from this
  • def teller
    • @teller ||= Teller.new(cash_slot)
  • end
• to
  • @teller ||= UserInterface.new
    And, we make sure that UserInterface can respond to withdraw_from messages
Breakage

- Of course, now with this change, our Cucumber acceptance test fails
  - When we withdraw money, we invoke our UserInterface class and it currently does nothing
  - As a result
    - no money gets put into the cash slot, and
    - our balance is not updated
- But, we will fix this one step at a time
  - First, we need to know what our UI will look like
Prototype UI

- We’re going to pretend our web app currently looks like this

- Before we create this form
  - we will create the code that tests it

- This fits in with the style of TDD
  - we first need a failing test case
  - then we’ll make the changes necessary to cause the test to pass

- We’ll make use of the Capybara’s domain specific language (DSL) that is designed to test web applications to write our “test” (the code that interacts with this form)

![Capybara's DSL module](image-url)
Using Capybara in UserInterface

• To do this, we create a UserInterface class (in world_extensions.rb)
  - class UserInterface
    - include Capybara::DSL
    - def withdraw_from(account, amount)
      - visit '/'
      - fill_in 'Amount', :with => amount
      - click_button 'Withdraw'
    - end
  - end

The DSL makes writing the test simple!
Running the Test

• If we run the test now

  • we will see it fail

  • but with an error message generated by Capybara

    • cannot fill in, no text field, text area or password field with id, name, or label 'Amount' found

• The test fails but we’re actually

  • launching a web server, calling it, parsing its return value, and failing because we didn’t get back the form we expected!

  • all in one test, all due to the power of the frameworks involved
Behind the Scenes

• Now, we know why the test is failing
  • Sinatra is currently configured to return just this string
    • 'Welcome to our nice bank.'
  • Capybara is looking for an HTML form that has a form element with the “Amount” label
    • It doesn’t find it, so it fails
• But, as our application gets more complex, we won’t necessarily be able to predict what web page is being generated by our test code
  • We need a way therefore to see what page was presented
  • To do that, we need to learn about a new Cucumber feature: Hooks
Hooks (I)

• Hooks are methods you can define in your Cucumber support code that will run before or after each scenario
  • You use the keywords “Before” and “After” to define them
  • If we return to our calculator example and add hooks.rb to its support directory with this code
    • Before do
      • puts "Go!"
    • end
    • After do
      • puts "Stop!"
    • end
  • then you will see “Go!” and “Done!” printed for each scenario  DEMO
Hooks (II)

• Hooks are thus similar to the methods setup() and teardown() found in JUnit
  
  • You can create tagged hooks that will only run if a scenario with their tag is about to execute
    
    • Before(‘@admin’) do
      
      • ...

      • end
    
    • will only execute its code
      
      • if a feature/scenario tagged with @admin is run
Hooks (III)

- You can get information about the scenario that is about to run by adding a scenario parameter to the Before and After methods
  
  - After do |scenario|
    
    - puts "Oh dear" if scenario.failed?
  
  - end

- This code will examine the scenario that just finished running and see if it failed
  
  - if it did, it prints “Oh dear”

- We can use this feature in our ATM test cases to print out the web page that was generated by our application when a scenario fails (like it is now)
Displaying the Web Page

- To do that, we’re going to add a file called debugging.rb to our features/support directory and add the following hook:
  - After do |scenario|
    - save_and_open_page if scenario.failed?
  - end

- The method “save_and_open_page” is provided by Capybara. Internally, it makes use of the launchy framework to open a web browser and display the web page that is causing the test case to fail.

- And, sure enough, if we run cucumber now, a .html page is automatically saved and our default browser launches and displays it.
  - Fun! Note: This proves that Sinatra is being launched and our web app is being run each time we run cucumber!
Create the Form

• Back in nicebank.rb, we will now update our Sinatra web app to generate a form and we will also add a method to generate a “fail” response when the “Withdraw” button is pushed (thus submitting the form)

• **DEMO**

• With those changes in place, we will see in our web browser that
  • the form is being generated
  • an error is being generated
  • the text of our error is appearing in cucumber’s output

• Due to the power of the frameworks, the integration between test code and app is seamless
Fix the Test Case (I)

• Okay, we now have everything in place to write the code that will fully link the test code and the application

• What we need to do is the following

  • Make sure that the domain objects being used by the step definitions are the same objects being used by the web application

  • To do that, we will use something called “settings” in a Sinatra web application to store instances of ruby classes

  • Think of it as a hash table
Fix the Test Case (II)

- Here’s what currently happens
  - The world object auto-creates an account object and a cash slot
  - The world object currently creates a UserInterface object when asked to create a teller object
- Here’s what we need to happen
  - The world object can continue to create the account object
  - The web app will create a teller object automatically
    - when processing the withdraw form action;
  - It will create a Cash Slot and store that as a setting automatically
  - It will expect to find an account object in its settings
    - it will generate an error if it doesn’t find an account object
  - The step definitions will now need to set the account object and make use of the new cash slot
Fix the Test Case (III)

• In nicebank.rb change the code for the withdraw action to

  • set :cash_slot, CashSlot.new
  • set :account do
    • fail 'account has not been set'
  • end

• post '/withdraw' do

  • teller = Teller.new(settings.cash_slot)
  • teller.withdraw_from(settings.account, params[:amount].to_i)

  • end

Running cucumber now causes the test to fail with the “account has not been set” error message; that’s because we have not yet updated the world object to do the right thing
Fix the Test Case (IV)

• Now, update the world object
  • For the cash slot helper method, we now use the instance created by Sinatra
    • def cash_slot
      • Sinatra::Application.cash_slot
    • end
  • In our UserInterface class, we make sure that Sinatra uses the account class that is passed into the withdraw_from method
    • Sinatra::Application.account = account
• Run cucumber and the test now passes
How? (I)

• What’s the big picture of the system as it now stands?

• 1. We invoke cucumber
  • a. It invokes env.rb
    • i) which runs our system in nicebank.rb
      • a) which launches Sinatra and defines our app
      • b) and creates an instance of Cash Slot and stores it as a setting
    • ii) and connects Capybara with our Sinatra app
  • b. Cucumber processes all of its support files, recording transforms, hooks, step definitions, etc.
How? (II)

• c. Cucumber finds our feature file and finds a scenario within
• d. It matches steps and during those steps it
  • i) calls the account helper function of the world object
    • a) creating an account object
  • ii) calls the teller helper function of the world object
    • a) creating a user interface object
  • iii) calls withdraw_from passing in an amount and the account
    • a) this uses Capybara to set the account on Sinatra
    • b) and fill out the form and submit it
    • c) which causes the “withdraw” handler to execute
      • this creates a teller object which uses the previously created account and cash slot objects to perform the withdrawal
How? (II)

• d. It matches steps and during those steps it
  • iv) it verifies that the cash slot and the account have their correct values; the world object makes sure those steps use the correct objects
  • v) It declares success and returns
• e. The “After” hook runs but does nothing because the scenario passed
  • If it had failed, the After hook would have displayed the web page that was generated by Sinatra
• f. The cucumber command finished and returns a success status to the shell that invoked it

• Wow!
Summary

• Brief introduction on how to integrate Cucumber with a user interface
  • In this case a web app, powered by Sinatra and accessed via Capybara
  • Saw how to use Sinatra settings to help share state between the application and the step definitions
• Learned about Cucumber hooks and used it to help us with a failing test case
Coming Up Next

- Lecture 23: More Cucumber
- Lecture 24: Agent Model of Concurrency