

Start with a Feature

- Feature: Cash Withdrawal
 - Scenario: Successful withdrawal from an account in credit
 - Given I have deposited \$100 in my account
 - When I request \$20
 - Then \$20 should be dispensed
- Use this feature to drive the development of a domain model for the ATM
 - We can then build services and a UI on top of the domain model
 - While letting cucumber drive the whole process

What Next?

- Ask cucumber!
 - Reports 1 undefined scenario and 3 undefined steps
 - Suggests the following step definitions; let's place these in steps.rb

```
Given /^I have deposited \$(\d+) in my account$/ do |arg1|
  pending
end
```

```
When /^I request \$(\d+)\$/ do |arg1|
  pending
end
```

```
Then /^\$(\d+) should be dispensed$/ do |arg1|
  pending
end
```

Developing the First Step

- In order to make progress, we will need an Account class
 - Based on TDD principles, we try to access the class before we create it
- Background
 - To create a new instance of a Ruby class, you invoke `new()` on it
 - Also, to convert a string into an integer in Ruby, you use the `to_i()` method
 - In Ruby, parentheses for method calls are mainly optional
- With that, we make the body of our first step definition, the following
 - **`Account.new(amount.to_i)`**
- `amount` is the argument being passed into the step definition

The Result? Failure!

- We invoke cucumber to see the test fail
 - In particular, ruby says:
 - uninitialized constant Account (NameError)
 - This just means that Ruby has no idea what we mean by “Account”, since the class has not been created yet
- We can solve this problem by creating the class right inside steps.rb
 - This will just get us started, eventually we will move Account to a better place

The class

- Creating a class in Ruby is easy;
 - class Account
 - def initialize(amount)
 - end
 - end
- And, because we are practicing TDD/BDD, we do not try to get the Account to do anything, we just define it
 - Note: in Ruby, initialize() is the constructor. It will be called during the process that is triggered by a call to new()
- Now what happens?

Step passed

- Ruby is more than happy to create an instance of a class that is not captured by any variable
 - As a result, no exception is thrown and the step is marked as passed
 - The second step is marked as pending
 - The third step is then skipped
- However, even though cucumber is happy, we're not happy
 - The account that is created doesn't stick around
 - It doesn't know it's balance
 - It can't be used in subsequent steps
- So, we have work to do!

Semantic Sense Needed

- The concerns on the previous slide are just logistical
 - We also have semantic concerns
- Our code does not honor the language of the step
 - The step talks about **depositing funds** into **my** account
 - but the code passes the funds to the Account class's constructor
 - nothing is being deposited anywhere; no customer either
- Having to convert the amount to an integer is less than ideal
 - We know from the regular expression that amount represents a number
 - It would be nice to have that conversion done for us

Fix the Class

- To ensure that we honor the semantics of the step
- First, we fix the class
 - Get rid of the constructor (we'll use the default constructor for now)
 - Add a method called `deposit` (that does nothing; TDD keeps it simple!)
- Second, we update the step definition to reflect these changes
 - **`my_account = Account.new`**
 - **`my_account.deposit(amount.to_i)`**
- If we run cucumber again, the step will pass and the internal semantics of the code are better aligned with the step

When do we add code?

- In TDD, **we never add code until there's a test case that's failing**
- To do that in this context
 - let's **add an assertion** to the end of our step definition
 - We have deposited funds, let's check the Account's balance
- The book uses RSpec
 - **my_account.balance.should eq(amount.to_i),**
 - **"Expected the balance to be #{amount} but it was #{my_account.balance}"**
- Background
 - Recall, parens are optional; **balance** is a method, so is **should**
 - In strings, **#{var}**, is string interpolation; it injects the value of var into the string

Failing Test Case; Time to Add Code

- First the assertion fails because `Account` does not have a **`balance()`** method
 - We add an empty implementation of one
- The test still fails but now its due to the assertion failing
 - Expected the balance to be 100 but it was
(RSpec::Expectations::ExpectationNotMetError)
- Now, we can FINALLY do something more than create method skeletons
- In **`deposit()`**: `@balance = amount`
- In **`balance()`**: `@balance`
- Note: `@var` is an instance variable
- Note: methods automatically return value of last evaluated expression

Transforms

- Last semantic concern: having to convert strings captured by regular expressions to integers
 - It leads to lots of code duplication: **to_i()** calls everywhere
- To fix this, Cucumber provides a method called Transform that allows us to define how certain regular expression patterns will be handled
 - In particular, whenever a step is matched to a step definition
 - Cucumber checks to see if any Transforms match the step's captured arguments
 - If so, the captured string, is passed to the transform
 - Then whatever value comes back from the transform is passed to the step

Example

- Transforms go into your step definition file
 - they will be evaluated along with the rest of your step definitions
- We need a transform that detects numbers and converts them automatically
- **Transform `/^\d+$/ do |number|`**
- **`number.to_i`**
- **`end`**
- The regular expression above matches strings consisting only of one or more digits
 - with the transform in place, we can remove the multiple **`to_i()`** calls in our step definition

Interesting Potential

- Transforms have interesting potential at making our step definitions more focused and expressive
- Image you have phrases like “User Ken” in your step definitions
 - You can capture them and then find the associated User object automatically
- **Transform `/^User ([a-zA-Z]+)$/ do |name|`**
 - **`Users.find(name)`**
- **end**
- Now each step definition that captures “User Ken” will automatically have the User object whose name is “Ken” passed to it

New Source of Duplication

- However, Transforms **introduce a new form of duplication**
 - The regular expression that appears in the Transform statement is the same as the regular expression that appears in the step definition
- This duplication can lead to maintenance headaches
 - because we can change one regular expression
 - and forget to change the other
 - especially problematic if the expression is used by multiple step definitions
- To solve this, the call to Transform returns its regular expression
 - This can be captured and used in the step definitions instead

Examples

- **CAPTURE_A_NUMBER = Transform /[^]\d+\$/ do |number|**
 - **number.to_i**
- **end**
- **Given /[^]I have deposited \\${#{CAPTURE_A_NUMBER}} in ...**
- **CAPTURE_CASH_AMOUNT = Transform /[^]\\$(\d+)\$/ do |digits|**
 - **digits.to_i**
- **end**
- **Given /[^]I have deposited (#{CAPTURE_CASH_AMOUNT}) in ...**
- The regular expression pattern of the call to Transform is captured as a string and then injected into the regular expression of the step definition via interpolation

On to Step 2

- We can of course upgrade our step definition for step 2 to make use of our transform
 - **When /^I request (#{CAPTURE_CASH_AMOUNT})\$/ do |amount|**
- We can also sketch out some code to handle this step
 - **teller = Teller.new**
 - **teller.withdraw_from(my_account, amount)**
- In the step, we say “request” but in the code (and our feature) we say “withdraw”. Let’s just change the step and its step definition
 - **When I withdraw \$20**
 - **When /^I withdraw (#{CAPTURE_CASH_AMOUNT})\$/ do |amount|**

Create the Teller class

- Cucumber now tells us that the step is failing because Teller does not exist
 - class Teller
 - def withdraw_from(account, amount)
 - end
 - end
- Now, cucumber complains that “my_account” is undefined
 - We created a my_account instance in the first step definition
 - but that was a local variable that has no scope outside of that definition

Use Instance Variables

- We saw the solution to this problem in our previous lectures
- Rather than creating `my_account` as a local variable
 - `my_account`
- we need to create it as an instance variable
 - `@my_account`
- If we update `steps.rb` to use instance variables, then everything works
 - DEMO
- BUT...

Instance Variables Considered Harmful

- The problem with using instance variables to communicate state across step definitions is that it can lead to fragile steps (and other problems)
 - Remember, step definitions are shared across ALL scenarios
 - If you have step definition A create an instance variable used by step definition B, then you have to guarantee that step A always appears before step B
- We can solve this problem by using Cucumber's World object to make sure that for any particular scenario all variables are created as needed
 - We do this by creating helper methods on the World object

The World (I)

- In previous lectures, I discussed how the instance variables created in one step definition were accessible in other step definitions
 - At the time, I said “Cucumber must be creating an object that provides context across step definitions;
 - these instance variables are being created on that object
- I was right!
- Cucumber creates an object called World at the start of each scenario
 - The step definitions execute as if they were methods on this object
 - When they create instance variables, they are creating instance variables on this object

The World (II)

- To solve the problems associated with instance variables not being created correctly, we can create helper methods and store them on the World object
 - We do this using a mechanism in Ruby called modules
 - Our module will define one or more methods
 - We then “mix in” the module into the World object
 - the methods of our module then become directly available

Ruby Idiom Explained (I)

- In our module, we are going to use a statement like this
 - `@my_account ||= Account.new`
- This is a ruby idiom to make sure that an instance variable is created once and only once
- To understand the line of code, consider this example
 - `a += 10`
 - This is equivalent to “`a = a + 10`”
- The line above then is short for
 - `@my_account = @my_account || Account.new`

Ruby Idiom Explained (II)

- The or operator (||) in Ruby is a short-circuit operator
- The statement
 - `@my_account = @my_account || Account.new`
- is equivalent to the following pseudocode
 - Does `@my_account` exist?
 - If yes then `@my_account = @my_account`
 - If no, then `@my_account = Account.new`
- The `Account.new` statement will only execute once

Back to the Example

- To ensure that `my_account` is always created and available, we will create the following module
 - **module AccountUtils**
 - **def my_account**
 - **@my_account ||= Account.new**
 - **end**
 - **end**
- We then add this module to the World object by calling
 - **World(AccountUtils)**

Update Step Definitions

- With the previous code in place, we have now ensured that the World knows about a **method** called `my_account`
 - When called, that method returns a single instance of the Account class
- Our step definitions can now be updated to the following
 - In the first step definition
 - **`my_account.deposit(amount)`**
 - In the second step definition
 - **`teller.withdraw_from(my_account, amount)`**
- Very important: `my_account` in the above two lines is a **METHOD CALL** not an instance variable or a temporary variable

On to Step 3

- The first two steps are passing
 - Note: the code does not do anything yet
 - but we will not fix that problem until we have a clearly failing test case
- In step 3, we need an object that can be used to check if the ATM dispenses the requested cash
 - We will call this the cash slot
- The step definition for step 3 will have code that looks like this
 - `cash_slot.contents.should == amount`
- This means that cucumber will not know about the method `cash_slot`

Making Step 3 Work (I)

- We will update our module to have a helper method that will create a cash_slot;
- We'll create a cash slot class and have it raise an exception
 - class CashSlot
 - def contents
 - raise("I'm empty!")
 - end
 - end
- We now have a failing test case
- **DEMO**

Making Step 3 Work (II)

- Remembering Where We Are
 - In Step 1, we create an account and ensure it has the correct balance
 - In Step 2, we create a teller and have it perform the withdrawal
 - In Step 3, we check a cash slot to see if it has the correct amount
- This fails right now because we have no code to make this happen
 - We need to update the Teller class to
 - point at the cash slot
 - and put the correct amount of money in the cash slot during a withdrawal

Making Step 3 Work (III)

- The teller class will now look like this
 - class Teller
 - def initialize(cash_slot)
 - @cash_slot = cash_slot
 - end
 - def withdraw_from(account, amount)
 - @cash_slot.dispense(amount)
 - end
 - end

Making Step 3 Work (IV)

- I now need to update the code that creates the Teller to pass it a `cash_slot`
 - That's when I notice that Teller is now the only object being created in a step definition
 - Let's create a helper method to manage its creation
 - And update the relevant step definitions to reflect these changes
- We then need to add a method to CashSlot called `dispense()` and ensure that the existing method called `contents` is linked to it
- DEMO

Time to Refactor

- All three steps of the scenario pass BUT
 - our code is in horrible shape
 - we have multiple types of code mixed together in a single file (steps.rb)
 - cucumber specific code, domain code, and test code
- Let's refactor
 - The domain code should leave in the root level of our project inside of a lib directory
 - Create the lib directory and put all three classes in nice_bank.rb
 - Change steps.rb to have the following line at the top
 - `require File.join(File.dirname(__FILE__), '..', '..', 'lib', 'nice_bank')`

Refactoring (I)

- The domain code should leave in the root level of our project inside of a **lib** directory
 - Create the lib directory and put all three classes in **nice_bank.rb**
 - Change steps.rb to have the following line at the top
 - `require File.join(File.dirname(__FILE__), '..', '..', 'lib', 'nice_bank')`
 - This is Ruby code
 - to create a file reference to `../..lib/nice_bank.rb`
 - and then load its code (require its use)
- Run cucumber to make sure everything still works

Refactoring (II)

- Cucumber has a folder that is meant to contain code that supports the step definitions
 - It is called features/support
- The code in this directory is loaded in an undefined manner EXCEPT that a file called **env.rb** is always executed first if it is present
 - A fundamental concern of booting our testing environment is loading the application under test
- Therefore
 - create the features/support directory
 - create the env.rb file within it and move the require statement to it
- Run cucumber to verify that everything still works

Refactoring (III)

- We now have a place for our cucumber-specific support code
 - Our Transform method can be moved to `features/support/transforms.rb`
 - Plus our DomainUtils module and the call to `World()` can be moved to the support directory in a file called `world_extensions.rb`
- These names are solely to help us as developers
 - We can call them “`foo.rb`” and “`bar.rb`” and everything will still function correctly

Refactoring (IV)

- The final refactoring is perhaps not necessary at this stage
 - but the book recommends organizing step definitions according to the primary domain entity they operate on
- So, they recommend moving our three step definitions into
 - `account_steps.rb`
 - `teller_steps.rb`
 - `cash_slot_steps.rb`
- Doing this at this stage, however, may impose some pain if our domain model is still in flux (which it likely is at this point)

One More Thing: Teller.withdraw_from()

- Our scenario passed but there's a problem with Teller.withdraw_from()
- Here's the code
 - **def withdraw_from(account, amount)**
 - **@cash_slot.dispense(amount)**
 - **end**
- What's the problem?

The Case of the Unused Parameter

- The account parameter is not being used
 - We are not actually withdrawing the money from the account!!
 - Our system, in its current state, can be used to dispense any amount of money from the bank
- How did we miss this?!
 - Our scenario didn't check for it!

New Scenario

- Feature: Cash Withdrawal
 - Scenario: Successful withdrawal from an account in credit
 - Given I have deposited \$100 in my account
 - When I withdraw \$20
 - Then \$20 should be dispensed
 - **And the balance of my account should be \$80**
- No one is to blame for this; it can be an easy thing to miss
 - It would have surfaced eventually but the unused parameter was enough to point the way

Cucumber Guides the Way

- Running cucumber we are back to the scenario being undefined
 - Take its suggestion for the step definition and add it to `account_steps.rb`
 - Update it to use our Transform
 - And copy the assertion from the first step definition into it
 - It already does what we need, checking the amount with the balance
 - We will need to fix the duplication of the assertion later
- Running cucumber again provides us with a failing scenario
 - Time to fix the code!

Fixing the Code

- First, we head to our teller class
 - He needs to debit the amount of the withdrawal from the account
- Second, we head to our account class to add a **debit()** method to it
- The result?
 - Test passed!
- DEMO

- Are we done?

Not Quite!

- After we make a change to our system, we should check to see if there are opportunities for refactoring
 - especially when we are making a change to make a failing test case pass
 - often times we are in a hurry to fix the failing test case
 - when we are rushed, we can make changes that do not fit with the existing design of our system
- Looking at our Account class, for example, we can see that we have a method called deposit() and another method called debit()
 - The opposite of debit is credit, but we don't have that method
 - And deposit doesn't really deposit funds, it initializes the account
- It is a prime target for refactoring!

Note: we won't fix this second problem until chapter 9

Refactoring Account (I)

- Let's change the name of the deposit() method to credit()
 - We have test cases, so let's just run cucumber to **see what breaks**
- The very first step definition breaks
 - because Account.deposit() no longer exists
- Since all we did was change the name of the method, we change it here and
 - everything works again
- However, the step and the step definition are no longer in sync
 - We talk about depositing in the step but we are crediting in the def
 - This issue goes back to ubiquitous language; we want consistent terms

Refactoring Account (II)

- We need to change the step from
 - Given I have deposited \$100 in my account
- to
 - Given my account has been credited with \$100
- And, once we do that, we must change the step definition to match
 - While we are there, we can also take care of the duplicate assertion
 - We will simply credit the account and let some other step take care of checking the balance
 - which is already in place, since realizing this other step was missing is what got this refactoring started in the first place!

All Done

- With this refactoring, we are back to our scenario passing and we're confident that our system is as simple and as organized as we can make it
 - To make the system evolve, we just need to add additional scenarios
- The book provides some criteria to try to assess whether we are at a good state given the tests we have. These criteria were created by Kent Beck and specify **what it takes to achieve a simple design**; A software system's design is simple if
 - It passes all tests
 - It reveals our intentions
 - It contains no duplication
 - It uses the fewest number of classes and methods

Summary

- Learned more about how to use Cucumber with a more extensive example
 - Transforms are a useful way to remove repetitive code from step definitions
 - We can assign names to transforms and use those names in the regular expressions of our step definitions
 - This allows us to avoid a second type of duplication
 - Step definitions share state via the customizable World object
 - features/support is used to store code that wires step definitions with the underlying system; support/env.b is used to boot up the testing process
- The next part of the example will add a user interface to the application and will help us test it as well

Coming Up Next

- Lecture 18: Review of Midterm
- Lecture 19: Software Transactional Memory