# Cucumber: Finishing the Example

CSCI 5828: Foundations of Software Engineering
Lecture 23 — 04/09/2012

# Goals

- Review the contents of Chapters 9 and 10 of the Cucumber textbook
  - Testing Asynchronous Systems
  - Testing Databases

# Before We Get Started: Update Gems

- Our example will make use of a new gem called “service_manager”
  - To make sure we can use it, we add
    - gem 'service_manager', '0.6.2'
  - to our Gemfile and then run “bundle install” to make sure our environment is ready

# ATM: Continuing the Example (I)

- At the end of our last lecture, our ATM system was implemented to handle
  - a single scenario, where $20 is withdrawn from a $100 account
- The system itself was implemented as a web app
  - using the Sinatra web service framework
  - instances of the domain model are shared between the test code and the web app
  - Capybara was used to test the web app
- All of this occurred behind the abstraction of step definitions that
  - only refer to the problem domain
  - not a particular implementation or UI of a system

# ATM: Continuing the Example (II)

- We will now increase the complexity of our implementation
  - to demonstrate how to use Cucumber
    - to test systems that have
      - asynchronous components and
      - databases
- With respect to the former, when the system-under-test is asynchronous
  - we have to deal with the fact that our test code will
    - ask for an operation to be performed
    - and then somehow detect when this has happened

# An Asynchronous ATM

- To make our ATM example asynchronous, we will
  - create a “repository” class that holds the current balance of the account
  - create a transaction queue class to hold credit/debit transactions for the account
  - create a transaction processor that pulls transactions off the queue and updates the balance
- This means when a debit or credit is performed,
  - the balance is NOT updated immediately
  - instead, a new transaction is put on the queue
    - where it will be handled at some point in the future by the processor

# Implications

- The primary concern with testing this new system is
  - synchronizing test code with the actions of the system
- I can no longer perform a debit and then immediately check the balance
  - it is likely that our debit transaction is still on the queue
  - as a result, the balance will not match our expectations
- This type of asynchronous interaction can lead to flickering scenarios
  - sometimes they pass and sometimes they fail depending on the timing
- As a result, we must provide a way for the test code
  - to synchronize with the ATM

# Two Approaches

- There are two options for adding this sort of synchronization
  - We can listen
    - With this technique, the system is engineered to generate events
    - The test code registers for the appropriate events and performs an operation
      - it then blocks until the appropriate event has been generated
      - or fails with a timeout if a problem causes the system to crash
  - We can sample
    - With this technique, we loop, polling the system until we detect that the change we were waiting for has occurred
      - This is known as a “busy wait” and is not as efficient as the former technique, but it is easier to implement

# Updates: Account

- The first change to our existing system is to change our Account class to make use of two new objects
    - the repository (the balance store) and the transaction processor
- The balance method will simply query the repository for the latest value
- The credit and debit methods will add a new transaction to the queue
    - Transactions are strings that look like this
        - “+20”, “-45”, “+100”, “-60”, etc.
- **DEMO**

# Updates: Transaction Queue

- The Transaction Queue is implemented simply
  - It creates a directory called messages
    - and stores each transaction as a file in that directory
    - the name of each file is the transaction id
    - ids start at 1 and increase indefinitely
    - when a transaction is read, its corresponding file is deleted
- **DEMO**; NOTE:
  - self.clear is a static method
  - File.open takes a block and passes the newly opened file to that block
  - read is meant to be invoked by an iterator
    - each time through it “yields” a string
      - which invokes a block passed in by the caller to process the string

# Updates: BalanceStore

- BalanceStore is a simple class that stores the current value of the account in a text file
  - A request for the current balance
    - results in reading the file, converting its contents to an integer, and returning that value
  - A request to update the balance
    - Invokes File.open (deleting the existing file) and writing the new value as a string
- **DEMO**; NOTE:
  - The method balance=() takes advantage of ruby's ability to convert
    - a.balance = 20
  - to
    - a.balance=(20)

# Updates: Transaction Processor

- The transaction processor can now be written
  - It makes use of both the BalanceStore and the TransactionQueue
  - It has a simple design
    - It loops forever
      - calling read() on the TransactionQueue
      - It sleeps for 1 second # to ensure our test fails
      - It converts the transaction to an integer
      - Calculates the new balance
      - Writes the new balance to the balance store; **DEMO**

# Making Sure Our Scenario Doesn't Leak

- Since our classes now create files in the file system
    - it is possible for our scenario to "leak" data
- In this context, that would mean, for instance,
    - running scenario A which leaves the account with a $500 balance
    - running scenario B which assumes the account starts with a balance of $100 but instead starts with a balance of $500
        - because scenario A forgot to clean up after itself
- We will use a hook to make sure that both classes delete any files that might have been created by previous scenarios (we'll also remove our previous hook)
    - The hook will set the balance to zero and clear the queue; **DEMO**

# Configuring Service Manager (I)

- To make our transaction service asynchronous
  - we will run it in a separate process
- That’s where Service Manager comes in (the gem we installed on slide 3)
  - We provide it with a config directory that tells it
    - what program to invoke
    - how to tell if that program was successfully invoked
    - and a few other details
  - We then make sure we start up the service manager when our tests are being run

# Configuring Service Manager (II)

- First, we create the config directory in the top level of our ATM project directory
  - **DEMO**
- Then, we create a file called services.rb in our features/support directory
  - and have it start-up the ServiceManager; **DEMO**
- This will, in turn, cause it to read its config information and launch our transaction processor
  - The transaction process will loop waiting for files to appear in the transaction queue's messages directory
- When we are shutting down, the ServiceManager will also shutdown the process that's running the transaction processor automatically

# You Know the Drill

- It’s finally time to run cucumber again
    - and ...
        - WATCH IT FAIL!
- Our system flies through
    - creating the account with $100
    - and withdrawing $20
    - But then fails when it tries to read the balance
        - It expected $80 but the balance is $0
- Looking in the messages directory after the test shows two unprocessed transactions: 1 with value “+100” and 2 with value “-20”

# Why Did It Fail? (I)

- The lack of synchronization (really the “sleep 1” statement)
    - Here, the transaction processor is sleeping for one second
        - while cucumber runs its test
        - the scenario will ALWAYS fail
    - We can flip where the sleep statement is
        - If we take it out of the transaction processor
        - and add it to the step definition, the scenario will ALWAYS pass
            - that’s because the processor has more than enough time to process the transactions while the step definition sleeps

# Why Did It Fail? (II)

- To truly see the race condition, we can do the following
  - take out the sleep statement altogether
  - run the following command
    - ruby -e “30.times { system ‘cucumber -f progress’ }”
- This runs the test 30 times in a row and
  - sometimes the scenario fails
  - and sometimes the scenario passes
- It flickers!

# How to Fix?

- We are going to use the sampling method to synchronize with the transaction processor
    - We're going to add a new method to the world object called "eventually"
        - eventually will run a block over and over until
            - it returns true, meaning the condition we were looking for occurred
            - or a time limit is exceeded, we then throw an exception causing the scenario to fail
    - We then change our final step definition, to pass its check that the balance is equal to $80 to the new eventually method; **DEMO**
- Now, if we run the test 30 times in a row, all tests pass!

# Next Up: Databases

- Now, we are going to update our ATM to use a database to keep track of the balances of multiple accounts
  - We're going to use a framework called ActiveRecord—developed as part of Ruby on Rails—to create an sqlite3 database
  - ActiveRecord makes accessing a database really easy
    - as long as you follow its conventions
      - A class called Account is stored in the accounts table
  - The class looks like this
    - class Account < ActiveRecord::Base
    - end
  - At run time, the class is dynamically modified to contain methods that allow access to the associated database table

# Update Gems

- Once more, we need to update our Gemfile
  - This time we add the gems for ActiveRecord and sqlite3
    - gem 'activerecord', '3.1.3'
    - gem 'sqlite3', '1.3.5'
  - Run “bundle install” to download these packages and their dependencies

# Updates: New Account Class

- We move the Account class out of nicebank.rb and make it an ActiveRecord subclass
    - It will associate with a database that has three columns:
        - id: unique id for each record, autogenerated by ActiveRecord
        - number: a unique account number
        - balance: the current balance for that account
    - We get rid of our file-based BalanceStore class
        - but still use the transaction processor to update the balance of an account
    - **DEMO**

# Creating the Database

- ActiveRecord makes use of a concept called migrations
  - to make sure a program is using the correct version of a database
  - One possible migration is to indicate how to create the database
    - if a database file doesn't exist when we start our program
- We place this migration in db/migrate
  - the migration itself contains code that describes the database and how to create the accounts table
  - DEMO

# Updates: Get Rid of BalanceStore

- We will be using a database now
  - so we don’t need our BalanceStore class
- We delete it
  - and update hooks.rb to no longer use it to initialize the balance of our account to zero
- Other clean up
  - We now need to tell the code in nicebank.rb and transaction_processor where the account class is located
    - We use a require_relative statement to handle that
    - And remove any remaining references to the BalanceStore class

# Run It To See It Fail

- We now have enough code in place to try running cucumber
  - We will see ActiveRecord notice that the database doesn’t exist
    - It will kick in and create it using the migration we defined
    - (All of this done automatically, via convention)
  - The scenario will then fail because
    - we haven’t updated the transaction processor to make use of the database
    - and there are still references to balance_store in our code, even though we got rid of the BalanceStore class
- If you run “strings db/bank.db”, you’ll see it indeed has an accounts table

# Updates: Transaction Processor

- We need to update the transaction processor
    - It now receives messages of the form
        - <amount>,<account number>
    - We have to parse out the amount and account number
    - Retrieve the account from the database
    - Update its balance
    - Save the change back to the database
- **DEMO**

# Updates: World Object

- Our account has a field called “number”
  - confusingly, its type has been set to string
- Currently, when we create an account, we are not assigning a value to this field, and so are transactions look like this:
  - “+100,” and “-20,”
- We will now change our world object to create an account whose “number” is set to “test”
  - **DEMO**
- We are doing this to demonstrate a few features about ActiveRecord and a difficultly about testing databases

# Failures (I)

- First, our validation step fails because
  - we create an account with a zero balance in our web app
  - we perform two transactions on it (add 100; subtract 20)
    - those get performed in a separate process by the transaction processor
  - we then check to see if the balance is $80 but our account object’s balance stays at $0
    - it doesn’t know the balance was changed by a different process
- To fix: we add a statement to reload the account’s values from the database
- We run cucumber again and...

# Failures (II)

- We fail again
    - This time our database validation code complains when we try to create another Account object whose “number” equals “test”
        - The reason
            - we told ActiveRecord that field must be unique
            - and we already have an account with that number
                - which was created on the PREVIOUS test
- We’re dealing with a leaky scenario
    - where results from a previous run of the scenario have leaked through to this run, causing it to fail!

# How to Fix? (I)

- We need to make sure we start each scenario with a clean database
  - We can do this one of two ways
    - transactions or truncation
  - With the transaction approach
    - you create a transaction at the start of your scenario
    - you perform a bunch of changes
    - test the result
    - and then roll the transaction back
    - all changes then “go away” because they don’t get committed
  - The problem?
    - Our system has two separate processes and two separate connections to the database; with transactions, they can’t see each other’s changes

# How to Fix? (II)

- With truncation, you simply make sure that your database is set back to its initial state (truncated)
    - You do this by ensuring that all tables have all of their data deleted
- The book uses a gem called Database Cleaner to take care of this
    - We’ll just use ActiveRecord directly
        - in a hook
            - to execute a “truncate table” command directly on the accounts table
- In our hook, we need to say:
    - ActiveRecord::Base.connection.execute("DELETE FROM accounts")
- We’ll put our hook in features/support/database.rb
    - **DEMO**

# Summary

- With this lecture, we reach the end of a detailed example for Cucumber
  - Remember, throughout this entire lecture, everything we did last time
    - launch a web service
    - load “/” to get a form, interpret the form, and submit a withdrawal
    - share domain objects between test code and web app
  - still occurred every time we launched the test
    - we then added an asynchronous transaction processor
    - and a database
- and via the abstraction provided by the world object, the text of the scenario never changed and never directly references any of the implementation

# Coming Up Next

- Lecture 24: The Agent (or Actor) Model of Concurrency
- Lecture 25: Creating Agile Software