Module 8
Test Driven Development

CS W169A: Software Engineering

1 Overview

In our discussions of the Agile lifecycle, we discussed two aspects of software assurance: validation ("Did you build the right thing?") and verification ("Did you build the thing right?"). Behavior Driven Design helps us communicate effectively with the customer to perform validation. In this worksheet, we'll focus on verification, specifically how to check if we've built the thing right using software testing.

2 FIRST in Practice

Recall, the FIRST principles are a set a guidelines for writing precise and effective unit tests. FIRST stands for fast, independent, repeatable, self-checking, and timely. Given the code below, determine what FIRST principle is not being followed, and why. Determine how you might edit the test to resolve its inadequacy.

In the `HomeController.rb` file

```ruby
def index
  if Time.now.tuesday?
    render 'special_index'
  else
    render 'index'
  end
end
```

In the `HomeControllerSpec.rb` file

```ruby
it "should␣render␣special␣template␣on␣Tuesdays" do
  get 'index'
  if Time.now.tuesday?
    response.should render_template('special_index')
  else
    response.should render_template('index')
  end
end
```

The main issue with the above test is that it is not Repeatable. The test is checking for whether a special template is rendered on Tuesday. However, the test output is dependent on what day of the week it is when the test is run. To elaborate, running this test on Monday would require `render_template['index']`, but running on Tuesday would require `render_template('special_index')`. This inconsistency violates repeatability.

To fix this, we could edit the test so that we can inject a mock date and verify it does the right thing given the fake date. You could then write a comprehensive test that tests for each day of the week with fake dates. Note that simply creating a separate test for each day of the week would not work, as the test results would still depend on the current date / day of the week, something out of our control.

3 Calling Triple A

The most popular pattern for the anatomy of a unit test is the Arrange, Act, Assert method. A simple example of AAA in JavaScript: In this exercise, let's implement each of these stages for a `BankAccount` class.

Here is the `BankAccount` class code:

```
class BankAccount
  attr_reader :total

  def initialize(amount)
    @total = amount
  end

  def cash(amount)
    @total += amount
  end
end
```

Fill in the below unit test, following the Triple A paradigm, to test whether the `cash` method works. Initialize the bank account with amount 100. `Cash` an amount of 1. Indicate that the expected total after `cash` is called is 101.
Solution:

```
RSpec.describe BankAccount do
  # Arrange Here
  subject { BankAccount.new 100 }

  describe '#cash' do
    it 'adds␣the␣passed␣amount␣to␣total' do
      # Act Here
      subject.cash(1)
      # Assert Here
      expect(subject.total).to eq(101)
    end
  end
end
```

4 TDD with RSpec

We've already seen a variant of the AAA anatomy in Cucumber, where the preconditions are typically Given steps, the actions taken are typically When steps, and the postconditions are typically expressed in Then steps.

In the RSpec framework, each test case is called an example, and although the terminology is a bit different, the concepts are the same. Each example will:

- Arrange: Set up some preconditions. Like Cucumber, preconditions can be placed in a `before(:each)` block that is run before each example.
- Act: Execute some code (i.e. call class method)..
- Assert: Check 1+ expectations to verify behavior correctness. RSpec can express many expectation types::
  - Equality: Computed value equals some known value

- Set Inclusion: Computed collection includes some element
- Method Invoked: Particular method should get called
- Mutation: Some expressions' values should have changed (i.e. collection length after new element).

Testing Concept	**In Cucumber/Capybara**	**In RSpec**
Single test case	`Scenario` block	example using `it` (or the alias `specify`)
Collection of related test cases	Feature (`.feature` file)	`describe` or `context` group of examples
Preconditions within a test	Steps identified by `Given` keyword	Statements iwthin example that setup pre-conditions
Preconditions used by many related test	`Background` section of feature file	`before(:each)` section of a `describe` or `context` block
Postconditions	Steps identified by `Then`	expectations, often using RSpec `expect`

4.1 Practice RSpec TDD

Given a class definition for a `LinkedList`, with two methods:

1. `add`: Adds an item to the list
2. `count`: Returns number of items in the list

Write RSpec tests for both methods.

Solution:

```
describe "LinkedList" do
  describe "adding␣an␣item" do
    it "should␣increase␣the␣count␣by␣1" do
      @list = LinkedList.new
      expect { @list.add('x') }.to change { @list.count }.by(1)
    end
  end
  describe "#count" do
    it "should␣start␣at␣zero" do
      expect(@list.count).to eq(0)
    end
  end
end
```

Passing a block `@list.add('x')` to expect or change delays the evaluation of those expressions. Why is this necessary in this example?

The change expectation needs to be able to execute `@list.count` before `@list.add`, to record the original value, and then again after `@list.add`, to record the new value. By making the argument to expect an anonymous lambda (procedure), we allow change to call this procedure after first calling `@list.count`. By making the argument to change an anonymous lambda, we allow change to call `@list.count` multiple times, that is, before and after `@list.add`.

5 Seams

Suppose you have a social messaging network app in which each Member maintains a list of friends (also Members). When any Member calls `tell_my_friends(message)`, the app should arrange to deliver that message by calling the `hear_news(message)` on each friend. The `hear_news` method propagates the message to their friends as well.

```
class Member
  def tell_my_friends(message)
    # notify all my friends of the message
  end
  def hear_news(message)
    # this gets called when one of my friends posts a message
    #   it makes sure the message is also forwarded to their friends
  end
  def friends
    # returns a collection of all my friends, each is a Member
  end
end
```

How would you test that this code works correctly? **HINT**: your strategy should be to "spy" on the `receive_news` method for each friend to make sure that it is called on all my friends. Note that you have to set up a "spy" for that method using the `receive()` expectation *before* you call the method that might generate those calls.

Solution:

```
describe "sending␣news" do
  before(:each) do
    @person = Member.first
    @message = Message.new("Great␣news!")
  end
  it "sends␣the␣news␣to␣all␣my␣friends" do
    @person.friends.each do |friend|
      expect(@person).to receive(:hear_news).with(@message).and_call_original
    end
    @person.tell_my_friends(@message)
  end
  # bonus: ensure that all Members who are NOT my friends
  #   do NOT receive the news
  it "does␣not␣send␣the␣news␣to␣people␣who␣are␣not␣my␣friends" do
    @not_my_friends = Member.all - @person.friends  # set difference!
    @not_my_friends.each do |stranger|
      expect(@person).not_to receive(:hear_news)
    end
  end
end
```

Use `and_call_original` to call the unstubbed method to get everyone to call their contacts. If you did not include it, since `expect().to receive` creates a seam, calling `hear_news` on someone will not cause them to continue calling people. Food for thought: What happens if you omit `with(@message)` from the expectation in line 8? What happens if you include it in the expectation in line 17?

6 Fixtures & Factories

To test effectively, we will need some mock data. That's where factories and fixtures come into play! A fixture is some static data that can be preloaded as test data before any tests are run. A factory is a method that creates a particular kind of object just-in-time when you need it for a particular test. Read submodule 8.6 of the textbook for a great shortlist of testing keywords.

6.1 Conceptual

Discuss pros and cons regarding the use of either Factories or Fixtures in tests.

Solution:

Fixtures let you keep all your test data in one place, but they can result in tests becoming dependent on fixture data without you realizing it. For example, a test that happens to rely on counting the number of items in a certain table may break if the fixtures for that table are changed, even though the code being tested hasn't changed. Factories are are helpers to make objects with default attributes if needed per test. Since factory data is associated with individual tests or small groups of tests (such as a describe or context block), tests are more likely to stay Independent.

Both fixtures and factories have the drawback that you sometimes have to set up complex relationships for a test. For example, in testing a social network app, you might have to set up several "users" and establish the necessary relationships between them so that some of them are friends. Factory frameworks, such as FactoryGirl for Rails, can greatly simplify such tasks.

6.2 Mock Data Case Study

6.2.1 Manual Creation

Suppose we have a `Animal` model with a single `name` attribute. If I want to test objects of the model with both valid and invalid states, that's pretty straightforward with manual data creation:

```
valid_animal = Animal.new(name: 'Harold')
invalid_animal = Animal.new(name: '')
```

However, this approach can become tediously and unscalable when dealing with more complex class inheritance and models with more variegated fields. For instance, let's say a `Zoo` model is made up of a `safari_zone` and a `jungle_zone`, both of which have multiple `Animal` objects. Suppose zoo also has additional fields including `isOpen`, `ticket_price`, and `capacity`.

```
sf_zoo = Zoo.create!(
    safari_zone: [
        Animal.create(name: 'Zach', species: 'Zebra', isNocturnal: false),
        Animal.create(name: 'Harry', species: 'Hyena', isNocturnal: true),
    ],
    jungle_zone: [
        Animal.create(name: 'Harold', species: 'Hippo', isNocturnal: false),
        Animal.create(name: 'Martin', species: 'Monkey', isNocturnal: true),
        Animal.create(name: 'Fred', species: 'Frog', isNocturnal: false),
    ],
    isOpen: true,
    ticket_price: 50,
    capacity: 3400,
)
```

Imagine having to come up with the above and more. We had to spend time arbitrarily coming up with details that may not even necessarily be relevant. What's worse is that if anything about the `Animal` or `Zoo` models change, our old data may become unusable, or at least requires updating.

6.2.2 Factories

The advantage of the factory is that you have methods that generate new class instances for you. With the Factory Bot gem (link), we can create instances of a class while only having to specify values relevant to our tests.

Let's say we writing a test on the `Zoo` object that counts the number of animals that will be awake during the Zoo's park hours based on whether the animal is nocturnal or not. Instead of manually creating a zoo object like the above, we could do the following:

```
zoo = FactoryBot.create(
    :zoo,
    safari_zone: [
        FactoryBot.create(:Animal, isNocturnal: false),
        FactoryBot.create(:Animal, isNocturnal: true),
    ],
    jungle_zone: [
        FactoryBot.create(:Animal, isNocturnal: false),
        FactoryBot.create(:Animal, isNocturnal: true),
        FactoryBot.create(:Animal, isNocturnal: false),
    ],
)
```

Once again, we're specifying only the details that are relevant to the test! An added bonus is that even if the `Zoo` model is updated with more fields or existing fields (not `isNocturnal`) are changed, the Factory Bot mock is unaffected and the test based on this mock object would still work!

6.2.3 Fixtures

Fixtures add a layer of abstraction between the data and the mock objects by placing the test data in a separate YAML file. For the above example, we might do something like the following:

In `config/database/animals.yml`

```
zebra:
    name: 'Zach',
    species: 'Zebra'
    isNocturnal: false,
...
```

Then, in the test file

```
zoo = Zoo.create!(
    safari_zone: [Animal.create!(:zebra), Animal.create!(:hyena)],
    jungle_zone: [Animal.create!(:hippo), Animal.create!(:monkey),
        Animal.create(:frog)],
)
```