Automated Unit Testing

Topics Covered: RSpec, unit testing, mocking, stubbing.

In this session we will introduce you to the process of unit testing code written using the Ruby on Rails web framework.

If you have difficulties please refer to the material on https://learn.shefcompsci.org.uk, the Rails guides at http://guides.rubyonrails.org, or ask one of the demonstrators.

Please Note:
This class requires a specific sample application, as it contains predefined unit tests required to complete the tasks within this worksheet. Please clone the sample from the following repository:
`git@git.shefcompsci.org.uk:com3420-2017-18/materials/unit-testing.git`
Change into the **unit-testing** directory and run the following command:
`bin/setup_labclass`
in order to setup the project.

Unit Testing

There are many different types of test and each has its own specific purpose. Unit testing is used to test small components of an application (i.e. a single method) in isolation from the rest of the application. Unit tests help to ensure that each method is doing its job correctly. One of the main benefits of unit testing is being able to identify exactly where in your application a bug is caused. When a test that covers a much larger quantity of an application fails it can be difficult to find the exact location of the problem.

RSpec Unit Test Structure

Open **spec/models/product_spec.rb**. The file will look something like this:

```
require 'rails_helper'

describe Product do

  describe '#value_of_stock' do

    it 'calculates the total value of the stock' do
       product = Product.new(cost: 10, stock_level: 300)
      expect(product.value_of_stock).to eq 3000
    end

  end

...
```

This is an example unit test that has already been created for you. The testing framework that we are using is called RSpec. RSpec is one of the most popular testing frameworks used with Rails, but there are several to choose from.

- **require 'rails_helper'** – By default a test file runs independently. We need this line to load the Rails framework and all of the application code.
- **describe Product** – This identifies which class we are testing. In this case it is the *Product* model.
- **describe '#value_of_stock'** – This is the method we are testing. The # is a convention used to indicate that it is an instance method. If we wanted to test a class method (i.e. `def self.some_method`) then we would write a '.' instead (e.g. `describe '.some_method').`

In this case the **value_of_stock** method is designed to calculate the total sales value if you sold all of the product. This is done by multiplying the stock level by the cost of the product.

- **it 'calculates the total value of the stock'** – This is a **test case**. In here we will write test code for this case.
- **product = Product.new(cost: 10, stock_level: 300)** – This creates a new instance of Product, with cost 10 and stock level 300. Most tests will require some setup like this. You need to know the state of the object that you are testing in order to know the outcome you are expecting.
- **expect(product.value_of_stock).to eq 3000** – This is an **assertion.** An assertion is an expression that will be either true or false. It is how you indicate the **expectation** of the test case. If the expectation is met then the test case will pass; if not it will fail.

This test case expects **`product.value_of_stock`** to equal 3000. It will create a new instance of Product then call **`product.value_of_stock`**. If the method returns 3000 the test case will pass; if it returns anything else it will fail.

Running Tests

You can run all of the unit tests in this file with the command:

```
bundle exec rspec spec/models/product_spec.rb
```

The output will show **passing** test cases as a green dot and **failing** test cases as a red F. You will also see **pending** test cases as a yellow *. Pending test cases are those you have started writing but not finished. There are a few different ways you can mark a test case as pending, but one of the most common is to type **skip** at the top. Note that an empty test case will pass unless it is marked as pending. You will see many passing test cases although a lot of these are empty.

Below all of this you will see a list of all failing and pending test cases which provides further details such as line numbers.

Task 1 - Write a simple unit test

In the **spec/models/product_spec.rb** file you will see a unit test for the **`price_with_vat`** method which contains two empty test cases. The application uses the 'cost' field of a product to record the price without any VAT. This method calculates the price of the product once VAT has been added on. The VAT rate is hard-coded at 20%.

Complete the two empty test cases and check they pass.

The first case tests the typical scenario where a product will need VAT adding on at the standard rate. The second case tests the **edge case** where a product may have a value of 0. An edge case is a scenario where the parameters are on the extreme end of the expected boundaries or require special handling in some way.

Mocking and Stubbing

As previously mentioned, unit tests are used to test your code in isolation from other code. For the above examples this was straightforward as the methods had no dependencies. There will be cases however where the method being tested will interact with other classes. Mocking and stubbing are two techniques that can be used to ensure that unit tests always test code in isolation.

Mocking

Mocking involves using a different object in place of another. This is useful when the method being tested has a dependency on another object. A mock object simulates the behaviour of a real object, but we decide exactly how the mock object is going to behave so we can focus on the method we are testing.

An example from this application occurs when testing the **`shipping_and_handling`** method of the product model. The shipping and handling cost differs depending on the category of the product. Each category has a flat **shipping cost** and also a **handling fee** which is a percentage of the product's cost. The purpose of this method is to use these two figures to calculate the exact shipping and handling cost for a product.

The **`shipping_and_handling`** method is dependant on a category object, but we are not testing categories in this unit test. If we were to make an error in the category model we would want our category tests to fail, not our product tests. It is for these reasons that we would use a mock category object.

You can see an example of this in the **spec/models/product_spec.rb** file:

```
it 'calculates the shipping cost + the set percentage' do
  ...
  mock_category = double('category', shipping_cost: 10, handling_percentage: 5)
  ...
end
```

The **double** method creates a mock object which can be used in place of a category object. Mock objects are also referred to as **test doubles**.

- The first argument is a string which gives a name to the mock object. It does not affect the functionality in any way; it is simply a label to improve readability.
- The second argument is a hash which defines the methods the mock object should have and what each should return. This case requires two methods, **shipping_cost** which will always return 10, and **handling_percentage** which will always return 5.

Stubbing

Stubbing involves telling a method exactly what to return instead of carrying out its usual behaviour. This is useful when the method being tested calls another method. We are able to **stub** the method call to return exactly what we expect so we can focus on the method we are testing.

Stubbing is very similar to mocking. The difference is that instead of creating another object with expected behaviour, we are making one method return exactly what we expect.

The **`shipping_and_handling`** unit test shows an example of this:

```
it 'calculates the shipping cost + the set percentage' do
  ...
  allow(product).to receive(:category).and_return(mock_category)
  ...
end
```

The test case creates a mock category object to use in place of a real category object, but we need to do something to make the product instance use it. This can be done by stubbing the **category** method of the product to make it return the mock category object.

- **allow(product)** – Pass in the object on which stubbing will be performed.
- **.to receive(:category) –** Identify the method to stub.
- **.and_return(mock_category) –** Pass in what is to be returned by the stub method.

In this test case **product.category** will always return the **mock_category** object. This ensures we know the exact values the **shipping_and_handling** method will receive, so we can define an assertion for the test case.

Task 2 - Write some unit test cases using mocking and stubbing

In the **spec/models/product_spec.rb** file you will see three empty test cases for the **`shipping_and_handling`** method.

Use mocking and stubbing to complete these test cases and check they pass.

Task 3 - Add a new method with a corresponding unit test

Create a new method in the product model called **value_of_stock_including_vat**. Write the necessary code to calculate that figure, which can make use of the existing **price_with_vat** method.

Write a new unit test with some test cases to check the new method and ensure it behaves as expected. If you used the existing **price_with_vat** method within the new method you should consider stubbing it in the test cases. The focus of unit testing is to test code in isolation; if you were to use **price_with_vat** in the new method without stubbing then a bug in it would cause the test cases for both methods to fail.

Additional tasks

The below tasks can be completed during the class if you complete the above tasks before the end of the class, or can be completed outside of the class for additional experience of writing unit tests.

Task 4 - Handle changing VAT rates

In recent years the standard VAT rate has changed a number of times. Create a new method in the product model called **current_vat_rate** which takes a date as a parameter and returns the standard VAT rate applicable for that date (see the table below). Add a unit test with appropriate test cases for this method. Once you are satisfied that the method works correctly, amend the **price_with_vat** and **value_of_stock_including_vat** methods to use it instead of the hard-coded VAT rate. Remember to update the corresponding unit tests and check they still pass.

From	To	Standard VAT Rate
19/03/1991	30/11/2008	17.5%
01/12/2008	31/12/2009	15%
01/01/2010	03/01/2011	17.5%
04/01/2011		20%

Task 5 - Handle special VAT rates

Certain categories of products are subject to a reduced VAT rate or a zero VAT rate (see https://www.gov.uk/vat-rates). Add a database migration and amend the category model so that it has a field to record the VAT rate for these categories of products. If the category is not subject to a reduced/zero rate will we leave the field empty and use the method created in task 4 above to determine the applicable standard VAT rate. Update the appropriate methods to use this new field, and add unit test cases to ensure that VAT is calculated correctly for reduced/zero rate products.

Task 6 - Refactor

As all of the code in the application is tested, it should be safe to refactor. Consider if there is any refactoring that can be done to the methods in your product model to avoid duplicated code or make them more efficient. Once your have done so check whether the tests still pass. If there are any failures you may need to update the tests or fix the code.