Module 8 - Test Driven Development

Topics covered:

- Introduction to Module 8: Test Driven Development
- Practice questions for module 8

TDD: Test Driven Development

- TDD: Practice of first writing tests before you write new functional code
- TDD is an aspect of agile software development
- In agile development, unlike plan-and-document, the role of the Quality Assurance team (QA) team is not to implement all the tests.
- Testing is a shared responsibility of the agile team.

TDD FIRST Principle

- The FIRST principle in TDD lists a set of agreed upon features of a good test.
  - **Fast**: test cases should be easy and quick to run.
  - **Independent**: The order in which we run tests should not interfere with the test results.
  - **Repeatable**: Test behavior should not be influenced by external factors such as today's date.
  - **Self-checking**: Test should automatically report whether they failed/passed and should not rely on manual verification.
  - **Timely**: Test creation/update should not be postponed. Instead write/update tests at the same time you write/update code.

Red-Green-Refactor

- Red-Green-Refactor refers to the recommended sequence of steps for writing test in TDD
  1. Write test for the behavior you expect the code to have.
  2. Red step: Run the test and verify that it fails since the behavior has not yet been implemented.
  3. Green step: Write the simplest possible code that achieves expected behavior and passes the test.
  4. Refactor step: Refactor/optimize your code and your test while ensuring that tests are passing.

System Under Test

- System under tests (SUT) is a term that refers to the object being tested. The object could be a single method, group of methods, entire class etc.
- A test suite refers to a collection of test cases where each test case checks a specific behavior of a SUT.
- A unit test is a fine-grained test case for which the SUT is a single method.
- In general every test case follows the Arrange, Act, Assert (3A) structure

3A Test Case Structure

1. ***Arrange***: Set up necessary pre-conditions for the test case eg. dependency injection (ie setting up required variables).
2. ***Act***: Exercise the SUT.
3. ***Assert***: Verify the behavior matches expectation.

RSpec

- RSpec is a ruby testing framework.
- ***Describe*** block used to group together a suite of related test cases.
- ***Context*** block used to group together tests cases within a describe block that share the same state.
- ***It*** block used to specify a single test case.
- ***Before*** block is used to set up necessary pre-conditions.

How to set up a 3A testing structure for this controller method?

```
class MoviesController < ApplicationController
	def review_movie
		search_string = params[:search]
		begin
			matches = Movie.find_in_tmdb(search_string)
			if matches.empty? # nothing was found
				redirect_to review_movie_path , :alert => "No
			elsif matches.length == 1
				@movie = matches [0]
				render 'review_movie'
			else # more than 1 match
				@movies = matches
				render 'select_movie'
			end

		end
		rescue Movie::ConnectionError => err

			redirect_to review_movie_path, :alert => "Error contacting
		TMDb: #{err.message}"

		end
	end
end
```

Arrange

How to set up a 3A testing structure for this controller method?

```
class MoviesController < ApplicationController
	def review_movie
		search_string = params[:search]
		begin
			matches = Movie.find_in_tmdb(search_string)
			if matches.empty? # nothing was found
				redirect_to review_movie_path , :alert => "No
			elsif matches.length == 1
				@movie = matches [0]
				render 'review_movie'
			else # more than 1 match
				@movies = matches
				render 'select_movie'
			end

		end
		rescue Movie::ConnectionError => err

			redirect_to review_movie_path, :alert => "Error contacting
		TMDb: #{err.message}"

		end
	end
end
```

Arrange

Act

How to set up a 3A testing structure for this controller method?

```
class MoviesController < ApplicationController
	def review_movie
		search_string = params[:search]
		begin
			matches = Movie.find_in_tmdb(search_string)
			if matches.empty? # nothing was found
				redirect_to review_movie_path , :alert => "No
			elsif matches.length == 1
				@movie = matches [0]
				render 'review_movie'
			else # more than 1 match
				@movies = matches
				render 'select_movie'
			end

		end
		rescue Movie::ConnectionError => err

			redirect_to review_movie_path, :alert => "Error contacting
		TMDb: #{err.message}"

		end
	end
end
```

Arrange

Act

Assert

How to set up a 3A testing structure for this controller method?

```
class MoviesController < ApplicationController
        def review_movie
                search_string = params[:search]
                begin
                        matches = Movie.find_in_tmdb(search_string)
                        if matches.empty? # nothing was found
                                redirect_to review_movie_path , :alert => "No
                        elsif matches.length == 1
                                @movie = matches [0]
                                render 'review_movie'
                        else # more than 1 match
                                @movies = matches
                                render 'select_movie'
                        end

                end
                rescue Movie::ConnectionError => err

                        redirect_to review_movie_path, :alert => "Error contacting
                TMDb: #{err.message}"

                end
        end
end
```

4 cases to test in asserting phase. But it essentially depends on the an external data stream.

What should we do??

How to set up a 3A testing structure for this controller method?

```
class MoviesController < ApplicationController
       def review_movie
              search_string = params[:search]
              begin
                     matches = Movie.find_in_tmdb(search_string)
                     if matches.empty? # nothing was found
                            redirect_to review_movie_path , :alert => "No
                     elsif matches.length == 1
                            @movie = matches [0]
                            render 'review_movie'
                     else # more than 1 match
                            @movies = matches
                            render 'select_movie'
                     end

              end
              rescue Movie::ConnectionError => err

                     redirect_to review_movie_path, :alert => "Error contacting
              TMDb: #{err.message}"

              end
       end
end
```

4 cases to test in asserting phase. But it essentially depends on the an external data stream.

What should we do??

We have to find a way to control the behavior *of the call* to find_in_tmdb.

How do we do that?

How to set up a 3A testing structure for this controller method?

```
class MoviesController < ApplicationController
       def review_movie
              search_string = params[:search]
              begin
                     matches = Movie.find_in_tmdb(search_string)
                     if matches.empty? # nothing was found
                            redirect_to review_movie_path , :alert => "No
                     elsif matches.length == 1
                            @movie = matches [0]
                            render 'review_movie'
                     else # more than 1 match
                            @movies = matches
                            render 'select_movie'
                     end

              end
              rescue Movie::ConnectionError => err

                     redirect_to review_movie_path, :alert => "Error contacting
              TMDb: #{err.message}"

              end
       end
end
```

4 cases to test in asserting phase. But it essentially depends on the an external data stream.

What should we do??

We have to find a way to control the behavior *of the call* to find_in_tmdb.

How do we do that?

Isolating Code in Test

Some properties of a SUT may require us to isolate it when testing

Property of SUT	Testing strategy
SUT has ***depended-on components*** (DOCs) eg. methods in other classes.	Test case should isolate these dependencies.
SUT ***has side-effects*** on the application state.	Ensure relevant state set up before Act phase.Test case should verify correct side-effect occurred.
SUT relies on ***non-deterministic or time-dependent state*** of the environment it is executed on eg. Date/Time or PRNG.	Test case should control these values.

Seams

- A place where you can change app's behavior without changing the source code.
- Useful in testing - Isolate behavior of some code from that of the other code it depends on.
- expect… to receive OR allow..to receive uses Ruby's open classes to create a seam for isolating controller action from behavior of possibly buggy model method.
    - Example - Assume we add a POST action ***MoviesController.review_movie*** in RottenPotatoes Rails app that calls an external TMDb API using ***Movie.find_in_tmdb*** class method.
    - When writing a test case for the controller we want to use seams to isolate controller action from Movie.find_in_tmdb
- Stubs and Mocks are used to achieve this behavior, and Rpsec resets all mocks and stubs after each test case. Why?

Seams - Rspec example stubs Movie.find_in_tmdb

https://gist.github.com/u/saasbook

```
1 describe MoviesController do
2   describe 'looking up movie' do
3     it 'redirects to search page if no match' do
4       allow(Movie).to receive(:find_in_tmdb).and_return( [] )
5       post 'review_movie', {'search_string' => 'I Am Big Bird'}
6       expect(response).to redirect_to(review_movie_path)
7     end
8   end
9 end
```

Figure 8.6: This RSpec example (test case) stubs `Movie.find_in_tmdb` to isolate the controller action from its collaborators for the purposes of unit testing.

Seams

On line 4, we “override” the find_in_tmdb by setting up a method stub.

In this case instead of calling the real method we call the “fake” method whose behavior we can control for each test case.

Use the ***allow(Movie)*** call to set up a stub that does not track whether the ***find_in_tmdb*** method is called.

Use the ***expect(Movie)*** instead on line 4 if we want to ensure ***find_in_tmdb*** method is called. Also known as “Spies”.

All seams setup for a test case are reset after the test case is run.

https://gist.github.com/u/saasbook

```
1 describe MoviesController do
2   describe 'looking up movie' do
3     it 'redirects to search page if no match' do
4       allow(Movie).to receive(:find_in_tmdb).and_return( [] )
5       post 'review_movie', {'search_string' => 'I Am Big Bird'}
6       expect(response).to redirect_to(review_movie_path)
7     end
8   end
9 end
```

Figure 8.6: This RSpec example (test case) stubs `Movie.find_in_tmdb` to isolate the controller action from its collaborators for the purposes of unit testing.

Doubles a.k.a Mocks

- Doubles are appropriate when you need a stand-in with a small amount of functionality to isolate the code under test from its dependencies.
- Like all seams, it gives us just enough functionality to test a specific behavior.
- In our example, we will use it to make “stand-in” objects or “stunt-double” objects of a class.
- Our current code does not set any instance variables; Here is how we can use doubles to check if the value returned from a stub gets propagated correctly.

Doubles a.k.a Mocks

```
it 'makes search results available to template' do
  @fake_results = [instance_double('Movie'),
                   instance_double('Movie')]
  allow(Movie).to receive(:find_in_tmdb).and_return(@fake_results)
  post 'search_tmdb', {:search_terms => 'hardware'}
  expect(assigns(:movies)).to eq(@fake_results)
end
```

- assigns() rspec rails method for testing controllers
  - Pass symbol that names the controller instance variable
  - Returns value that controller assigned to that variable.

Seams and Doubles

- ***Test Double*** : generic term for any kind of object that is used in place of a real object for testing purposes.
- ***Fake*** : Object with working implementation that uses some sort of shortcut. Example: using an in-memory DB instead of a full RDBMS. Common in static, compiled languages like C and Go where monkey patching is difficult or not possible.
- ***Stubs*** : provide canned answers to calls usually not responding to anything outside what is programmed for the test. Example: Method stub that returns a given value always.
- ***Spies*** : Stubs that also record some information based on how they were called. Example: keep track of parameters passed in.
- ***Mocks*** : Pre-programmed to expect certain parameters which they should respond to in specific way.

Source: https://martinfowler.com/articles/mocksArentStubs.html

Stubbing out external services

- Reasons to stub out external services (even when doing full-system testing)
  1. ***Make tests fast***. Using stubs leads to faster calls since we avoid network calls.
  2. ***Make tests repeatable tests***. If we call external APIs that we do not control, our tests may fail due to factors not related to our code eg. if the service is down.
  3. ***Avoid abusing external services***. Invoking external services in test may overwhelm the service with unnecessary requests.
- WebMock - Ruby gem that is used to mock external services by intercepting remote API calls and responding with content from files.

Factories and Fixtures

- ***Factories***: Bits of code framework designed to allow quick creation of full-featured objects in tests.
- ***Factories*** typically provide valid instances of models that would pass validations tests.
- ***FactoryBot*** gem (previously called FactoryGirls) is used to create factories.
- ***Fixtures***: Rails default way used to prepare and reuse test data. Fixtures are defined in the spec/fixtures folder.
- Using fixtures is generally considered an anti-pattern except for data that is required for the app to run.

Coverage

- Code coverage is a measure of how much code an suite of automated tests is running.
- SimpleCov gem used to analyze code coverage in Rails apps.
- Command: rake stats used to display the ratio of number of lines of Rails application code to RSpec & Cucumber tests.

Terms in Test Coverage

- There are different code coverage standards\
  1. ***S0 coverage*** checks whether each method is invoked at least once in the test suite
  2. ***S1 coverage*** checks whether each method is invoked from every possible call site.
  3. ***C0 coverage*** checks whether each statement is executed at least once.
  4. ***C1 coverage*** checks whether each branch in code has been taken at least once.
  5. ***C2 coverage*** checks whether every code path possible has been exercised.
  6. ***Modified Condition/Decision Coverage*** (MCDC) combines a subset of above

Unit tests, Integration tests & System tests

- ***Unit tests***: lowest level of testing. Typically involves a single method.
- ***Integration tests***: middle level of testing. Typically involves multiple methods.
- ***System tests***: highest level of testing. Typically done using Cucumber.
- Beyond tests for correctness, there are other categories of tests: smoke tests (minimal tests that ensures important parts work), compatibility tests, regression tests, stress testing, accessibility testing.
- ***Black box*** tests have no knowledge of internal implementation, just check external contract while ***white box*** tests know internal implementation and check edge cases.
- ***Fuzzing***: testing using randomized inputs to check if system breaks.

Question 1: FIRST Principle

All of the following are general features of good tests. Which one is NOT referred to in the FIRST principle?

A. Tests should have short execution time.
B. We should be able to run tests in any order.
C. Tests should be easily readable.
D. Tests should not require manual verification.

Question 2: Red-Green-Refactor steps

In which stage of the Red-Green-Refactor practice do we first write functional code for the behavior under test?

A. Step 1
B. Red step
C. Green step
D. Refactor step

Question 3: 3A Test Structure

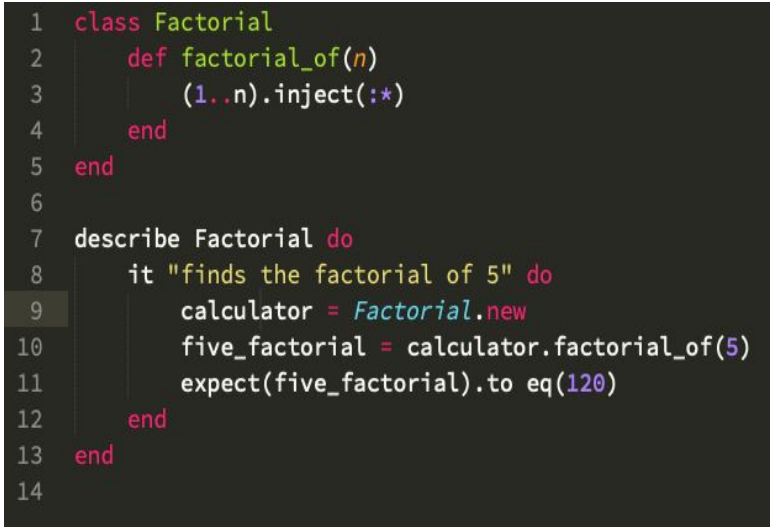
Consider the following lines of code and the corresponding RSpec test.

```
1  class Factorial
2      def factorial_of(n)
3          (1..n).inject(:*)
4      end
5  end
6
7  describe Factorial do
8      it "finds the factorial of 5" do
9          calculator = Factorial.new
10         five_factorial = calculator.factorial_of(5)
11         expect(five_factorial).to eq(120)
12     end
13 end
14
```

Question 3: Test Structure - Arrange

Which of the following lines MOST LIKELY corresponds to Arrange step?

A. Line 3
B. Line 9
C. Line 10
D. Line 11

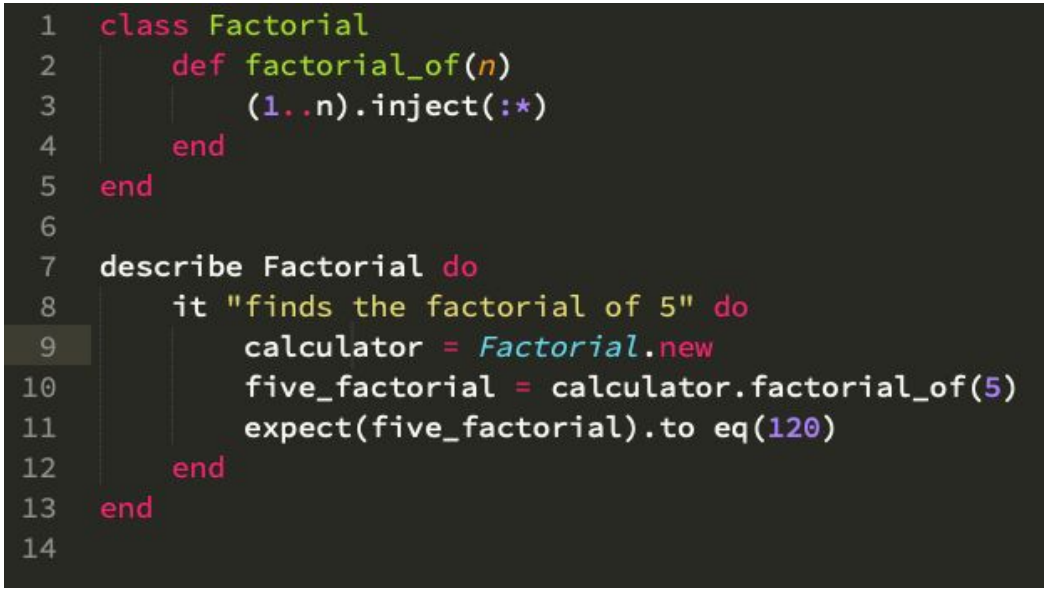


```
1  class Factorial
2      def factorial_of(n)
3          (1..n).inject(:*)
4      end
5  end
6
7  describe Factorial do
8      it "finds the factorial of 5" do
9          calculator = Factorial.new
10         five_factorial = calculator.factorial_of(5)
11         expect(five_factorial).to eq(120)
12     end
13 end
14
```

Question 3: Test Structure - Act

Which of the following lines MOST LIKELY corresponds to the Act step?

A. Line 3
B. Line 9
C. Line 10
D. Line 11



```
1  class Factorial
2      def factorial_of(n)
3          (1..n).inject(:*)
4      end
5  end
6
7  describe Factorial do
8      it "finds the factorial of 5" do
9          calculator = Factorial.new
10         five_factorial = calculator.factorial_of(5)
11         expect(five_factorial).to eq(120)
12     end
13 end
14
```

Question 3: Test Structure - Assert

Which of the following lines MOST LIKELY corresponds to the Assert step?

A. Line 3
B. Line 9
C. Line 10
D. Line 11

```
1  class Factorial
2      def factorial_of(n)
3          (1..n).inject(:*)
4      end
5  end
6
7  describe Factorial do
8      it "finds the factorial of 5" do
9          calculator = Factorial.new
10         five_factorial = calculator.factorial_of(5)
11         expect(five_factorial).to eq(120)
12     end
13 end
14
```

Question 4: Factories and Fixtures

Mark all of the following that are FALSE.

A. Factories create valid instances of a class that have some default values that you can selectively override for testing.
B. Fixtures are generally used for data that may be changed by test code whereas factories are used for data that may not be changed by test code.
C. Testing frameworks typically run “test teardown” after each test to truncate the database and keep tests independent.
D. Fixtures should typically be avoided in test except for cases when the data is required for the app to run at all.
E. None of the above.

Question 5: Coverage Terms

Which of the following statements is correct. Select ALL that apply.

A. C0 coverage checks whether each method is invoked at least once.
B. S0 coverage checks whether each statement is executed at least once.
C. S1 coverage checks whether a method is invoked at all places it is called.
D. C2 coverage checks whether each possible permutation of conditional variables and hence paths in code are tested.
E. None of the above.

Question 6: Testing levels

Which of the following statements are TRUE. Mark all that apply.

A. Unit tests have excellent error localization since they only target single methods.
B. We frequently use test doubles in system tests to make them run faster.
C. Integration tests frequently have better coverage resolution than system tests since they test more code paths.
D. Unit test cases run faster than integration test cases since integration tests use test doubles more extensively.
E. None of the above.

Attendance

Attendance link -

https://tinyurl.com/discussion-week-9

Passcode: FIRST