Lab 6 - Advanced JPA, Logging and Mockito

SENG301 Software Engineering II

Neville Churcher, Morgan English, Fabian Gilson, Karl Moore, Jack Patterson, Saskia van der Peet

30th March 2023

Learning objectives

This week's lab aims to introduce you to *Mockito*[1] mocking framework. By the end of this lab, you should be able to:

- Understand the purpose of mocking and stubbing when writing tests;
- Understand the basic functionality of *Mockito* testing framework;
- Implement a test class using *Mockito*.

Last, this lab will illustrate how to use loggers (with *Apache log4j2*[2]):

- Show how to set up multiple types of loggers for multiple purposes;
- How to use loggers inside the code at different levels (e.g., to print debug or error traces).

1 Introduction

When developing larger scale software systems, full usage scenarios may require the interactions of different components or classes, such as REST APIs. Mock objects are useful to write automated (user acceptance) tests where some involved parties can be substituted by default implementations (i.e. stubs) that are *mocking* an actual behaviour. This enables to isolate some behaviour of specific classes within test scenarios and therefore create self-contained tests (cfr. Lecture 9).

A concrete example is when creating unit/acceptance tests, you do not want to rely on external resources, e.g., REST API, that may or may not reply or that may require API keys to be set up (and sometimes paid for). This is even more critical in an automated context (i.e. Continuous Integration) when tests may be run thousands of times. So, you usually want to avoid calling third party APIs within unit/acceptance tests where failures to fetch data from these APIs would make your pipeline fail unexpectedly (cfr. Lecture 10).

Mocking can also be used in a Test-Driven Development loop while developing a feature where part of the behaviour of a succession of methods can be mocked and therefore tested without having the full implementation ready yet. *Mockito*, is one of the available mocking tools. There are a number of other available frameworks, such as *jMock*[3]. We have chosen to use *Mockito* due to its wide community support and compatibility with *JUnit*, *Cucumber* and *Spring*.

[1] See `https://site.mockito.org/`
[2] See `https://logging.apache.org/log4j/2.x/`
[3] See `http://jmock.org`

2 Domain, story and acceptance criteria

This Lab will build on the example case used in *Lab 4* and *Lab 5*. Please refer to these handouts if need be. We reproduce an updated domain model in Figure 2 where we introduce new entities for fetching cards from an API, these can be found in the `cards` package in code archive supplied with this handout.

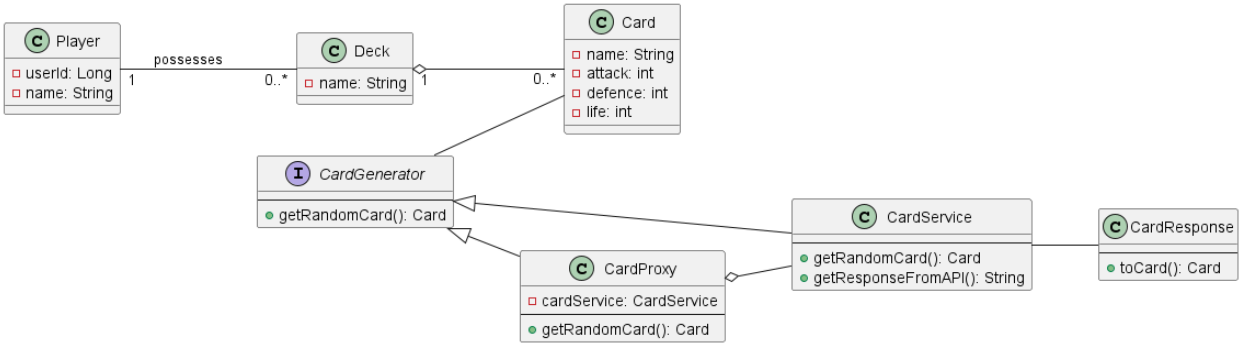

Figure 1: Domain model of a Yu-Gi-Oh! Clone App (updated from lab 5)

2.1 User stories

Remember that we have one Persona, *Alex*, a Yu-Gi-Oh player.

U3 As Alex, I want to draw random cards from an external API so that I can build a deck from them.
- AC.1 I can draw a random card that has valid name, attack, defence, and life stats.
- AC.2 If I find a suitable card, I can add the card to my deck.
- AC.3 I can decide to ignore the card and not add it to my deck.

3 Set up *Gradle* dependencies

Next to this handout, you will find a `.zip` archive with an updated version of the expected result from lab 5 (including the test of U1 AC2 and 3). In a nutshell, the project is structured as follows:

`main` contains the sources of your project
- **`java`** java source files including the `model`, `accessor`, `util` and `location` packages (API calls)
- **`resources`** application configuration files (i.e. `hibernate.cfg.xml` with *sqlite* and `log4j2.xml`)

`test` contains all test sources
- **`java`** java test sources (including the `feature` and class files from *Lab 4*)
- **`resources`** test configuration files (i.e. `cucumber.properties` and `hibernate.cfg.xml` with *h2*)

`build.gradle` the dependencies configuration file

3.1 Initial `gradle` file

We only reproduce the new elements from the `build.gradle` file relevant to this lab. Refer to prior labs if needed.

```
dependencies {
    // Use JUnit Jupiter for testing.
    // Logging facility (SENG301 lecture 9 and lab 6)
    implementation 'org.apache.logging.log4j:log4j-core:2.17.0'

    // JSON deserialization (for external REST API, lab 6)
    implementation group: 'com.fasterxml.jackson.core', name: 'jackson-core', version: '2.12.2'
    implementation group: 'com.fasterxml.jackson.core', name: 'jackson-annotations', version: '2.12.2'
    implementation group: 'com.fasterxml.jackson.core', name: 'jackson-databind', version: '2.12.2'

```

```
    //Mocking (lab 6)
    implementation 'org.mockito:mockito-core:5.2.0'
    testImplementation 'org.mockito:mockito-inline:5.2.0'

}
```

Listing 1: Partial `build.gradle` file with dependencies for logging and *Jackson* JSON handling.

3.2 Using an API

For this lab we will be using an API `https://db.ygoprodeck.com/api/v7/cardinfo.php` to fetch Yu-Gi-Oh card information. For more information about the api, see the official documentation here `https://ygoprodeck.com/api-guide/`. Feel free to try this out in your browser.

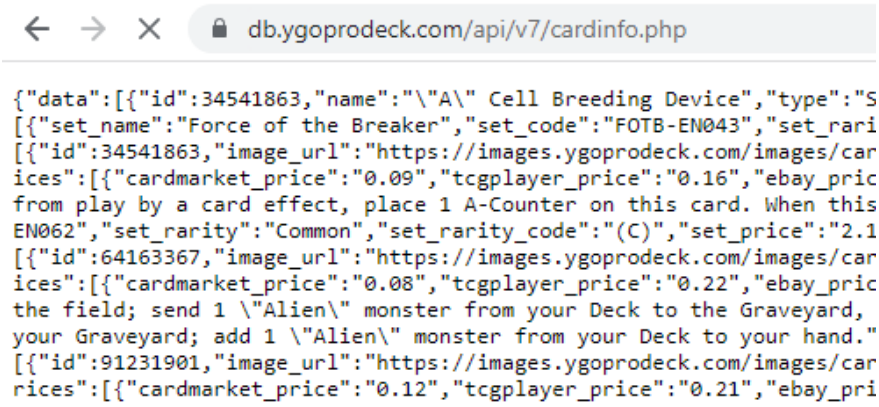

Figure 2: Example API request in browser

As you can see this gives back far too much information to properly understand as a human, in our application we will programmatically fetch only 1 card at a time for simplicity. However it is suggested you have a look at the structure of the response you receive from the server in a JSON viewing tool such as `https://jsonviewer.stack.hu/`

An alternative way is to use `curl`[4], a lightweight command line tool to transfer data over HTTP. See listing 2 for an example.

```
$ curl https://db.ygoprodeck.com/api/v7/cardinfo.php
{"data":[{"id":34541863,"name":"\"A\" Cell Breeding Device","type":"Spell Card",
//Trace truncated
```

Listing 2: `GET` cards through the Yu-GHi-Oh! API with `curl`.

3.3 Running the application

How the application runs has been changed compared to the previous labs, now instead of simply running and receiving an output you will have to interact with the application and 'play' the game yourself. To start, run (`$ ./gradlew run --quiet --console=plain`) the project to see what this programme does. Note that we have added `--quiet --console=plain` to the command as parameters. This removes the normal gradle logs so that we can more easily interact with our application through the command line. A similar trace as Listing 3 should be visible in your terminal. Have a little play around with the application to understand how it works.

```
#######################################################
  Welcome to Yu-Gi-Oh! Clone App
#######################################################
Available Commands:
```

[4] See `https://stackoverflow.com/a/27238493` for an example with a pretty printing tool offered by python.

```
"create_player <name>" to create a new player
"create_deck <player_name> <deck_name>" create a deck with <deck_name> for player <player_name>
"draw <player_name> <deck_name>" draw a random card to add to deck
"print <player_name>" print player by name
"exit", "!q" to quit
"help" print this help text
```

Listing 3: Execution trace of `App.java`.

An example interaction with the game to create a player and deck then draw a random card is shown in Listing 4.

```
create_player Morgan
Created player 1: Morgan
create_deck Morgan MyFirstDeck
Created deck 1: MyFirstDeck for Morgan
draw Morgan MyFirstDeck
You drew...
Yashinoki -- Atk: 800 Def: 600 Life: 0 -- Currently: attacking
Do you want to add this card to your deck? Y/N
Y
Card saved
```

Listing 4: Example interaction with app.

4 Logging

Logging is an important aspect of software, that is often over-looked by inexperienced developers. Logging allows us to "log" or record messages during operation to a file. Up until this point you may have simply used print statements, however these come with drawbacks:

- It often isn't as time efficient to add print statements when compared to debugging with IDE integrated tools.
- Printing a lot of information that is not easily identifiable and doesn't follow strict patterns can be hard to understand, lack much needed information for debugging purposes, and be hard to search efficiently.
- When delivering a product to the user we don't want them to be overwhelmed with messages printing to the console (or similar) that they do not understand.

So how does logging help, you may ask?

- We can easily specify more information, such as a time stamp, the current class or method, a level of criticality, and more.
- The use of a file (with a specific format such as json) means that it is easier to search for or filter specific messages (often an important ability is to filter by criticality).
- Files also make it easier to find issues in production. It is much easier to retrieve files from your production server or instance. You may have experienced this in practice where upon an application crash a window asks you to (or automatically) selects a log file to send to the developers so they can better understand the bug and what caused it.
- Logs may also be sent to multiple outputs including sockets, ensuring that developers can have access to the logs for a corrupted or shutdown machine.

In this project we make use of Log4J2, which you may have heard of over their security flaw in recent years[5], however this issue has since been patched in newer versions. To set up our logger we define a configuration file in our `main/resources` folder, for this example we will use xml but there are several ways to do this.

The configuration file will not be discussed in detail but there are many comments, including commented out configurations so that you should take the time to look into it. However it is important that we know where the output is going.

Within the `log` folder several files will be produced after you run the application

[5] See https://theconversation.com/what-is-log4j-a-cybersecurity-expert-explains-the-latest-internet-vulnerability-how-bad-it-is-and-whats-at-stake-173896

- `app.log` - This records the logging statements we add in our code (linked to the `applog` *RollingFile*[6] appender, i.e. logger instance).
- `other.log` - This records all other log statements, often filled by logs from other libraries (linked to the `allother` *RollingFile* appender, i.e. logger instance).
- `sql.log` - This records all of the sql related actions done by hibernate. Note that this is very bad practice in production, as it may log sensitive data[7] (linked to the `sqllog` *RollingFile* appender, i.e. logger instance).

If you take a look around the codebase you should notice that the `System.out` and `System.error` calls have been replaced by relevant logging types.

5 Using *Mockito* to mock external APIs

5.1 An introduction to *Mockito*

Mockito is a mocking framework for Java to help with effective unit testing. This is achieved by providing a *"test double"* through *"mocks"* or *"spies"* that reduce or remove dependencies between units of code to allow for proper testing of isolated units without being dependent on actual running implementations of dependencies of the *System Under Test* (SUT).

Mocking a class, e.g., `Mockito.mock(MyClass.class)` will create a complete test double for that class to be *"instrumented"* within the tests. This means that there is no real object in play, only a mock (i.e. dummy) object. Still, the mock must respect the pre- and post-conditions of the real object's methods.

When we spy a class, e.g., `Mockito.spy(new MyClass())`, we provide a wrapper around a real instance for which we can also define a valid, but dummy behaviour for one or more methods (i.e. we override one of the method for the purpose of the test). This allows us to control what values a method call will return for that particular *"spied"* method. With *Mockito*, spies are therefore real objects that we can reuse as-is or replace some methods by mocked (i.e. dummy) methods. Note that *Mockito*'s implementation of spies is different from their definition in *JMock*[8].

5.2 Update `build.gradle` file

Add the dependencies below under the `dependencies` clause of your `build.gradle` file.

```
//Mocking (lab 6)
implementation 'org.mockito:mockito-core:5.2.0'
testImplementation 'org.mockito:mockito-inline:5.2.0'
```

Listing 5: *Mockito* dependency.

5.3 Create a new `feature` file for the new ACs

As you did in *Lab 5*, create a new `u3-draw.feature` file that will contain the acceptance criteria reproduced in Listing 6 for the story from Section 2.1. Place this file under `src/test/resources/uc/seng301/cardbattler/lab6/cucumber`.

```
Feature: U3 As Alex, I want to draw random cards from an external API so that I can build a deck from them
  Scenario: AC1 - I can draw a random card that has valid name, attack, and defense
    Given I have a player "Jane"
    And I have an empty deck "MyFirstDeck"
    When I draw a randomly selected card
    Then I receive a random monster card with valid stats

  Scenario: AC2 - If I find a suitable card, I can add the card to my deck
```

[6] See `https://howtodoinjava.com/log4j2/log4j2-rollingfileappender-example/`
[7] See `https://twitter.com/TwitterSupport/status/992132808192634881`
[8] See Martin Fowler's piece referenced from SENG301 optional material `https://martinfowler.com/articles/mocksArentStubs.html`.

```
    Given I have a player "Jane"
    And I have an empty deck "MyFirstDeck"
    And While playing I draw a randomly selected card
    When I decide I want to add the card to my deck
    Then The card is added to my deck

  Scenario: AC3 - I can decide to ignore a card and not add it to my deck
    Given I have a player "Jane"
    And I have an empty deck "MyFirstDeck"
    And While playing I draw a randomly selected card
    When I decide I want to ignore the card
    Then The card is not added to my deck
```

Listing 6: Acceptance criteria for U3 in *Gherkin* (Cucumber) syntax

You can run the *Cucumber* task (`$ ./gradlew cucumber`) to generate the methods' skeletons. Copy this code into a new test class under `src/test/java/uc/seng301/cardbattler/lab6/cucumber`, as you did in *Lab 5*,

5.4 Implement AC1

To implement AC1 we will make use of spies (`Mockito.spy()`). As mentioned above, a spy allows us to only mock certain methods on an object, in this case we only want to mock `CardService.getResponseFromAPI()` so that all other methods of the implementation can be tested.

The steps:

- Given I have a player "Jane"
- And I have a deck "MyFirstDeck"

Are left to you to implement, you may want to look back at the implementation from Lab 5. You will also need a similar `setup()` method annotated with `@Before`.

5.5 Using *Mockito* to fake API calls

We implemented a simple HTTP client in `CardService` that will fetch cards from the API for you (i.e. the service behind the `draw` command). Take a few minutes to look into that class.

Mockito can supply default result values for any methods that it is *"spying"*. This is particularly useful in our example if you don't want to call the API each time in your tests. To this end, you need to:

- Create an instance of CardService as a Mockito.spy()
- Define what method to override on the spy, and what is should respond

In Listing 7, we show how to declare a spy of `CardService`. Below code should be placed in the `setup()` method of your test class. Do not forget to actually declare the `CardService` instance as a class field to call it from the related scenario step.

```
cardGeneratorSpy = Mockito.spy(new CardService());
```

Listing 7: *"Spy"* a class with *Mockito* (within `setup()`).

The second step is to define our return object that our Mocked class will return. In this example we return a minimal string result of a query done in the past (properties that we are not interested in such as `card_sets` have been removed).

```
String apiResponse = "{\"data\":[{\"id\":44073668,\"name\":\"Takriminos\",\"type\":\"Normal Monster\",\"frameType\":\"
    normal\",\"desc\":\"A member of a race of sea serpents that freely travels through the sea.\",\"atk\":1500,\"def
    \":1200,\"level\":4,\"race\":\"Sea Serpent\",\"attribute\":\"WATER\"}]}\n";
Mockito.doReturn(apiResponse).when(cardGeneratorSpy).getResponseFromAPI();
```

Listing 8: API string to return on spied method (within step definition).

Finally we can call `CardGenerator.getRandomCard()` (Note that `CardGenerator` is the interface our `CardService` implements) as shown in Listing 9 and under the hood it will call our spied method instead of the API.

```
card = ((CardGenerator) cardGeneratorSpy).getRandomCard();
```

Listing 9: Calling `getRandomCard()` using our spied method (within step definition).

We can see all of this in practice with the `I draw a randomly selected card` step definition in Listing 10.

```
@When("I draw a randomly selected card")
public void i_draw_a_randomly_selected_card() {
    String apiResponse = "{\"data\":[{\"id\":44073668,\"name\":\"Takriminos\",\"type\":\"Normal Monster\",\"frameType\":\"
        normal\",\"desc\":\"A member of a race of sea serpents that freely travels through the sea.\",\"atk\":1500,\"def
        \":1200,\"level\":4,\"race\":\"Sea Serpent\",\"attribute\":\"WATER\"}]}\n";
    Mockito.doReturn(apiResponse).when(cardGeneratorSpy).getResponseFromAPI();
    card = ((CardGenerator) cardGeneratorSpy).getRandomCard();
    Assertions.assertNotNull(card);
}
```

Listing 10: "I draw a randomly selected card" step definition

To finish this scenario implement the *Then* step definition to check the `card` is valid.

5.6 Implement the remaining steps

Your last task is to implement the remaining steps for ACs 2 and 3 to make the full acceptance test pass.

AC2 and 3 require testing the Game.draw() function. Doing so means that we not only have to provide the same `CardGenerator` spy, but also a mock for the `CommandLineInterface` class to provide dummy behaviour for either saying “yes” to keep the card or “no” to discard it. The relevant code you need is included in Listing 11, make sure to read the comments to know where everything goes.

```
// As a private class variable
private CommandLineInterface cli;
// Initialise the mock object in setup()
cli = Mockito.mock(CommandLineInterface.class);
// Define the mock behaviour in the "I decide I want to add the card to my deck" step to return "Y" when getNextLine() is
    called
Mockito.when(cli.getNextLine()).thenReturn("Y");
```

Listing 11: Example of Mockito.Mock

Note that when `Game.draw()` adds a card to a deck, it is not done so the `deck` instance you have in the test. This means that you will have to fetch the updated version from the database before you validate if a card has or hasn’t been added.

6 To go further

As an interesting additional resource to go further with *Mockito* with dependency injection, for mocking REST calls in Spring (SENG302), you are encouraged to look at:

- `https://www.baeldung.com/mockito-annotations`
- `https://www.baeldung.com/spring-mock-rest-template`
- `https://github.com/Jet-C/spring-demo`