**Assignment #3: Mocking**

**Due date: Feb 21st, 2021, 11:59 p.m.**

# Introduction

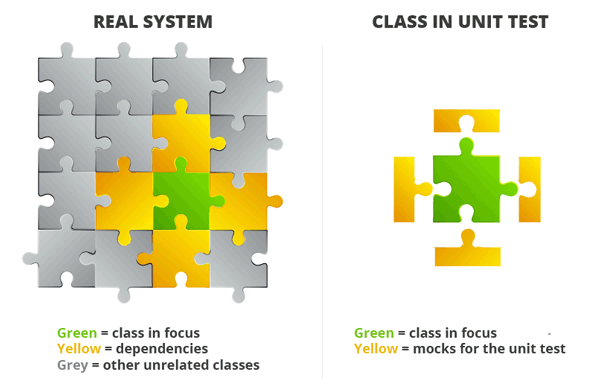
In this assignment, you will learn how to learn what Mocking is and how to mock some objects.

Definition

Mocking is creating objects that simulate the behavior of real objects. An object that you want to test may have dependencies on other complex objects. In unit testing, we use mocking to isolate the behavior of the objects you want to test by replacing the other objects by mocks, simulating the behavior of the real objects.

Goals

* Classes are not isolated. They may use services and methods that are from other classes. We want to mock the behavior of these dependencies.



Mechanism

The mechanism of mocking is to mock the services and methods from other classes and simulate the real behavior of them using some mocking frameworks and use that mocked methods and services to do unit testing in isolation. This is where Mocking frameworks come into play.

Tools

* Frameworks available for mocking in unit testing include PowerMock, EasyMock, and Mockito.

**In this assignment, we will use the Mockito framework for mocking.**

**The documentation for the Mockito framework can be found here:** [**https://javadoc.io/static/org.mockito/mockito-core/3.2.4/org/mockito/Mockito.html**](https://javadoc.io/static/org.mockito/mockito-core/3.2.4/org/mockito/Mockito.html)

Mockito - Features

* Mockito is a proxy based framework

A proxy is just an object which will be used instead of the original object. If a method of the proxy object is called than the proxy object can decide what it will do with this call:

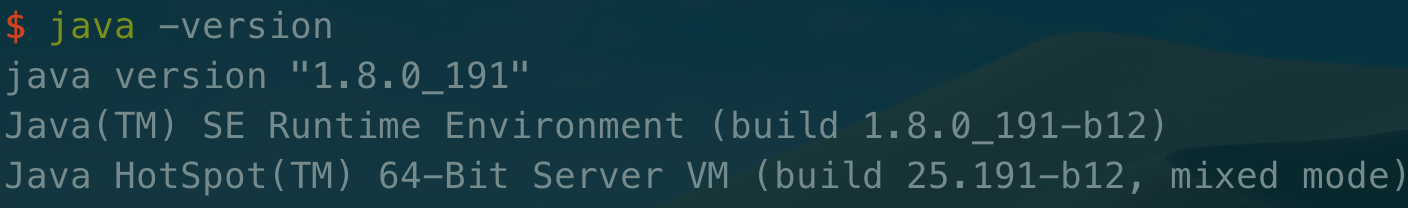
* delegate it to the original object
* handles the call itself

A proxy doesn’t require an instance of an interface/class if the proxy handles all method invocations itself.

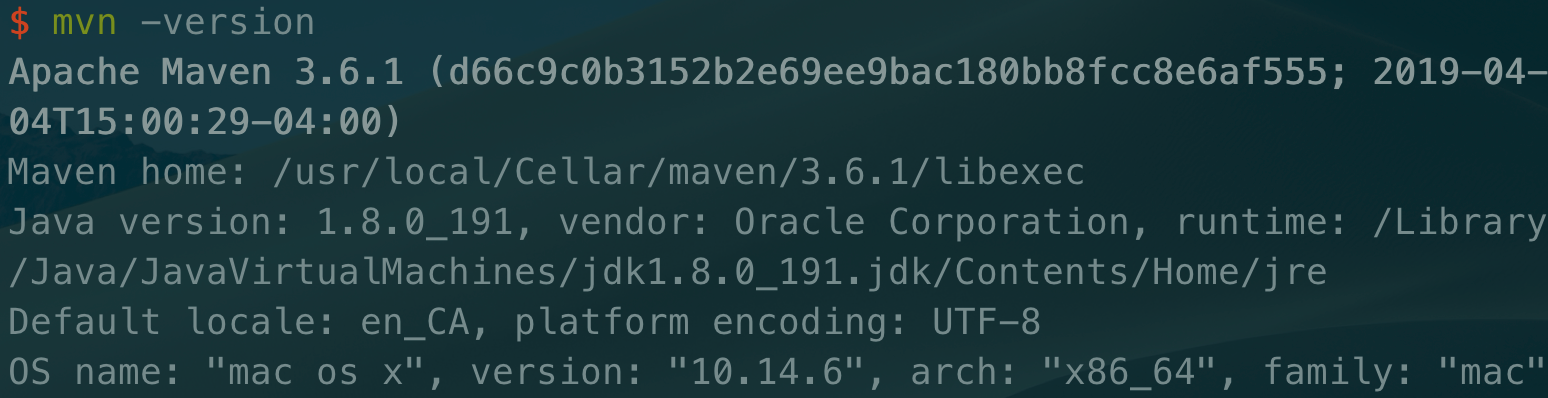
# Environment Requirements

Please make sure that you have Maven and Java installed on your computer.

* OS: macOS/Windows 10
* Eclipse: version 2019-12 (4.14.0)
* Java: version 1.8 or superior



* Maven: version 3.6.1 or superior



*Note: Only the above environment is tested by us, not sure if other versions work well.*

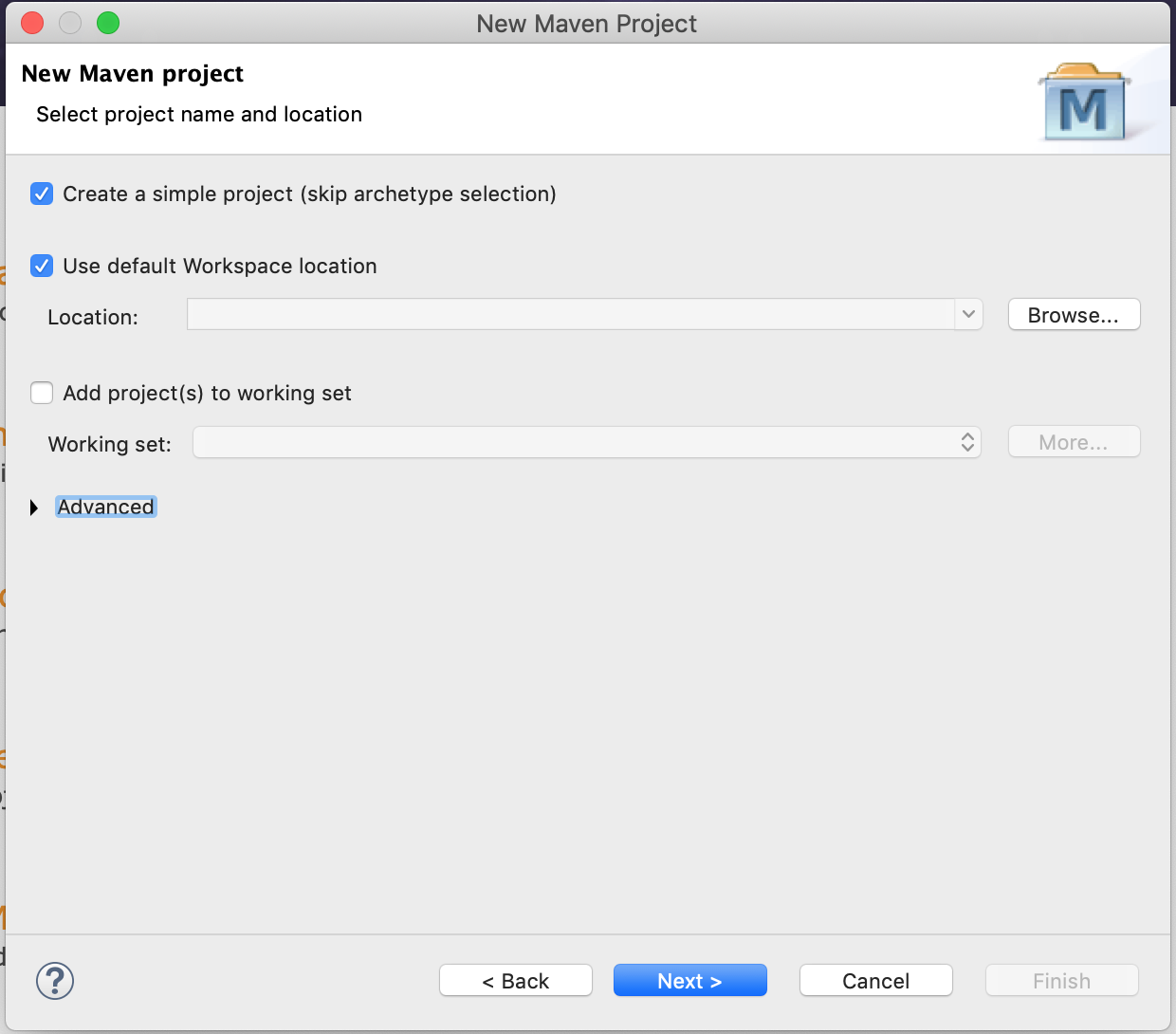
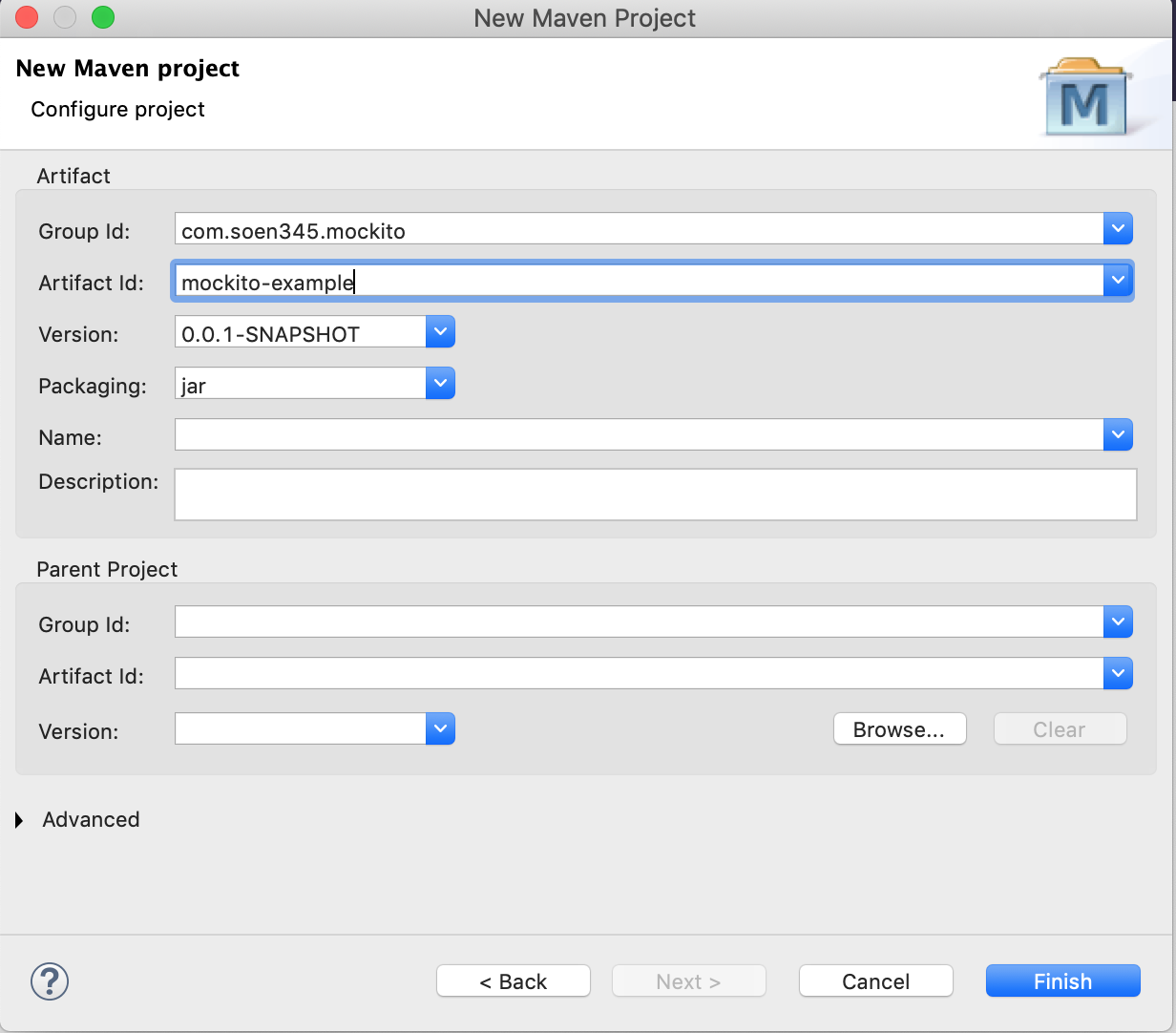
# Experiments

Please make sure that you have Maven and Java installed on your computer.

**Part A** *(Integrate Mockito and JUnit testing tool into an example project)*

1. **Open any workspace in Eclipse**
2. **File → New → Other → Maven → Maven Project**

Maven makes it easy for us to manage dependencies



Open up pom.xml file

Add in before </project> to download the dependencies for JUnit and Mockito framework:

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.mockito</groupId>

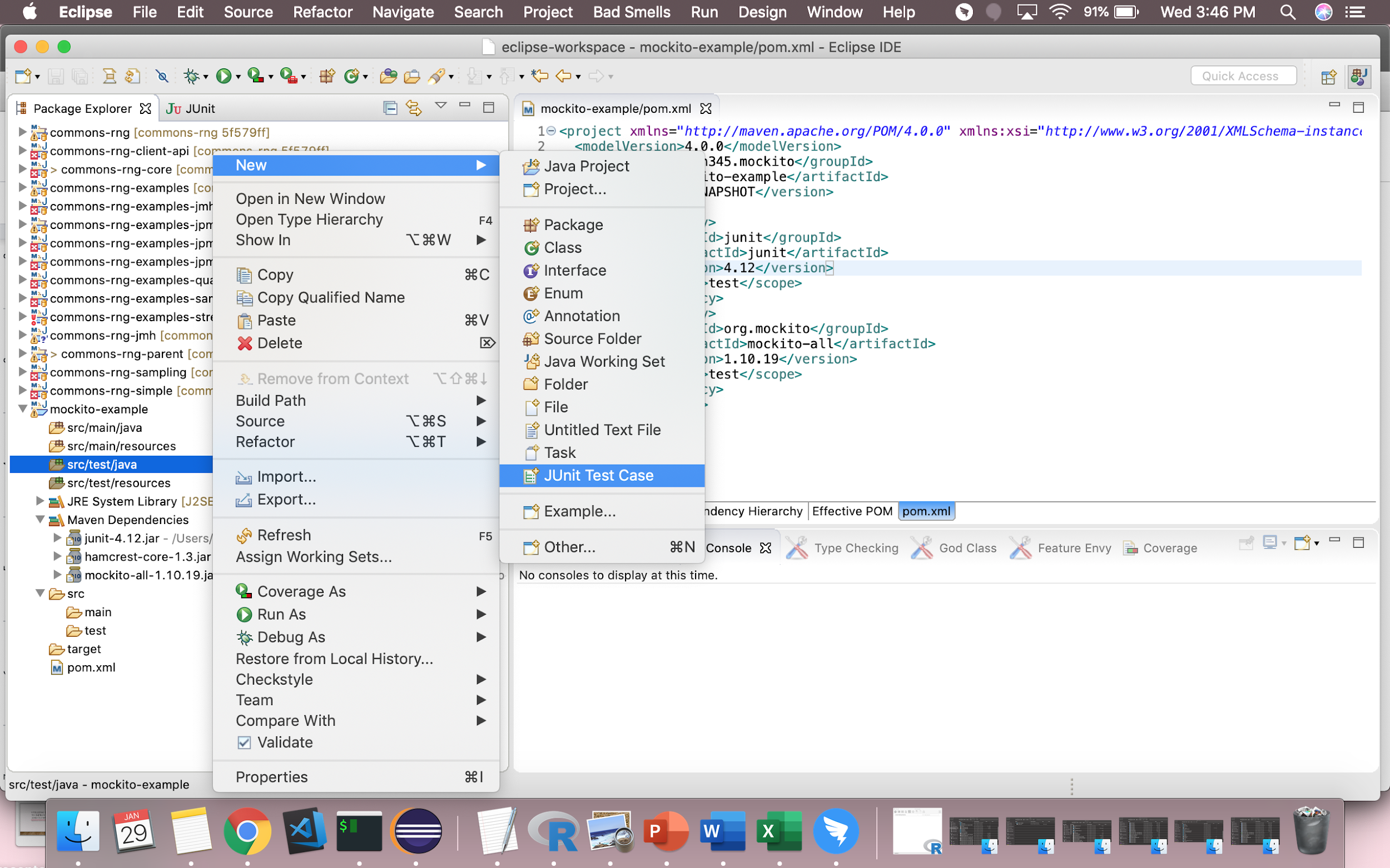
<artifactId>mockito-all</artifactId>

<version>1.10.19</version>

<scope>test</scope>

</dependency>

</dependencies>

**Check that JUnit is working -** The test actually does not contain any Mockito functions yet:

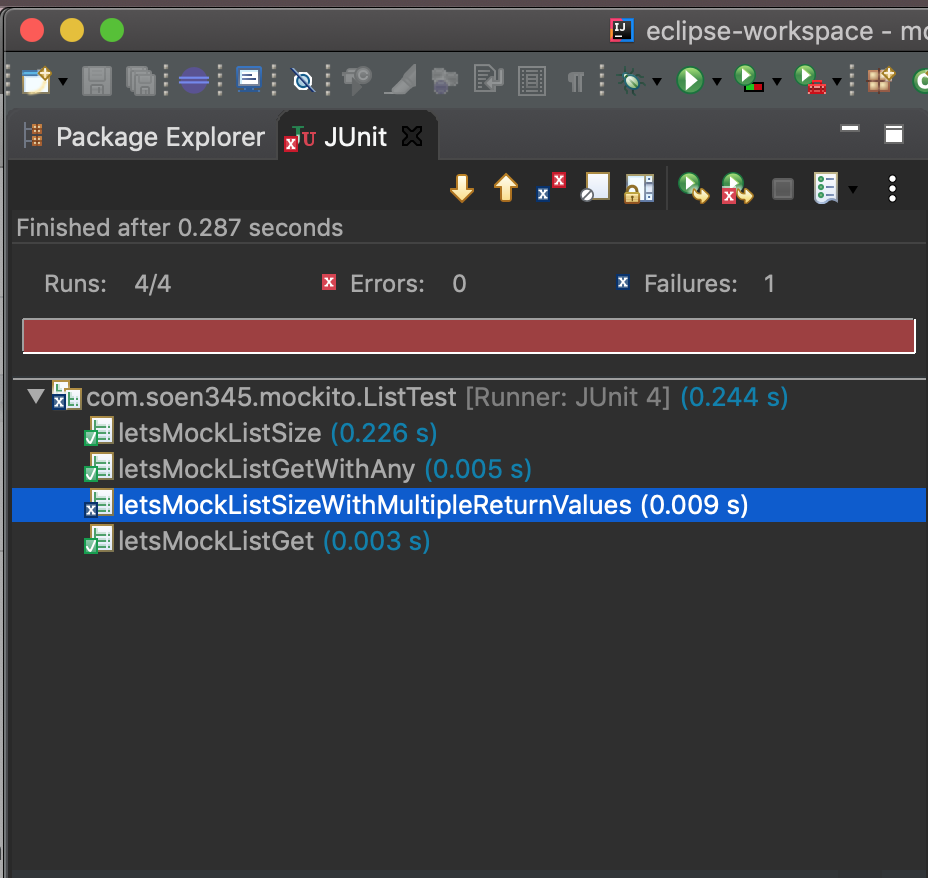
## **Part B** *(Paste in your first basic Mockito test cases)*

Create a JUnit test case named **ListTestTest**

Paste in the following code from the Gist:

<https://gist.github.com/sophiaquach16/f7e9200f9af8c873a17fd28088957aba>

When run, this should be shown:



For each of the four unit tests, you can see the line List list = mock(List.class);

The mock function takes in a parameter of type **Object**.class to mock

To make **letsMockListSizeWithMultipleReturnValues()** pass, add this line after the commented line:

when(list.size()).thenReturn(10, 20);

Now run the tests again and it should pass.

By default, for all methods that return a value, a mock will return either null, a primitive/primitive wrapper value, or an empty collection, as appropriate. For example 0 for an int/Integer and false for a boolean/Boolean.

Once stubbed, the method will always return a stubbed value, regardless of how many times it is called, unless overridden by specifying another value through another stub.

The last stubbing is more important - when you stubbed the same method with the same arguments many times. Other words: **the order of stubbing matters** but it is only meaningful rarely, e.g. when stubbing exactly the same method calls or sometimes when argument matchers are used, etc.

## **Part C** *(Mocking a class implementation)*

Now that we have mocked the List class from Java, we will see how mocking a dependency works. In real life, rather than mocking a List object, we would have instances of an actual one.

[TodoService.java](https://gist.github.com/sophiaquach16/91b138666e4607eb8f5fa59ca375d2c3#file-todoservice-java) interface:

<https://gist.github.com/sophiaquach16/91b138666e4607eb8f5fa59ca375d2c3>

[TodoBusinessImplMockitoTest.java](https://gist.github.com/sophiaquach16/ecf9ac5eec836225bd80c3b99dc55500#file-todobusinessimplmockitotest-java) test case:

<https://gist.github.com/sophiaquach16/ecf9ac5eec836225bd80c3b99dc55500>

[TodoBusinessImpl.java](https://gist.github.com/sophiaquach16/e104cacfcf1a61959646a6534fabc4f3#file-todobusinessimpl-java) class:

<https://gist.github.com/sophiaquach16/e104cacfcf1a61959646a6534fabc4f3>

## **Part D** *(Mocking in a TDD environment. A complete exercise)*

The TA will not demonstrate this exercise. But still you can ask general questions.

Using the mockito framework you will complete the unit test for **ConcordiaAppDepot,** an app store.

Anybody can upload their app, but users need to register with **ConcordiaAppDepot** with a PayPal user account. **ConcordiaAppDepot will charge a percent of the app price as hosting and publishing fees for basic members. See fees below.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Plan** | **Basic** | **Standard** | **Premium** | **Professional** |
| Fee % | 30 | 15 | 10 | 8 |
| Storage | 1 GB | 5 GB | 10 GB | unlimited |
| Min paid fee | 20 | 100 | 200 | >1000 |

**Anybody can download apps, but they have to pay the price using a PayPal site before downloading the app**

**ConcordiaAppDepot needs a scheduler service;** it will run every month and calculate the payable amount for the app developers and then deposit the amount to the developers’ Paypal accounts. PayPal charges on the transaction amount and the number of transactions, so minimize the number of transactions. If a developer launched three apps in a month, instead of sending three invoices, we just send one with three line items.

Specifications:

The Service perform the following operations

* Retrieve transaction information for all downloads in the last 30 days
* Retrieve membership information of the app developers
* Calculate the payable amount for each app developer
* Send payment advice to PayPal

**Download the software project from the following** [link](https://drive.google.com/file/d/1iOlG2mLBxee0maff_U91-lqGJFQsFA85/view?usp=sharing).

You are going to work on class /ConcordiaAppDepot/src/test/java/**ReconciliationServiceTest**.java

The class under test is /ConcordiaAppDepot/src/main/java/**ReconciliationService**.java

You will mock the following interfaces: FinancialTransactionDAO, MembershipDAO, PayPalFacade.

Complete the unit tests in **ReconciliationServiceTest. Each test is uniquely identified by a number. The assert statements are left in the unit test code as a reference.**

1. We will concentrate on when DAO (Direct data access) returns 1 transaction. Stub the retrieval method to return a transaction.
2. Add a test to verify that the reconcile method calls membershipDAO to

fetch the membership details for a developer. In the test, create a list of transactions and add only one transaction with the developer's ID as DEV001. Stub financialTransactioDAO to retrieve this list. During test execution, the Service will get this list and then it should ask\* membershipDAO to get the details of developer DEV001. We will verify that in the test using Mockito's verify() API.

1. Now add a test to verify multiple transactions. This is a very important stage in TDD. Once we are done with one, we should test our code against many. In a test we will create two transactions: one for John and another one for Bob. We expect that for both developers membershipDAO will be called. We are going to use ArgumentCaptor and Times to verify the invocation. Verification will check the number of invocations by passing new Times(2), then the argument captor will capture arguments for all invocations. Finally, we will ask the argument captor to return the list of invocations and from that list we will verify whether membershipDAO was invoked for both Bob and John.
2. Verify that the pay advice was sent. Create a transactional list for David, a developer. The createTxDto(...) method creates a Transaction Dto instance from the developer ID, PayPal ID, and app price.
3. We need to calculate the amount payable. How do we test this? Stub MembershipDAO to return a Basic membership DTO object. This means 30 percent is deducted from the original app price.If the app price is CAD 100.00, then PayPal payment advice should be CAD 70.00. Use Mockito's ArgumentCaptor method to verify that.
4. Now it's time to test multiple transactions—one with CAD 200.00 and another with CAD 150.00, and Standard and Premium memberships. The deductible is 15 percent and 10 percent respectively. The memberShip(double percent) method creates membershipStatusDto. Stub the membershipDAO instance to return membership deductible 15 percent for John and 10 percent for Dave. Use ArgumentCaptor to capture the PayPalFacade call. Then, verify that the correct deductible was computed and passed to facade for both the developers.
5. One thing is still missing. How can we minimize the PayPal Transactions?If a developer develops two apps, we should invoke PayPal facade only once, not twice. PayPal charges against each transaction and also multiple transaction calls can create performance issues. Add a test for the developer Janet, who has two apps: FishPond and TicTacToe. Default membership is Basic, with 30 percent deductible.

After you finish all the experiment parts (**part A to part D**), you need to answer the corresponding questions in the following report and provide appropriate screenshots and source files to support your answers. Please submit it as a **PDF** file on moodle with the file name of ‘**Assignment3\_yourname\_yourstudentID.pdf**’.

Last but not least, plagiarism is prohibited. Have fun!

**Assignment #3 report**

Student number:

Student name:

Question 1 [0.25%]: For *@Test* **public void letsMockListGet()**, does the commented line

// assertNull(list.get(1)); pass or fail when uncommented? Why? If it fails, how do we make it pass? What is the value of list.get(1)?

Question 2 [0.25%]: Mockito has something called “ArgumentMatchers”. How do they work? There is an example in the test case *@Test* **public void letsMockListGetWithAny()**, which passes when run. How does this work?

Question 3 [0.25%]: Copy the three files above into the same test directory as **Part B**

Observe the three files and run the unit tests in [TodoBusinessImplMockitoTest.java](https://gist.github.com/sophiaquach16/ecf9ac5eec836225bd80c3b99dc55500#file-todobusinessimplmockitotest-java). Describe the interactions.

Question 4 [4%]: Complete the unit tests according to instructions in **part D**. You need to attach screenshots to show that your tests pass, and attach your source code. Do not forget to add your name and student id in the header of your test class ReconciliationServiceTest.

TODO 1= 0.25%

TODO 2 = 0.5%

TODO 3 = 1%

TODO 4= 1%

TODO 5 = 1%

TODO 6 = 0.5%

TOTAL 5% of the course grade.

# 

# Graduate attributes (to be filled by TA)

|  |  |  |  |
| --- | --- | --- | --- |
| Graduate attribute | Description | Score out 100% | Comments |
| [PA-4] Analysis (uncertainty and incomplete knowledge) | Students analyze a problematic situation and propose solutions in scenarios that differ from those seen in class |  |  |
| [UET-1] Ability to use appropriate engineering tools, techniques and resources | Students demonstrate their ability to use a well known mocking framework to test services that are not available |  |  |
| [ITW-4] Delivering results | Deliver results in time and according to instructions |  |  |
| [CO-3] Documentation | The report follows the template proposed. The answer to the questions are well written and the conclusions obtained are well documented |  |  |