|  |
| --- |
| Department of Software Engineering  Mehran University of Engineering and Technology, Jamshoro |

|  |  |  |  |
| --- | --- | --- | --- |
| Course: SW426 - Software Quality Engineering | | | |
| Instructor | Rabia Iftikhar | **Practical/Lab No.** | 07 |
| Date | 18-08-2020 | **CLOs** | CLO-3 |
| Signature |  | **Assessment Score** | 1 Mark |

|  |  |
| --- | --- |
| Topic | Creating Mocks using Mockito |
| Objectives | * To get familiar with mocks * To explore Mockito framework |

|  |
| --- |
| Lab Discussion: Theoretical concepts and Procedural steps |

**What is mocking?**

Mocking is a way to test the functionality of a class in isolation. Mocking does not require a database connection or properties file read or file server read to test a functionality. Mock objects do the mocking of the real service. A mock object returns a dummy data corresponding to some dummy input passed to it.

## Mockito

Mockito facilitates creating mock objects seamlessly. It uses Java Reflection in order to create mock objects for a given interface. Mock objects are nothing but proxy for actual implementations.

**Why mock?**

Most of the classes we come across have dependencies and often times methods delegates some of the work to other methods in other classes, and we call these classes dependencies. When unit testing such methods, if we only used JUnit, our tests will also depend on those methods as well. We want the unit tests to be independent of all other dependencies e.g. we want to test the method addCustomer in CustomerService class, and within this addCustomer method, the save method of the CustomerDao class is invoked. We don’t want to call the real implementation of the CustomerDao save() method for a few reasons:

* + We only want to test the logic inside the addCustomer() in isolation.
  + We may not yet have implemented it.
  + We don’t want the unit test of the addCustomer() fail if there is a defect in save() method in the CustomerDao.

So we should somehow mock the behavior of the dependencies. This is where mocking frameworks comes in to play.

**Example**

**Step 1 − Create an interface called CalculatorService to provide mathematical functions**

* ***CalculatorService.java***

public interface CalculatorService {

public double add(double input1, double input2);

public double subtract(double input1, double input2);

public double multiply(double input1, double input2);

public double divide(double input1, double input2);

}

**Step 2 − Create a JAVA class to represent MathApplication**

* ***MathApplication.java***

public class MathApplication {

private CalculatorService calcService;

public void setCalculatorService(CalculatorService calcService){

this.calcService = calcService;

}

public double add(double input1, double input2){

return calcService.add(input1, input2);

}

public double subtract(double input1, double input2){

return calcService.subtract(input1, input2);

}

public double multiply(double input1, double input2){

return calcService.multiply(input1, input2);

}

public double divide(double input1, double input2){

return calcService.divide(input1, input2);

}

}

**Step 3 − Test the MathApplication class**

* Let's test the MathApplication class, by injecting in it a mock of calculatorService. Mock will be created by Mockito.
* Here we've added two mock method calls, add() and subtract(), to the mock object via when(). However during testing, we've called subtract() before calling add(). When we create a mock object using create(), the order of execution of the method does not matter.
* ***File: MathApplicationTester.java***

package com.tutorialspoint.mock;

import static org.mockito.Mockito.mock;

import static org.mockito.Mockito.verify;

import static org.mockito.Mockito.when;

import org.junit.Assert;

import org.junit.Before;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.mockito.runners.MockitoJUnitRunner;

// @RunWith attaches a runner with the test class to initialize the test data

@RunWith(MockitoJUnitRunner.class)

public class MathApplicationTester {

private MathApplication mathApplication;

private CalculatorService calcService;

@Before

public void setUp(){

mathApplication = new MathApplication();

calcService = mock(CalculatorService.class);

mathApplication.setCalculatorService(calcService);

}

@Test

public void testAddAndSubtract(){

//add the behavior to add numbers

when(calcService.add(20.0,10.0)).thenReturn(30.0);

//subtract the behavior to subtract numbers

when(calcService.subtract(20.0,10.0)).thenReturn(10.0);

//test the subtract functionality

Assert.assertEquals(mathApplication.subtract(20.0, 10.0),10.0,0);

//test the add functionality

Assert.assertEquals(mathApplication.add(20.0, 10.0),30.0,0);

//verify call to calcService is made or not

verify(calcService).add(20.0,10.0);

verify(calcService).subtract(20.0,10.0);

}

}

**Step 4 − Execute test cases**

Create a java class file named TestRunner to execute Test case(s).

* ***TestRunner.java***

import org.junit.runner.JUnitCore;

import org.junit.runner.Result;

import org.junit.runner.notification.Failure;

public class TestRunner {

public static void main(String[] args) {

Result result = JUnitCore.runClasses(MathApplicationTester.class);

for (Failure failure : result.getFailures()) {

System.out.println(failure.toString());

}

System.out.println(result.wasSuccessful());

}

}

**Step 5 − Verify the Result**

## 

|  |
| --- |
| Lab Tasks |

1. Rewrite and execute the code in handouts and observe the results.
2. Write a code capturing the functionality of online shopping system. The code should call the functions like add to cart, remove from cart, add to favorites, check out, etc. Test all these functions in isolation using Mockito.

|  |  |
| --- | --- |
| Lab Tasks Assessment/Rubrics along with Score/Marks | |
| *Rubric Description* | ***Rubric Marks*** |
| 1. Test coverage | 0.5 |
| 1. Testing standards | 0.25 |
| 1. Test completeness | 0.25 |