JAVA ACADEMY - XIDERAL MONTERREY, N.L

WEEK 4
MOCKITO FRAMEWORK

Made by:
Luis Miguel Sánchez Flores

INTRODUCTION

To ensure that the developed code has been implemented correctly, **unit tests** are developed to test out each component separately and examine different scenarios with different parameters to ensure that the program is working as expected. For simple applications, the use of unit tests is enough to prove that the software is correct and ready for deployment. However, the same thing cannot be said for more elaborated and complex programs, as sometimes these kinds of programs rely on external dependencies to work, and it can be quite challenging testing the program's components individually due to this dependency, it may be necessary to write verbose and tedious code in order to test the component with confidence. As such, frameworks such as **Mockito** emerged as an answer to replace the tedious code by mimicking the dependency capabilities, allowing developers to focus more on effectively managing their unit tests.

Mockito is a Java-based open-source library that provides a simple API to create mock objects. Mock objects are objects that simulate the behavior of real objects in a controlled way. They are commonly used in unit tests to isolate the code under test from its dependencies.
Mockito provides a rich set of features to create mock objects, verify method calls, and test behavior-driven development (BDD) style tests. It is widely used in Java-based applications and can be easily integrated with popular Java testing frameworks such as JUnit and TestNG.

Mockito is a powerful, and easy-to-use tool that can help developers write effective unit tests for their code. By leveraging the techniques, it can create robust test cases that will assist to identify errors with anticipation in the development cycle, and to ensure the quality code.

Getting Started

To get started with Mockito, the framework is imported as a dependency in either a Maven or Gradle project.
Mockito complements the JUnit library to check the behaviors and interactions between the components of the program without the need to know in great detail the dependencies, managing to execute unit tests in an effective and fast way.

To demonstrate the Mockito framework, we will simulate a service layer used by a controller as a dependency to delegate and display the relevant Customer info towards the users. Customer will represent the “application” entity:

Customer.java (Entity)

```java
// Simple Customer Entity


// Use of Lombok to quickly create the
// constructors, getters and setters of the class:
@NoArgsConstructor
@Data
@AllArgsConstructor
public class Customer {


    private int id;

    private String firstName;

    private String lastName;

    private String email;

    private String gender;

    private String contactNo;

    private String country;

    private String dob;

```

CustomerService.java (Service to mock)

```java
// Serivce to mock with Mockito:

public interface CustomerService {

    List<Customer> findAll();
    Customer findById(int id);
    List<Customer> findByDOB(String dob);
    Customer updateCustomer(Customer customer);
    void deleteCustomer(int id) throws RuntimeException;

}
```

Controller.java

```java
// Controller in which it depends on CustomerService
// to demonstrate the client(s) to the user.

public class Controller {

    // Dependency
    CustomerService customerService;


    public Controller(CustomerService customerService) {
        this.customerService = customerService;
    }
```

Creating mock objects

With the creation of the JUnit Test case inside the test folder in the project, Mockito is imported statically in order to invoke the framework's methods directly:

```
1 package example;
2
3 import static org.junit.jupiter.api.Assertions.*;
4 import static org.mockito.Mockito.*;
5
```

This is where we can generate the imitated object using the `mock()` function to mimic the object within the unit tests and simplify the process. The method takes a class literal as an argument to generate the imitation of that particular class.

```
CustomerService example = mock(CustomerService.class);
```

An alternative way to generated the mock object in the test class is through the use of `MockitoAnnotations`, which simplifies the initialization of the objects to be imitated during the tests, indicating with the `@Mock` annotation the objects to mimic, along with the `@InjectMocks` annotation to inject the mimicked attribute into the test object automatically:

```java
class MockitoTest {

    @Mock
    CustomerService customerService;

    @InjectMocks
    Controller controller;
```

To initialize the annotated attributes, we need to enable them through the `openMocks()` function, which is executed before each of the unit tests included:

```java
@BeforeEach
void setUp() {
    // Initialize fields with the Mockito annotations:
    MockitoAnnotations.openMocks(this);
}
```

Stubbing

With the mimicked object we can decide the desired behavior by means of the `when()` and `then()` functions, in which we specify which method and arguments should be mimicked, and what the expected result should be, respectively.

This process of defining the behavior of the object and the type of result we expected from it is known as **stubbing**.

In the following example, stubbing is employed to mimic the return of a list of customers of the same gender, returning the male customers with the argument of “Male” entered in the mimicked function:

```java
@Test
void returnListOfMaleEmployees() {

    final String MALE_GENDER = "Male";

    // Mock the function call and the desired data returned, given a specific argument
    // When the customerService calls findByGender(MALE_GENDER), it should return the list of male customers:

    // if findByGender() is called with a "Male" argument, then it should return a list of only male customers
    when(customerService.findByGender(MALE_GENDER)).thenReturn(List.of(
            exampleList.get(0), // <- John Doe
            exampleList.get(2), // <- Victor Van Dam
            exampleList.get(3) // <- Fredrick Durst
    ));

    // Get all male customers with the controller:
    List<Customer> maleCustomers = controller.findCustomersByGender(MALE_GENDER);

    // Check if the retrieved list only includes male customers:
    boolean areAllMale = maleCustomers.stream().allMatch(c -> c.getGender().equals("Male"));

    assertTrue(areAllMale);
}
```

The `when()` y `then()` functions only work with the methods that return a value. For stubbing void methods, Mockito offers `doNothing()`, `doThrow()` and `doAnswer()`. In this case, `doThrow()` was used to simulate a RuntimeException when invoking the `deleteCustomer()` of the service:

```java
// Testing out Mockito stubbing for void methods
@Test
void testDeletingCustomerError() {

    assertThrows(RuntimeException.class, () -> {
        // For now, throw the RuntimeException of the deleteCustomer method when called:
        doThrow().when(customerService).deleteCustomer(anyInt());
    });
}
```

It is worth mentioning that Mockito automatically returns a default value based on the type of data returned by the function of the imitated attribute:

```
// Mockito automatically assings a default value based on the method's return datatype.
// In this case, since findAll() returns a List, Mockito assigns a empty list as its default value:
@Test
void testEmptyList() {
    assertEquals(true, customerService.findAll().isEmpty());
}
```

Verification

We can also perform checks on the mimicked objects using the `verify()` function to check if certain actions or interactions occurred during the test. With `verify()` we can evaluate if the function was called a certain number of times, or if the function parameters were entered correctly.

For example, `verify()` was used to check whether the mimicked service layer had called the `findById()` function during the update process of a client, ensuring that the update process has a previous evaluation of the existence of the client via its ID:

```
@Test
void testCustomerUpdate() {
    Customer newCustomer = new Customer(1, "John", "Dale", "johndale@example.com", "Male", "123456999", "United States", "09-03-1990

    // For now, return a new customer for any existing Customer object:
    when(customerService.updateCustomer(any(Customer.class))).thenReturn(newCustomer);

    // Update the example customer with the new customer's details:
    Customer updatedCustomer = controller.updateCustomer(exampleCustomer);

    // Mockito offers the verify() methods to ensure that a specific method / behavior has been achieved
    // Here, we check if CustomerService invoked the findById() method to verify if the ID of the exampleCustomer
    // has been checked out during the update process:
    verify(customerService).findById(anyInt());
```

Along with the `verify()` function, **ArgumentMatchers** were used, which allow us to perform stubbing or verification in a more generic and flexible way. In this case, the ArgumentMatcher of `anyInt()` was used to check if the

`findById()` function was applied by the mocked service, regardless of the ID value used in the test.

Argument Capturing

Another useful feature of Mockito is its ability to capture and evaluate arguments that were introduced to a function during a test, through the use of an `ArgumentCaptor` object. This allows developers to verify if the code is working with the correct values, or employ tests on the captured arguments to ensure that they comply with the conditions/rules set.

In the following example, an `ArgumentCaptor` object is instantiated to capture arguments of the `String` type:

```
// Create an ArgumentCaptor object that captures Strings:
ArgumentCaptor<String> dobArgument = ArgumentCaptor.forClass(String.class);
```

The `findCustomersByDOB()` function of the controller is executed in a way that it internally calls the `findByDOB()` function of the mocked service:

```
final String DOB_TO_SEARCH = "09-03-1990";
```

```
// Call the controller method to find the customers by DOB, passing the following argument:
List<Customer> customers = controller.findCustomersByDOB(DOB_TO_SEARCH);
```

The arguments entered in the `findByDOB()` method are captured by the `verify()` function, as shown in the following image:

```
// Check if the findByDOB was called, and while doing so, capture the arguments:
verify(customerService).findByDOB(dobArgument.capture());
```

Finally, the argument obtained from the mocked service function is printed to the console:

```
// Print out the captured arguments:
System.out.println("Captured arguments in findByDOB : " + dobArgument.getValue());
```

When the test is executed, the entered argument, which is the DOB_TO_SEARCH string of “09-03-1990”, is printed onto the console:

```
Captured arguments in findByDOB : 09-03-1990
```