ABOUT MOHAMMAD MERAJ ZIA

I did my Engineering in Information Technology from IET, Lucknow, India. Currently doing MSc in Information Technology from Derby University. I have worked in Java/J2EE domain for the last 10 years. Have good understanding of Payment and Finance domains.

Test-Driven Development With Mockito

Posted by: Mohammad Meraj Zia in Mockito July 13th, 2016

In this example we will learn how to do a **T**est **D**riven **D**evelopment (TDD) using Mockito. A unit test should test a class in isolation. Side effects from other classes or the system should be eliminated if possible. Mockito lets you write beautiful tests with a clean & simple API. Tools and technologies used in this example are Java 1.8, Eclipse Luna 4.4.2

1. Introduction

Mockito is a popular mocking framework which can be used in conjunction with JUnit. Mockito allows us to create and configure mock objects. Using Mockito simplifies the development of tests for classes with external dependencies significantly. We can create the mock objects manually or we can use the mocking framewors like Mockito, EasyMock. jMock etc. Mock frameworks allow us to create mock objects at runtime and define their behavior. The classical example for a mock object is a data provider. In production a real database is used, but for testing a mock object simulates the database and ensures that the test conditions are always the same.

2. Test Driven Development

Test-Driven Development (TDD) is an evolutionary approach to development. It offers test-first development where the production code is written only to satisfy a test. TDD is the new way of programming. Here the rule is very simple; it is as follows:

1. Write a test to add a new capability (automate tests).
2. Write code only to satisfy tests.

3. Re-run the tests—if any test is broken, revert the change.
4. Refactor and make sure all tests are green.
5. Continue with step 1.

3. Creating a project

Below are the steps required to create the project.

- Open Eclipse. Go to File=>New=>Java Project. In the 'Project name' enter 'TDDMockito'.

Figure 1. Create Java Project

- Eclipse will create a 'src' folder. Right click on the 'src' folder and choose New=>Package. In the 'Name' text-box enter 'com.javacodegeeks'. Click 'Finish'.

Figure 2. New Java Package

- Right click on the package and choose New=>Class. Give the class name and click 'Finish'. Eclipse will create a default class with the given name.

3.1 Dependencies

For this example we need the junit and mockito jars. These jars can be downloaded from Maven repository. We are using 'junit-4.12.jar' and 'mockito-all-1.10.19.jar'. There are the latests (non-beta) versions available as per now. To add these jars in the classpath right click on the project and choose Build Path=>Configure Build Path. The click on the 'Add External JARs' button on the right hand side. Then go to the location where you have downloaded these jars. Then click ok.

4. Test first

Let's say we want to build a tool for Report generation. Please note that this is a very simple example of showing how to use mockito for TDD. It does not focus on developing a full report generation tool.

For this we will need three classes. The first one is the interface which will define the API to generate the report. The second one is the report entity itself and the third one is the service class. First we will start with writing the test.

We will inject the service class by using @InjectMocks.

```
1 @InjectMocks private ReportGeneratorService reportGeneratorService;
```

@InjectMocks mark a field on which injection should be performed. It allows shorthand mock and spy injection. Mockito will try to inject mocks only either by constructor injection, setter injection, or property injection in order and as described below. If any of the following strategy fail, then Mockito won't report failure i.e. you will have to provide dependencies yourself.

Constructor injection: the biggest constructor is chosen, then arguments are resolved with mocks declared in the test only. If the object is successfully created with the constructor, then Mockito won't try the other strategies. Mockito has decided not to corrupt an object if it has a parametered constructor. If arguments can not be found, then null is passed. If non-mockable types are wanted, then constructor injection won't happen. In these cases, you will have to satisfy dependencies yourself.

Property setter injection: mocks will first be resolved by type (if a single type matches injection will happen regardless of the name), then, if there are several property of the same type, by the match of the property name and the mock name. If you have properties with the same type (or same erasure), it's better to name all @Mock annotated fields with the matching properties, otherwise Mockito might get confused and injection won't happen. If @InjectMocks instance wasn't initialized before and have a no-arg constructor, then it will be initialized with this constructor.

Field injection: mocks will first be resolved by type (if a single type matches injection will happen regardless of the name), then, if there is several property of the same type, by the match of the field name and the mock name. If you have fields with the same type (or same erasure), it's better to name all @Mock annotated fields with the matching fields, otherwise Mockito might get confused and injection won't happen. If @InjectMocks instance wasn't initialized before and have a no-arg constructor, then it will be initialized with this constructor.

Now we will mock the interface using @Mock annotation:

```
1 @Mock private IReportGenerator reportGenerator;
```

Now we will define the argument captor on report entity:

```
1 @Captor private ArgumentCaptor<ReportEntity> reportCaptor;
```

The ArgumentCaptor class is used to capture argument values for further assertions. Mockito verifies argument values in natural java style: by using an equals() method. This is also the recommended way of matching arguments because it makes tests clean & simple. In some situations though, it is helpful to assert on certain arguments after the actual verification.

Now we will define a setup method which we will annotate with @Before. This we will use to initialize the mocks.

```
1 MockitoAnnotations.initMocks(this);
```

initMocks() initializes objects annotated with Mockito annotations for given test class.

In the test method we will call the generateReport() method of the ReportGeneratorService class passing the required parameters:

```
1 reportGeneratorService.generateReport(startDate.getTime(), endDate.getTime(),
reportContent.getBytes());
```

Below is the snippet of the whole test class:

ReportGeneratorServiceTest.java

```
01 package com.javacodegeeks;
02
03 import static org.junit.Assert.assertEquals;
04
05 import java.util.Calendar;
06
07 import org.junit.Before;
08 import org.junit.Test;
09 import org.mockito.ArgumentCaptor;
10 import org.mockito.Captor;
11 import org.mockito.InjectMocks;
12 import org.mockito.Mock;
13 import org.mockito.Mockito;
14 import org.mockito.MockitoAnnotations;
15
16 public class ReportGeneratorServiceTest {
17
18   @InjectMocks private ReportGeneratorService reportGeneratorService;
```

```
  @Mock private IReportGenerator reportGenerator;
  @Captor private ArgumentCaptor<ReportEntity> reportCaptor;

  @Before
  public void setUp() {
    MockitoAnnotations.initMocks(this);
  }

  @SuppressWarnings("deprecation")
  @Test
  public void test() {
    Calendar startDate = Calendar.getInstance();
    startDate.set(2016, 11, 25);
    Calendar endDate = Calendar.getInstance();
    endDate.set(9999, 12, 31);
    String reportContent = "Report Content";
    reportGeneratorService.generateReport(startDate.getTime(), endDate.getTime(),
reportContent.getBytes());

    Mockito.verify(reportGenerator).generateReport(reportCaptor.capture());

    ReportEntity report = reportCaptor.getValue();

    assertEquals(116, report.getStartDate().getYear());
    assertEquals(11, report.getStartDate().getMonth());
    assertEquals(25, report.getStartDate().getDate());

    assertEquals(8100, report.getEndDate().getYear());
    assertEquals(0, report.getEndDate().getMonth());
    assertEquals(31, report.getEndDate().getDate());

    assertEquals("Report Content", new String(report.getContent()));
  }

}
```

The test class will not compile as the required classes are missing here. Don't worry as this is how TDD works. First we write the test then we build our classes to satisfy the test requirements.

Now lets start adding the classes. First we will add the interface. This is the same interface which we mocked in our test class. The service class will have reference to this interface.

IReportGenerator.java

```
package com.javacodegeeks;

/**
* Interface for generating reports.
* @author Meraj
*/
public interface IReportGenerator {

  /**
  * Generate report.
  * @param report Report entity.
  */
  void generateReport(ReportEntity report);

}
```

Please note that this interface will also not compile as the ReportEntity class is still missing. Now lets add the entity class. This class represents the domain object in our design.

ReportEntity.java

```
package com.javacodegeeks;

import java.util.Date;

/**
* Report entity.
* @author Meraj
*/
public class ReportEntity {

  private Long reportId;
  private Date startDate;
  private Date endDate;
  private byte[] content;

  public Long getReportId() {
    return reportId;
  }

  public void setReportId(Long reportId) {
    this.reportId = reportId;
  }

  public Date getStartDate() {
    return startDate;
  }

  public void setStartDate(Date startDate) {
    this.startDate = startDate;
  }

  public Date getEndDate() {
    return endDate;
  }

```

```
  public void setEndDate(Date endDate) {
    this.endDate = endDate;
  }

  public byte[] getContent() {
    return content;
  }

  public void setContent(byte[] content) {
    this.content = content;
  }
}
```

Now lets add the service class:

ReportGeneratorService.java

```
package com.javacodegeeks;

import java.util.Date;

/**
* Service class for generating report.
* @author Meraj
*/
public class ReportGeneratorService {

  private IReportGenerator reportGenerator;

  /**
  * Generate report.
  * @param startDate start date
  * @param endDate end date
  * @param content report content
  */
  public void generateReport(Date startDate, Date endDate, byte[] content) {
    ReportEntity report = new ReportEntity();
    report.setContent(content);
    report.setStartDate(startDate);
    report.setEndDate(endDate);
    reportGenerator.generateReport(report);
  }

}
```

Now all the classes will compile and we can run our test class.

5. Download the source file

This was an example of using Mockito to do Test Driven Development.

Download
You can download the full source code of this example here: **TDD Mockito**

Tagged with: MOCKITO TDD