1 - Introdução a Objetos Mocks

O termo objeto Mock (em português, objeto simulado) é utilizado para descrever um caso especial de objetos que imitam objetos reais para teste.

Eles são criados para testar o comportamento de outros objetos. Em outras palavras, eles são objetos falsos que simulam o comportamento de uma classe ou objeto "real" para que possamos focar o teste na unidade a ser testada.

Os testes simulados podem ser empregados nas seguintes situações:

- A técnica de Desenvolvimento Guiado por Testes (Test Driven Development - TDD) diz que o desenvolvedor deve primeiramente escrever um caso de teste automatizado para testar o comportamento do objeto a ser programado, para posteriormente codificar as classes reais.
    - A questão é, como é possível testar um código que ainda não foi programado?
        - Os objetos Mock podem simular o comportamento dos objetos, desta forma, é possível testar o comportamento da classe a ser codificada.
- Nos testes unitários os objetos Mock podem simular o comportamento de objetos reais complexos e que são difíceis de serem obtidos, tais como:
    - Objetos que envolvem conexões remotas, desta forma, o tempo de construção do objeto pode ser indeterminado, e com isso, o tempo para testar esses objetos em diversas situações pode ser inviável;
    - Objetos que dependem da leitura de bases de dados grandes;
    - Objetos que possuem valores não determinísticos, tais como, temperatura e direção do vento;
    - Objetos que possuem estados difíceis de reproduzir, tal como, situações de erro.

Os objetos Mocks possuem as seguintes limitações:

- Ao modificar o "código a ser testado" tem-se de fazer as mesmas modificações nos objetos Mocks. Manutenções incorretas nos objetos Mocks podem mascarar erros que só seriam percebidos quando os testes unitários utilizassem instâncias de classes reais;
- Os objetos Mocks não respeitam a ideia do baixo acomplamento, ou seja, os testes com objetos Mock são completamente acoplados com a implementação do objeto a ser testado. Dessa forma, ao refatorar a classe real os testes com o Mock podem não refletir a realidade.

Os objetos Mock podem ser criados através de frameworks que facilitam a sua criação. Praticamente todas as principais linguagens possuem frameworks para a criação de objetos Mock. Para o Java podem ser usados, entre outros, o EasyMock, Mockito, PowerMock, JMockit e JMock. Aqui usaremos o Mockito, por ser considerado mais agradável e simples que os outros projetos. Uma desvantagem do Mockito é que ele não suporta o Mock de métodos estáticos. Para mais detalhes sobre a biblioteca acesse http://site.mockito.org/mockito/docs/current/org/mockito/Mockito.html.

2 - Incluir a Biblioteca Mockito no Projeto

Acesse o endereço http://mvnrepository.com/artifact/org.mockito/mockito-all para fazer o download do arquivo mockito-all-1.10.19.jar, que contém a biblioteca mockito.

Para programar os testes primeiramente crie um projeto de nome Aula4 no IDE Eclipse, não esqueça de incluir a biblioteca JUnit, assim como na Figura 1. Na sequência:

- Crie um source folder de nome test;
- Adicione os pacotes de nome aula nos folders src e test;
- Crie um folder (não é source folder) de nome lib;
- Copie o arquivo mockito-all-1.10.19.jar para o folder lib;
- Na sequência é necessário incluir a biblioteca mockito-all-1.10.19.jar no classpath do projeto Aula4. Então clique com o botão direito do mouse sobre o nome do projeto Aula4 e acesse a opção properties, e siga os passos da Figura 2.

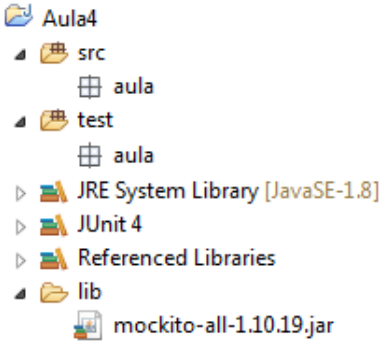

Figura 1 – Estrutura do projeto Aula4.

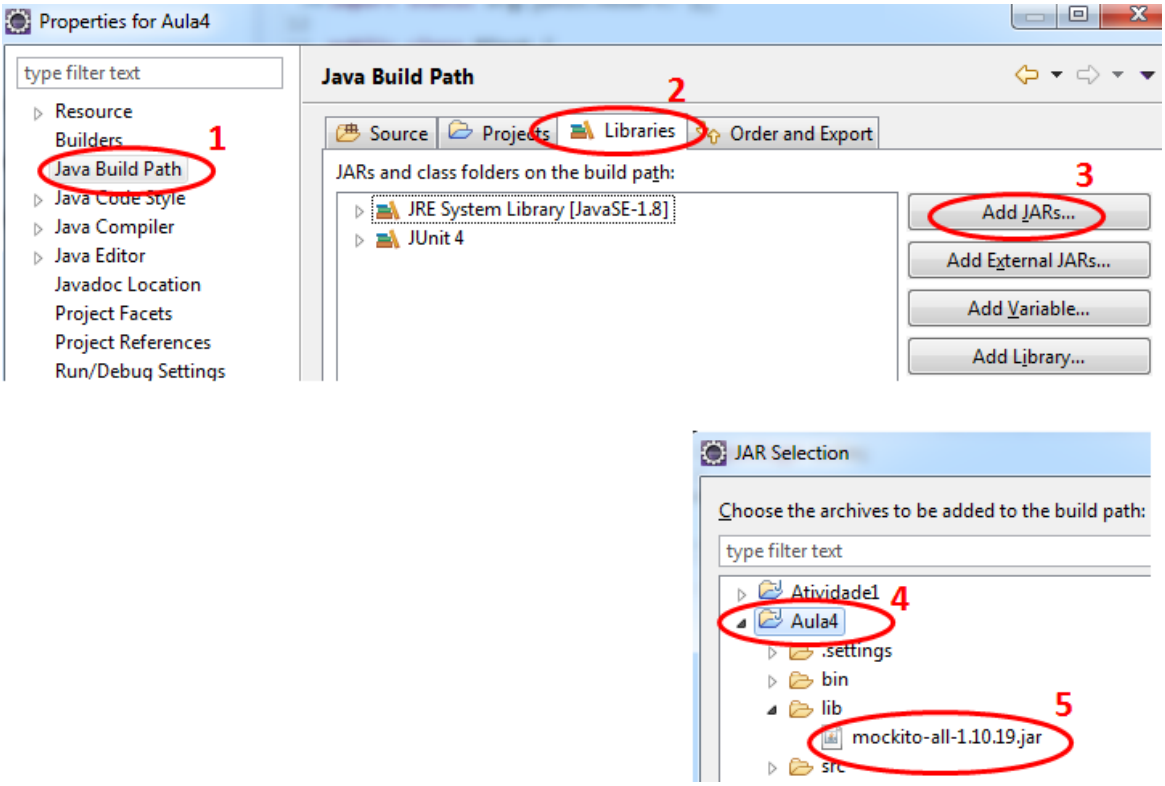

Figura 2 – Sequência de passos para adicionar a biblioteca mockito no classpath do projeto Aula4.

3 - Teste Usando Mockito e JUnit

A biblioteca mockito cria objetos fictícios de classes concretas, abstratas e interfaces. A biblioteca usa Reflexões Java para criar os objetos fictícios, pois métodos abstratos não possuem corpo, logo classes abstratas e interfaces não podem ser instanciadas.

A Figura 3 mostra o código da classe abstrata Operacao e a Figura 4 mostra uma classe para testar os métodos da classe Operacao. Para fazer um teste usando um objeto simulado são necessários os 4 passos a seguir:

i. Crie um atributo do tipo a ser testado e anote-o com @Mock. Neste exemplo o caso a ser testado é Operacao:

```
@Mock
private Operacao op = null;
```

ii. Use o método mock para criar uma instancia da classe a ser testada. Observação, não precisa ser necessariamente uma classe concreta, pode ser abstrata ou interface:

```
op = mock(Operacao.class);
```

iii. Use o método when para simular um comportamento para o método a ser testado (neste exemplo é o método maior com os parâmetros 4 e 2) e thenReturn para indicar o valor a ser retornado pela chamada especificada por when (neste exemplo o retorno 4 está vinculado apenas a chamada maior(4, 2)):

```
when(op.maior(4, 2)).thenReturn(4);
```

iv. Use o método verify para checar a quantidade de vezes que um método foi chamado com determinados parâmetros. Neste exemplo, verify irá checar se o método maior foi chamado exatamente 1 vez com os parâmetros 4 e 2:

```
verify(op, times(1)).maior(4,2);
```

O método verify pode ser invocado passando os seguintes parâmetros:

```
verify(op, times(1)).maior(4,2);
```

- O primeiro parâmetro precisa ser um objeto do tipo a ser simulado (Mock), que neste exemplo é op;
- O método times(nro) é usado para indicar a quantidade de vezes que queremos verificar que o método em teste foi chamado. As duas chamadas a seguir possuem o mesmo significado, pois o segundo parâmetro é times(1) por padrão:

```
verify(op).maior(4,2);
verify(op, times(1)).maior(4,2);
```

- Para indicar a quantidade de vezes que esperamos que o método em teste seja chamado podemos usar ainda:
  - never(): equivalente a times(0);
  - atLeastOnce(): pelo menos uma vez;
  - atLeast(n): pelo menos n vezes;
  - atMost(n): no máximo n vezes.

O método reset é usado para resetar o objeto Mock. Para testar adicione o método test4 na classe OperacaoTest. Veja que o teste maior(1,1) irá falhar, pois o objeto op foi resetado.

```
@Test
public void test4() {
   when(op.maior(1, 1)).thenReturn(1);
   reset(op);
   assertSame(1, op.maior(1, 1));
}
```

O método thenThrow faz a chamada do método lançar uma exceção. No método test5 está sendo testado se a chamada do método maior(1,1) irá lançar uma exceção do tipo RuntimeException.

```
@Test(expected= RuntimeException.class)
public void test5() {
   when(op.maior(1, 1)).thenThrow(new RuntimeException());
   /* a chamada a seguir irá lançar uma exceção do tipo RuntimeException */
   assertSame( 1, op.maior(1, 1));
}
```

```
package aula;
public abstract class Operacao {
   public abstract int maior(int a, int b);

   public abstract int incrementa(int a);

   public abstract void imprimir(String msg);

   public boolean isPar(Integer a){
      return a%2 == 0;
   }
}
```

Figura 3 – Código da classe abstrata Operacao.

```
package aula;
package aula;
import static org.junit.Assert.*;
import static org.mockito.Mockito.*;
import org.junit.Before;
import org.junit.Test;
import org.mockito.Mock;

public class OperacaoTest {
   /* a anotação @Mock indica que este
    * atributo deverá receber um objeto Mock */
   @Mock
   private Operacao op = null;

   @Before
   public void setUp() throws Exception {
      /* o método mock cria um objeto do tipo Operacao */
      op = mock(Operacao.class);
      /* o método when é usado para indicarmos quais
       * métodos e com quais parâmetros serão testados.
       * O método thenReturn indica o valor retornado
       * quando o método especificado no when for chamado */
      when(op.maior(4, 2)).thenReturn(4);
      when(op.maior(2, 4)).thenReturn(4);
   }

   @Test
   public void test1() {
      /* checa se o método maior(4,2) foi chamado zero vezes */
      verify(op, times(0)).maior(4,2);
      /* assertSame é um método do framework JUnit, ou seja,
       * não tem relação com o objeto Mock*/
      assertSame( 4, op.maior(4, 2));
/* checa se o método maior(4,2) foi chamado exatamente 1 vez */
      verify(op, times(1)).maior(4,2);
   }

   @Test
   public void test2() {
      verify(op, times(0)).maior(4,2);
      assertSame( 4, op.maior(4, 2));
      verify(op, times(1)).maior(4,2);
   }

   @Test
   public void test3() {
      verify(op, times(0)).maior(2,4);
      assertSame( 4, op.maior(2, 4));
      verify(op, times(1)).maior(2,4);
   }
}
```

Figura 4 – Código da classe OperacaoTest.

O método doThrow provê ao objeto Mock a possibilidade de lançar uma exceção. Ele geralmente é usado para testar métodos com retorno void. Para testar adicione o método test6, a seguir, na classe OperacaoTest. Veja que o test6 irá lançar uma exceção do tipo Exception.

```
@Test(expected=Exception.class)
public void test6() {
   /* o método doThrow provê a capacidade do objeto Mock de lançar uma exceção */
   doThrow(new Exception("Teste de exceção")).when(op).maior(4,2);
}
```

Um método concreto pode ser simulado ou ter a sua implementação concreta sendo invocada a partir de um objeto Mock. No método test7, a seguir, o método isPar é simulado, já no método test8 fez se a chamado do método concreto. O método thenCallRealMethod é usado para indicar que o objeto Mock deverá chamar o método real e não o simulado.

```
@Test
public void test7() {
   /* será chamado o método simulado isPar */
   when(op.isPar(3)).thenReturn(true);
   assertTrue(op.isPar(3));
   verify(op, times(1)).isPar(3);
}

@Test
public void test8() {
   /* indica que deverá ser chamado o método concreto isPar */
   when(op.isPar(3)).thenCallRealMethod();
   assertFalse(op.isPar(3));
   verify(op, times(1)).isPar(3);
}
```

Existem métodos para indicar qualquer valor em um tipo de dado, tal como, anyInt(), anyFloat(), anyString(), entre outros. No exemplo do test9 qualquer chamada com número inteiro irá chamar o método isPar concreto.

```
@Test
public void test9() {
   /* anyInt() é qualquer int, integer ou null */
   when(op.isPar(anyInt())).thenCallRealMethod();
   assertFalse(op.isPar(null));
   verify(op, times(1)).isPar(anyInt());
}
```

É possível definir os valores retornados em chamadas consecutivas do mesmo método. No test10 definiu-se o valor retornado para as 5 primeiras chamadas do método incrementa(1). Sendo que a última irá lançar uma exceção do tipo Exception.

```
@Test(expected=Exception.class)
public void test10() {
   /* a 1a chamada do método incrementa(1) irá retornar 1
    * a 2a irá retornar 2
    * a 3a irá retornar 3
    * a 4a irá retornar 4
    * a 5a irá lançar uma exceção */
   when(op.incrementa(1)).thenReturn(1,2,3,4).thenThrow(new Exception());
```

```
    System.out.println( op.incrementa(1) );
    System.out.println( op.incrementa(1) );
    System.out.println( op.incrementa(1) );
    System.out.println( op.incrementa(1) );
    System.out.println( op.incrementa(1) );
}
```

O método verify pode receber o método timeout para fazer a chamada durar o tempo especificado. No exemplo a seguir, a chamada duraria 1 segundo, ou seja, o teste irá falhar, pois ele será interrompido após 0.5 segundo.

```
@Test(timeout=500)
public void test11(){
    verify(op, timeout(1000)).maior(4,2);
    assertSame( 4, op.maior(4, 2));
}
```

A biblioteca Mockito não pode simular:

- Métodos privados (https://github.com/mockito/mockito/wiki/Mockito-And-Private-Methods);
- Métodos estáticos;
- Métodos finais;
- Classes finais;
- Construtores.