Especialização em Tecnologias Aplicadas a Sistemas de Informação com Metodologias Ágeis
Disciplina: Design Patterns, Smells e Refactoring
Professor: Guilherme Lacerda

Workshop Ferramenta Mockito

1. AUTORES

Alécio Dalprá - aleciodalpra@gmail.com

Fábio Castilhos - fabcastilhos@gmail.com

2. INTRODUÇÃO

O Mockito é um framework de código aberto, desenvolvido pela Google para uso em testes de unidade na linguagem Java. Possui algumas iniciativas para utilização em outras linguagens como *Python*, *Flex*, *JavaScript*, *Scala*, *Objective C*, *Pearl*, *PHP* e *MATLAB*.

Com o Mockito é possível entregar um código de teste mais fácil de ler e modificar, reduzindo ou minimizando acoplamentos. Ele simula a utilização um objeto real para uma classe específica, auxiliando nos testes de unidade quando precisamos verificar a comunicação entre objetos.

O Mockito difere um pouco dos outros frameworks, pois tenta eliminar o padrão "esperar/executar/verificar" que é seguido por grande parte dos *frameworks*. Sua utilização é bastante simplificada, e para começar a utilizá-lo basta baixar o pacote no site do projeto. Possui também uma documentação bastante útil, recheada de exemplos.

Como principais características do Mockito podemos citar:

- O setup para a criação de um *mock*, diferentemente de outros *frameworks*, é bastante simples.
- Possibilidade de criar *mocks* tanto de classes concretas como de *interfaces*.
- Criação de *mocks* por meio de anotações.
- Facilidade na verificação de erros: a verificação de testes que falham é simples, bastando olhar o *stack trace* disponibilizado.
- Permite verificar a quantidade de vezes que um método foi chamado.
- Permite a verificação por meio de matchers de argumentos.

3. MÉTODOS E FUNÇÕES

O Mockito fornece uma série de métodos para comportamentos de *mocks* específicos:

Método	**Descrição**
thenReturn(T valueToBeReturned)	Retorna valor dado
thenThrow(Throwable toBeThrown) thenThrow(Class<?extends Throwable> toBeThrown)	Lança uma exceção
then(Answer answer) thenAnswer(Answer answer)	Usa o código criado pelo usuário para responda
thenCallRealMethod()	Chama o método real quando trabalhando com parcial mock /spy
doThrow(Throwable toBeThrown) doThrow(Class<?extends Throwable> toBeThrown)	Lança uma exceção dada
doAnswer(Answer answer)	Usa o código criado pelo usuário para responder
doCallRealMethod()	Trabalhando com espião
doNothing()	Faz nada
doReturn(Object toBeReturned)	Retorna valor dado (não para vazio métodos)

Por padrão, o Mockito verifica se um determinado método foi chamado uma vez e apenas uma vez. Com ele é possível criar *mocks* de verificação personalizado:

Nome	**O método de verificação foi:**
times(int wantedNumberOfInvocations)	Chamado exatamente 'n' vezes (um por padrão) fl
never()	Nunca foi chamado.
atLeastOnce()	Chamado pelo menos uma vez.
atLeast(int minNumberOfInvocations)	Chamado pelo menos 'n' vezes.
atMost(int maxNumberOfInvocations)	Chamado no máximo 'n' vezes.
only()	O único método chamado pelo *mock*
timeout(int millis)	Interagiu em um intervalo de tempo especificado.

O Mockito também permite redefinir um valor padrão retornado nos métodos:

Resposta Padrão	**Descrição**
RETURNS_DEFAULTS	Retorna um valor "vazio" padrão (por exemplo, nulo, o, falso, coleção vazia): usado por padrão
RETURNS_SMART_NULLS	Cria um espião de um determinado objeto
RETURNS_MOCKS	Retorna um valor "vazio" padrão, mas um mock em vez de nulo
RETURNS_DEEP_STUBS	Permite um simples esboço profundo (por exemplo, Dado (ourMock.getObject (). GetValue ()). WillReturn (s))
CALLS_REAL_METHODS	Ligue para um método real de objeto mockado

4. ANOTAÇÕES

Mockito oferece três anotações - @Mock, @Spy, @Captor - para simplificar a processo de criação de objetos relevantes usando métodos estáticos. A anotação @InjectMocks simplifica a falsificação e a injeção de espião. Pode injetar objetos usando injeção de construtor, injeção de *setter* ou injeção de campo.

Notação	**Responsabilidade**
@Mock	Cria uma *mock* de um determinado tipo.
Spy	Cria *spy* de um determinado objeto.
@Captor	Cria um argumento *captor* de um determinado tipo.
@InjectMocks	Cria um objeto InjectMocks de um determinado tipo.

5. LIMITAÇÕES

O Mockito possui algumas limitações que merecem ser lembradas. Elas são geralmente restrições técnicas, que podem levar a criação de códigos pouco verificáveis. Com o Mockito não se deve criar:

- *Mocks* de classes finais.
- *Mocks* de *enums*.
- *Mocks* de métodos finais.
- *Mocks* de métodos estáticos.
- *Mocks* de métodos privados.
- *Mocks* de hashCode() e equals().

No entanto, ao trabalhar com um código legado mal escrito, lembre-se de que algumas das limitações mencionadas acima podem ser mitigadas usando as bibliotecas do *PowerMock* ou *JMockit*.

6. INSTRUÇÕES DE INSTALAÇÃO

Os artefatos Mockito estão disponíveis no *Maven Central Repository* (MCR). A maneira mais fácil de disponibilizar o MCR no seu projeto é colocar a seguinte configuração no seu gerenciador de dependências:

MAVEN	`<dependency>`
	`<groupId>org.mockito</groupId>`
	`<artifactId>mockito-core</artifactId>`
	`<version>1.9.5</version>`
	`<scope>test</scope>`
	`</dependency>`

GRADLE	`testCompile "org.mockito: mockito -core: 1.9.5"`
IVY	`<dependency org="org.mockito" name="mockito-core" rev="1.9.5" conf="test->default"/>`

Também é possível instalar o *framework* Mockito executando a sequência de passos descritos a seguir:

Passo 1: Instalar Mockito *framework*

1. Baixar a versão mais recente do *framework* Mockito, acessando: https://code.google.com/p/mockito/downloads/list

2. Descompactar o pacote mockito-2.0.2.zip e salvar o arquivo .jar em sua unidade C: C: \> Mockito.

SO	Nome do Arquivo
Windows	`mockito-all-1.9.5-beta.jar`
Linux	`mockito-all-1.9.5-beta.jar`
Mac	`mockito-all-1.9.5-beta.jar`

3. Definir a variável de ambiente **Mockito_HOME** para apontar para a localização do diretório base onde o Mockito *framework*, onde os arquivos de dependência foram armazenados:

SO	Saída
Windows	`Defina a variável de ambiente Mockito_HOME para C: \ Mockito`
Linux	`Exportar Mockito_HOME = / usr / local / Mockito`
Mac	`Exportar Mockito_HOME = / Library / Mockito`

Passo 2: Definir variável CLASSPATH

4. Definir a variável de ambiente **CLASSPATH** para apontar para o local onde o .jar do Mockito *framework* foi armazenado:

SO	Saída
Windows	`Defina a variável de ambiente CLASSPATH para% CLASSPATH%;% Mockito_HOME% \ mockito-all-1.9.5-beta.jar;.;`
Linux	`Exportar CLASSPATH = $ CLASSPATH: $ Mockito_HOME / mockito-all-1.9.5-beta.jar :.`
Mac	`Exportar CLASSPATH = $ CLASSPATH: $ Mockito_HOME / mockito-all-1.9.5-beta.jar :.`

7. EXEMPLO DE USO

7.1. Exemplo 1: Teste de Estoque com Mockito

O Mockito é simples e fácil de usar. No exemplo a seguir, iremos criar uma simulação passo a passo de Estoque para obter o preço falso de alguns produtos:

Passo 1: Crie uma classe JAVA denominada Estoque.java

```
1  package br.com.uniritter.mockito;
2
3  public class Estoque {
4     private String estoqueId;
5     private String nome;
6     private int quantidade;
7
8     public Estoque(String estoqueId, String nome, int quantidade){
9        this.estoqueId = estoqueId;
10       this.nomeme = nome;
11       this.quantidade = quantidade;
12    }
13
14    public String getEstoqueId() {
15       return estoqueId;
16    }
17
18    public void setEstoqueId(String estoqueId) {
19       this.estoqueId = estoqueId;
20    }
21
22    public int getQuantidade() {
23       return quantidade;
24    }
25
26    public String getTicker() {
27       return nome;
28    }
29 }
```

Passo 2: Crie uma interface denominada EstoqueService.java para obter o preço de uma ação

```
1  package br.com.uniritter.mockito;
2
3  public interface EstoqueService {
4     public double getPreco(Estoque estoque);
5  }
```

Passo 3: Crie a classe denominada CarteiraProdutos.java para representar a carteira dos clientes

```java
package br.com.uniritter.mockito;

import java.util.List;

public class CarteiraProdutos {
   private EstoqueService estoqueService;
   private List<Estoque> estoques;

   public EstoqueService getEstoqueService() {
      return estoqueService;
   }

   public void setEstoqueService(EstoqueService estoqueService) {
      this.estoqueService = estoqueService;
   }

   public List<Estoque> getEstoques() {
      return estoques;
   }

   public void setEstoques(List<Estoque> estoques) {
      this.estoques = estoques;
   }

   public double getValorMercado(){
      double valorMercado = 0.0;

      for(Estoque estoque:estoques){
         valorMercado += estoqueService.getPreco(estoque) * estoque.getQuantidade();
      }
      return valorMercado;
   }
}
```

Passo 4: Teste a classe CarteiraProdutosTest.java

Vamos testar a classe CarteiraProdutos, injetando nele uma falsificação de estoque. O Mockito criará simulações.

```java
package br.com.uniritter.mockito;

import java.util.ArrayList;
import java.util.List;
import static org.mockito.Mockito.*;
import br.com.uniritter.mockito.Estoque;
import br.com.uniritter.mockito.EstoqueService;
import br.com.uniritter.mockito.CarteiraProdutos;

public class CarteiraProdutosTest {

   CarteiraProdutos carteiraProdutos;
   EstoqueService estoqueService;

   public static void main(String[] args){
      CarteiraProdutosTest teste = new CarteiraProdutosTest();
      teste.setAcima();
      System.out.println(teste.testeValorMercado()?"PASSOU":"FALHOU");
   }

   public void setAcima(){
      // Crie um objeto CarteiraProdutos que seja testavel
      carteiraProdutos = new CarteiraProdutos();

      // Crie o objeto mock do servico Estoque
      estoqueService = mock(EstoqueService.class);

      // Configure o Estoque para a Carteira de Produtos
      carteiraProdutos.setEstoqueService(estoqueService);
   }

   public boolean testeValorMercado(){

      // Cria uma lista de produtos a serem adicionados à Carteira de Produtos
      List<Estoque> estoques = new ArrayList<Estoque>();
      Estoque koenigseggEstoque = new Estoque("1","Koenigsegg CCXR Trevita", 10);
      Estoque lamborghiniEstoque = new Estoque("2","Lamborghini Veneno",100);
      Estoque lykanEstoque = new Estoque("3","Lykan Hypersport ",100);
      Estoque bugattiEstoque = new Estoque("4","Bugatti Veyron Mansory Vivere", 10);
      Estoque ferrariEstoque = new Estoque("5","Ferrari FXX K",100);

      estoques.add(koenigseggEstoque);
      estoques.add(lamborghiniEstoque);
      estoques.add(lamborghiniEstoque);
      estoques.add(bugattiEstoque);
      estoques.add(ferrariEstoque);

      // Adicione ações a Carteira de Produtos
      carteiraProdutos.setEstoques(estoques);

      // Mock o comportamento do serviço de Estoque para retornar o valor de v produtos
      when(estoqueService.getPreco(koenigseggEstoque)).thenReturn(50.00);
      when(estoqueService.getPreco(lamborghiniEstoque)).thenReturn(1000.00);
      when(estoqueService.getPreco(lykanEstoque)).thenReturn(1000.00);
      when(estoqueService.getPreco(bugattiEstoque)).thenReturn(1000.00);
      when(estoqueService.getPreco(ferrariEstoque)).thenReturn(1000.00);

      double valorMercado = carteiraProdutos.getValorMercado();
      return valorMercado == 400500.0;
   }
}
```

Passo 5: Verifique o resultado

a. Compile as classes usando o compilador javac da seguinte maneira:

```
C:\Mockito_WORKSPACE>javac Estoque.java EstoqueService.java
CarteiraProdutos.java CarteiraProdutosTest.java
CarteiraProdutosTest.java
```

b. Agora execute o CarteiraProdutosTest. java para ver o resultado:

```
C:\Mockito_WORKSPACE>java CarteiraProdutosTest
```

c. Verifique a saída

```
PASSOU
```

7.2. Exemplo 2: Teste de uma Calculadora com Mockito

Neste exemplo iremos demonstrar uma simulação de uma Calculadora utilizando o Mockito:

Passo 1: Crie uma classe JAVA denominada Calculadora.java

```java
package br.com.uniritter.mockito;

public class Calculadora {
    private String descricao;

    public String getDescricao(){
        return descricao;
    }
    public void setDescricao(String descricao){
        this.descricao = descricao;
    }
    public double adicao(double numero1, double numero2){
        return numero1 + numero2;
    }

    public double subtracao(double numero1, double numero2){
        return numero1 - numero2;
    }

    public double multiplicacao(double numero1, double numero2){
        return numero1 * numero2;
    }

    public double divisao(double numero1, double numero2){
        return numero1 / numero2;
    }
}
```

Passo 2: Crie uma classe de teste denominada CalculadoraTests.java

```
1  package br.com.uniritter.mockito;
2
3  import static org.mockito.Mockito.*;
4  import org.junit.Assert;
5  import org.junit.Test;
6
7  public class CalculadoraTest {
8
9      @Test
10     public void Test01(){
11         Calculadora mockedCalc = mock(Calculadora.class);
12
13         // Usando o mock object
14         mockedCalc.setDescricao("Calculadora Mockito");
15         mockedCalc.getDescricao();
16
17         //VerificaÃ§Ãµes
18         verify(mockedCalc).setDescricao("Calculadora Mockito");
19         verify(mockedCalc).getDescricao();
20     }
21
22     @Test
23     public void Test02(){
24
25         Calculadora mockedCalc = mock(Calculadora.class);
26         when(mockedCalc.adicao(3.0, 5.0).thenReturn(8.0));
27
28         Assert.assetEquals(8.0, mockedCalc.adicao(3.0, 5.0), 0.0);
29         Assert.assetEquals(11.0, mockedCalc.adicao(6.0, 5.0), 0.0);
30     }
31 }
```

Foi adicionado comando para criar um retorno "mockado" na linha 14 e 14. O retorno desse mock é testado nas linhas 18 e 19, onde a verificação espera que se retorne, na primeira linha "Calculadora Mockito" e na segunda nenhum valor.

O segundo teste foi "mockado" o retorno da operação de adição ao somar 3 + 5 (linha 27). Na linha 28 é feita a validação. Caso se passe valores diferentes de 3 + 5, o teste falhará, pois, o retorno esperado na linha 26 é 8.

OBRIGADO!