IMD0412 – INTRODUÇÃO AO TESTE DE SOFTWARE

PROF. EIJI ADACHI M. BARBOSA

Roteiro de Laboratório –

Mock Objects

Objetivo:

O objetivo desta atividade é exercitar os conceitos de *test doubles* vistos em sala de aula. Mais especificamente, iremos exercitar a criação de *mock objects* com o framework Mockito. Neste tutorial, cobriremos exemplos bem básicos de uso do Mockito para que nos familiarizemos com seus principais métodos e classes. Para manter os exemplos simples, utilizaremos a interface java.util.List, uma vez que ela é bastante comum e de fácil uso. Obviamente, os exemplos mostrados a seguir servem apenas o propósito de demonstrar o uso básico do Mockito. Em todos os exemplos desta seção, crie na classe SimpleMockExamples os métodos que são mostrados aqui neste tutorial, lembrando de dar um "import static org.mockito.Mockito.*", além de outros import necessários.

Observação:

Junto a este enunciado, veio um projeto do Eclipse para você iniciar sua implementação. Para tanto, basta você descompactar o .zip e importar o projeto pelo menu *File > Import > Existing Project*. Caso você use outra IDE, o projeto é um projeto Maven, então basta descompactar o .zip e usar a opção da sua IDE de importar um projeto Maven.

Criando mock objects.

Para criarmos um mock object usando o Mockito, devemos utilizar o método org.mockito.Mockito.mock, conforme o código abaixo:

```
@Test
public final void testCreatingMockWithMethod ()
{
	List mockList = Mockito.mock(List.class);
	mockList.add("MyStr");
	System.out.println(mockList.get(0));
	System.out.println(mockList.contains("MyStr"));
}
```

No código acima, o método Mockito.mock recebe um tipo de classe (List.class) e retorna um *mock object* deste tipo. Isto é o suficiente para criarmos um *mock object*. Ao executar o código acima, você deverá ver impresso na tela do console *null* e *false*. Isto ocorre porque o objeto *mockList* é apenas um dublê de List, e não uma implementação real de List. Ou seja, ele não armazenou de fato a string "MyStr" na invocação mockList.add("MyStr"); ele simplesmente "finge" que é uma List, mas como não configuramos o mock object ele ainda não consegue fingir um comportamento similar ao de uma List. Mais adiante, veremos como podemos configurar um mock object para que ele simule um determinado comportamento. Por enquanto, estamos interessados apenas em como criar mock objects com o Mockito.

Além do método Mockito.mock, uma outra forma de criarmos mock objects é por meio da anotação org.mockito.Mock, como mostrado no exemplo abaixo (lembre de dar um *"import org.mockito.Mock"*):

```
@Mock
private List mockList;

@Test
public final void testCreatingMockWithAnnotation()
{
	mockList.add("MyStr");

	System.out.println(mockList.get(0));
	System.out.println(mockList.contains("MyStr"));
}
```

Ao executar o código acima pela primeira vez, o teste deverá falhar com uma NullPointerException inesperada. Isto ocorre porque o mock object não foi de fato criado. Para isto, é necessário anotarmos nossa classe de teste com a anotação @RunWith(MockitoJUnitRunner.class). Desta forma, a classe MockitoJUnitRunner será responsável por criar o mock object e injetá-lo na nossa classe de testes.[1] Após acrescentar essa anotação, rode novamente o seu teste. Você deverá ver mais uma vez impresso na tela *null* e *false.*

Especificando comportamento.

Nos exemplos anteriores, apenas criamos *mock objects*; nós não chegamos a especificar o seu comportamento. Para especificarmos o comportamento dos mock objects, usamos o método Mockito.when, como mostrado no código abaixo:

```
@Test(expected=IndexOutOfBoundsException.class)
public final void testSpecifyingBehavior()
{
	// Especificando o comportamento do mock object
	Mockito.when(mockList.get(0)).thenReturn("MyStr1");
	Mockito.when(mockList.get(1)).thenReturn("MyStr2");
	Mockito.when(mockList.get(2)).thenThrow(new IndexOutOfBoundsException());

	// Exercitando o comportamento do mock object
	System.out.println(mockList.get(0));// prints MyStr1
	System.out.println(mockList.get(1));// prints MyStr2
	System.out.println(mockList.get(9));// prints null
	System.out.println(mockList.get(2));// throws exception
}
```

No código acima, as três primeiras linhas usam o método Mockito.when para especificar o comportamento do mock object.[2] A primeira linha especifica que quando alguém invocar o método get do objeto mockList passando o argumento 0, este objeto deverá retornar "MyStr1". A segunda linha especifica comportamento similar ao especificado na primeira linha. Já a terceira linha especifica que quando alguém invocar o método get do objeto mockList passando o argumento 3, este objeto deverá lançar uma exceção do tipo IndexOutOfBoundsException. As últimas quatro linhas do código simplesmente invocam métodos sobre o objeto mockList, isto é, apenas exercitam o comportamento definido para este mock object. Este seria o trecho do código em que o mock object é de fato usado para testar um determinado SUT.

1 Leia mais sobre o que é injeção de dependência em https://en.wikipedia.org/wiki/Dependency_injection
2 Uma boa prática é fazer um import static do método Mockito.when, de modo a deixar o código mais legível.

Ao executar o código acima, você deverá ver impresso na tela do console: *MyStr1*, *MyStr2* e *null*. Os retornos MyStr1 e MyStr2 foram exatamente o que especificamos nas primeiras duas linhas para os casos de invocações get(0) e get(1). Já o null corresponde à invocação get(3). Neste caso, como não especificamos nada para chamadas à get com argumento igual a 3, o Mockito retorna um valor default, que neste caso é *null*. Se você criar uma nova especificação para o comportamento de invocações a get(3), você verá que o valor impresso na tela será igual ao que você especificou. Por fim, a invocação get(2) lança uma exceção do tipo IndexOutOfBoundsException, como especificamos na terceira linha do código. O lançamento desta exceção não faz nosso teste falhar porque especificamos neste caso de teste que esperávamos tal exceção. Remova o *expected* da anotação @Test e execute novamente o caso de teste. Você verá que o teste passará a falhar.

Lembrem que em sala de aula discutimos que mock objects apresenta características de stub e de spy, além de suas características próprias. O que acabamos de aprender nesta são as características de stub que o mock object apresenta. Ou seja, mock objects são capazes de retornar valores quando seus métodos são invocados. Nos exemplos acima, nós definimos o comportamento dos mock objects definindo exatamente quais valores devem ser retornados para chamadas a métodos recebendo argumentos específicos. Nós fomos bastante específicos em nossa especificação. Caso queiramos definir mock objects com comportamento mais simples, podemos usar o conceito de Argument Matchers do Mockito, como mostrado no código abaixo (para este exemplo, dê um *import static org.mockito.Mockito.**):[3]

```
@Test
public final void testSpecifyingBehaviorWithArgumentMatchers()
{
        when(mockList.get(anyInt())).thenReturn("MyStrMatcher");

        System.out.println(mockList.get(0));
        System.out.println(mockList.get(1));
        System.out.println(mockList.get(9));
        System.out.println(mockList.get(2));
}
```

Nos exemplos anteriores, nós especificamos o comportamento do mock object definindo valores específicos dos argumentos recebidos pelo método get, e os valores retornados para cada valor específico. Já no código acima, nós usamos o método anyInt para especificarmos que para qualquer chamada do método get que receba um inteiro, o objeto mockList deverá retornar "MyStrMatcher". Se você executar o código acima, você deverá ver impresso no seu console quatro vezes esta string.

Especificando comportamento de chamadas sucessivas.

O Mockito também permite especificarmos o comportamento de um mock object por meio da definição de quais valores devem ser retornados em chamadas sucessivas de um mesmo método, como mostrado no código abaixo:

[3] Mais informações sobre Argument Matchers são encontradas em http://static.javadoc.io/org.mockito/mockito-core/2.10.0/org/mockito/Mockito.html#argument_matchers

```
@Test
public final void testSpecifyingBehaviorWithSuccessiveInvocations()
{
        when(mockList.get(anyInt())).thenReturn("S1", "S2", "S3");

        System.out.println(mockList.get(0));
        System.out.println(mockList.get(1));
        System.out.println(mockList.get(9));
        System.out.println(mockList.get(2));
        System.out.println(mockList.get(2));
}
```

No exemplo acima, passamos ao método thenReturn uma sequência de strings para especificar quais valores devem ser retornados caso sucessivas chamadas ao método get sejam realizadas no objeto mockList. Se você executar o código acima, você verá impresso na tela do seu console: *S1, S2, S3, S3* e *S3*. Perceba que a string *S3* é impressa três vezes. Isto ocorre porque especificamos apenas três chamadas consecutivas, enquanto nós exercitamos nosso mock object com cinco chamadas consecutivas. Para as quarta e quinta chamadas, o Mockito usa como valor de retorno o último valor especificado no método *thenReturn*.

Especificando comportamento excepcional de métodos void.

Se quisermos especificar o comportamento excepcional de métodos void, uma vez que não faz sentido especificar o retorno de métodos void, nós devemos usar o método doThrow, como mostrado no código abaixo:

```
@Test(expected = UnsupportedOperationException.class)
public final void testSpecifyingBehaviorOfVoidMethods()
{
        doThrow(new UnsupportedOperationException()).when(mockList).clear();

        mockList.clear();
}
```

No código acima, nós primeiros definimos a exceção que deve ser lançada, para depois definirmos qual método irá lançar tal exceção quando for invocado. Esta mesma forma de especificar o comportamento de um mock object, pode ser usada para definir o comportamento excepcional de métodos que não são void. Ou seja, para métodos não void, podemos especificar seu comportamento excepcional de duas formas: when(método).thenThrow(exceção) e doThrow(exceção).when(método). As duas formas são equivalentes.[4]

Verificando comportamento.

Nas seções passadas, nós apenas especificamos o comportamento dos mock objects; nós não chegamos a verificar se a forma como o mock object foi usado de fato atendeu às nossas expectativas. A verificação do comportamento de um mock object é feito utilizando o método verify, como mostrado no código a seguir:

[4] Na verdade, há toda uma família de métodos do*: doThrow, doReturn, doAnswer, etc. Estes métodos servem para especificar o comportamento de qualquer método (void ou não void). Mais sobre isto no link: http://static.javadoc.io/org.mockito/mockito-core/2.10.0/org/mockito/Mockito.html#do_family_methods_stubs

```
@Test
public final void testVerifyingBehavior()
{
	// Exercitando o mock object
	mockList.add("A");
	mockList.clear();
	// Verificando se determinadas invocações foram feitas durante o exercício
	verify(mockList).clear();
	verify(mockList).add("B");
}
```

No código anterior, as duas primeiras linhas apenas exercitam o mock object, da mesma forma que fizemos nas seções passadas. Já as duas linhas seguintes, definem duas expectativas que devem ser atendidas durante o exercício do objeto mockList: (i) o método clear() deve ser invocado, e (ii) o método add deve ser invocado recebendo o argumento "B". Se você executar o código anterior, você verá que o caso de teste falha com a seguinte mensagem:

Argument(s) are different! Wanted:
mockList.add("B");
-> at imd0412.recommender.SimpleMockExamples.testVerifyingBehavior(SimpleMockExamples.java:88)
Actual invocation has different arguments:
mockList.add("A");
-> at imd0412.recommender.SimpleMockExamples.testVerifyingBehavior(SimpleMockExamples.java:84)

Ou seja, a mensagem de erro diz que você verificou se o mock object teve o método add exercitado com o argumento "B", mas o mock object foi de fato exercitado com o argumento "A". Ou seja, sua expectativa não foi atendida. Modifique a última linha do código, verificando que a invocação recebeu o argumento "A". Execute novamente o caso de teste e você verá que o teste passará com sucesso.

Verificando o número de invocações.

O Mockito também permite que nós verifiquemos quantas vezes determinados métodos são invocados num mock object. Para isto, o Mockito provê os métodos: times, never, atLeastOnce, atLeast e atMost. O código a seguir exemplifica estes métodos (leia atentamente os comentários no código):

```
@Test
public final void testVerifyingNumberOfInvocations()
{
       // Exercitando o mock object
        mockList.add("once");
        mockList.add("twice");
        mockList.add("twice");

        // As duas verificações são equivalentes: times(1) é usada por default
        verify(mockList).add("once");
        verify(mockList, times(1)).add("once");

        // Verifica o número exato de invocações
        verify(mockList, times(2)).add("twice");

        // Verifica que uma invocação nunca ocorreu.
        // never() é um 'apelido' para times(0)
        verify(mockList, never()).add("never happened");

        // Verificando que ao menos algumas invocações ocorreram - atLeast()
        // e que no máximo algumas invocações ocorreram - atMost()
        verify(mockList, atLeastOnce()).add("twice");
        verify(mockList, atLeast(2)).add("twice");
        verify(mockList, atMost(5)).add("twice");
}
```

No código acima, nós fazemos diferentes tipos de verificação:

- método times - verificações para que determinados métodos tenham sido invocados um número exato de vezes
- método never – verificações para que determinadas invocações nunca ocorram
- método atLeast – verificações para que determinadas métodos tenham sido invocados um número mínimo de vezes
- método atMost– verificações para que determinadas métodos tenham sido invocados um número máximo de vezes

Faça mudanças no código acima de modo a fazer com que o caso de teste falhe. Faça isso tanto mudando a forma como o mock object é exercitado, quanto mudando as verificações.

Verificando a ordem das invocações.

O Mockito também permite verificarmos se as invocações dos métodos de um mock object obedeceram uma determinada ordem. Para isto, devemos usar um objeto do tipo org.mockito.InOrder, como mostra o código a seguir:

```
@Test
public final void testVerifyingOrderOfInvocations()
{
        // Exercita o mock object
        mockList.add("A");
        mockList.add("B");

        // Cria um 'verificador' de ordem
        InOrder inOrder = inOrder(mockList);

        // Verificações a seguir checam se foi obedecida esta ordem de invocações
        inOrder.verify(mockList).add("A");
        inOrder.verify(mockList).add("B");
        inOrder.verify(mockList).clear();
}
```

No código anterior, as duas últimas linhas definem a ordem de invocações que se deseja verificar: definimos que esperamos primeiro uma invocação ao método add recebendo o argumento "A", seguido de uma invocação ao método add recebendo o argumento "B", seguido de uma invocação ao método clear. Se você executar o código acima, você receberá a seguinte mensagem de erro:

org.mockito.exceptions.verification.VerificationInOrderFailure:
Verification in order failure
Wanted but not invoked:
mockList.clear();
-> at imd0412.recommender.SimpleMockExamples.testVerifyingOrderOfInvocations(SimpleMockExamples.java:128)
Wanted anywhere AFTER following interaction:
mockList.add("B");

Ou seja, a mensagem de erro diz que, após a invocação add("B"), era esperado uma invocação ao método clear, a qual nunca ocorreu. Corrija esta falha no caso de teste exercitando o mock objeto com uma invocação ao método clear na ordem esperada.

Espiando implementações concretas.

Nós vimos nos exemplos anteriores que os mock objects criados pelo Mockito possuem comportamento de stub, ao serem configurados para retornarem valores específicos para determinadas chamadas, e comportamento de spy, ao serem verificados quanto a forma que foram usados pelos testes (métodos verify). O Mockito também permite "espionar" instâncias concretas através da criação de dublês do tipo Spy. O exemplo abaixo mostra como criar um spy usando o Mockito (obs.: para executar o exemplo abaixo, lembre-se de anotar sua classe de testes com @RunWith(MockitJUnitRunner.class)):

```
@Spy
    List<String> spyList = new ArrayList<String>();

    @Test
    public void spyingList () {
        spyList.add("MyStr1");
        spyList.add("MyStr2");

        Mockito.verify(spyList).add("MyStr1");
        Mockito.verify(spyList).add("MyStr2");

        assertEquals(2, spyList.size());
    }
```

No exemplo acima, a partir da anotação @Spy o Mockito cria uma lista-espiã chamada "spyList". Para fazer a espionagem, o restante da execução do teste deve manipular a instância "spyList"; toda invocação sobre a instância "spyList" será registrada. Ao contrário de um mock object, no Mockito, o spy é um objeto "de verdade", por isso não é necessário configurar o seu comportamento, como fizemos com os mock objects dos exemplos passados.

No exemplo acima, usamos o método Mockito.verify para verificar se o estado final da lista-espiã está de acordo com o esperado, da mesma forma que fizemos anteriormente para os mock objects. Perceba nas invocações ao método assertEquals que o tamanho da lista-espiã é igual a 2, ou seja, os elementos foram de fato inseridos sem que fosse necessário configurar um comportamento para o método add (se tivéssemos feito invocação ao método add de um mock object, o tamanho da lista permaneceria igual a 0).

Comentários finais.

Neste tutorial, cobrimos o básico do framework Mockito. Cobrimos como criar mock objects, e como especificar e verificar o seu comportamento. Como discutido em sala de aula, mock objects são test doubles que agregam características de stub e spy, além de ter suas próprias características. Neste contexto, qual a sua opinião sobre o Mockito: ele de fato provê suporte para a criação de mock objects, da forma como são definidos no livro xUnit Patterns?

Nos exemplos que vimos aqui, nós não "mockamos" métodos estáticos. Pesquise como é possível fazer isto. Dica: o Mockito não suporta isso (pelo menos até onde eu conheço ;-). Procure outros frameworks que apoiam a criação de mock objects.

Referências.

- http://static.javadoc.io/org.mockito/mockito-core/2.10.0/org/mockito/Mockito.html
- http://www.vogella.com/tutorials/Mockito/article.html