Neste curso, estudaremos o Mockito. Ele é um dos frameworks de mock mais populares do mercado.

O primeiro passo é é criar um mock do objeto que queremos simular. Em nosso caso, queremos criar um mock de LeilaoDao:

LeilaoDao daoFalso = mock(LeilaoDao.class);

Veja que fizemos uso do método mock. Esse método deve ser importado estaticamente da classe do próprio Mockito:

import static org.mockito.Mockito.\*;

Pronto! Temos um mock criado! Precisamos agora ensiná-lo a se comportar da maneira que esperamos. Vamos ensiná-lo, por exemplo, a devolver a lista de leilões criados quando o método correntes() for invocado:

Calendar antiga = Calendar.getInstance();

antiga.set(1999, 1, 20);

Leilao leilao1 = new CriadorDeLeilao().para("TV de plasma")

.naData(antiga).constroi();

Leilao leilao2 = new CriadorDeLeilao().para("Geladeira")

.naData(antiga).constroi();

List<Leilao> leiloesAntigos = Arrays.asList(leilao1, leilao2);

// criando o mock!

LeilaoDao daoFalso = mock(LeilaoDao.class);

// ensinando o mock a reagir da maneira que esperamos!

when(daoFalso.correntes()).thenReturn(leiloesAntigos);

Veja que o método when(), também do Mockito, recebe o método que queremos simular. Em seguida, o método thenReturn() recebe o que o método falso deve devolver. Veja que simples! Agora, quando invocarmos daoFalso.correntes(), ele devolverá leiloesAntigos!

Levemos esse código agora para nosso método de teste:

@Test

public void deveEncerrarLeiloesQueComecaramUmaSemanaAtras() {

Calendar antiga = Calendar.getInstance();

antiga.set(1999, 1, 20);

Leilao leilao1 = new CriadorDeLeilao().para("TV de plasma")

.naData(antiga).constroi();

Leilao leilao2 = new CriadorDeLeilao().para("Geladeira")

.naData(antiga).constroi();

List<Leilao> leiloesAntigos = Arrays.asList(leilao1, leilao2);

// criamos o mock

LeilaoDao daoFalso = mock(LeilaoDao.class);

// ensinamos ele a retornar a lista de leiloes antigos

when(daoFalso.correntes()).thenReturn(leiloesAntigos);

EncerradorDeLeilao encerrador = new EncerradorDeLeilao();

encerrador.encerra();

assertTrue(leilao1.isEncerrado());

assertTrue(leilao2.isEncerrado());

assertEquals(2, encerrador.getQuantidadeDeEncerrados());

}

Mas, ao executar o teste, ele falha! Isso ocorre porque a classe EncerradorDeLeilao não faz uso do mock que criamos. Veja que ela ainda instancia o DAO falso. Precisamos fazer com que o EncerradorDeLeilao receba o mock na hora do teste e receba a classe de verdade quando o sistema estiver em produção.

Uma solução é receber o LeilaoDao no construtor. Nesse caso, o teste passaria o mock para o Encerrador:

public class EncerradorDeLeilao {

private int encerrados;

private final LeilaoDao dao;

public EncerradorDeLeilao(LeilaoDao dao) {

this.dao = dao;

}

public void encerra() {

List<Leilao> todosLeiloesCorrentes = dao.correntes();

for(Leilao leilao : todosLeiloesCorrentes) {

if(comecouSemanaPassada(leilao)) {

encerrados++;

leilao.encerra();

dao.salva(leilao);

}

}

}

// codigo continua aqui

}

public class EncerradorDeLeilaoTest {

@Test

public void deveEncerrarLeiloesQueComecaramUmaSemanaAtras() {

Calendar antiga = Calendar.getInstance();

antiga.set(1999, 1, 20);

Leilao leilao1 = new CriadorDeLeilao().para("TV de plasma")

.naData(antiga).constroi();

Leilao leilao2 = new CriadorDeLeilao().para("Geladeira")

.naData(antiga).constroi();

List<Leilao> leiloesAntigos = Arrays.asList(leilao1, leilao2);

LeilaoDao daoFalso = mock(LeilaoDao.class);

when(daoFalso.correntes()).thenReturn(leiloesAntigos);

EncerradorDeLeilao encerrador = new EncerradorDeLeilao(daoFalso);

encerrador.encerra();

assertTrue(leilao1.isEncerrado());

assertTrue(leilao2.isEncerrado());

assertEquals(2, encerrador.getQuantidadeDeEncerrados());

}

}

Agora sim! Nosso teste passa! Ao invocar o método encerra(), o DAO que é utilizado é o mock; ele, por sua vez, nos devolve a lista "de mentira", e conseguimos executar o teste!

Perceba como agora foi fácil escrevê-lo, afinal o Mockito facilitou a nossa vida! Conseguimos simular o comportamento do DAO e testar a classe que queríamos sem precisarmos montar cenários no banco de dados!

Mock objects são uma ótima alternativa para facilitar a escrita de testes de unidade para classes que dependem de outras classes!

Quando você invoca um método no mock que não foi previamente ensinado a responder algo, o Mockito faz o seguinte:

* Se o método retorna um inteiro, double, ou um tipo primitivo qualquer, ele retornará 0
* Se o método retorna uma lista, o Mockito retornará uma lista vazia
* Se o método retorna uma outra classe qualquer, o Mockito retorna null.

Sempre defina os comportamentos que você espera no seu mock.

**Métodos estáticos e mocks**

É impossível mockar métodos estáticos!

Por esse motivo, desenvolvedores fanáticos por testes evitam ao máximo criar métodos estáticos!

Fuja deles! Além de serem difíceis de serem testados, ainda não são uma boa prática de orientação a objetos.

**Mockando interfaces**

A grande vantagem é trabalhar sempre voltado para interfaces, que é um importante princípio de orientação a objetos.

Sempre que for trabalhar com mocks, pense em criar interfaces entre suas classes. Dessa forma, seu teste e código passam a depender apenas de um contrato, e não de uma classe concreta.

**Mockito e o Verify**

Escreva um teste que garanta que o método dao.atualiza() é invocado pela classe EncerradorDeLeilao.

Para isso, faça uso do verify().

**Verify garantindo apenas 1 execução**

Para garantir que um método seja invocado apenas uma única vez num mock, devemos passar um segundo argumento para o método verify, que é o times(1):

verify(daoFalso, times(1)).atualiza(leilao1);

Repare que o método times recebe como argumento o número de vezes que o método deve ser invocado. Ou seja, você pode verificar que um método foi chamado exatamente 4 vezes, se for necessário.

**Controlando número de execução de métodos ainda mais**

O método atLeastOnce() vai garantir que o método foi invocado no mínimo uma vez. Ou seja, se ele foi invocado 1, 2, 3 ou mais vezes, o teste passa. Se ele não for invocado, o teste vai falhar.

O método atLeast(numero) funciona de forma análoga ao método acima, com a diferença de que passamos para ele o número mínimo de invocações que um método deve ter.

Por fim, o método atMost(numero) nos garante que um método foi executado até no máximo N vezes. Ou seja, se o método tiver mais invocações do que o que foi passado para o atMost, o teste falha.

**Garantindo que os métodos foram executados na ordem certa**

Exemplo de utilização:

// passamos os mocks que serao verificados

InOrder inOrder = inOrder(daoFalso, carteiroFalso);

// a primeira invocação

inOrder.verify(daoFalso, times(1)).atualiza(leilao1);

// a segunda invocação

inOrder.verify(carteiroFalso, times(1)).envia(leilao1);

**Mockando exceções**

//doThrow: indica que ele deve lançar uma exceção, o quando indica o momento do lançamento dela

Exemplo:

*doThrow*(**new** RuntimeException()).when(daoFalso).atualiza(leilao1);

**Checked Exceptions e Mocks**

o Mockito não consegue fazer com que o método lance Exception.

Exception é uma exceção checada no Java, e seu lançamento precisa ser explicitamente declarado, o que não acontece no método envia() do EnviadorDeEmail.

Por esse motivo a mensagem de erro do teste é: org.mockito.exceptions.base.MockitoException: Checked exception is invalid for this method!. Ou seja, o Mockito percebe isso e falha o teste!

Fique sempre atento às exceções que seu método pode lançar.

**Ignorando o parâmetro no Mock com any()**

Veja que no teste passado repetimos algumas linhas, como por exemplo:

doThrow(new RuntimeException()).when(daoFalso).atualiza(leilao1);

doThrow(new RuntimeException()).when(daoFalso).atualiza(leilao2);

// ...

verify(carteiroFalso, never()).envia(leilao1);

verify(carteiroFalso, never()).envia(leilao2);

Observe que para esses testes não importa se é leilao1 ou leilao2, a exceção será lançada para qualquer leilão. Podemos escrever isso com apenas uma linha informando ao Mockito que não importa qual leilão o mock vai receber usando o método any() da classe org.mockito.Matchers:

import static org.mockito.Matchers.\*;

// ...

doThrow(new RuntimeException()).when(daoFalso).atualiza(any(Leilao.class));

// ...

verify(carteiroFalso, never()).envia(any(Leilao.class));

**Mais sobre argumentos genéricos**

No exercício anterior usamos o método any() da classe org.mockito.ArgumentMatchers para dizer ao Mockito que nosso mock object poderia receber qualquer objeto do tipo Leilao.

A classe ArgumentMatchers tem muitos outros métodos. Veja a [lista de métodos dela](https://github.com/mockito/mockito/blob/release/2.x/src/main/java/org/mockito/ArgumentMatchers.java).

A classe Matchers possui diversos métodos que podem ser usados para especificarmos que argumentos nosso mock pode receber numa chamada de método. Isso permite escrever testes mais facilmente e deixa nosso código mais flexível.

Podemos, por exemplo, garantir que um mock vai ser chamado com uma String começando com "Importante:". Veja só:

verify(meuMock).imprimeMensagem(startsWith("Importante:"));

**O que devemos mockar?**

Geralmente optamos por mockar classes que são difíceis de serem testadas. Por exemplo, se não mockarmos um DAO ou uma classe que envia e-mail, dificilmente conseguiremos testar aquela classe.

Classes de domínio, como entidades e etc, geralmente não necessitam de mocks. Nesses casos, é até bom não mockarmos, pois se ela tiver algum bug, a chance de um teste pegar é maior.

**O que o ArgumentCaptor recebe?**

ArgumentCaptor<ABC> argumento = ArgumentCaptor.forClass(ABC.class);

Colocamos a classe que queremos capturar! Ou seja, se queremos capturar um Pagamento, colocamos ali Pagamento e Pagamento.class.

**Criando abstrações para testar**

Não é possível testar quando se tem métodos estáticos, neste caso, criando abstrações

o código fica um pouco mais complexo, mas conseguimos testar e garantir sua qualidade.

Uma ótima dica para se levar é: se está difícil testar, é porque nosso projeto de classes não está bom o suficiente.

Idealmente, deve ser fácil escrever um teste de unidade. Use seus conhecimentos de orientação a objetos, crie abstrações, escreva classes pequenas, diminua o acoplamento... Tudo isso facilitará o seu teste!