Mock Objects - parte 1

Site:	Moodle institucional da UTFPR	Impresso por:	MARLENE VASCONCELOS MORAES DE OLIVEIRA
Curso:	CETEJ158A - Teste de Software - J29 (2023_03) J30A (2024_01)	Data:	domingo, 25 ago. 2024, 09:12
Livro:	Mock Objects - parte 1		

Descrição

Até agora, escrever testes de unidade era fácil. E todos eles eram de certa forma parecidos. Um teste era basicamente um código que montava um cenário, instanciava um objeto, invocava um comportamento e verificava sua saída.

Mas será que é sempre fácil assim?

Índice

1. Introdução

Veja a classe EncerradorDeLeilao a seguir, responsável por encerrar leilões:

```
public class EncerradorDeLeilao {

    private int total = 0;

    public void encerra() {
        LeilaoDao dao = new LeilaoDao();
        List<Leilao> todosLeiloesCorrentes = dao.correntes();

        for (Leilao leilao : todosLeiloesCorrentes) {

            if (comecouSemanaPassada(leilao)) {

                leilao.encerra();
                total++;
                dao.atualiza(leilao);
            }
        }
    }

    private boolean comecouSemanaPassada(Leilao leilao) {
        return diasEntre(leilao.getData(), Calendar.getInstance()) >= 7;
    }

    private int diasEntre(Calendar inicio, Calendar fim) {
        Calendar data = (Calendar) inicio.clone();
        int diasNoIntervalo = 0;
        while (data.before(fim)) {
            data.add(Calendar.DAY_OF_MONTH, 1);
            diasNoIntervalo++;
        }
        return diasNoIntervalo;
    }

    public int getTotalEncerrados() {
        return total;
    }
}
```

Ela é razoavelmente simples de entender: esse código percorre toda a lista de leilões e, caso o leilão tenha sido iniciado semana passada, ele é encerrado e persistido no banco de dados através do DAO. Nosso DAO é convencional. Ele faz uso de JDBC e envia comandos SQL para o banco.

Pensar em cenários de teste para esse problema não é tão difícil:

- Uma lista com leilões a serem encerrados;
- Uma listas em nenhum leilão a ser encerrado;
- Uma lista com um leilão um dia antes de ser encerrado;
- Uma lista com um leilão no dia exato de ser encerrado.

Escrever esses testes não deve ser uma tarefa tão difícil. Vamos lá:

```
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();
// mas como passo os leilões criados para o EncerradorDeLeilao,
// já que ele os busca no DAO?
        EncerradorDeLeilao encerrador = new EncerradorDeLeilao();

        encerrador.encerra();

        assertTrue(leilao1.isEncerrado());
        assertTrue(leilao2.isEncerrado());
    }
}
```

O problema é: como passamos o cenário para a classe EncerradorDeLeilao, já que ela busca esses dados do banco de dados?

2. Simulando a infraestrutura

Uma solução seria adicionar todos esses leilões no banco de dados, um a um, para cada cenário de teste. Ou seja, faríamos diversos INSERTs no banco de dados e teríamos o cenário pronto lá. No nosso caso, vamos usar o próprio DAO para inserir os leilões. Além disso, já que o DAO criará novos objetos, precisamos buscar os leilões novamente no banco, para garantir que eles estão encerrados:

```
    @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();
        LeilaoDao dao = new LeilaoDao();
        dao.salva(leilao1);
        dao.salva(leilao2);
        EncerradorDeLeilao encerrador = new EncerradorDeLeilao(daoFalso);
        encerrador.encerra();
// busca no banco a lista de encerrados
        List<Leilao> encerrados = dao.encerrados();
// vamos conferir tambem o tamanho da lista!
        assertEquals(2, encerrados.size());
        assertTrue(encerrados.get(0).isEncerrado());
        assertTrue(encerrados.get(1).isEncerrado());
    }
```

Isso resolve o nosso problema, afinal agora o teste passa! Mas isso não é suficiente: se rodarmos o teste novamente, ele agora quebra!

Isso aconteceu porque, como persistimos o cenário no banco de dados, na segunda execução do teste já existiam leilões lá! Precisamos lembrar sempre de limpar o cenário antes do início de cada teste, dando um DELETE ou um TRUNCATE TABLE.

Veja quanto trabalho para testar uma simples regra de negócio! Isso sem contar que o teste agora leva muito mais tempo para ser executado, afinal conectamos em um banco de dados todas as vezes que rodamos o teste!

Precisamos conseguir testar a classe EncerradorDeLeilao sem depender do banco de dados. A grande pergunta é: como fazer isso?

Uma ideia seria simular o banco de dados. Veja que, para a classe EncerradorDeLeilao, não importa como o DAO faz o serviço dela. O encerrador está apenas interessado em alguém que saiba devolver uma lista de leilões e que saiba persistir um leilão. Como isso é feito, para o encerrador pouco importa.

Vamos criar uma classe que finge ser um DAO. Ela persistirá as informações em uma simples lista:

```
public class LeilaoDaoFalso {

    private static List<Leilao> leiloes = new ArrayList<Leilao>();

    public void salva(Leilao leilao) {
        leiloes.add(leilao);
    }

    public List<Leilao> encerrados() {

        List<Leilao> filtrados = new ArrayList<Leilao>();
        for (Leilao leilao : leiloes) {
            if (leilao.isEncerrado()) {
                filtrados.add(leilao);
            }
        }
        return filtrados;
    }

    public List<Leilao> correntes() {
        List<Leilao> filtrados = new ArrayList<Leilao>();
        for (Leilao leilao : leiloes) {
            if (!leilao.isEncerrado()) {
                filtrados.add(leilao);
            }
        }
        return filtrados;
    }

    public void atualiza(Leilao leilao) {
        /* faz nada! */ }
}
```

Agora vamos fazer com que o EncerradorDeLeilao e o EncerradorDeLeilaoTest usem o DAO falso. Além disso, vamos pegar o total de encerrados agora pelo próprio EncerradorDeLeilao, já que pegar pelo DAO não adianta mais (o DAO é falso!):

```
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();
// dao falso aqui!
        LeilaoDaoFalso daoFalso = new LeilaoDaoFalso();
        daoFalso.salva(leilao1);
        daoFalso.salva(leilao2);
        EncerradorDeLeilao encerrador = new EncerradorDeLeilao();
        encerrador.encerra();
        assertEquals(2, encerrador.getTotalEncerrados());
        assertTrue(leilao1.isEncerrado());
        assertTrue(leilao2.isEncerrado());
    }
}

public class EncerradorDeLeilao {

    public void encerra() {
// DAO falso aqui!
        LeilaoDaoFalso dao = new LeilaoDaoFalso();
        List<Leilao> todosLeiloesCorrentes = dao.correntes();
        for (Leilao leilao : todosLeiloesCorrentes) {
            if (comecouSemanaPassada(leilao)) {
                leilao.encerra();
                total++;
                dao.atualiza(leilao);
            }
        }
    }
// classe continua aqui...
}
```

Rodamos o teste novamente, e pronto! Veja que agora ele rodou rápido! Nosso teste está mais simples, mas ainda assim não é ideal. Sempre que criarmos um método novo no DAO, precisaremos escrevê-lo também no LeilaoDaoFalso. Se precisarmos fazer testes de casos excepcionais como, por exemplo, uma exceção lançada no método salva(), precisaríamos de vários DAOs falsos.

3. Mock Objects

A ideia de objetos falsos é boa; precisamos apenas encontrar uma maneira ideal de implementá-la. Objetos que simulam os comportamentos dos objetos reais são o que chamamos de mock objects. Mock objects ou, como foi traduzido para o português, objetos dublês, são objetos que fingem ser outros objetos. Eles são especialmente úteis em testes como esses, em que temos objetos que se integram com outros sistemas (como é o caso do nosso DAO, que fala com um banco de dados).

E o melhor: os frameworks de mock objects tornam esse trabalho extremamente simples! Neste curso, estudaremos o Mockito. Ele é um dos frameworks de mock mais populares do mercado.

A primeira coisa que devemos fazer é criar um mock do objeto que queremos simular. No 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. Olhe que simples. Agora, quando invocarmos daoFalso.correntes(), ele devolverá leiloesAntigos.

Vamos levar 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 leilões 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! Por quê? Porque a classe EncerradorDeLeilao não faz uso do mock que criamos! Veja que ela instancia o DAO falso ainda! 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);
            }
        }
    }
// código 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. Veja que, 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.

3.1. JMock ou Mockito?

Alguns desenvolvedores preferem o JMock. A API do JMock é bastante diferente da do Mockito. No fim, é questão do gosto. A ideia aqui é você aprender o que é e quando usar objetos dublês.

4. Mocks estritos e acoplamento

Algumas pessoas preferem fazer com que seus mocks tenham comporta- mento bem restrito. Ou seja, se o código de produção invocar um método que não foi previamente definido, o framework de mock deve fazer o teste falhar.

Prefira que seus testes deixem bem claro qual uso eles fazem do mock. Ou seja, se determinado método será invocado pelo código de produção, isso está explícito no código de teste. Se o método é void e é invocado pelo código de produção, então faça um verify ao final.

Perceba que, a partir do momento em que você usa mocks, você está deixando vazar detalhes da implementação da classe de produção. Afinal, o que antes estava encapsulado na classe de produção agora está aberto no código de testes: o seu teste sabe quais métodos serão invocados de cada dependência. Ou seja, se você resolver mudar a dependência de A para A2, vai provavelmente precisar alterar todos os seus testes. Dado que o trabalho acontecerá de qualquer maneira, prefira ter testes explícitos.

5. Fugindo de métodos estáticos

Conhecendo agora um pouco mais sobre mocks e frameworks de mocks, é fácil entender por que todos dizem para não fazer uso de métodos estáticos. Além de eles terem um cheiro de código procedural, os frameworks não conseguem mocká-los. Ou seja, se você tem um método que aparentemente parece ser um bom candidato a ser estático, pense em fazê-lo como método de instância e ter a possibilidade de simulá-lo no futuro.

É por isso também que desenvolvedores que conhecem bastante sobre testabilidade e orientação a objetos optam sempre por interfaces na hora de fazer uma dependência. É muito mais fácil mockar. 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.

6. Garantindo que métodos foram invocados

Agora que conseguimos simular nosso banco de dados, o teste ficou fácil de ser escrito. Mas ainda não testamos o método encerra() da classe EncerradorDeLeilao por completo. Veja o código a seguir:

```
    public void encerra() {
        List<Leilao> todosLeiloesCorrentes = dao.correntes();
        for (Leilao leilao : todosLeiloesCorrentes) {
            if (comecouSemanaPassada(leilao)) {
                leilao.encerra();
                total++;
                dao.atualiza(leilao);
            }
        }
    }
```

Testamos que leilões são ou não encerrados, mas ainda não temos garantia de que os leilões são atualizados pelo DAO após serem encerrados! Como garantir que o método atualiza() foi invocado?

Se não estivéssemos “mockando” o DAO, seria fácil: bastaria fazer um SELECT no banco de dados e verificar que a coluna foi alterada! Mas agora, com o mock, precisamos perguntar para ele se o método foi invocado!

Para isso, faremos uso do método verify do Mockito. Nele, indicaremos qual método queremos verificar se foi invocado! Veja o teste a seguir:

```
    @Test
    public void deveAtualizarLeiloesEncerrados() {
        Calendar antiga = Calendar.getInstance();
        antiga.set(1999, 1, 20);
        Leilao leilao1 = new CriadorDeLeilao().para("TV de plasma")
                .naData(antiga).constroi();
        RepositorioDeLeiloes daoFalso
                = mock(RepositorioDeLeiloes.class);
        when(daoFalso.correntes())
                .thenReturn(Arrays.asList(leilao1));
        EncerradorDeLeilao encerrador
                = new EncerradorDeLeilao(daoFalso);
        encerrador.encerra();
// verificando que o método atualiza foi realmente invocado!
        verify(daoFalso).atualiza(leilao1);
    }
```

Veja que passamos para o método atualiza() a variável leilao1. O Mockito é inteligente: ele verificará se o método atualiza() foi invocado com a variável leilao1 sendo passada. Caso passemos um outro leilão, por exemplo, o teste falhará.

Nesse momento, nosso teste passa! Por curiosidade, se comentarmos a linha dao.atualiza(leilao); na classe EncerradorDeLeilao, o teste falhará. Repare que a mensagem dirá que o método não foi invocado.

Pronto! Dessa forma conseguimos testar a invocação de métodos.

7. Contando o número de vezes que o método foi invocado

Podemos melhorar ainda mais essa verificação. Podemos dizer ao verify() que esse método deve ser executado uma única vez, e que, caso ele seja invocado mais de uma vez, o teste deve falhar:

```
    @Test
    public void deveAtualizarLeiloesEncerrados() {
        Calendar antiga = Calendar.getInstance();
        antiga.set(1999, 1, 20);
        Leilao leilao1 = new CriadorDeLeilao().para("TV de plasma")
                .naData(antiga).constroi();
        RepositorioDeLeiloes daoFalso
                = mock(RepositorioDeLeiloes.class);
        when(daoFalso.correntes())
                .thenReturn(Arrays.asList(leilao1));
        EncerradorDeLeilao encerrador
                = new EncerradorDeLeilao(daoFalso);
        encerrador.encerra();
        verify(daoFalso, times(1)).atualiza(leilao1);
    }
```

Veja o times(1). Ele também é um método da classe Mockito. Ali passamos a quantidade de vezes que o método deve ser invocado; poderia ser 2, 3, 4, ou qualquer outro número. Por curiosidade, se fizermos a classe EncerradorDeLeilao invocar duas vezes o DAO, nosso teste falhará. Ele nos avisará de que o método foi invocado 2 vezes, o que não era esperado.

Por meio do verify(), conseguimos testar quais métodos são invocados, garantindo o comportamento de uma classe por completo.

8. Outros métodos de verificação

O Mockito tem muitos outros métodos para ajudar a fazer a verificação. O 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 anterior, com a diferença de que passamos para ele o número mínimo de invocações que um método deve ter. Por fim, o atMost(numero) nos garante que um método foi executado até no máximo N vezes. Ou seja, se ele tiver mais invocações do que o que foi passado para o atMost, o teste falha.

Veja que existem diversas maneiras para garantir a quantidade de invocações de um método. Você pode escolher a melhor e mais elegante para seu teste. Consulte a documentação para entender melhor cada um deles.

9. Referência Bibliográfica

Extraído de:
Aniche, M. **Testes Automatizados de Software**. São Paulo: Casa do Código, 2015.