Mock Objects - parte 2

Descrição

Engraçado que quando desenvolvedores aprendem mocks, eles querem usá-los o tempo todo. Mas não é bem assim. 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. Já classes de domínio, como entidades 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.

Índice

1. Mocks que lançam exceções

Imagine agora a classe EncerradorDeLeilao, que precisa enviar um e-mail logo após encerrar o leilão. Para isso, receberemos o Carteiro, uma interface, no construtor e o invocaremos logo após persistir no DAO:

```
public interface Carteiro {

    void envia(Leilao leilao);
}

public class EncerradorDeLeilao {

    private int total = 0;
    private final RepositorioDeLeiloes dao;
    private final Carteiro carteiro;

    public EncerradorDeLeilao(
            RepositorioDeLeiloes dao,
            Carteiro carteiro) {
        this.dao = dao;
// guardamos o carteiro como atributo da classe
        this.carteiro = carteiro;
    }

    public void encerra() {
        List<Leilao> todosLeiloesCorrentes = dao.correntes();
        for (Leilao leilao : todosLeiloesCorrentes) {
            if (comecouSemanaPassada(leilao)) {
                leilao.encerra();
                total++;
                dao.atualiza(leilao);
// agora enviamos por email tambem!
                carteiro.envia(leilao);
            }
        }
    }
// ... código continua
}
```

Nesse momento, provavelmente todos os testes existentes até então devem quebrar. Isso acontecerá pois agora o construtor da classe EncerradorDeLeilao recebe um Carteiro. Resolver é fácil: basta passar um mock de Carteiro para todos eles. A mudança deve ser parecida com a seguinte:

```
Carteiro carteiroFalso = mock(Carteiro.class);
EncerradorDeLeilao encerrador = new EncerradorDeLeilao(daoFalso, carteiroFalso);
```

O método encerra() agora, além de encerrar um leilão, ainda persiste a informação na base de dados e notifica o sistema de envio de e-mails. Já sabemos que, sempre que lidamos com infraestrutura, precisamos nos pre- caver de possíveis problemas: o banco pode estar fora do ar, o SMTP pode recusar nosso envio de e-mail e assim por diante.

Nosso sistema, entretanto, deve saber se recuperar da falha e continuar para garantir que nosso EncerradorDeLeilao não pare por causa de um erro de infraestrutura. Para isso, vamos adicionar um try-catch dentro do loop e, caso o DAO lance uma exceção, o encerrador continuará a tratar os próximos leilões da lista. Vamos lá:

```
    public void encerra() {
        List<Leilao> todosLeiloesCorrentes = dao.correntes();
        for (Leilao leilao : todosLeiloesCorrentes) {
            try {
                if (comecouSemanaPassada(leilao)) {
                    leilao.encerra();
                    total++;
                    dao.atualiza(leilao);
                    carteiro.envia(leilao);
                }
            } catch (Exception e) {
                //salvo a exceção no sistema de logs
                // e o loop continua
            }
        }
    }
```

2. Simulando exceções

Como sempre, vamos garantir que esse comportamento realmente funciona. Sabemos que, se passarmos dois leilões e o DAO lançar uma exceção no primeiro, ainda sim o segundo deve ser processado. Para simular essa exceção, faremos uso do método doThrow() do Mockito. Esse método recebe um parâmetro: a exceção que deve ser lançada. Em seguida, passamos para ele a mesma instrução when() com que já estamos acostumados. Veja o exemplo:

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

Estamos dizendo ao mock que, quando o método atualiza(leilao1) for invocado no daoFalso, o Mockito deve então lançar a Exception que foi passada para o doThrow. Simples assim!

Vamos ao teste agora:

```
    @Test
    public void deveContinuarAExecucaoMesmoQuandoDaoFalha() {
        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();
        RepositorioDeLeiloes daoFalso
                = mock(RepositorioDeLeiloes.class);
        when(daoFalso.correntes())
                .thenReturn(Arrays.asList(leilao1, leilao2));
        doThrow(new RuntimeException()).when(daoFalso)
                .atualiza(leilao1);
        EnviadorDeEmail carteiroFalso
                = mock(EnviadorDeEmail.class);
        EncerradorDeLeilao encerrador
                = new EncerradorDeLeilao(daoFalso, carteiroFalso);
        encerrador.encerra();
        verify(daoFalso).atualiza(leilao2);
        verify(carteiroFalso).envia(leilao2);
    }
```

Veja que ensinamos nosso mock a lançar uma exceção quando o leilao1 for passado; mas nada deve acontecer com o leilao2. Ao final, verificamos que o DAO e o carteiro receberam leilao2 (afinal, a execução deve continuar!). Nosso teste passa!

Por curiosidade, comente o try-catch e rode o teste novamente. Ele falhará.

Pronto! Garantimos que nosso sistema continua funcionando mesmo se uma exceção ocorrer! É comum termos tratamentos diferentes dada uma exceção; o Mockito faz com que esses testes sejam fáceis de serem escritos! Sem ele, como faríamos esse teste? Desligaríamos o MySQL para simular banco de dados fora do ar? Mocks realmente são úteis.

2.1. Lembre-se das boas práticas

No começo, os testes quebraram todos por causa da mudança de construtor. É difícil evitar isso, mas você pode facilitar a manutenção. A linha do construtor do objeto poderia estar dentro do @Before, por exemplo. Dessa forma, a mudança ocorreria apenas em um ponto da bateria, diminuindo o retrabalho (chato, aliás, afinal acertar repetidas linhas de instanciação de objeto não é legal).

3. Capturando argumentos recebidos pelo mock

O sistema mudou mais uma vez (te lembra o mundo real?). Precisamos agora de um batch job que pegue leilões encerrados e gere um pagamento associado ao valor desse leilão. Para isso, vamos criar a classe Pagamento e a interface RepositorioDePagamentos, que representa o contrato de acesso aos pagamentos já armazenados:

```
public class Pagamento {

    private double valor;
    private Calendar data;

    public Pagamento(double valor, Calendar data) {
        this.valor = valor;
        this.data = data;
    }

    public double getValor() {
        return valor;
    }

    public Calendar getData() {
        return data;
    }
}

public interface RepositorioDePagamentos {

    void salva(Pagamento pagamento);
}
```

Agora vamos a classe que gera a lista de pagamentos de acordo com os leilões encerrados:

```
public class GeradorDePagamento {

    private final RepositorioDePagamentos pagamentos;
    private final RepositorioDeLeiloes leiloes;
    private final Avaliador avaliador;

    public GeradorDePagamento(RepositorioDeLeiloes leiloes,
            RepositorioDePagamentos pagamentos,
            Avaliador avaliador) {
        this.leiloes = leiloes;
        this.pagamentos = pagamentos;
        this.avaliador = avaliador;
    }

    public void gera() {
        List<Leilao> leiloesEncerrados = leiloes.encerrados();
        for (Leilao leilao : leiloesEncerrados) {
            avaliador.avalia(leilao);
            Pagamento novoPagamento
                    = new Pagamento(avaliador.getMaiorLance(),
                            Calendar.getInstance());
            pagamentos.salva(novoPagamento);
        }
    }
}
```

Veja que a regra de negócio é bem simples: ela pega todos os leilões encerrados, avalia o leilão para descobrir o maior lance, gera um novo pagamento com o valor e a data corrente e o salva no repositório.

Agora, precisamos testar essa nossa nova classe. Vamos lá:

```
public class GeradorDePagamentoTest {

    @Test
    public void deveGerarPagamentoParaUmLeilaoEncerrado() {
        RepositorioDeLeiloes leiloes
                = mock(RepositorioDeLeiloes.class);
        RepositorioDePagamentos pagamentos
                = mock(RepositorioDePagamentos.class);
        Avaliador avaliador
                = mock(Avaliador.class);
        Leilao leilao = new CriadorDeLeilao()
                .para("Playstation")
                .lance(new Usuario("José da Silva"), 2000.0)
                .lance(new Usuario("Maria Pereira"), 2500.0)
                .constroi();
        when(leiloes.encerrados())
                .thenReturn(Arrays.asList(leilao));
        when(avaliador.getMaiorLance())
                .thenReturn(2500.0);
        GeradorDePagamento gerador
                = new GeradorDePagamento(leiloes, pagamentos, avaliador);
        gerador.gera();
// como fazer assert no Pagamento gerado?
    }
}
```

O problema é como fazer o assert no Pagamento gerado pela classe GeradorDePagamento. Afinal, ele é instanciado internamente e não temos como recuperá-lo no nosso método de teste.

Mas repare que a instância é passada para o RepositorioDePagamentos, que é um mock! Podemos pedir ao Mock para guardar esse objeto para que possamos recuperá-lo a fim de realizar as asserções! A classe do Mockito que faz isso é chamada de ArgumentCaptor, ou seja, capturador de argumentos.

Para a utilizarmos, precisamos instanciá-la, passando qual a classe que será recuperada. No nosso caso, é a Pagamento. Em seguida, fazemos uso do verify() e checamos a execução do método que recebe o atributo. Como parâmetro, passamos o método capture() do ArgumentCaptor:

```
// criamos o ArgumentCaptor que sabe capturar um Pagamento
ArgumentCaptor<Pagamento> argumento = ArgumentCaptor.forClass(Pagamento.class);
// capturamos o Pagamento que foi passado para o método salvar
verify(pagamentos).salvar(argumento.capture());
```

Simples assim! Agora o argumento contém a instância de Pagamento criada! Basta pegarmos a instância do Pagamento através do método argumento.getValue():

```
Pagamento pagamentoGerado = argumento.getValue();
```

Pronto! Agora temos tudo para escrever o teste:

```
    @Test
    public void deveGerarPagamentoParaUmLeilaoEncerrado() {
        RepositorioDeLeiloes leiloes
                = mock(RepositorioDeLeiloes.class);
        RepositorioDePagamentos pagamentos
                = mock(RepositorioDePagamentos.class);
        Avaliador avaliador
                = mock(Avaliador.class);
        Leilao leilao = new CriadorDeLeilao()
                .para("Playstation")
                .lance(new Usuario("José da Silva"), 2000.0)
                .lance(new Usuario("Maria Pereira"), 2500.0)
                .constroi();
        when(leiloes.encerrados())
                .thenReturn(Arrays.asList(leilao));
        when(avaliador.getMaiorLance())
                .thenReturn(2500.0);
        GeradorDePagamento gerador
                = new GeradorDePagamento(leiloes, pagamentos, avaliador);
        gerador.gera();
        ArgumentCaptor<Pagamento> argumento
                = ArgumentCaptor.forClass(Pagamento.class);
        verify(pagamentos).salvar(argumento.capture());
        Pagamento pagamentoGerado = argumento.getValue();
        assertEquals(2500.0, pagamentoGerado.getValor(), 0.00001);
    }
```

Veja que o ArgumentCaptor possibilita capturar a instância que foi passada para o Mock. Isso é especialmente útil em situações como essas: nossa classe de produção instancia um novo objeto, que é passado para uma das dependências. Assim, conseguimos garantir que o objeto criado está correto.

4. Isolando para testar

Imagine mais uma regra de negócio: a data do pagamento nunca pode ser de fim de semana; se “hoje” for sábado ou domingo, devemos empurrar a data para o primeiro dia útil. A implementação não é difícil: basta verificarmos o dia da semana e empurrar 1 dia se for domingo, 2 dias se for sábado:

```
public class GeradorDePagamento {

    private final RepositorioDePagamentos pagamentos;
    private final RepositorioDeLeiloes leiloes;
    private final Avaliador avaliador;

    public GeradorDePagamento(RepositorioDeLeiloes leiloes,
            RepositorioDePagamentos pagamentos,
            Avaliador avaliador) {
        this.leiloes = leiloes;
        this.pagamentos = pagamentos;
        this.avaliador = avaliador;
    }

    public void gera() {
        List<Leilao> leiloesEncerrados = leiloes.encerrados();
        for (Leilao leilao : leiloesEncerrados) {
            avaliador.avalia(leilao);
// agora empurramos para o próximo dia útil
            Pagamento novoPagamento
                    = new Pagamento(avaliador.getMaiorLance(),
                            primeiroDiaUtil());
            pagamentos.salva(novoPagamento);
        }
    }

    private Calendar primeiroDiaUtil() {
        Calendar data = Calendar.getInstance();
        int diaDaSemana = data.get(Calendar.DAY_OF_WEEK);
        if (diaDaSemana == Calendar.SATURDAY) {
            data.add(Calendar.DAY_OF_MONTH, 2);
        } else if (diaDaSemana == Calendar.SUNDAY) {
            data.add(Calendar.DAY_OF_MONTH, 1);
        }
        return data;
    }
}
```

Agora vamos testar. O que faremos é verificar que a data gerada para o pagamento é uma segunda-feira. Vamos usar o ArgumentCaptor, que estudamos no capítulo passado:

```
@Test
    public void deveEmpurrarParaOProximoDiaUtil() {
        RepositorioDeLeiloes leiloes
                = mock(RepositorioDeLeiloes.class);
        RepositorioDePagamentos pagamentos
                = mock(RepositorioDePagamentos.class);
        Leilao leilao = new CriadorDeLeilao()
                .para("Playstation")
                .lance(new Usuario("José da Silva"), 2000.0)
                .lance(new Usuario("Maria Pereira"), 2500.0)
                .constroi();
        when(leiloes.encerrados())
                .thenReturn(Arrays.asList(leilao));
        GeradorDePagamento gerador
                = new GeradorDePagamento(leiloes, pagamentos, new Avaliador());
        gerador.gera();
        ArgumentCaptor<Pagamento> argumento
                = ArgumentCaptor.forClass(Pagamento.class);
        verify(pagamentos).salva(argumento.capture());
        Pagamento pagamentoGerado = argumento.getValue();
        assertEquals(Calendar.MONDAY,
                pagamentoGerado.getData().get(Calendar.DAY_OF_WEEK));
    }
```

Nosso teste falha! Veja a implementação do método primeiroDiaUtil. Ele verifica se o dia de hoje é sábado ou domingo. Ou seja, esse teste só passará se você estiver fazendo esse curso no fim de semana!

5. Criando abstrações para facilitar o teste

A pergunta é: como mudar a data atual? Precisamos fazer com que ela seja sábado para esse teste! Poderíamos mudar a data da nossa máquina, mas essa não é uma solução elegante. Sabemos também que é im- possível mockar um método estático, ou seja, não conseguimos mockar Calendar.getInstance().

Uma possível solução para o problema é criar uma abstração de um relógio; uma interface que teria um método hoje(), por exemplo, responsável por devolver a data atual. Teríamos também uma implementação concreta, que simplesmente faria Calendar.getInstance(). Essa abstração é facilmente mockável e, se nosso GeradorDePagamento fizer uso da abstração em vez de usar o Calendar diretamente, conseguiríamos testá-la.

Vamos criar a interface Relogio, passá-la como dependência para GeradorDePagamento e pedir a hora atual para ela. Para manter a compatibilidade, manteremos o construtor antigo também:

```
public interface Relogio {

    Calendar hoje();
}

public class RelogioDoSistema implements Relogio {

    public Calendar hoje() {
        return Calendar.getInstance();
    }
}

public class GeradorDePagamento {

    private final RepositorioDePagamentos pagamentos;
    private final RepositorioDeLeiloes leiloes;
    private final Avaliador avaliador;
    private final Relogio relogio;

    public GeradorDePagamento(RepositorioDeLeiloes leiloes,
            RepositorioDePagamentos pagamentos,
            Avaliador avaliador,
            Relogio relogio) {
        this.leiloes = leiloes;
        this.pagamentos = pagamentos;
        this.avaliador = avaliador;
        this.relogio = relogio;
    }

    public GeradorDePagamento(RepositorioDeLeiloes leiloes,
            RepositorioDePagamentos pagamentos,
            Avaliador avaliador) {
        this(
                leiloes,
                pagamentos,
                avaliador,
                new RelogioDoSistema()
        );
    }
// ...

    private Calendar primeiroDiaUtil() {
        Calendar data = relogio.hoje();
        int diaDaSemana = data.get(Calendar.DAY_OF_WEEK);
        if (diaDaSemana == Calendar.SATURDAY) {
            data.add(Calendar.DAY_OF_MONTH, 2);
        } else if (diaDaSemana == Calendar.SUNDAY) {
            data.add(Calendar.DAY_OF_MONTH, 1);
        }

        return data;
    }
}
```

Agora, tendo o Relogio como dependência, conseguimos facilmente criar um mock para ele e fazer com que o método hoje() devolva a data de sábado.

Vamos lá:

```
    @Test
    public void deveEmpurrarParaOProximoDiaUtil() {
        RepositorioDeLeiloes leiloes
                = mock(RepositorioDeLeiloes.class);
        RepositorioDePagamentos pagamentos
                = mock(RepositorioDePagamentos.class);
        Relogio relogio = mock(Relogio.class);
// dia 7/abril/2012 é um sabado
        Calendar sabado = Calendar.getInstance();
        sabado.set(2012, Calendar.APRIL, 7);
// ensinamos o mock a dizer que "hoje" é sabado!
        when(relogio.hoje()).thenReturn(sabado);
        Leilao leilao = new CriadorDeLeilao()
                .para("Playstation")
                .lance(new Usuario("José da Silva"), 2000.0)
                .lance(new Usuario("Maria Pereira"), 2500.0)
                .constroi();
        when(leiloes.encerrados())
                .thenReturn(Arrays.asList(leilao));
        GeradorDePagamento gerador
                = new GeradorDePagamento(leiloes, pagamentos, new Avaliador(), relogio.hoje());
        gerador.gera();
        ArgumentCaptor<Pagamento> argumento
                = ArgumentCaptor.forClass(Pagamento.class);
        verify(pagamentos).salva(argumento.capture());
        Pagamento pagamentoGerado = argumento.getValue();
        assertEquals(Calendar.MONDAY,
                pagamentoGerado.getData().get(Calendar.DAY_OF_WEEK));
        assertEquals(9,
                pagamentoGerado.getData().get(Calendar.DAY_OF_MONTH));
    }
```

Veja que criamos um Calendar para o dia 07/04/2012, que é um sábado, e ensinamos o mock de Relogio a responder ao método hoje() com essa data. Agora conseguimos simular o “dia de hoje”!

Nosso teste agora passa!

Sempre que tivermos dificuldade de testar algum trecho de código (geralmente os que fazem uso de métodos estáticos), é comum criarmos abstrações para facilitar o teste. A abstração de relógio é muito comum em sistemas bem testados.

5.1. Criar abstrações sempre?

É uma pergunta interessante. Devemos criar abstrações o tempo todo? Geralmente, é uma troca bastante justa. 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!

6. Referência Bibliográfica

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