


# Testes Automatizados de Software

Um guia prático



Casa do  
Código

—  —  
SÉRIE CAELUM

MAURÍCIO ANICHE

© Casa do Código

Todos os direitos reservados e protegidos pela Lei nº9.610, de 10/02/1998.

Nenhuma parte deste livro poderá ser reproduzida, nem transmitida, sem autorização prévia por escrito da editora, sejam quais forem os meios: fotográficos, eletrônicos, mecânicos, gravação ou quaisquer outros.

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

# Quem sou eu?

Meu nome é Mauricio Aniche, e trabalho com desenvolvimento de software há por volta de 10 anos. Em boa parte desse tempo, atuei como consultor para diferentes empresas do mercado brasileiro e internacional. Com certeza, as linguagens mais utilizadas por mim ao longo da minha carreira foram Java, C# e C.

Como sempre pulei de projeto em projeto (e, por consequência, de tecnologia em tecnologia), nunca fui a fundo em nenhuma delas. Pelo contrário, sempre foquei em entender princípios que pudessem ser levados de uma para outra, para que no fim, o código saísse com qualidade, independente da tecnologia.

Em meu último ano da graduação, 2007, comecei a ler mais sobre a ideia de testes automatizados e TDD. Achei muito interessante e útil a ideia de se escrever um programa para testar seu programa, e decidi praticar tudo isso, por conta própria, para entender melhor como funcionava.

Gostei muito do que vi. De 2007 em diante, resolvi praticar, pesquisar e divulgar melhor minhas ideias sobre o assunto. Comecei devagar, apenas blogando o que estava na minha cabeça e sobre o que gostaria de *feedback* de outros desenvolvedores. Mas para fazer isso de maneira mais decente, resolvi ingressar no programa de Mestrado da Universidade de São Paulo. Lá, pesquisei sobre os efeitos da prática de TDD no design de classes.

Ao longo desse tempo participei da grande maioria dos eventos relacionados ao assunto. Palestrei nos principais eventos de métodos ágeis do país (como Agile Brazil, Encontro Ágil), de desenvolvimento de software (QCON SP e DNAD), entre outros menores. Cheguei a participar de eventos internacionais também; fui o único palestrante brasileiro no Primeiro Workshop

Internacional sobre TDD, em 2010, na cidade de Paris. Isso mostra também que tenho participado dos eventos acadêmicos. Em 2011, apresentei um estudo sobre TDD no WBMA (Workshop Brasileiro de Métodos Ágeis), e em 2012, no maior simpósio brasileiro sobre engenharia de software, o SBES.

Atualmente trabalho pela Caelum, como consultor e instrutor. Também sou aluno de doutorado pela Universidade de São Paulo, onde continuo a pesquisar sobre a relação dos testes de unidade e qualidade do código.

Portanto, esse é meu relacionamento com TDD e testes. Nos últimos anos tenho olhado-o de todos os pontos de vista possíveis: de praticante, de acadêmico, de pesquisador, de apaixonado, de frio. Esse livro é o relato de tudo que aprendi nesses últimos anos.

# Sumário

<b>1</b>	<b>Testes de Unidade</b>	<b>1</b>
1.1	Um código qualquer . . . . .	1
1.2	Implementando uma nova funcionalidade . . . . .	3
1.3	O que aconteceu? . . . . .	5
1.4	Olá mundo, JUnit! . . . . .	8
1.5	Convenções na escrita de testes . . . . .	14
1.6	Mas será que sou produtivo assim? . . . . .	15
1.7	Testando o que realmente é necessário . . . . .	16
1.8	Classes de equivalência . . . . .	17
1.9	Import estático do Assert . . . . .	19
1.10	A próxima funcionalidade: os 3 maiores lances . . . . .	20
1.11	Testando casos especiais . . . . .	24
1.12	A bateria de testes nos salvou mais uma vez! . . . . .	25
1.13	Quais são as dificuldades? . . . . .	25
1.14	Cuidando dos seus testes . . . . .	26
1.15	Test Data Builders . . . . .	29
1.16	@After, @BeforeClass e @AfterClass . . . . .	31
1.17	Acoplamento entre testes e produção . . . . .	31
1.18	Cuide bem dos seus testes . . . . .	32
1.19	Testando exceções . . . . .	32
1.20	Melhorando a legibilidade dos testes . . . . .	34
1.21	100% de cobertura de testes? . . . . .	37
		iii

<b>2</b>	<b>Praticando Test-Driven Development (TDD)</b>	<b>39</b>
2.1	Testes, do jeito que você já sabe . . . . .	40
2.2	Mudando a maneira de desenvolver . . . . .	41
2.3	Test-Driven Development . . . . .	45
2.4	Efeitos no design de classes . . . . .	46
2.5	Baby steps . . . . .	47
2.6	Devo ver o teste falhar? . . . . .	48
2.7	TDD 100% do tempo? . . . . .	49
2.8	Onde posso ler mais sobre isso? . . . . .	49
<b>3</b>	<b>Mock Objects</b>	<b>51</b>
3.1	Simulando a infraestrutura . . . . .	53
3.2	Mock Objects . . . . .	58
3.3	Mocks estritos e acoplamento . . . . .	62
3.4	Fugindo de métodos estáticos . . . . .	62
3.5	Garantindo que métodos foram invocados . . . . .	63
3.6	Contando o número de vezes que o método foi invocado . . . . .	65
3.7	Outros métodos de verificação . . . . .	67
3.8	Mocks que lançam exceções . . . . .	68
3.9	Simulando exceções . . . . .	70
3.10	Capturando argumentos recebidos pelo mock . . . . .	73
3.11	Isolando para testar . . . . .	77
3.12	Criando abstrações para facilitar o teste . . . . .	79
3.13	O que mockar e o que não mockar? . . . . .	83
<b>4</b>	<b>Testes de Integração</b>	<b>85</b>
4.1	Devemos mockar um DAO? . . . . .	86
4.2	Testando DAOs . . . . .	89
4.3	Testando cenários mais complexos . . . . .	94
4.4	Praticando com consultas mais complicadas . . . . .	98
4.5	Testando alteração e deleção . . . . .	102
4.6	Organizando testes de integração . . . . .	104

<b>5</b>	<b>Testes de Sistema</b>	<b>107</b>
5.1	Automatizando o primeiro teste de sistema . . . . .	111
5.2	Novamente, as vantagens do teste automatizado . . . . .	117
5.3	Boas práticas: Page Objects . . . . .	118
5.4	Testando formulários complexos . . . . .	124
5.5	Classes de teste pai . . . . .	129
5.6	Como limpar o banco em testes de sistema? . . . . .	130
5.7	Requisições Ajax . . . . .	130
5.8	Builders em testes de sistema . . . . .	135
5.9	API para criação de cenários . . . . .	137
<b>6</b>	<b>Testes de serviços web</b>	<b>139</b>
6.1	Usando o Rest-Assured . . . . .	140
6.2	Testando JSONs . . . . .	143
6.3	Enviando dados para o WebService . . . . .	145
6.4	Outros recursos do Rest-Assured . . . . .	148
<b>7</b>	<b>E agora?</b>	<b>151</b>
7.1	Não há mais nenhum tipo de teste? . . . . .	151
7.2	Não preciso nunca de testes manuais? . . . . .	152
7.3	Testes de sistema o tempo todo? . . . . .	152
7.4	Onde posso falar mais sobre o assunto? . . . . .	153
7.5	Obrigado! . . . . .	154





## CAPÍTULO 1

# Testes de Unidade

Com certeza, todo desenvolvedor de software já escreveu um trecho de código que não funcionava. E pior, muitas vezes só descobrimos que o código não funciona quando nosso cliente nos reporta o bug. Nesse momento, perdemos a confiança no nosso código (já que o número de bugs é alto) e o cliente perde a confiança na equipe de desenvolvimento (já que ela não entrega código de qualidade). Mas será que isso é difícil de acontecer?

### 1.1 UM CÓDIGO QUALQUER

Para exemplificar isso, imagine que hoje trabalhamos em um sistema de leilão. Nesse sistema, um determinado trecho de código é responsável por devolver o maior lance de um leilão. Veja a implementação deste código:

```
class Avaliador {
```

```
private double maiorDeTodos = Double.NEGATIVE_INFINITY;

public void avalia(Leilao leilao) {

    for(Lance lance : leilao.getLances()) {
        if(lance.getValor() > maiorDeTodos) {
            maiorDeTodos = lance.getValor();
        }
    }
}

public double getMaiorLance() {
    return maiorDeTodos;
}

}

class TesteDoAvaliador {

    public static void main(String[] args) {
        Usuario joao = new Usuario("Joao");
        Usuario jose = new Usuario("José");
        Usuario maria = new Usuario("Maria");

        Leilao leilao = new Leilao("Playstation 3 Novo");

        leilao.propoe(new Lance(joao,300.0));
        leilao.propoe(new Lance(jose,400.0));
        leilao.propoe(new Lance(maria,250.0));

        Avaliador leiloeiro = new Avaliador();
        leiloeiro.avalia(leilao);

        // imprime 400.0
        System.out.println(leiloeiro.getMaiorLance());
    }
}
```

Esse código funciona. Ao receber um leilão, ele varre a lista buscando o maior valor. Veja que a variável `maiorDeTodos` é inicializada com o menor

valor que cabe em um `double`. Isso faz sentido, já que queremos que, na primeira vez que o `for` seja executado, ele caia no `if` e substitua o menor valor do `double` pelo primeiro valor da lista de lances.

## 1.2 IMPLEMENTANDO UMA NOVA FUNCIONALIDADE

A próxima funcionalidade a ser implementada é a busca pelo menor lance de todos. Essa regra de negócio faz sentido estar também na classe `Avaliador`. Basta acrescentarmos mais uma condição, desta vez para calcular o menor valor:

```
class Avaliador {

    private double maiorDeTodos = Double.NEGATIVE_INFINITY;
    private double menorDeTodos = Double.POSITIVE_INFINITY;

    public void avalia(Leilao leilao) {

        for(Lance lance : leilao.getLances()) {
            if(lance.getValor() > maiorDeTodos) {
                maiorDeTodos = lance.getValor();
            }
            else if(lance.getValor() < menorDeTodos) {
                menorDeTodos = lance.getValor();
            }
        }
    }

    public double getMaiorLance() { return maiorDeTodos; }
    public double getMenorLance() { return menorDeTodos; }
}

class TesteDoAvaliador {

    public static void main(String[] args) {
        Usuario joao = new Usuario("Joao");
        Usuario jose = new Usuario("José");
        Usuario maria = new Usuario("Maria");
```

```
Leilao leilao = new Leilao("Playstation 3 Novo");

leilao.propoe(new Lance(joao,300.0));
leilao.propoe(new Lance(jose,400.0));
leilao.propoe(new Lance(maria,250.0));

Avaliador leiloeiro = new Avaliador();
leiloeiro.avalial(leilao);

// imprime 400.0
System.out.println(leiloeiro.getMaiorLance());
// imprime 250.0
System.out.println(leiloeiro.getMenorLance());
}
}
```

Tudo parece estar funcionando. Apareceram na tela o menor e maior valor corretos. Vamos colocar o sistema em produção, afinal está testado!

Mas será que o código está realmente correto? Veja agora um outro teste, muito parecido com o anterior:

```
class TesteDoAvaliador {

    public static void main(String[] args) {
        Usuario joao = new Usuario("Joao");
        Usuario jose = new Usuario("José");
        Usuario maria = new Usuario("Maria");

        Leilao leilao = new Leilao("Playstation 3 Novo");

        leilao.propoe(new Lance(maria,250.0));
        leilao.propoe(new Lance(joao,300.0));
        leilao.propoe(new Lance(jose,400.0));

        Avaliador leiloeiro = new Avaliador();
        leiloeiro.avalial(leilao);

        // imprime 400.0
```

```
        System.out.println(leiloeiro.getMaiorLance());  
        // INFINITY  
        System.out.println(leiloeiro.getMenorLance());  
    }  
}
```

Veja que, para um cenário um pouco diferente, nosso código não funciona! A grande pergunta é: no mundo real, será que teríamos descoberto esse bug facilmente, ou esperaríamos nosso cliente nos ligar bravo porque a funcionalidade não funciona? Infelizmente, bugs em software são uma coisa mais comum do que deveriam ser. Bugs nos fazem perder a confiança do cliente, e nos custam muito dinheiro. Afinal, precisamos corrigir o bug e recuperar o tempo perdido do cliente, que ficou parado, enquanto o sistema não funcionava.

### 1.3 O QUE ACONTECEU?

Por que nossos sistemas apresentam tantos bugs assim? Um dos motivos para isso é a falta de testes, ou seja, testamos muito pouco! Equipes de software geralmente não gostam de fazer (ou não fazem) os devidos testes. As razões para isso são geralmente a demora e o alto custo para testar o software como um todo. E faz todo o sentido: pedir para um ser humano testar todo o sistema é impossível: ele vai levar muito tempo para isso!

Como resolver esse problema? Fazendo a máquina testar! Escrevendo um programa que teste nosso programa de forma automática! Uma máquina, com certeza, executaria o teste muito mais rápido do que uma pessoa! Ela também não ficaria cansada ou cometeria erros!

Um teste automatizado é muito parecido com um teste manual. Imagine que você está testando manualmente uma funcionalidade de cadastro de produtos em uma aplicação web. Você, com certeza, executará três passos diferentes. Em primeiro lugar, você pensaria em um cenário para testar. Por exemplo, “vamos ver o que acontece com o cadastro de funcionários quando eu não preencho o campo de CPF”. Após montar o cenário, você executará a ação que quer testar. Por exemplo, clicar no botão “Cadastrar”. Por fim, você olharia para a tela e verificaria se o sistema se comportou da maneira que

you esperava. Nesse nosso caso, por exemplo, esperaríamos uma mensagem de erro como “CPF inválido”.

Um teste automatizado é muito parecido. Você sempre executa estes três passos: **monta o cenário, executa a ação e valida a saída**. Mas, acredite ou não, já escrevemos algo muito parecido com um teste automatizado neste capítulo. Lembra da nossa classe `TesteDoAvaliador`? Perceba que ela se parece com um teste, afinal ela monta um cenário e executa uma ação. Veja o código:

```
class Teste {  
  
    public static void main(String[] args) {  
        // cenário: 3 lances em ordem crescente  
        Usuario joao = new Usuario("Joao");  
        Usuario jose = new Usuario("José");  
        Usuario maria = new Usuario("Maria");  
  
        Leilao leilao = new Leilao("Playstation 3 Novo");  
  
        leilao.propoe(new Lance(maria,250.0));  
        leilao.propoe(new Lance(joao,300.0));  
        leilao.propoe(new Lance(jose,400.0));  
  
        // executando a ação  
        Avaliador leiloeiro = new Avaliador();  
        leiloeiro.avalial(leilao);  
  
        // exibindo a saída  
        System.out.println(leiloeiro.getMaiorLance());  
        System.out.println(leiloeiro.getMenorLance());  
    }  
}
```

E veja que ele é automatizado! Afinal, é a máquina que monta o cenário e executa a ação (nós escrevemos o código, sim, mas na hora de executar, é a máquina que executa!). Não gastamos tempo nenhum para executar esse teste. O problema é que ele ainda não é inteiro automatizado. A parte final (a validação) ainda é manual: o programador precisa ver o que o programa

imprimiu na tela e checar se o resultado bate com o esperado.

Para melhorar isso, precisamos fazer a própria máquina verificar o resultado. Para isso, ela precisa saber qual a saída esperada. Ou seja, a máquina deve saber que o maior esperado, para esse caso, é 400, e que o menor esperado é 250. Vamos colocá-la em uma variável e então pedir pro programa verificar se a saída é correta:

```
class TesteDoAvaliador {  
  
    public static void main(String[] args) {  
        // cenário: 3 lances em ordem crescente  
        Usuario joao = new Usuario("Joao");  
        Usuario jose = new Usuario("José");  
        Usuario maria = new Usuario("Maria");  
  
        Leilao leilao = new Leilao("Playstation 3 Novo");  
  
        leilao.propoe(new Lance(maria,250.0));  
        leilao.propoe(new Lance(joao,300.0));  
        leilao.propoe(new Lance(jose,400.0));  
  
        // executando a ação  
        Avaliador leiloeiro = new Avaliador();  
        leiloeiro.avalua(leilao);  
  
        // comparando a saída com o esperado  
        double maiorEsperado = 400;  
        double menorEsperado = 250;  
  
        System.out.println(maiorEsperado ==  
            leiloeiro.getMaiorLance());  
        System.out.println(menorEsperado ==  
            leiloeiro.getMenorLance());  
    }  
}
```

Ao rodar esse programa, a saída já é um pouco melhor:

```
true  
false
```

Veja que agora ela sabe o resultado esperado e verifica se a saída bate com ele, imprimindo `true` se o resultado bateu e `false` caso contrário. Estamos chegando perto. O desenvolvedor ainda precisa olhar todos os trues e falses e ver se algum deu errado. Imagina quando tivermos 1000 testes iguais a esses? Será bem complicado!

Precisávamos de uma maneira mais simples e elegante de verificar o resultado de nosso teste. Melhor ainda se soubéssemos exatamente quais testes falharam e por quê. Para resolver esse problema, utilizaremos o **JUnit**, o framework de testes de unidade mais popular do mundo Java. O JUnit é uma ferramenta bem simples. Tudo que ele faz é ajudar na automatização da última parte de um teste, a validação, e a exibir os resultados de uma maneira bem formatada e simples de ser lida.

## 1.4 OLÁ MUNDO, JUNIT!

Para facilitar a interpretação do resultado dos testes, o JUnit pinta uma barra de verde, quando tudo deu certo, ou de vermelho, quando algum teste falhou. Além disso, ele nos mostra exatamente quais testes falharam e qual foi a saída incorreta produzida pelo método. O JUnit é tão popular que já vem com o Eclipse.

Nosso código anterior está muito perto de ser entendido pelo JUnit. Precisamos fazer apenas algumas mudanças: 1) um método de teste deve sempre ser público, de instância (isto é, não pode ser `static`) e não receber nenhum parâmetro; 2) deve ser anotado com `@Test`. Vamos fazer estas alterações:

```
class AvaliadorTest {

    @Test
    public void main() {
        // cenário: 3 lances em ordem crescente
        Usuario joao = new Usuario("Joao");
        Usuario jose = new Usuario("José");
        Usuario maria = new Usuario("Maria");

        Leilao leilao = new Leilao("Playstation 3 Novo");
```



```
leilao.propoe(new Lance(maria,250.0));
leilao.propoe(new Lance(joao,300.0));
leilao.propoe(new Lance(jose,400.0));

// executando a ação
Avaliador leiloeiro = new Avaliador();
leiloeiro.avalial(leilao);

// comparando a saída com o esperado
double maiorEsperado = 400;
double menorEsperado = 250;

System.out.println(maiorEsperado ==
    leiloeiro.getMaiorLance());
System.out.println(menorEsperado ==
    leiloeiro.getMenorLance());
}
}
```

Repare algumas coisas nesse código. Veja que mudamos o nome da classe: agora ela se chama `AvaliadorTest`. É convenção que o nome da classe geralmente seja **NomeDaClasseSobTesteTest**, ou seja, o nome da classe que estamos testando (no caso, `Avaliador`) mais o sufixo `Test`.

Seguindo a ideia da convenção do nome da classe, vamos querer colocar mais métodos para testar outros cenários envolvendo essa classe. Mas veja o nome do método do teste: continua `main`. Será que esse é um bom nome? Que nome daremos para os outros testes que virão?

Quando rodamos os testes pelo JUnit, ele nos mostra o nome do método que ele executou e um ícone indicando se aquele teste passou ou não.



Olhando para essa figura, dá para saber o que estamos testando? Ou então que parte do sistema está quebrada? Os nomes dos testes não nos dão nenhuma dica sobre o que está sendo testado. Precisamos ler o código de cada teste para descobrir. O ideal seria que os nomes dos testes já nos dessem uma boa noção do que estamos testando em cada método. Assim, conseguimos descobrir mais facilmente o que está quebrado no nosso sistema. Vamos renomear o método:

```
class AvaliadorTest {  
  
    @Test  
    public void deveEntenderLancesEmOrdemCrescente() {  
        // ...  
    }  
}
```

A última coisa que precisamos mudar no código para que ele seja entendido pelo JUnit são os `System.out.println()`. Não podemos imprimir na tela, pois assim nós é que seríamos responsáveis por validar a saída. Quem deve fazer isso agora é o JUnit! Para isso, utilizaremos a instrução `Assert.assertEquals()`. Esse é o método utilizado quando queremos que o resultado gerado seja igual à saída esperada. Em código:

```
import org.junit.Assert;
```

```
class AvaliadorTest {

    @Test
    public void deveEntenderLancesEmOrdemCrescente() {
        // cenário: 3 lances em ordem crescente
        Usuario joao = new Usuario("Joao");
        Usuario jose = new Usuario("José");
        Usuario maria = new Usuario("Maria");

        Leilao leilao = new Leilao("Playstation 3 Novo");

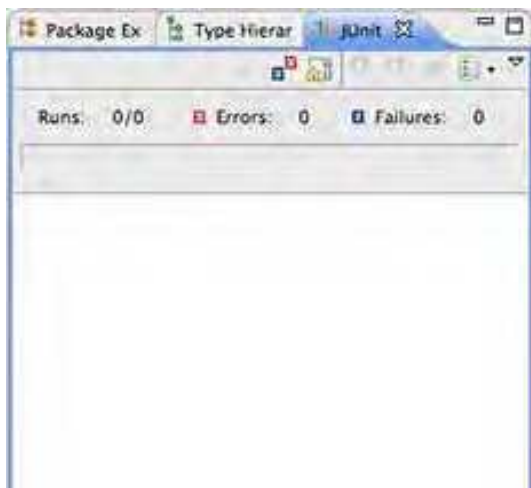
        leilao.propoe(new Lance(maria, 250.0));
        leilao.propoe(new Lance(joao, 300.0));
        leilao.propoe(new Lance(jose, 400.0));

        // executando a ação
        Avaliador leiloeiro = new Avaliador();
        leiloeiro.avalial(leilao);

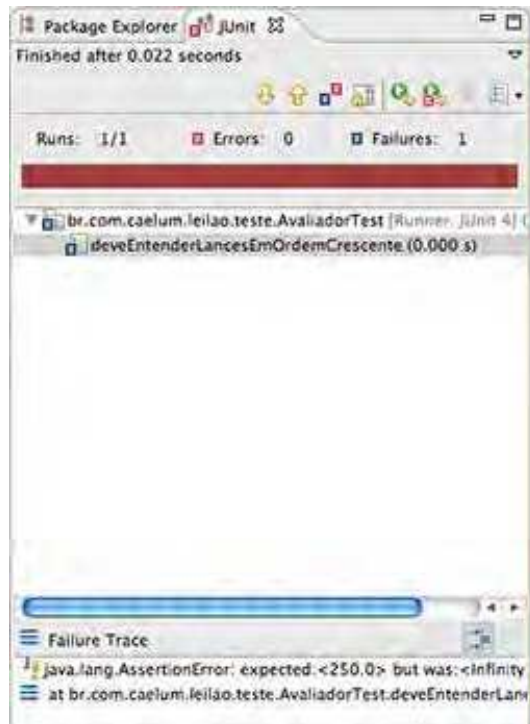
        // comparando a saída com o esperado
        double maiorEsperado = 400;
        double menorEsperado = 250;

        Assert.assertEquals(maiorEsperado,
            leiloeiro.getMaiorLance(), 0.0001);
        Assert.assertEquals(menorEsperado,
            leiloeiro.getMenorLance(), 0.0001);
    }
}
```

Repare que importamos a classe `Assert` e utilizamos o método duas vezes: para o maior e para o menor lance. Vamos agora executar o teste! Para isso, basta clicar com o botão direito no código da classe de teste e selecionar `Run as -> JUnit Test`. Veja que a *view* do JUnit se abrirá no Eclipse com o resultado do teste.



Nosso teste não passa. Veja a mensagem de erro dada pelo JUnit:

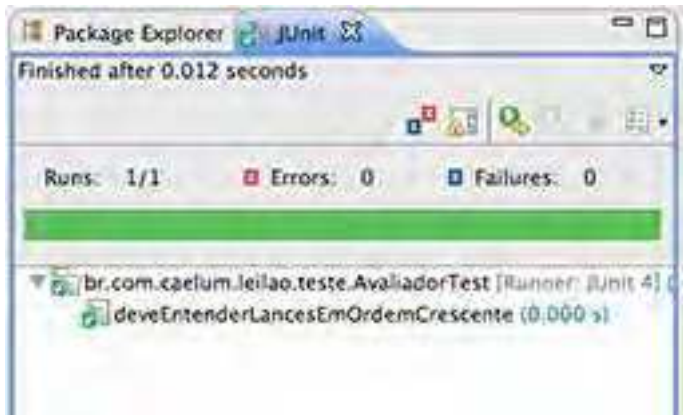


Vamos corrigir o bug. No nosso código, temos um `else` sobrando! Ele faz toda a diferença. Vamos corrigir o código:

```
public void avalia(Leilao leilao) {  
  
    for(Lance lance : leilao.getLances()) {  
        if(lance.getValor() > maiorDeTodos) {  
            maiorDeTodos = lance.getValor();  
        }  
        if(lance.getValor() < menorDeTodos) {  
            menorDeTodos = lance.getValor();  
        }  
    }  
}
```

```
}
```

Rodamos o teste novamente. Agora ele está verde: passou!



Veja o tempo que o teste levou para ser executado: alguns milissegundos. Isso significa que podemos rodar esse teste **O TEMPO TODO**, que é rápido e não custa nada. É a máquina que roda! Agora imagine um sistema com 5000 testes como esses. Ao clicar em um botão, o JUnit, em alguns segundos, executará 5000 testes! Quanto tempo levaríamos para executar o mesmo teste de maneira manual?

## 1.5 CONVENÇÕES NA ESCRITA DE TESTES

Como você percebeu, teste é código, e código pode ser escrito de muitas formas diferentes. Mas para facilitar a vida de todos nós, há algumas convenções que geralmente seguimos.

Por exemplo, mesmo aqueles que gostam de ter seus códigos escritos em português batizam suas classes de teste com o sufixo `Test`. Não “testes”, ou “TesteDaClasse”. Por quê? Porque essa é a convenção, e qualquer desenvolvedor que olhar uma classe, por exemplo, `NotaFiscalTest`, sabe que ela contém testes automatizados para a classe `NotaFiscal`.

A separação do código de teste e código de produção é outra prática comum. Em Java, é normal termos *source folders* separados; em C#, os testes ficam em DLLs diferentes. É também bastante comum que o pacote (ou namespace, em C#) da classe de teste seja o mesmo da classe de produção. Isso facilita a busca pelas classes de teste, afinal você sabe que todos os testes do pacote `br.com.caelum` estão em `br.com.caelum` no outro *source folder*.

O `assertEquals` é também outra coisa padrão. Apesar de não parecer natural, a ordem certa dos parâmetros é `assertEquals(esperado, calculado)`. Ou seja, o valor esperado é o primeiro, e o valor calculado é o segundo. Mas se estou comparando a igualdade, qual a diferença? A diferença é na hora de mensagem de erro. O JUnit mostrará o valor esperado e o calculado em uma frase bonita. Se você inverter os parâmetros, a mensagem também ficará invertida.

### MAS POR QUE NÃO PACOTES SEPARADOS?

Alguns desenvolvedores preferem colocar seus testes em pacotes diferentes dos da classe de produção. A razão para isso é que assim o teste só conseguirá invocar os métodos públicos; no mesmo pacote, ele conseguirá invocar métodos default, por exemplo.

Eu, em particular prefiro colocar no mesmo pacote e, por educação, testar apenas os métodos públicos. Falarei mais sobre isso à frente.

## 1.6 MAS SERÁ QUE SOU PRODUTIVO ASSIM?

Depende da sua definição de produtividade. Se produtividade significa linhas de código de produção escritas por dia, talvez você seja menos produtivo. Agora, se sua definição é algo como “linhas de código escritas com qualidade”, então, muito provavelmente, você será mais produtivo com testes.

É difícil garantir qualidade de um sistema sem testes automatizados, por todos os motivos já citados ao longo deste capítulo.

Além disso, alguns estudos mostram que programadores que escrevem testes, a longo prazo, são mais produtivos do que os que não escrevem e até gastam menos tempo depurando o código! Isso faz sentido: imagine um

desenvolvedor que testa manualmente. Quantas vezes por dia ele executa o MESMO teste? O programador que automatiza o teste gasta seu tempo apenas 1 vez: escrevendo-o. Depois, executar o teste é rápido e barato. Ou seja, ao longo do tempo, escrever testes automatizados vai fazer você economizar tempo.

Ou seja, você é sim produtivo. Não produtivo é ficar fazendo o mesmo teste manual 20 vezes por dia, e ainda assim entregar software com bug.

## 1.7 TESTANDO O QUE REALMENTE É NECESSÁRIO

No fim do capítulo passado, escrevemos nosso primeiro teste automatizado de unidade para a classe `Avaliador` e garantimos que nosso algoritmo sempre funcionará para uma lista de lances em ordem crescente. Mas será que só esse teste é suficiente?

Para confiarmos que a classe `Avaliador` realmente funciona, precisamos cobri-la com mais testes. Nesse momento, temos somente um teste. O cenário do teste nesse caso são 3 lances com os valores 250, 300, 400. Mas será que se passarmos outros valores, ele continua funcionando? Vamos testar com 1000, 2000 e 3000. Na mesma classe de testes `AvaliadorTest`, apenas adicionamos um outro método de testes. É assim que fazemos: cada novo teste é um novo método. **Não** apagamos o teste anterior, mas sim criamos um novo.

```
import org.junit.Assert;

class AvaliadorTest {

    @Test
    public void deveEntenderLancesEmOrdemCrescente() {
        // código aqui ainda...
    }

    @Test
    public void deveEntenderLancesEmOrdemCrescenteComOutrosValores() {
        Usuario joao = new Usuario("Joao");
        Usuario jose = new Usuario("José");
```



```
Usuario maria = new Usuario("Maria");

Leilao leilao = new Leilao("Playstation 3 Novo");

leilao.propoe(new Lance(maria,1000.0));
leilao.propoe(new Lance(joao,2000.0));
leilao.propoe(new Lance(jose,3000.0));

Avaliador leiloeiro = new Avaliador();
leiloeiro.avalial(leilao);

Assert.assertEquals(3000, leiloeiro.getMaiorLance(), 0.0001);
Assert.assertEquals(1000, leiloeiro.getMenorLance(), 0.0001);
}
}
```

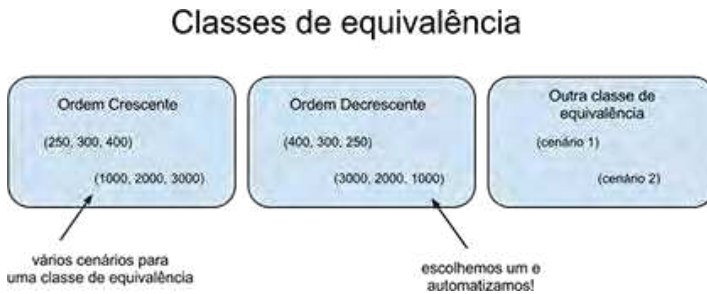
Ao rodar o teste, vemos que ele passa. Mas será que é suficiente ou precisamos de mais testes para ordem crescente? Poderíamos escrever vários deles, afinal o número de valores que podemos passar para esse cenário é quase infinito! Poderíamos ter algo como:

```
class AvaliadorTest {
    @Test public void deveEntenderLancesEmOrdemCrescente1() { }
    @Test public void deveEntenderLancesEmOrdemCrescente2() { }
    @Test public void deveEntenderLancesEmOrdemCrescente3() { }
    @Test public void deveEntenderLancesEmOrdemCrescente4() { }
    @Test public void deveEntenderLancesEmOrdemCrescente5() { }
    // muitos outros testes!
}
```

## 1.8 CLASSES DE EQUIVALÊNCIA

Infelizmente testar todas as combinações é impossível! E se tentarmos fazer isso (e escrevermos muitos testes, como no exemplo anterior), dificultamos a manutenção da bateria de testes! O ideal é escrevermos apenas **um único teste** para cada possível cenário diferente! Por exemplo, um cenário que levantamos é justamente **lances em ordem crescente**. Já temos um teste para ele: `deveEntenderLancesEmOrdemCrescente()`. Não precisamos de outro

para o mesmo cenário! Na área de testes de software, chamamos isso de **classe de equivalência**. Precisamos de um teste por classe de equivalência. A figura a seguir exemplifica isso:



A grande charada então é encontrar essas classes de equivalência. Para esse nosso problema, por exemplo, é possível enxergar alguns diferentes cenários:

- Lances em ordem crescente;
- Lances em ordem decrescente;
- Lances sem nenhuma ordem específica;
- Apenas um lance na lista.

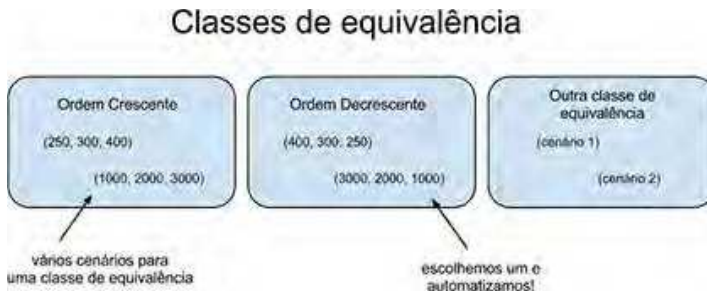
Veja que cada um é diferente do outro; eles testam “cenários” diferentes! Vamos começar pelo teste de apenas um lance na lista. O cenário é simples: basta criar um leilão com apenas um lance. A saída também é fácil: o menor e o maior valor serão idênticos ao valor do único lance.

```
import org.junit.Assert;

class AvaliadorTest {

    @Test
    public void deveEntenderLancesEmOrdemCrescente() {
```

para o mesmo cenário! Na área de testes de software, chamamos isso de **classe de equivalência**. Precisamos de um teste por classe de equivalência. A figura a seguir exemplifica isso:



A grande charada então é encontrar essas classes de equivalência. Para esse nosso problema, por exemplo, é possível enxergar alguns diferentes cenários:

- Lances em ordem crescente;
- Lances em ordem decrescente;
- Lances sem nenhuma ordem específica;
- Apenas um lance na lista.

Veja que cada um é diferente do outro; eles testam “cenários” diferentes! Vamos começar pelo teste de apenas um lance na lista. O cenário é simples: basta criar um leilão com apenas um lance. A saída também é fácil: o menor e o maior valor serão idênticos ao valor do único lance.

```
import org.junit.Assert;

class AvaliadorTest {

    @Test
    public void deveEntenderLancesEmOrdemCrescente() {
```

```
    // código aqui ainda...
}

@Test
public void deveEntenderLeilaoComApenasUmLance() {
    Usuario joao = new Usuario("Joao");
    Leilao leilao = new Leilao("Playstation 3 Novo");

    leilao.propoe(new Lance(joao, 1000.0));

    Avaliador leiloeiro = new Avaliador();
    leiloeiro.avaliao(leilao);

    Assert.assertEquals(1000, leiloeiro.getMaiorLance(), 0.0001);
    Assert.assertEquals(1000, leiloeiro.getMenorLance(), 0.0001);
}
}
```

## 1.9 IMPORT ESTÁTICO DO ASSERT

Ótimo! O teste passa! Mas agora repare: quantas vezes já escrevemos `Assert.assertEquals()`? Muitas! Um dos pontos em que vamos batalhar ao longo do curso é a qualidade do código de testes; ela deve ser tão boa quanto a do seu código de produção. Vamos começar por diminuir essa linha. O método `assertEquals()` é estático, portanto, podemos importá-lo de maneira estática! Basta fazer uso do `import static`! Veja o código:

```
import static org.junit.Assert.assertEquals;

class AvaliadorTest {

    // outros testes ainda estão aqui...

    @Test
    public void deveEntenderLeilaoComApenasUmLance() {
        Usuario joao = new Usuario("Joao");
        Leilao leilao = new Leilao("Playstation 3 Novo");
```

```
leilao.propoe(new Lance(joao,1000.0));

Avaliador leiloeiro = new Avaliador();
leiloeiro.avalua(leilao);

// veja que não precisamos mais da palavra Assert!
assertEquals(1000, leiloeiro.getMaiorLance(), 0.0001);
assertEquals(1000, leiloeiro.getMenorLance(), 0.0001);
    }
}
```

Pronto! Muito mais sucinto! Importar estaticamente os métodos da classe `Assert` é muito comum, e você encontrará muitos códigos de teste assim!

## 1.10 A PRÓXIMA FUNCIONALIDADE: OS 3 MAIORES LANCES

Neste momento, precisamos implementar a próxima funcionalidade do `Avaliador`. Ele precisa agora retornar os três maiores lances dados! Veja que a implementação é um pouco complicada. O método `pegaOsMaioresNo()` ordena a lista de lances em ordem decrescente, e depois pega os 3 primeiros itens:

```
public class Avaliador {

    private double maiorDeTodos = Double.NEGATIVE_INFINITY;
    private double menorDeTodos = Double.POSITIVE_INFINITY;
    private List<Lance> maiores;

    public void avalua(Leilao leilao) {
        for(Lance lance : leilao.getLances()) {
            if(lance.getValor() > maiorDeTodos)
                maiorDeTodos = lance.getValor();
            if (lance.getValor() < menorDeTodos)
                menorDeTodos = lance.getValor();
        }

        pegaOsMaioresNo(leilao);
    }
}
```

```
    }

    private void pegaOsMajoresNo(Leilao leilao) {
        maiores = new ArrayList<Lance>(leilao.getLances());
        Collections.sort(majores, new Comparator<Lance>() {
            public int compare(Lance o1, Lance o2) {
                if(o1.getValor() < o2.getValor()) return 1;
                if(o1.getValor() > o2.getValor()) return -1;
                return 0;
            }
        });
        maiores = maiores.subList(0, 3);
    }

    public List<Lance> getTresMajores() {
        return this.majores;
    }

    public double getMajiorLance() {
        return maiorDeTodos;
    }

    public double getMenorLance() {
        return menorDeTodos;
    }
}
```

Vamos agora testar! Para isso, criaremos um método de teste que dará alguns lances e ao final verificaremos que os três maiores selecionados pelo Avaliador estão corretos:

```
public class AvaliadorTest {
    // outros testes aqui

    @Test
    public void deveEncontrarOsTresMajoresLances() {
        Usuario joao = new Usuario("João");
        Usuario maria = new Usuario("Maria");
        Leilao leilao = new Leilao("Playstation 3 Novo");
    }
}
```

```

        leilao.propoe(new Lance(joao, 100.0));
        leilao.propoe(new Lance(maria, 200.0));
        leilao.propoe(new Lance(joao, 300.0));
        leilao.propoe(new Lance(maria, 400.0));

        Avaliador leiloeiro = new Avaliador();
        leiloeiro.avalua(leilao);

        List<Lance> maiores = leiloeiro.getTresMaiores();

        assertEquals(3, maiores.size());
    }
}

```

Vamos rodar o teste, e veja a surpresa: o novo teste passa, mas o anterior quebra! Será que perceberíamos isso se não tivéssemos a bateria de testes de unidade nos ajudando? Veja a segurança que os testes nos dão. Implementamos a nova funcionalidade, mas quebramos a anterior, e percebemos na hora!

Vamos corrigir: o problema é na hora de pegar apenas os 3 maiores. E se a lista tiver menos que 3 elementos? Basta alterar a linha a seguir e corrigimos:

```

maiores = maiores.subList(0,
    maiores.size() > 3 ? 3 : maiores.size()
);

```

Pronto, agora todos os testes passam.

Veja a asserção do nosso teste: ele verifica o tamanho da lista. Será que é suficiente? Não! Esse teste só garante que a lista tem três elementos, mas não garante o conteúdo desses elementos. Sempre que testamos uma lista, além de verificar seu tamanho precisamos verificar o conteúdo interno dela. Vamos verificar o conteúdo de cada lance dessa lista:

```

import static org.junit.Assert.assertEquals;
// outros imports aqui

```

```
public class AvaliadorTest {
    // outros testes aqui

    @Test
    public void deveEncontrarOsTresMajoresLances() {
        Usuario joao = new Usuario("João");
        Usuario maria = new Usuario("Maria");
        Leilao leilao = new Leilao("Playstation 3 Novo");

        leilao.propoe(new Lance(joao, 100.0));
        leilao.propoe(new Lance(maria, 200.0));
        leilao.propoe(new Lance(joao, 300.0));
        leilao.propoe(new Lance(maria, 400.0));

        Avaliador leiloeiro = new Avaliador();
        leiloeiro.avalua(leilao);

        List<Lance> maiores = leiloeiro.getTresMajores();

        assertEquals(3, maiores.size());
        assertEquals(400, maiores.get(0).getValor(), 0.00001);
        assertEquals(300, maiores.get(1).getValor(), 0.00001);
        assertEquals(200, maiores.get(2).getValor(), 0.00001);
    }
}
```

O teste passa! Mesmo que não entendamos bem a implementação, pelo menos temos a segurança de que, aparentemente, ela funciona. Mas será que só esse teste é suficiente!?



### UM ÚNICO ASSERT POR TESTE?

Uma regra bastante popular na área de teste é “tenha um único assert por teste”. Segundo os que defendem a ideia, ao ter apenas um assert por teste, você faz o seu teste ser mais focado.

A minha regra é um pouco mais diferente: faça asserts em apenas um único objeto no seu teste. Em sistemas orientados a objeto, sua unidade é a classe, e muitas vezes um único comportamento altera diversos atributos da classe. É por isso que tínhamos asserts para o “menor” e “maior” lance, afinal, ambos estavam no objeto `Leiloeiro`.

O que você deve evitar ao máximo é ter mais de um teste diferente dentro do mesmo método de teste. Isso só deixa seu código de teste maior e mais confuso. Nesses casos, separe-os em dois testes.

## 1.11 TESTANDO CASOS ESPECIAIS

É sempre interessante tratar de casos especiais no teste. Por exemplo, tratamos o caso da lista com um elemento separado do caso da lista com vários elementos. Isso faz sentido? Por quê? Consegue ver outros casos como esse, que merecem atenção especial?

Tratar o caso da lista com um elemento separado do caso da lista com vários elementos faz todo o sentido. É muito comum, durante a implementação, pensarmos direto no caso complicado, e esquecermos de casos simples, mas que acontecem. Por esse motivo é importante os testarmos.

Quando lidamos com listas, por exemplo, é sempre interessante tratarmos o caso da lista cheia, da lista com apenas um elemento, da lista vazia.

Se estamos lidando com algoritmos cuja ordem é importante, precisamos testar ordem crescente, decrescente, randômica.

Um código que apresente um `if (salario >= 2000)`, por exemplo, precisa de três diferentes testes:

- Um cenário com salário menor do que 2000
- Um cenário com salário maior do que 2000

- Um cenário com salário igual a 2000

Afinal, quem nunca confundiu um  $>$  por um  $>=$ ? Justamente por isso, o grande desafio da área de testes é pensar em todas essas possíveis situações. Quais são os valores de entrada que farão seu sistema não se comportar da maneira adequada? Encontrar todos eles é sem dúvida uma tarefa complicada.

### **1.12 A BATERIA DE TESTES NOS SALVOU MAIS UMA VEZ!**

A bateria de testes automatizados nos ajuda a encontrar problemas na nossa implementação de forma muito rápida: basta clicarmos em um botão, e alguns segundos depois sabemos se nossa implementação realmente funciona ou não.

Sem uma bateria de testes, dificilmente pegaríamos esse bug em tempo de desenvolvimento. Testes manuais são caros e, por esse motivo, o desenvolvedor comumente testa apenas a funcionalidade atual, deixando de lado os testes de regressão (ou seja, testes para garantir que o resto do sistema ainda continua funcionando mesmo após a implementação da nova funcionalidade).

Por isso, a discussão deste tópico torna-se importante. Os diversos cenários que você automatizou em forma de teste são complicados, e provavelmente são aqueles que o desenvolvedor esquecerá na hora de dar manutenção no código. Você já percebeu a segurança que o teste lhe dá, e o quanto isso é importante na hora da manutenção? Você pode mexer à vontade no seu algoritmo, pois se algo der errado, o teste avisará. Então o teste está lá não só pra garantir que seu software funciona naquele momento, mas que funcionará para sempre, mesmo após diversas manutenções.

### **1.13 QUAIS SÃO AS DIFICULDADES?**

Desenvolvedores que estão aprendendo a testar geralmente sentem dificuldades no momento de levantar e escrever cenários para o teste. Lembre-se que

um teste automatizado é muito parecido com um teste manual. Do mesmo jeito que você pensa no cenário de um teste manual (por exemplo, visitar a página de cadastro, preencher o campo CPF com “123”, clicar no botão etc.), você faz no automatizado.

Foque-se na classe que você está testando. Pense sobre o que você espera dela. Como ela deve funcionar? Se você passar tais parâmetros para ela, como ela deve reagir?

Uma sugestão que sempre dou é ter uma lista, em papel mesmo, com os mais diversos cenários que você precisa testar. E, à medida que você for implementando-os, novos cenários aparecerão. Portanto, antes de sair programando, pense e elenque os testes.

## 1.14 CUIDANDO DOS SEUS TESTES

A partir do momento em que você entendeu a vantagem dos testes e começou a usá-los no seu dia a dia, perceberá então que a bateria só tenderá a crescer. Com isso, teremos mais segurança e qualidade na manutenção e evolução do nosso código.

Entretanto, teste é código. E código mal escrito é difícil de ser mantido, atrapalhando o desenvolvimento. Com isso em mente, pense nos testes que escrevemos até o momento. Observe que temos a seguinte linha em todos os métodos dessa classe:

```
Avaliador leiloeiro = new Avaliador();
```

Mas, e se alterarmos o construtor da classe `Avaliador`, obrigando a ser passado um parâmetro? Precisaríamos alterar em todos os métodos, algo bem trabalhoso. Veja que, em nossos códigos de produção, sempre que encontramos código repetido em dois métodos, centralizamos esse código em um único lugar.

Usaremos essa mesma tática na classe `AvaliadorTest`, isolando essa linha em um único método:

```
public class AvaliadorTest {  
  
    private Avaliador leiloeiro;
```

```
// novo método que cria o avaliador
private void criaAvaliador() {
    this.leiloeiro = new Avaliador();
}

@Test
public void deveEntenderLancesEmOrdemCrescente() {
    // ... código ...

    // invocando método auxiliar
    criaAvaliador();
    leiloeiro.avaliao(leilao);

    // asserts
}

@Test
public void deveEntenderLeilaoComApenasUmLance() {
    // mesma mudança aqui
}

@Test
public void deveEncontrarOsTresMaioresLances() {
    // mesma mudança aqui
}
}
```

Veja que podemos fazer uso de métodos privados em nossa classe de teste para melhorar a qualidade do nosso código, da mesma forma que fazemos no código de produção. Novamente, todas as boas práticas de código podem (e devem) ser aplicadas no código de teste.

Com essa alteração, o nosso código de teste ficou mais fácil de evoluir, pois teremos que mudar apenas o método `criaAvaliador()`. Contudo, os métodos de teste não ficaram menores ou mais legíveis.

Nesses casos, onde o método auxiliar “inicializa os testes”, ou seja, instancia os objetos que serão posteriormente utilizados pelos testes, podemos pedir para o JUnit rodar esse método automaticamente, antes de executar cada teste.

Para isso, basta mudarmos nosso método auxiliar para `public` (afinal, o JUnit precisa enxergá-lo) e anotá-lo com `@Before`. Então, podemos retirar a invocação do método `criaAvaliador()` **de todos os métodos de teste**, já que o próprio JUnit irá fazer isso.

Vamos aproveitar e levar a criação dos usuários também para o método auxiliar. O intuito é deixar nosso método de teste mais fácil ainda de ser lido.

```
public class AvaliadorTest {

    private Avaliador leiloeiro;
    private Usuario joao;
    private Usuario jose;
    private Usuario maria;

    @Before
    public void criaAvaliador() {
        this.leiloeiro = new Avaliador();
        this.joao = new Usuario("João");
        this.jose = new Usuario("José");
        this.maria = new Usuario("Maria");
    }

    @Test
    public void deveEntenderLancesEmOrdemCrescente() {

        Leilao leilao = new Leilao("Playstation 3 Novo");

        leilao.propoe(new Lance(joao, 250.0));
        leilao.propoe(new Lance(jose, 300.0));
        leilao.propoe(new Lance(maria, 400.0));

        // parte 2: ação
        leiloeiro.avalia(leilao);

        // parte 3: validação
        assertEquals(400.0, leiloeiro.getMaiorLance(), 0.00001);
        assertEquals(250.0, leiloeiro.getMenorLance(), 0.00001);
    }
}
```

```
@Test
public void deveEntenderLeilaoComApenasUmLance() {
    // usa os atributos joao, jose, maria e leiloeiro.
}

@Test
public void deveEncontrarOsTresMajoresLances() {
    // usa os atributos joao, jose, maria e leiloeiro.
}
}
```

Ao rodarmos essa bateria de testes, o JUnit executará o método `criaAvaliador()` 3 vezes: uma vez antes de cada método de teste!

Repare como conseguimos ler cada método de teste de maneira mais fácil agora. Toda a instanciação de variáveis está isolada em um único método. É uma boa prática manter seu código de teste fácil de ler e isolar toda a inicialização dos testes dentro de métodos anotados com `@Before`.

## 1.15 TEST DATA BUILDERS

Podemos melhorar ainda mais nosso código de teste. Veja que criar um `Leilao` não é uma tarefa fácil nem simples de ler. E note em quantos lugares diferentes fazemos uso da classe `Leilão`: `AvaliadorTest`, `LeilaoTest`.

Podemos isolar o código de criação de leilão em uma classe específica, mais legível e clara. Podemos, por exemplo, fazer com que nosso método fique algo como:

```
@Test
public void deveEncontrarOsTresMajoresLances() {
    Leilao leilao = new CriadorDeLeilao()
        .para("Playstation 3 Novo")
        .lance(joao, 100.0)
        .lance(maria, 200.0)
        .lance(joao, 300.0)
        .lance(maria, 400.0)
        .constroi();
}
```

```
        leiloeiro.avaliao(leilao);

        List<Lance> maiores = leiloeiro.getTresMaiores();
        assertEquals(3, maiores.size());
        assertEquals(400.0, maiores.get(0).getValor(), 0.00001);
        assertEquals(300.0, maiores.get(1).getValor(), 0.00001);
        assertEquals(200.0, maiores.get(2).getValor(), 0.00001);
    }
}
```

Observe como esse código é mais fácil de ler, e mais enxuto que o anterior! E escrever a classe `CriadorDeLeiloes` é razoavelmente simples! O único segredo talvez seja possibilitar que invoquemos um método atrás do outro. Para isso, basta retornarmos o `this` em todos os métodos!

Vamos à implementação:

```
public class CriadorDeLeilao {

    private Leilao leilao;

    public CriadorDeLeilao() { }

    public CriadorDeLeilao para(String descricao) {
        this.leilao = new Leilao(descricao);
        return this;
    }

    public CriadorDeLeilao lance(Usuario usuario, double valor) {
        leilao.propoe(new Lance(usuario, valor));
        return this;
    }

    public Leilao constroi() {
        return leilao;
    }
}
```

A classe `CriadorDeLeilao` é a responsável por instanciar leilões para os nossos testes. Classes como essas são muito comuns e são conhecidas como

**Test Data Builders.** Este é um padrão de projeto para código de testes. Sempre que temos classes que são complicadas de serem criadas ou que são usadas por diversas classes de teste, devemos isolar o código de criação das mesmas em um único lugar, para que mudanças na estrutura dessa classe não impactem em todos os nossos métodos de teste.

## 1.16 @AFTER, @BEFORECLASS E @AFTERCLASS

Ao contrário do `@Before`, métodos anotados com `@After` são executados após a execução do método de teste. Utilizamos métodos `@After` quando nossos testes consomem recursos que precisam ser finalizados. Exemplos podem ser testes que acessam banco de dados, abrem arquivos, abrem sockets etc.

Analogamente, métodos anotados com `@BeforeClass` são executados apenas uma vez, antes de todos os métodos de teste. O método anotado com `@AfterClass`, por sua vez, é executado uma vez, após a execução do último método de teste da classe. Eles podem ser bastante úteis quando temos algum recurso que precisa ser inicializado apenas uma vez e que pode ser consumido por todos os métodos de teste sem a necessidade de ser reinicializado.

Apesar de esses testes não serem mais considerados testes de unidade, afinal eles falam com outros sistemas, desenvolvedores utilizam JUnit para escrever testes de integração. Os mesmos são discutidos mais à frente nos capítulos sobre testes de integração.

## 1.17 ACOPLAMENTO ENTRE TESTES E PRODUÇÃO

Algo que você deve começar a perceber é que o código de teste é altamente acoplado ao nosso código de produção. Isso significa que uma mudança no código de produção pode impactar profundamente em nosso código de testes. Se não cuidarmos dos nossos testes, uma simples mudança pode impactar em MUITAS mudanças no código de testes.

É por isso que neste capítulo discutimos métodos auxiliares e *test data builders*. Todos eles são maneiras para fazer com que nosso código de testes evolua mais facilmente.



## 1.18 CUIDE BEM DOS SEUS TESTES

A ideia deste capítulo é mostrar pra você a importância de cuidar dos seus códigos de teste. Refatore-os constantemente. Lembre-se: uma bateria de testes mal escrita pode deixar de ajudar e começar a atrapalhar. Eu mostrei uma ou outra prática, mas você pode (e deve) usar todo o conhecimento que tem sobre código de qualidade e aplicá-lo na sua bateria de testes.

Daqui para frente, tentarei ao máximo só escrever testes, usando essas boas práticas.

## 1.19 TESTANDO EXCEÇÕES

Nem sempre queremos que nossos métodos de produção modifiquem estado de algum objeto. Algumas vezes, queremos tratar casos excepcionais. Em nosso código atual, o que aconteceria caso o leilão passado não recebesse nenhum lance? O atributo `maiorDeTodos`, por exemplo, ficaria com um número muito pequeno (no caso, `Double.NEGATIVE_INFINITY`). Isso não faz sentido! Nesses casos, muitos desenvolvedores podem optar por lançar uma exceção.

Podemos verificar se o leilão possui lances. Caso não possua nenhum lance, podemos lançar uma exceção:

```
public class Avaliador {  
  
    private double maiorDeTodos = Double.NEGATIVE_INFINITY;  
    private double menorDeTodos = Double.POSITIVE_INFINITY;  
    private List<Lance> maiores;  
  
    public void avalia(Leilao leilao) {  
        // lançando a exceção  
        if(leilao.getLances().size() == 0)  
            throw new RuntimeException(  
                "Não é possível avaliar um leilão sem lances"  
            );  
  
        for(Lance lance : leilao.getLances()) {  
            if(lance.getValor() > maiorDeTodos)
```

```
        maiorDeTodos = lance.getValor();
        if (lance.getValor() < menorDeTodos)
            menorDeTodos = lance.getValor();
    }

    tresMaiores(leilao);
}

// código continua aqui...
}
```

A pergunta agora é: como testar esse trecho de código? Se escrevermos um teste da maneira que estamos acostumados, o teste falhará, pois o método `avalia()` lançará uma exceção. Além disso, como fazemos o assert? Não existe um `assertException()` ou algo do tipo:

```
@Test
public void naoDeveAvaliarLeiloesSemNenhumLanceDado() {
    Leilao leilao = new CriadorDeLeilao()
        .para("Playstation 3 Novo")
        .constroi();

    leiloeiro.avalia(leilao);

    // como fazer o assert?
}
```

Uma alternativa é fazermos o teste falhar caso a exceção não seja lançada. Podemos fazer isso por meio do método `Assert.fail()`, que falha o teste:

```
@Test
public void naoDeveAvaliarLeiloesSemNenhumLanceDado() {
    try {
        Leilao leilao = new CriadorDeLeilao()
            .para("Playstation 3 Novo")
            .constroi();

        leiloeiro.avalia(leilao);
        Assert.fail();
    }
```

```

    }
    catch(RuntimeException e) {
        // deu certo!
    }
}

```

O teste agora passa, afinal o método lançará a exceção, a execução cairá no `catch(RuntimeException e)`, e como não há `asserts` lá dentro, o teste passará. Essa implementação funciona, mas não é a melhor possível.

A partir do JUnit 4, podemos avisar que o teste na verdade passará se uma exceção for lançada. Para isso, basta fazermos uso do atributo `expected`, pertencente à anotação `@Test`. Dessa maneira, eliminamos o *try-catch* do nosso código de teste, e ele fica ainda mais legível:

```

@Test(expected=RuntimeException.class)
public void naoDeveAvaliarLeiloesSemNenhumLanceDado() {
    Leilao leilao = new CriadorDeLeilao()
        .para("Playstation 3 Novo")
        .constroi();

    leiloeiro.avalia(leilao);
}

```

Veja que passamos a exceção que deve ser lançada pelo teste. Caso a exceção não seja lançada ou não bata com a informada, o teste falhará. Agora já sabemos como testar métodos que lançam exceções em determinados casos.

## 1.20 MELHORANDO A LEGIBILIDADE DOS TESTES

Ótimo. Vamos agora continuar a melhorar nosso código de teste. Nossos testes já estão bem expressivos, mas algumas coisas ainda não são naturais. Por exemplo, nossos `asserts`. A ordem exigida pelo JUnit não é “natural”, afinal normalmente pensamos no valor que calculamos e depois no valor que esperamos. Além disso, a palavra `assertEquals()` poderia ser ainda mais expressiva. Veja o teste a seguir e compare os dois `asserts`:

```

class AvaliadorTest {

```

```
@Test
public void deveEntenderLancesEmOrdemCrescente() {

    Leilao leilao = new CriadorDeLeilao()
        .para("Playstation 3 Novo")
        .lance(joao, 250)
        .lance(jose, 300)
        .lance(maria, 400)
        .constroi();

    leiloeiro.avalial(leilao);

    assertThat(leiloeiro.getMenorLance(), equalTo(250.0));
    assertEquals(400.0, leiloeiro.getMaiorLance(), 0.00001);
}
}
```

Veja que o primeiro assert é muito mais legível. Se lermos essa linha como uma frase em inglês, temos **garanta que o menor lance é igual a 250.0**. Muito mais legível!

Para conseguirmos escrever asserts como esse, podemos fazer uso do projeto **Hamcrest**. Ele contém um monte de instruções como essas, que simplesmente nos ajudam a escrever um teste mais claro. Alterando o método de teste na classe `AvaliadorTest` para que ele use as asserções do Hamcrest, veja como ele fica mais legível:

```
import static org.hamcrest.MatcherAssert.assertThat;
import static org.junit.Assert.assertEquals;
import static org.hamcrest.Matchers.*;

class AvaliadorTest {
    @Test
    public void deveEntenderLancesEmOrdemCrescente() {

        Leilao leilao = new CriadorDeLeilao()
            .para("Playstation 3 Novo")
            .lance(joao, 250)
            .lance(jose, 300)
            .lance(maria, 400)
```

```

        .constroi();

        leiloeiro.avalia(leilao);

        assertThat(leiloeiro.getMenorLance(), equalTo(250.0));
        assertThat(leiloeiro.getMaiorLance(), equalTo(400.0));
    }
}

```

Veja agora o teste que garante que o avaliador encontra os três maiores lances dados para um leilão:

```

@Test
public void deveEncontrarOsTresMajoresLances() {
    Leilao leilao = new CriadorDeLeilao()
        .para("Playstation 3 Novo")
        .lance(joao, 100)
        .lance(maria, 200)
        .lance(joao, 300)
        .lance(maria, 400)
        .constroi();

    leiloeiro.avalia(leilao);

    List<Lance> maiores = leiloeiro.getTresMajores();
    assertEquals(3, maiores.size());

    assertEquals(400.0, maiores.get(0).getValor(), 0.00001);
    assertEquals(300.0, maiores.get(1).getValor(), 0.00001);
    assertEquals(200.0, maiores.get(2).getValor(), 0.00001);
}

```

Podemos mudar esses asserts para algo muito mais expressivo. Veja que estamos conferindo se a lista `maiores` contém os 3 lances esperados. Um detalhe é que para que o **matcher** (o nome pelo qual esses métodos estáticos `hasItems`, `equalTo` etc. são chamados) funcione, é necessário implementar o método `equals()` na classe `Lance`:

```

assertThat(maiores, hasItems(
    new Lance(maria, 400),

```

```
        new Lance(joao, 300),  
        new Lance(maria, 200)  
    ));  
}
```

Lembre-se sempre de deixar seu teste o mais legível possível. O Hamcrest é uma alternativa. Apesar de o desenvolvedor precisar conhecer os matchers dele para que consiga utilizar bem o framework, os testes que fazem uso de Hamcrest geralmente são mais fáceis de ler. Por esse motivo, o uso de Hamcrest é muito comum entre os desenvolvedores, e é inclusive encorajado. O Hamcrest possui muitos outros matchers e você pode conferi-los na documentação do projeto, em <http://code.google.com/p/hamcrest/wiki/Tutorial>.

### 1.21 100% DE COBERTURA DE TESTES?

Se conseguimos agora escrever testes para nosso código (e mais para a frente, você verá que realmente conseguirá escrever para todo o código), será que faz sentido termos testes para todas as nossas linhas de código? Ou seja, todo método de produção ter ao menos um teste automatizado passando por ele? Existe até uma métrica bastante conhecida, chamada *cobertura de código*, que nos indica a porcentagem de código que está coberta por ao menos um teste. Uma cobertura de 90% indica que existem 10% de código que não são exercitados por nenhum teste automatizado.

Apesar de a ideia parecer ótima, 100% de cobertura de código não deve ser sua meta absoluta. Afinal, temos trechos de código que não precisam diretamente de testes. Um bom exemplo são *getters* e *setters* na linguagem Java. Geralmente você usa o próprio Eclipse para gerá-los, eles possuem uma única linha de código padrão e não sofrem modificações no futuro. Por que teríamos um teste pra eles? Quando eles vão parar de funcionar?

É uma decisão difícil escolher qual trecho de código não precisa ser testado. O fato é que se você precisar priorizar, teste aqueles métodos que são complicados e/ou importantes. Use o número de cobertura para ajudá-lo a identificar trechos como esse que não estão testados. Mas não fique focado em chegar aos 100%, porque não é isso que garantirá que seu sistema seja a prova de defeitos.



## CAPÍTULO 2

# Praticando Test-Driven Development (TDD)

Será que conseguimos pensar em uma outra maneira de desenvolver e escrever nossos testes, que não a que estamos acostumados? Veja a classe `Leilao`, por exemplo. Ela não tem teste nenhum:

```
public class Leilao {  
  
    private String descricao;  
    private List<Lance> lances;  
  
    public Leilao(String descricao) {  
        this.descricao = descricao;  
        this.lances = new ArrayList<Lance>();  
    }  
}
```



```
public void propoe(Lance lance) {
    lances.add(lance);
}

public String getDescricao() {
    return descricao;
}

public List<Lance> getLances() {
    return Collections.unmodifiableList(lances);
}
}
```

## 2.1 TESTES, DO JEITO QUE VOCÊ JÁ SABE

Apesar de ser simples, no contexto de um sistema de leilão, essa classe é de extrema importância, portanto, merece ser testada. O método `propoe()`, em especial, é essencial para o leilão e provavelmente sofrerá mudanças no decorrer do projeto. Por isso, ela precisa ser testada para termos certeza de que ela está funcionando conforme se espera. Inicialmente, vamos analisar se um determinado lance que foi proposto ficará armazenado no leilão. Para isso, temos dois casos a serem averiguados: a realização de apenas um lance e a de mais de um lance.

O código para realizar tal teste não tem muito segredo. Começaremos instanciando um leilão e serão propostos alguns lances nele:

```
import static org.junit.Assert.assertEquals;

import org.junit.Test;

public class LeilaoTest {

    @Test
    public void deveReceberUmLance() {
        Leilao leilao = new Leilao("Macbook Pro 15");
        assertEquals(0, leilao.getLances().size());
    }
}
```

```
leilao.propoe(new Lance(new Usuario("Steve Jobs"), 2000));

assertEquals(1, leilao.getLances().size());
assertEquals(2000,
    leilao.getLances().get(0).getValor(), 0.00001);
}

@Test
public void deveReceberVariosLances() {
    Leilao leilao = new Leilao("Macbook Pro 15");
    leilao.propoe(new Lance(new Usuario("Steve Jobs"), 2000));
    leilao.propoe(new Lance(new Usuario("Steve Wozniak"), 3000));

    assertEquals(2, leilao.getLances().size());
    assertEquals(2000,
        leilao.getLances().get(0).getValor(), 0.00001);
    assertEquals(3000,
        leilao.getLances().get(1).getValor(), 0.00001);
}
}
```

Agora implementaremos duas novas regras de negócio no processo de lances em um leilão:

- Uma pessoa não pode propor dois lances em sequência;
- Uma pessoa não pode dar mais do que cinco lances no mesmo leilão.

## 2.2 MUDANDO A MANEIRA DE DESENVOLVER

Contudo, dessa vez, vamos fazer diferente. Estamos muito acostumados a implementar o código de produção e testá-lo ao final. Mas será que essa é a única maneira de desenvolver um projeto? Vamos tentar inverter e **começar pelos testes**! Além disso, vamos tentar ao máximo **ser o mais simples possível**, ou seja, pensar no cenário mais simples naquele momento e implementar sempre o código mais simples que resolva o problema.

Vamos começar pelo cenário. Um novo leilão cujo mesmo usuário dê dois lances seguidos e, por isso, o último lance deve ser ignorado:

```
@Test
public void naoDeveAceitarDoisLancesSeguidosDoMesmoUsuario() {
    Leilao leilao = new Leilao("Macbook Pro 15");
    Usuario steveJobs = new Usuario("Steve Jobs");

    leilao.propoe(new Lance(steveJobs, 2000));
    leilao.propoe(new Lance(steveJobs, 3000));

    assertEquals(1, leilao.getLances().size());
    assertEquals(2000,
        leilao.getLances().get(0).getValor(), 0.00001);
}
```

Ótimo, teste escrito. Ao rodar o teste, ele falha. Mas tudo bem, já estávamos esperando por isso! Precisamos fazê-lo passar agora, da maneira mais simples possível. Vamos modificar o método `propoe()`. Agora ele, antes de adicionar o lance, verificará se o último lance da lista não pertence ao usuário que está dando o lance naquele momento. Veja que fazemos uso do método `equals()` na classe `Usuario`. Você poderia implementá-lo usando o próprio atalho do Eclipse (*Generate hashCode and equals*):

```
public void propoe(Lance lance) {
    if(lances.isEmpty() ||
        !lances.get(lances.size()-1)
            .getUsuario()
            .equals(lance.getUsuario()))
    {

        lances.add(lance);
    }
}
```

Se rodarmos o teste agora, ele passa! Nosso código funciona! Mas veja que o código que produzimos não está muito claro. O `if` em particular está confuso. Agora é uma boa hora para melhorar isso, afinal temos certeza de que o código atual funciona! Ou seja, se melhorarmos o código e rodarmos o teste novamente, ele **deverá continuar verde**.

Vamos, por exemplo, extrair o código responsável por pegar o último elemento da lista em um método privado:

```
public void propoe(Lance lance) {
    if(lances.isEmpty() || !ultimoLanceDado().
        getUsuario().equals(lance.getUsuario())) {
        lances.add(lance);
    }
}

private Lance ultimoLanceDado() {
    return lances.get(lances.size()-1);
}
```

Perfeito. Vamos para a próxima regra de negócio: um usuário só pode dar no máximo 5 lances para um mesmo leilão. De novo, começaremos pelo teste. Vamos criar um leilão e fazer um usuário dar 5 lances nele. Repare que, devido à regra anterior, precisaremos intercalá-los, já que o mesmo usuário não pode fazer dois lances em sequência:

```
@Test
public void naoDeveAceitarMaisDoQue5LancesDeUmMesmoUsuario() {
    Leilao leilao = new Leilao("Macbook Pro 15");
    Usuario steveJobs = new Usuario("Steve Jobs");
    Usuario billGates = new Usuario("Bill Gates");

    leilao.propoe(new Lance(steveJobs, 2000));
    leilao.propoe(new Lance(billGates, 3000));
    leilao.propoe(new Lance(steveJobs, 4000));
    leilao.propoe(new Lance(billGates, 5000));
    leilao.propoe(new Lance(steveJobs, 6000));
    leilao.propoe(new Lance(billGates, 7000));
    leilao.propoe(new Lance(steveJobs, 8000));
    leilao.propoe(new Lance(billGates, 9000));
    leilao.propoe(new Lance(steveJobs, 10000));
    leilao.propoe(new Lance(billGates, 11000));

    // deve ser ignorado
    leilao.propoe(new Lance(steveJobs, 12000));
}
```

```
assertEquals(10, leilao.getLances().size());

int ultimo = leilao.getLances().size() - 1;
Lance ultimoLance = leilao.getLances().get(ultimo);

assertEquals(11000.0,
    ultimoLance.getValor(), 0.00001);
}
```

Ao rodarmos, o teste falhará! Excelente, vamos agora fazê-lo passar escrevendo o código mais simples:

```
public void propoe(Lance lance) {

    int total = 0;
    for(Lance l : lances) {
        if(l.getUsuario().equals(lance.getUsuario())) total++;
    }

    if(lances.isEmpty() ||
        (!ultimoLanceDado()
            .getUsuario().equals(lance.getUsuario())
            && total < 5)) {
        lances.add(lance);
    }
}
```

Pronto! O teste passa! Mas esse código está claro o suficiente? Podemos melhorar! De novo, uma ótima hora para refatorarmos nosso código. Vamos extrair a lógica de contar o número de lances de um usuário em um método privado:

```
public void propoe(Lance lance) {
    if(lances.isEmpty() || (
        !ultimoLanceDado().getUsuario().equals(lance.getUsuario()) &&
        qtdDelancesDo(lance.getUsuario()) < 5)) {
        lances.add(lance);
    }
}
```

```
}

private int qtdDelancesDo(Usuario usuario) {
    int total = 0;
    for(Lance lance : lances) {
        if(lance.getUsuario().equals(usuario)) total++;
    }
    return total;
}
```

Rodamos o teste, e tudo continua passando. Podemos melhorar ainda mais o código, então vamos continuar refatorando. Dessa vez, vamos extrair aquela segunda condição do `if` para que a leitura fique mais clara:

```
public void propoe(Lance lance) {
    if(lances.isEmpty() || podeDarLance(lance.getUsuario())) {
        lances.add(lance);
    }
}

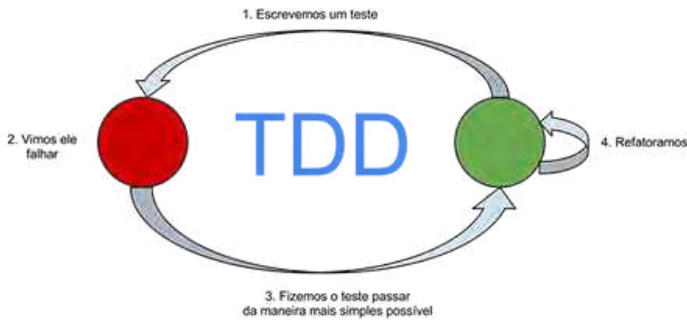
private boolean podeDarLance(Usuario usuario) {
    return !ultimoLanceDado().getUsuario().equals(usuario)
        && qtdDelancesDo(usuario) < 5;
}
```

Pronto! Os testes continuam passando! Poderíamos continuar escrevendo testes antes do código, mas vamos agora pensar um pouco sobre o que acabamos de fazer.

## 2.3 TEST-DRIVEN DEVELOPMENT

Em primeiro lugar, escrevemos um teste; rodamos e o vimos falhar; em seguida, escrevemos o código mais simples para passar o teste; rodamos novamente, e dessa vez ele passou; por fim, refatoramos nosso código para que ele ficasse melhor e mais claro.

A figura a seguir mostra o ciclo que acabamos de fazer:



Esse ciclo de desenvolvimento, onde escrevemos um teste antes do código, é conhecido por **Test-Driven Development**. A popularidade da prática de TDD tem crescido cada vez mais entre os desenvolvedores, uma vez que ela traz diversas vantagens:

- Se sempre escrevermos o teste antes, garantimos que todo nosso código já “nasce” testado;
- Temos segurança para refatorar nosso código, afinal sempre refatoraremos com uma bateria de testes que garante que não quebraremos o comportamento já existente;
- Como o teste é a primeira classe que usa o seu código, você naturalmente tende a escrever código mais fácil de ser usado e, por consequência, mais fácil de ser mantido.
- Efeitos no design. É comum percebermos que o projeto de classes de sistemas feitos com TDD é melhor do que sistemas feitos sem TDD.

## 2.4 EFEITOS NO DESIGN DE CLASSES

Muitos desenvolvedores famosos, como Kent Beck, Martin Fowler e Michael Feathers, dizem que a prática de TDD faz com que seu design de classes melhore. A grande pergunta é: como?

Em alto nível, a ideia é que, quando você começa pelo teste, você escreve naturalmente um código que é mais fácil de ser testado. E um código mais fácil de ser testado apresenta algumas características interessantes:

- Ele é mais coeso. Afinal, um código que faz muita coisa é mais difícil de ser testado.
- Ele é menos acoplado. Pois testar código acoplado é mais difícil.
- A interface pública é simples. Você não quer invocar 10 métodos para conseguir testar o comportamento.
- As precondições são mais simples. Você também não quer montar imensos cenários para testar o método.

Por esses motivos, ao escrever classes que favoreçam a testabilidade, você naturalmente está criando classes mais simples e mais bem desenhadas. É naturalmente mais complicado do que isso, mas esse é um assunto que abordei no meu outro livro, *Test-Driven Development: Teste e Design no Mundo Real*.

### **DEVO TESTAR MÉTODOS PRIVADOS?**

A resposta é direta: não. Se você sente vontade de testar aquele método privado isolado do resto, é o teste lhe dizendo que esse código está no lugar errado. Nesses casos, extraia esse trecho de código para uma classe específica, e teste-a naturalmente. Lembre-se: se está difícil testar, é porque você pode fazer melhor.

## **2.5 BABY STEPS**

*Baby steps* é o nome que damos à ideia de sempre tentarmos escrever o código mais simples que faz o teste passar, e começar pelo cenário de teste mais simples naquele momento. Dar esses passos pequenos pode ser muito benéfico para a sua implementação. Começar pelo teste mais simples nos possibilita



evoluir o código aos poucos (geralmente gostamos de começar pelo caso mais difícil, o que pode dificultar).

Além disso, ao implementar códigos simples, aumentamos a facilidade de manutenção do nosso código. Afinal, código simples é muito mais fácil de se manter do que códigos complexos. Muitas vezes nós, programadores, escrevemos códigos complicados desnecessariamente. TDD nos lembra o tempo todo de ser simples.

Muitas pessoas, no entanto, dizem que tomar passos de bebê o tempo todo pode ser contraproducente. Segundo o próprio autor da prática, Kent Beck, você deve tomar passos pequenos sempre que sua “confiança sobre o código” estiver baixa. Se você está trabalhando em um trecho de código e está bastante confiante sobre ele, você pode dar passos um pouco maiores. Mas cuidado: passos grandes não devem ser a regra, e sim a exceção.

### **CUIDADO COM TRECHOS SIMPLES**

Sempre que escrevemos um trecho de código, achamos que ele é simples. Afinal, nós o escrevemos, ele está inteiro na nossa cabeça. Não há dúvidas. Mas quantas vezes não voltamos a um código que escrevemos 1 mês atrás e não entendemos nada? A única maneira de trabalhar com qualidade e segurança em códigos assim é tendo uma bateria de testes que garanta qualquer mudança feita.

Portanto, não se deixe enganar. Se o método contém uma regra de negócio, teste-o. Você agradecerá no futuro.

## **2.6 DEVO VER O TESTE FALHAR?**

Devemos ver o teste falhar, pois é uma das maneiras que temos para garantir que nosso teste foi implementado corretamente. Afinal, um teste automatizado é código, e podemos implementá-lo incorretamente.

Ao ver que o teste que esperamos falhar realmente falha, temos a primeira garantia de que o implementamos de maneira correta. Imagine se o teste que esperamos que falhe na prática não falha. O que aconteceu?

Para completar o “teste” do teste, podemos escrever o código de produção mais simples que o faz passar. Dessa forma, garantimos que o teste fica vermelho ou verde quando deve.

## 2.7 TDD 100% DO TEMPO?

Não. Como toda prática de engenharia de software, ela deve ser usada no momento certo. TDD faz muito sentido ao implementar novas funcionalidades, ao corrigir bugs, ao trabalhar em códigos complexos etc.

Mas às vezes não precisamos seguir o fluxo ideal da prática de TDD. Por exemplo, às vezes queremos só escrever um conjunto de testes que faltaram para determinada funcionalidade. Nesse momento, não faríamos TDD, mas sim escreveríamos testes.

Em códigos extremamente simples, talvez a prática de TDD não seja necessária. Mas lembre-se: cuidado para não achar que “tudo é simples”, e nunca praticar TDD.

## 2.8 ONDE POSSO LER MAIS SOBRE ISSO?

Você pode ler mais sobre o assunto em meu livro *Test-Driven Development: Testes e Design no Mundo Real*. Lá eu discuto com detalhes cada uma dessas seções: produtividade, efeitos na qualidade interna, efeitos no design etc. Não repetirei todo o discurso aqui, afinal ele é longo.



## CAPÍTULO 3

# Mock Objects

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? 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 lista sem 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**?

### 3.1 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();

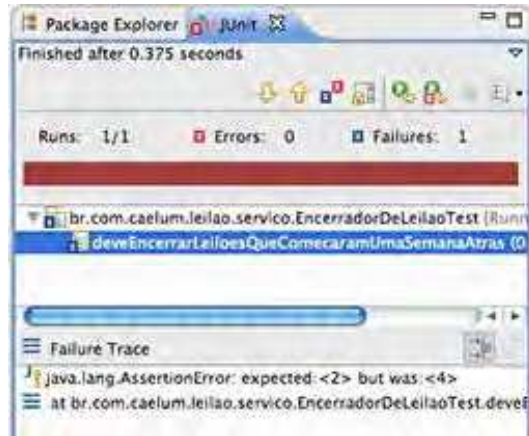
    LeilaoDao dao = new LeilaoDao();
    dao.salva(leilao1);
    dao.salva(leilao2);

    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.

## 3.2 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.

### 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.

### 3.3 MOCKS ESTRITOS E ACOPLAMENTO

Algumas pessoas preferem fazer com que seus mocks tenham comportamento 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 vaziar 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.

### 3.4 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.

### 3.5 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())
```



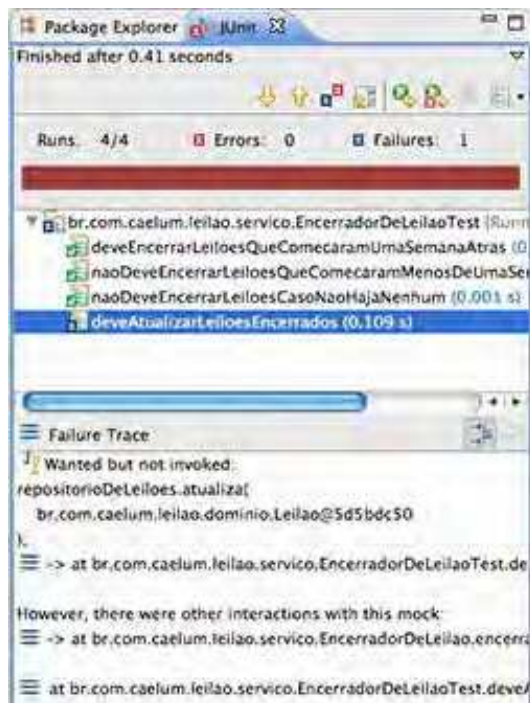
```
        .thenReturn(Arrays.asList(leilao1));

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 com a mensagem a seguir. Repare que ela diz que o método não foi invocado.



Pronto! Dessa forma conseguimos testar a invocação de métodos.

### 3.6 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);
```

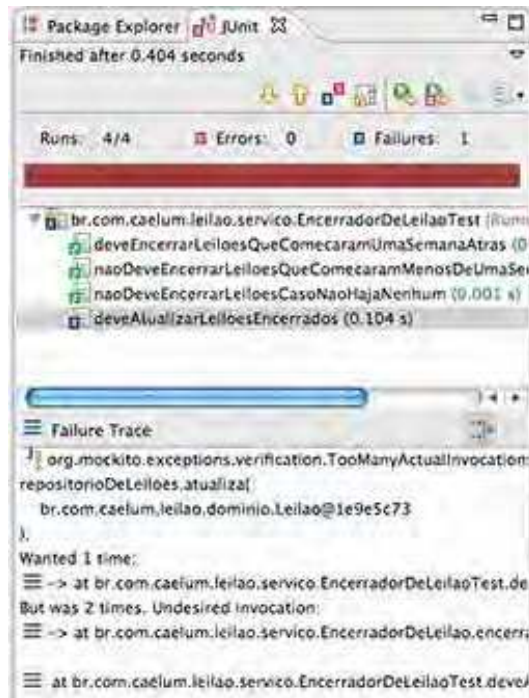
```
Leilao leilao1 = new CriadorDeLeilao().para("TV de plasma")
    .naData(antiga).constroi();

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.

### 3.7 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.

### 3.8 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, carteiro);
```

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 precaver 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);
            }
        } catch (Exception e) {
            // ... código continua
        }
    }
}
```

```

        carteiro.envia(leilao);
    }
}
catch(Exception e) {
    // salvo a exceção no sistema de logs
    // e o loop continua!
}
}
}

```

### 3.9 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);

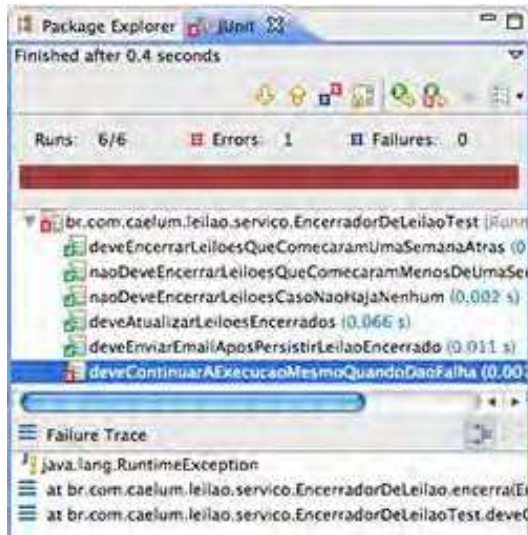
encerrador.encerra();

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.

### 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).

### 3.10 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(avalizador.getMaiorLance())
    .thenReturn(2500.0);

GeradorDePagamento gerador =
    new GeradorDePagamento(leiloes, pagamentos, avalizador);
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.

### 3.11 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!

### 3.12 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 impossível 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 relógio;

    public GeradorDePagamento(RepositorioDeLeiloes leiloes,
        RepositorioDePagamentos pagamentos,
        Avaliador avaliador,
        Relogio relógio) {

        this.leiloes = leiloes;
        this.pagamentos = pagamentos;
        this.avaliador = avaliador;
        this.relogio = relógio;
    }

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

    // ...

    private Calendar primeiroDiaUtil() {
```

```
Calendar data = relógio.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 `Relógio` 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);
    Relógio relógio = mock(Relógio.class);

    // dia 7/abril/2012 é um sábado
    Calendar sabado = Calendar.getInstance();
    sabado.set(2012, Calendar.APRIL, 7);

    // ensinamos o mock a dizer que "hoje" é sábado!
    when(relógio.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));

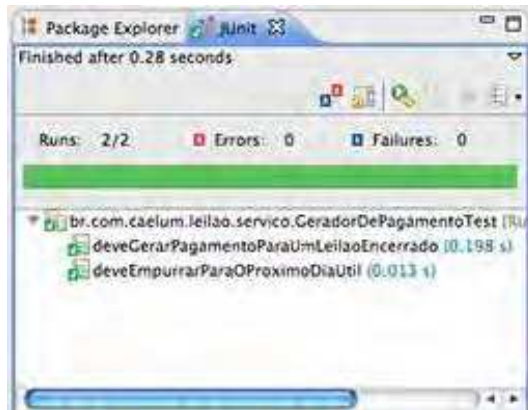
GeradorDePagamento gerador =
    new GeradorDePagamento(leiloes, pagamentos, new Avaliador(), relogio,
        gerador.gera());

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 (geral-

mente 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.

### **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!

### **3.13 O QUE MOCKAR E O QUE NÃO MOCKAR?**

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.

Também não é comum mockar métodos totalmente auxiliares, como aquele simples método que faz uma operação matemática qualquer. Nesses casos, o trabalho de mockar é maior do que o de usar a implementação concreta.

Perceba a troca. Se você não mockar a classe, você precisará montar o cenário completo para ambas as classes (a que você está testando e a de que ela depende, que você optou por não mockar), e se a classe B quebrar, os testes de A também quebrarão juntos.

Eu e o Hugo Corbucci, coautor do livro de TDD em Ruby, já demos palestras sobre o assunto. Você pode vê-la em <http://www.aniche.com.br/>

[2014/06/mockar-ou-nao-mockar/](#).

## CAPÍTULO 4

# Testes de Integração

Até esse ponto, todas as classes de negócio foram testadas isoladamente, com testes de unidade. Algumas delas, inclusive, eram mais complicadas, dependiam de outras classes, e nesses casos fizemos uso de **Mock Objects**. Mocks são muito importantes quando queremos testar a classe isolada do “resto”, ou seja, das outras classes de que ela depende e faz uso. Mas a pergunta que fica é: será que vale a pena mockar as dependências de uma classe no momento de testá-la?

Veja um DAO, por exemplo. Um DAO é uma classe que esconde toda a complexidade de se comunicar com o banco de dados. É ela que contém os comandos SQLs que explicarão ao banco o que fazer com o conjunto de dados que está lá. Um DAO depende de um sistema externo: o banco de dados.

Veja um exemplo de DAO a seguir, que salva e busca por usuários do sistema. Repare que ele usa Hibernate para acessar o banco de dados. O Hibernate é uma ferramenta que facilita o acesso a banco de dados. Se você

não o conhece, não tem problema; não precisará entender seus detalhes para escrever o teste.

```
public class UsuarioDao {

    private final Session session;

    public UsuarioDao(Session session) {
        this.session = session;
    }

    public Usuario porId(int id) {
        return (Usuario) session.load(Usuario.class, id);
    }

    public Usuario porNomeEEEmail(String nome, String email) {
        return (Usuario) session.createQuery(
            "from Usuario u where u.nome = :nome and x.email = :email"
            .setParameter("nome", nome)
            .setParameter("email", email)
            .uniqueResult();
    }

    public void salvar(Usuario usuario) {
        session.save(usuario);
    }
}
```

## 4.1 DEVEMOS MOCKAR UM DAO?

Será que faz sentido testar nosso DAO e “mockar o banco de dados”? Vamos tentar testar o método `porNomeEEEmail()`, que busca um usuário pelo nome e e-mail. Usaremos o JUnit, framework com que já estamos acostumados.

Como todo teste, ele tem cenário, ação e validação. O cenário será mockado; faremos com que a `Session` retorne um usuário. A ação será invocar o método `porNomeEEEmail()`. A validação será garantir que o método retorne um `Usuario` com os dados corretos.

Para isso, precisamos instanciar um `UsuarioDao`. Repare que essa

classe depende de uma `Session` do Hibernate. A `Session` é análoga ao `Connection`, ou seja, é a forma de falar com o banco de dados. Todo sistema geralmente tem sua forma de conseguir uma conexão com o banco de dados; o nosso não é diferente.

Conforme visto no curso anterior, vamos mockar a `Session` do Hibernate. No caso, mockaremos a classe `Session` e a `Query`:

```
@Test
public void deveEncontrarPeloNomeEEEmailMockado() {
    Session session = Mockito.mock(Session.class);
    Query query = Mockito.mock(Query.class);
    UsuarioDao usuarioDao = new UsuarioDao(session);
}
```

Em seguida, vamos setar o comportamento desses mocks para que funcionem de acordo. Precisaremos simular os métodos `createQuery()`, `setParameter()` e `thenReturn()` (que são os métodos usados pelo DAO):

```
@Test
public void deveEncontrarPeloNomeEEEmailMockado() {
    Session session = Mockito.mock(Session.class);
    Query query = Mockito.mock(Query.class);
    UsuarioDao usuarioDao = new UsuarioDao(session);

    Usuario usuario = new Usuario
        ("João da Silva", "joao@dasilva.com.br");
    String sql = "from Usuario u where u.nome = :nome
        and x.email = :email";

    Mockito.when(session.createQuery(sql))
        .thenReturn(query);
    Mockito.when(query.uniqueResult())
        .thenReturn(usuario);
    Mockito.when(query.setParameter("nome", "João da Silva"))
        .thenReturn(query);
    Mockito.when(query.setParameter("email", "joao@dasilva.com.br"))
        .thenReturn(query);
}
```



Por fim, vamos invocar o método que queremos testar, e validar a saída:

```
@Test
public void deveEncontrarPeloNomeEEEmailMockado() {
    Session session = Mockito.mock(Session.class);
    Query query = Mockito.mock(Query.class);
    UsuarioDao usuarioDao = new UsuarioDao(session);

    Usuario usuario = new Usuario
        ("João da Silva", "joao@dasilva.com.br");
    String sql = "from Usuario u where u.nome = :nome
        and x.email = :email";

    Mockito.when(session.createQuery(sql))
        .thenReturn(query);
    Mockito.when(query.uniqueResult())
        .thenReturn(usuario);
    Mockito.when(query.setParameter("nome", "João da Silva"))
        .thenReturn(query);
    Mockito.when(query.setParameter("email", "joao@dasilva.com.br"))
        .thenReturn(query);

    Usuario usuarioDoBanco = usuarioDao
        .porNomeEEEmail("João da Silva", "joao@dasilva.com.br");

    assertEquals(usuario.getNome(),
        usuarioDoBanco.getNome());
    assertEquals(usuario.getEmail(),
        usuarioDoBanco.getEmail());
}
```

Excelente. Se rodarmos o teste, ele passa! Isso quer dizer que conseguimos simular o banco de dados e facilitar a escrita do teste, certo? Errado!

Olhe a consulta SQL com mais atenção: `from Usuario u where u.nome = :nome and x.email = :email`. Veja que o `x.email` está errado! Deveria ser `u.email`. Isso seria facilmente descoberto se não estivéssemos simulando o banco de dados, mas sim usando um banco de dados real! A SQL seria imediatamente recusada!

## 4.2 TESTANDO DAOs

A resposta da primeira pergunta, portanto, é **NÃO**. Se o único objetivo do DAO é falar com o banco de dados, não faz sentido simular justamente o serviço externo com que ele se comunica. Nesse caso, precisamos testar a comunicação do nosso DAO com um banco de dados de verdade; queremos garantir que nossos `INSERTs`, `SELECTs` e `UPDATEs` estão corretos e funcionam da maneira esperada. Se simulássemos um banco de dados, não saberíamos ao certo se, na prática, ele funcionaria com nossas SQLs!

Escrever um teste para um DAO é parecido com escrever qualquer outro teste:

- 1) Precisamos montar um cenário;
- 2) Executar uma ação e
- 3) Validar o resultado esperado.

Vamos testar novamente o método `porNomeEEEmail()`, mas dessa vez batendo em um banco de dados real. Como exemplo, podemos usar o usuário “João da Silva”, com o e-mail “joao@dasilva.com.br”. Vamos já corrigir o método do DAO e fazer `u.email`, que é o certo:

```
public Usuario porNomeEEEmail(String nome, String email) {
    return (Usuario) session.createQuery(
        "from Usuario u where u.nome = :nome and u.email = :email")
        .setParameter("nome", nome)
        .setParameter("email", email)
        .uniqueResult();
}
```

Vamos ao teste. Começaremos por invocar esse método do DAO:

```
@Test
public void deveEncontrarPeloNomeEEEmail() {
    Usuario usuario = usuarioDao
        .porNomeEEEmail("João da Silva", "joao@dasilva.com.br");
}
```

Mas, para criar o DAO, precisamos passar uma `Session` do Hibernate; e dessa vez não vamos mockar. A classe `CriadorDeSessao` cria a `Session`. Devemos passá-la para o DAO. No teste:

```
@Test
public void deveEncontrarPeloNomeEEEmail() {
    Session session = new CriadorDeSessao().getSession();
    UsuarioDao usuarioDao = new UsuarioDao(session);

    Usuario usuario = usuarioDao.porNomeEEEmail(
        "João da Silva", "joao@dasilva.com.br");
}
```

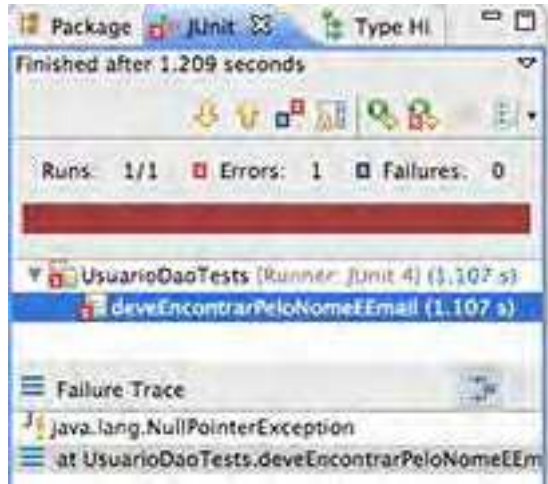
Ótimo. Se tudo deu certo, espera-se que a instância `usuario` contenha o nome e e-mail passados. Vamos escrever os `asserts`:

```
@Test
public void deveEncontrarPeloNomeEEEmail() {
    Session session = new CriadorDeSessao().getSession();
    UsuarioDao usuarioDao = new UsuarioDao(session);

    Usuario usuario = usuarioDao
        .porNomeEEEmail("João da Silva", "joao@dasilva.com.br");

    assertEquals("João da Silva", usuario.getNome());
    assertEquals("joao@dasilva.com.br", usuario.getEmail());
}
```

O teste está pronto, mas se o rodarmos, ele falhará.



O motivo é simples: para que o teste passe, o usuário “João da Silva” deve existir no banco de dados! Precisamos salvá-lo no banco antes de invocar o método `porNomeEEEmail`. Essa é a principal diferença entre testes de unidade e testes de integração: precisamos montar o cenário, executar a ação e validar o resultado esperado no software externo.

Para salvar o usuário, basta invocarmos o método `salvar()` do próprio DAO. Veja o código a seguir, onde criamos um usuário e o salvamos. Não podemos também esquecer de fechar a sessão com o banco de dados (afinal, sempre que consumimos um recurso externo, precisamos fechá-lo!):

```
@Test
public void deveEncontrarPeloNomeEEEmail() {
    Session session = new CriadorDeSessao().getSession();
    UsuarioDao usuarioDao = new UsuarioDao(session);

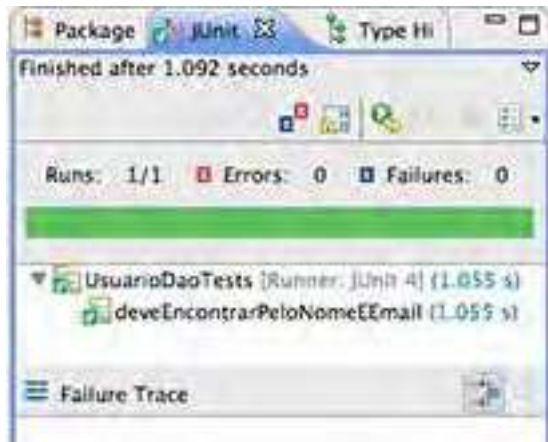
    // criando um usuário e salvando antes
    // de invocar o porNomeEEEmail
    Usuario novoUsuario = new Usuario
        ("João da Silva", "joao@dasilva.com.br");
    usuarioDao.salvar(novoUsuario);
}
```

```
// agora buscamos no banco
Usuario usuarioDoBanco = usuarioDao
    .porNomeEEEmail("João da Silva", "joao@dasilva.com.br");

assertEquals("João da Silva", usuarioDoBanco.getNome());
assertEquals("joao@dasilva.com.br", usuarioDoBanco.getEmail());

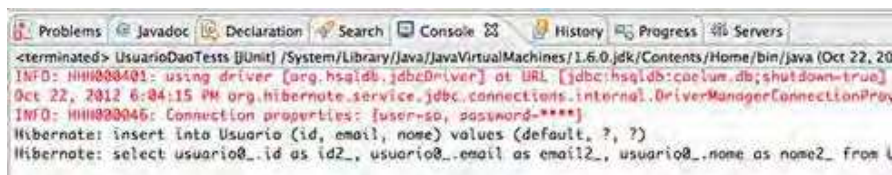
session.close();
}
```

Agora sim, o teste passa!



Veja que escrever um teste para um DAO não é tão diferente; é só mais trabalhoso, afinal precisamos nos comunicar com o software externo o tempo todo, para montar cenário, para validar se a operação foi efetuada com sucesso etc. Em nosso caso, criamos uma `Session` (uma conexão com o banco), inserimos um usuário no banco (um `INSERT`, da SQL), e depois uma busca (um `SELECT`).

Isso pode inclusive ser visto pelo log do Hibernate, no console do Eclipse:



```
<terminated> UsuarioDaoTests [JUnit] /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home/bin/java (Oct 22, 2012 6:04:15 PM)
INFO: HH000001: using driver [org.hsqldb.jdbcDriver] at URL [jdbc:hsqldb:caeium.db;shutdown=true]
Oct 22, 2012 6:04:15 PM org.hibernate.service.jdbc.connections.internal.DriverManagerConnectionProvider
INFO: HH000004: Connection properties: {user=sa, password=****}
Hibernate: insert into Usuario (id, email, nome) values (default, ?, ?)
Hibernate: select usuario0_.id as id2_, usuario0_.email as email2_, usuario0_.nome as nome2_ from
```

Chamamos esses testes de **testes de integração**, afinal estamos testando o comportamento da nossa classe integrada com um serviço externo real. Testes como esse são úteis para classes como nossos DAOs, cuja tarefa é justamente se comunicar com outro serviço (no caso, o banco de dados).

Muitas vezes ficamos na dúvida se devemos mockar ou fazer um teste de integração real. No último exemplo, mostrei que se mockássemos a sessão com o banco de dados teríamos um teste inútil, que não nos daria o feedback ideal. Lembre-se: se a tarefa da classe é comunicar com um outro serviço, a única maneira de garantir que ela funciona é fazendo-a comunicar-se de verdade com ele. E você precisa achar uma maneira de fazer isso acontecer. Os desafios são vários e totalmente contextuais: montar o cenário, fazer a validação etc. Mas quem disse que testar era tarefa fácil?

### DIFERENÇA ENTRE TESTES DE UNIDADE E INTEGRAÇÃO

Um teste de unidade isola a classe de suas dependências e a testa independente delas. Testes de unidade fazem sentido quando nossas classes contêm regras de negócio, mas dependem de infraestrutura. Nesses casos, fica fácil isolar a infraestrutura.

Já testes de integração testam a classe de maneira integrada ao serviço que usam. Um teste de DAO, por exemplo, que bate em um banco de dados de verdade, é considerado um teste de integração. Testes como esses são especialmente úteis para testar classes cuja responsabilidade é se comunicar com outros serviços.

Ao usar mocks, estamos “enganando” nosso teste. Um bom teste de DAO é aquele que garante que sua consulta SQL realmente funciona quando enviada para o banco de dados; e a melhor maneira de garantir isso é enviando-a para o banco! Repare que, quando usamos mock objects, nosso teste passou, mesmo estando com bug! Testes como esses não servem de nada, apenas atrapalham.

Sempre que tiver classes cuja responsabilidade é falar com outro sistema, teste-a realmente integrando com esse outro sistema.

## 4.3 TESTANDO CENÁRIOS MAIS COMPLEXOS

Vamos agora começar a testar nosso `LeilaoDao`. Um dos métodos desse DAO retorna a quantidade de leilões que ainda não foram encerrados. Veja:

```
public Long total() {  
    return (Long) session.createQuery("select count(l) from "+  
                                     "Leilao l where l.encerrado = false")  
}
```

Ele faz um simples `SELECT COUNT`. Para testar essa consulta, adicionaremos dois leilões: um encerrado e outro não encerrado. Dado esse cenário, esperamos que o método `total()` nos retorne 1. Vamos ao teste.

Repare que, para criar um Leilão, precisaremos criar um Usuário e persisti-lo no banco também, afinal o Leilão referencia um Usuário; para fazer

isso, utilizaremos o `UsuarioDao`, que sabe persistir um `Usuario`!

Essa é uma das dificuldades de se escrever um teste de integração: montar cenário é mais difícil. Dê uma olhada no código a seguir, ele é extenso, mas está comentado:

```
public class LeilaoDaoTests {
    private Session session;
    private LeilaoDao leilaoDao;
    private UsuarioDao usuarioDao;

    @Before
    public void antes() {
        session = new CriadorDeSessao().getSession();
        leilaoDao = new LeilaoDao(session);
        usuarioDao = new UsuarioDao(session);
    }

    @After
    public void depois() {
        session.close();
    }

    @Test
    public void deveContarLeiloesNaoEncerrados() {
        // criamos um usuário
        Usuario mauricio =
            new Usuario("Mauricio Aniche", "mauricio@aniche.com.br");

        // criamos os dois leilões
        Leilao ativo =
            new Leilao("Geladeira", 1500.0, mauricio, false);
        Leilao encerrado =
            new Leilao("XBox", 700.0, mauricio, false);
        encerrado.encerra();

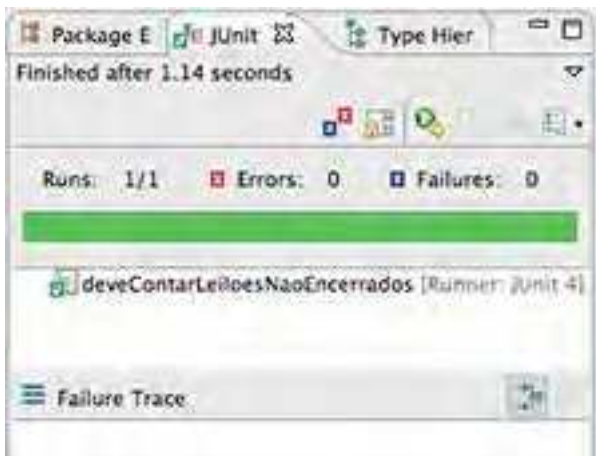
        // persistimos todos no banco
        usuarioDao.salvar(mauricio);
        leilaoDao.salvar(ativo);
        leilaoDao.salvar(encerrado);
    }
}
```



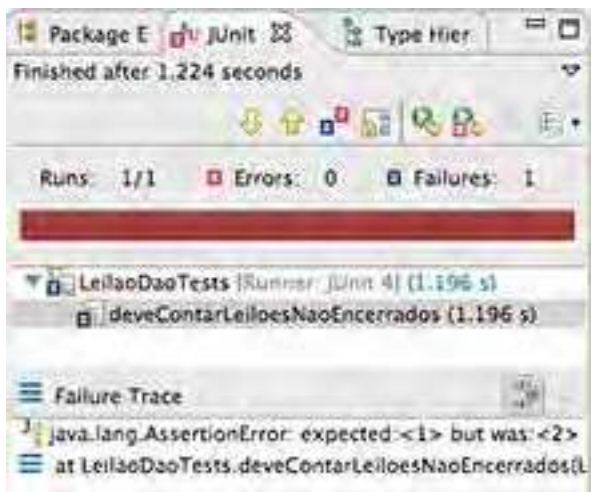
```
// invocamos a ação que queremos testar
// pedimos o total para o DAO
long total = leilaoDao.total();

assertEquals(1L, total);
}
}
```

Se rodarmos o teste, ele passa!



Mas esse teste ainda não está legal. Nesse momento, ele passa porque estamos rodando-o usando o banco de dados HSQLDB. Se estivéssemos rodando em um MySQL, por exemplo, esse teste poderia falhar. Cada vez que rodamos o teste, ele insere 2 linhas no banco de dados. Se rodarmos o teste 2 vezes, por exemplo, teremos 2 leilões não encerrados, o que faz com que o teste falhe:



A melhor maneira de garantir que, independente do banco em que você esteja rodando o teste, o cenário esteja sempre limpo para aquele teste. É ter a base de dados limpa. Um jeito simples de fazer isso é executar cada um dos testes dentro de um contexto de transação e, ao final do teste, fazer um “rollback”. Com isso, o banco rejeitará tudo o que aconteceu no teste e continuará limpo.

Isso é fácil de ser implementado. Basta mexermos nos métodos `@Before` e `@After`:

```
@Before
public void antes() {
    session = new CriadorDeSessao().getSession();
    leilaoDao = new LeilaoDao(session);
    usuarioDao = new UsuarioDao(session);

    // inicia transação
    session.beginTransaction();
}

@After
```

```
public void depois() {  
    // faz o rollback  
    session.getTransaction().rollback();  
    session.close();  
}
```

Pronto. Agora, mesmo no MySQL, esse teste passaria. Iniciar e dar rollback na transação durante testes de integração é prática comum. Faça uso do `@Before` e `@After` para isso, e dessa forma, seus testes ficam independentes e fáceis de manter.

### USAR OU NÃO USAR HSQLDB?

Os desenvolvedores se dividem muito entre usar HSQLDB ou o banco de produção nos testes. A vantagem do HSQLDB é clara: o teste roda muito mais rápido, afinal ele é um banco de dados em memória, muito mais leve.

No entanto, sou mais favorável a testes de integração que realmente façam uso do mesmo banco que a aplicação que será usada em produção. Apesar dos testes ficarem mais lentos, o feedback é maior. Na prática, sabemos que algumas consultas SQL são dependentes de banco, e que, em um caso extremo, o resultado pode ser diferente.

Portanto, se estou pagando o custo de fazer um teste de integração, prefiro que ele seja o mais real possível.

## 4.4 PRATICANDO COM CONSULTAS MAIS COMPLICADAS

Algumas consultas são mais difíceis de serem testadas, simplesmente porque seus cenários são mais complicados. Nesses casos, precisamos facilitar a criação de cenários.

Veja, por exemplo, o método `porPeriodo(Calendar inicio, Calendar fim)`, do nosso `LeilaoDao`. Ele devolve todos os leilões que foram criados dentro de um período e que ainda não foram encerrados:

```
public List<Leilao> porPeriodo(Calendar inicio, Calendar fim) {  
    return session.createQuery("from Leilao l where l.dataAbertura " +  
        "between :inicio and :fim and l.encerrado = false")  
        .setParameter("inicio", inicio)  
        .setParameter("fim", fim)  
        .list();  
}
```

Para testarmos esse método, precisamos pensar em alguns cenários, como:

- Leilões não encerrados com data dentro do intervalo devem aparecer;
- Leilões encerrados com data dentro do intervalo não devem aparecer;
- Leilões encerrados com data fora do intervalo não devem aparecer;
- Leilões não encerrados com data fora do intervalo não devem aparecer.

Vamos começar pelo primeiro cenário. Criaremos 2 leilões não encerrados, um com data dentro do intervalo, outro com data fora do intervalo, e vamos garantir que só o primeiro estará lá dentro. O método é grande, mas está comentado:

```
@Test  
public void deveTrazerLeiloesNaoEncerradosNoPeriodo() {  
  
    // criando as datas  
    Calendar comecoDoIntervalo = Calendar.getInstance();  
    comecoDoIntervalo.add(Calendar.DAY_OF_MONTH, -10);  
    Calendar fimDoIntervalo = Calendar.getInstance();  
    Calendar dataDoLeilao1 = Calendar.getInstance();  
    dataDoLeilao1.add(Calendar.DAY_OF_MONTH, -2);  
    Calendar dataDoLeilao2 = Calendar.getInstance();  
    dataDoLeilao2.add(Calendar.DAY_OF_MONTH, -20);  
  
    Usuario mauricio = new Usuario("Mauricio Aniche",  
        "mauricio@aniche.com.br");
```

```
// criando os leilões, cada um com uma data
Leilao leilao1 =
    new Leilao("XBox", 700.0, mauricio, false);
leilao1.setDataAbertura(dataDoLeilao1);
Leilao leilao2 =
    new Leilao("Geladeira", 1700.0, mauricio, false);
leilao2.setDataAbertura(dataDoLeilao2);

// persistindo os objetos no banco
usuarioDao.salvar(mauricio);
leilaoDao.salvar(leilao1);
leilaoDao.salvar(leilao2);

// invocando o método para testar
List<Leilao> leiloes =
    leilaoDao.porPeriodo(comecoDoIntervalo, fimDoIntervalo);

// garantindo que a query funcionou
assertEquals(1, leiloes.size());
assertEquals("XBox", leiloes.get(0).getNome());
}
```

Ele passa. Vamos ao próximo cenário: leilões encerrados devem ser ignorados pela consulta. Nesse caso, criaremos apenas um leilão encerrado, dentro do intervalo. Esperaremos que a query não devolva nada:

```
@Test
public void naoDeveTrazerLeiloesEncerradosNoPeriodo() {

    // criando as datas
    Calendar comecoDoIntervalo = Calendar.getInstance();
    comecoDoIntervalo.add(Calendar.DAY_OF_MONTH, -10);
    Calendar fimDoIntervalo = Calendar.getInstance();
    Calendar dataDoLeilao1 = Calendar.getInstance();
    dataDoLeilao1.add(Calendar.DAY_OF_MONTH, -2);

    Usuario mauricio = new Usuario("Mauricio Aniche",
        "mauricio@aniche.com.br");
```

```
// criando os leilões, cada um com uma data
Leilao leilao1 =
    new Leilao("XBox", 700.0, mauricio, false);
leilao1.setDataAbertura(dataDoLeilao1);
leilao1.encerra();

// persistindo os objetos no banco
usuarioDao.salvar(mauricio);
leilaoDao.salvar(leilao1);

// invocando o método para testar
List<Leilao> leiloes =
    leilaoDao.porPeriodo(comecoDoIntervalo, fimDoIntervalo);

// garantindo que a query funcionou
assertEquals(0, leiloes.size());
}
```

Montar cenários é o grande segredo de um teste de integração. Queries mais complexas exigirão cenários de teste mais complexos. Nos capítulos anteriores, estudamos sobre como melhorar a escrita dos testes, como `Test Data Builders` etc. Você pode (e deve) fazer uso deles para facilitar a escrita dos cenários!

Geralmente, o grande desafio é justamente montar o cenário e validar o resultado esperado no sistema externo. No caso de banco de dados, precisamos fazer `INSERTs`, `DELETEs`, e `SELECTs` para validar. Além disso, ainda precisamos manter o estado do sistema consistente, ou seja, devemos limpar o banco de dados constantemente para que um teste não atrapalhe o outro.

Veja que usamos o JUnit da mesma forma. A diferença é que justamente precisamos nos comunicar com o outro sistema.

### COMO LIMPAR O BANCO DE DADOS?

Entre um teste e outro, precisamos que o banco de dados esteja limpo; afinal, não queremos que os dados de um teste influenciem/ atrapalhem o outro. Por isso, algo que geralmente não é opcional é limpar o banco de dados para que o próximo teste consiga executar sem problemas.

Você pode fazer isso de diversas maneiras diferentes, por exemplo, dando rollback no banco de dados ao final de cada teste, ou mesmo executando uma sequência de `TRUNCATE TABLEs`, que limpam todas as tabelas. Não há uma melhor maneira. Você pode escolher a que lhe agrada mais. Depois é fácil: basta colocar esse código no `@Before` das suas classes de teste.

### SCRIPTS PARA POPULAR O BANCO?

Aqui sempre populamos nosso banco via código Java. Mas outra possibilidade é fazer uso de scripts de importação. Scripts são arquivos `.SQL` que já contêm um conjunto de `INSERTs`, e `CREATE TABLEs` e tudo necessário para colocar o banco de dados em algum estado para o teste.

Em particular, também não gosto dessa abordagem, pois manter esses scripts não é fácil. No momento em que você acrescentar coluna em uma tabela qualquer, você precisará regerar esses scripts (afinal, sair colocando uma coluna a mais em todas instruções SQL é trabalho muito braçal e sujeito a erros).

Portanto, gosto de criar cenários via código Java, pois eles, sim, são fáceis de manter e evoluir. Adicionar ou remover novos atributos é muito mais fácil.

## 4.5 TESTANDO ALTERAÇÃO E DELEÇÃO

Até agora testamos operações de inserção e seleção. Chegou a hora de testar remoção e atualização. Veja o método `deleta()` no `UsuarioDao`:

```
public void deletar(Usuario usuario) {  
    session.delete(usuario);  
}
```

Ele deleta o usuário que é passado. Para testar uma deleção, precisamos primeiro inserir um elemento, deletá-lo, e depois fazer uma consulta para garantir que ele não está lá. Vamos aos testes:

```
@Test  
public void deveDeletarUmUsuario() {  
    Usuario usuario =  
        new Usuario("Mauricio Aniche", "mauricio@aniche.com.br");  
  
    usuarioDao.salvar(usuario);  
    usuarioDao.deletar(usuario);  
  
    Usuario usuarioNoBanco =  
        usuarioDao.porNomeEEEmail("Mauricio Aniche", "mauricio@aniche.  
  
    assertNull(usuarioNoBanco);  
}
```

Excelente, o teste está da maneira que pretendíamos. Mas se rodarmos, ele falha! A pergunta é: por quê?

Muitas vezes nossa camada de acesso a dados não envia as consultas SQL para o banco até que você finalize a transação. É o que está acontecendo aqui. Precisamos forçar o envio do `INSERT` e do `DELETE` para o banco, para que depois o `SELECT` não traga o objeto! Essa operação é chamada de **flush**:

```
@Test  
public void deveDeletarUmUsuario() {  
    Usuario usuario =  
        new Usuario(  
            "Mauricio Aniche",  
            "mauricio@aniche.com.br");  
  
    usuarioDao.salvar(usuario);  
    usuarioDao.deletar(usuario);
```



```
// envia tudo para o banco de dados
session.flush();

Usuario usuarioNoBanco =
    usuarioDao.porNomeEEEmail(
        "Mauricio Aniche",
        "mauricio@aniche.com.br");

assertNull(usuarioNoBanco);
}
```

Agora sim nosso teste passa! É muito comum fazer uso de `flush` sempre que fazemos testes com banco de dados. Dessa forma, garantimos que a consulta realmente chegou ao banco de dados, e que as futuras consultas levarão mesmo em consideração as consultas anteriores.

### DEVEMOS TESTAR ATÉ MÉTODOS SIMPLES?

Tudo é questão de feedback. Testes para o método de alteração, por exemplo, podem fazer sentido quando o processo de alteração é complicado. Atualizar um usuário é bem simples e feito pelo próprio Hibernate. Não há muito como dar errado.

Agora, atualizar um leilão pode ser mais trabalhoso, afinal precisamos atualizar lances juntos. Nesses casos, testar uma alteração pode ser bem importante.

## 4.6 ORGANIZANDO TESTES DE INTEGRAÇÃO

Nossa bateria de testes de integração crescerá, mas já conseguimos reparar em alguns padrões nela: em todo começo de teste, abrimos uma `Session`, e no fim a fechamos. Em nosso primeiro curso, aprendemos que sempre que algo ocorre no começo e fim de todo o teste, a boa prática é não repetir código, mas sim fazer uso de métodos anotados com `@Before` e `@After`. Você

se lembra? Esses métodos são executados respectivamente antes e depois de todos os testes.

Vamos lá. Criaremos dois atributos na classe, uma `Session` e um `UsuarioDao` e faremos o método com `@Before` instanciar esses objetos. No método com `@After`, fecharemos a sessão. Com isso, os métodos de testes ficarão mais enxutos:

```
public class UsuarioDaoTests {

    private Session session;
    private UsuarioDao usuarioDao;

    @Before
    public void antes() {
        // criamos a sessão e a passamos para o dao
        session = new CriadorDeSessao().getSession();
        usuarioDao = new UsuarioDao(session);
    }

    @After
    public void depois() {
        // fechamos a sessão
        session.close();
    }

    @Test
    public void deveEncontrarPeloNomeEEEmail() {
        Usuario novoUsuario = new Usuario
            ("João da Silva", "joao@dasilva.com.br");
        usuarioDao.salvar(novoUsuario);

        Usuario usuarioDoBanco = usuarioDao
            .porNomeEEEmail("João da Silva", "joao@dasilva.com.br");

        assertEquals("João da Silva",
            usuarioDoBanco.getNome());
        assertEquals("joao@dasilva.com.br",
            usuarioDoBanco.getEmail());
    }
}
```

```
@Test
public void deveRetornarNuloSeNaoEncontrarUsuario() {
    Usuario usuarioDoBanco = usuarioDao
        .porNomeEEEmail("João Joaquim", "joao@joaquim.com.br");

    assertNull(usuarioDoBanco);
}
```

Excelente. Muito melhor, e os testes continuam passando. Lembre-se da discussão que permeia o livro inteiro sobre qualidade do código de testes.

## CAPÍTULO 5

# Testes de Sistema

Estamos muito acostumados a testar nossas aplicações de maneira manual. Empresas geralmente possuem imensos roteiros de script, e fazem com que seus analistas de teste executem esses scripts incansavelmente. Mas quais os problemas dessa abordagem?

Imagine agora um sistema web que toma conta de leilões, domínio com que trabalhamos até agora. Nele, o usuário da empresa pode adicionar novos usuários, cadastrar leilões e efetuar lances. Essa aplicação foi desenvolvida em Java e está pronta para ser executada. A imagem a seguir mostra a tela inicial do sistema:



A aplicação não é muito grande: possui cadastro de usuários, leilões e lances. Mas, apesar da pequena quantidade de funcionalidades, pense na quantidade de cenários que você precisa testar para garantir seu funcionamento:

- Cadastrar um usuário com sucesso;
- Editar um usuário;
- Validar o cadastro de usuário com nome ou e-mail inválido;
- Excluir um usuário;
- Exibir ficha completa de um usuário;
- ...

Se pensarmos em todos os cenários que devemos testar, percebemos que teremos uma quantidade enorme! Quanto tempo uma pessoa leva para executar todos esses cenários? Agora imagine o mesmo problema em uma aplicação grande. Testar aplicações grandes de maneira manual **leva muito tempo!** E, por consequência, **custa muito caro.**

Na prática, o que acontece é que, como testar sai caro, as empresas optam por não testar! No fim, entregamos software com defeito para nosso cliente! Precisamos mudar isso. Se removermos a parte humana do processo e fizermos com que a máquina execute o teste, resolvemos o problema: **a máquina vai executar o teste rápido, repetidas vezes, e de graça!**

A grande questão é: como ensinar a máquina a fazer o trabalho de um ser humano? Como fazê-la abrir o browser, digitar valores nos campos, preencher formulários, clicar em links etc.?

Para isso, faremos uso do **Selenium**. O Selenium é um framework que facilita e muito a vida do desenvolvedor que quer escrever um teste automatizado. A primeira coisa que uma pessoa faria para testar a aplicação seria abrir o browser. Com Selenium, precisamos apenas da linha a seguir:

```
WebDriver driver = new FirefoxDriver();
```

Nesse caso, estamos abrindo o Firefox! Em seguida, entraríamos em algum site. Vamos entrar no Google, por exemplo, usando o método `get()`:

```
driver.get("http://www.google.com.br/");
```

Para buscar o termo “Caelum”, precisamos digitar no campo de texto. No caso do Google, o nome do campo é “q”.



Para descobrir, podemos fazer uso do Inspector, no Chrome (ou do Firebug no Firefox). Basta apertar `Ctrl + Shift + I` (ou `F12`), e a janela abrirá. Nela, selecionamos a lupa e clicamos no campo cujo nome queremos descobrir. Ele nos levará para o HTML, onde podemos ver `name="q"`.



Com Selenium, basta dizermos o nome do campo de texto, e enviarmos o texto:

```
WebElement campoDeTexto = driver.findElement(By.name("q"));
campoDeTexto.sendKeys("Caelum");
```

Agora, basta submetermos o form! Podemos fazer isso pelo próprio campoDeTexto:

```
campoDeTexto.submit();
```

Pronto! Juntando todo esse código em uma classe Java simples, temos:

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;

public class TesteAutomatizado {

    public static void main(String[] args) {
        // abre firefox
        WebDriver driver = new FirefoxDriver();

        // acessa o site do google
```

```
driver.get("http://www.google.com.br/");

// digita no campo com nome "q" do google
WebElement campoDeTexto =
    driver.findElement(By.name("q"));

campoDeTexto.sendKeys("Caelum");

// submete o form
campoDeTexto.submit();

}
}
```

Se você não entendeu algum método invocado, não se preocupe. Vamos estudá-los com mais detalhes nos próximos capítulos. Nesse momento, rode a classe! Acabamos de fazer uma busca no Google de maneira automatizada!

Execute o teste novamente. Veja agora como é fácil, rápido e barato! Qual a vantagem? Podemos executá-los a tempo! Ou seja, a cada mudança que fazemos em nosso software, podemos testá-lo por completo, clicando apenas em um botão. Saberemos em poucos minutos se nossa aplicação continua funcionando!

Quantas vezes não entregamos software sem tê-lo testado por completo?

## 5.1 AUTOMATIZANDO O PRIMEIRO TESTE DE SISTEMA

Imagine uma tela de cadastro padrão, onde o usuário deve preencher um formulário qualquer. E que, ao clicar em “Salvar”, o sistema devolve o usuário para a listagem com o novo usuário cadastrado:





A funcionalidade funciona atualmente, mas nossa experiência nos diz que futuras alterações no sistema podem fazer com que a funcionalidade pare. Vamos automatizar um teste para o cadastro de um novo usuário. O cenário é o mesmo que acabamos de testar de maneira manual.

```
public static void main(String[] args) {  
  
}
```

A primeira parte do teste manual é entrar na página de cadastro de usuários. Vamos fazer o Selenium abrir o Firefox nessa página. Suponha que o endereço seja <http://localhost:8080/usuarios/new>:

```
WebDriver driver = new FirefoxDriver();  
driver.get("http://localhost:8080/usuarios/new");
```

Nessa página, precisamos cadastrar algum usuário, por exemplo, “Ronaldo Luiz de Albuquerque” com o e-mail “ronaldo2009@terra.com.br”. Para preencher esses valores de maneira automatizada, precisamos saber o nome dos campos de texto para que o Selenium saiba aonde colocar essa informação!

Aperte CTRL + U (no Firefox e Chrome) ou Ctrl + F12 (no Internet Explorer) para exibir o código-fonte da página. Veja que o nome dos campos de texto são “usuario.nome” e “usuario.email”. Com essa informação em mãos, precisamos encontrar esses elementos na página e preencher com os valores que queremos:

```
// encontrando ambos elementos na pagina  
WebElement nome = driver.findElement(By.name("usuario.nome"));  
WebElement email = driver.findElement(By.name("usuario.email"));  
  
// digitando em cada um deles  
nome.sendKeys("Ronaldo Luiz de Albuquerque");  
email.sendKeys("ronaldo2009@terra.com.br");
```

Veja o código. Para encontrarmos um elemento, utilizamos o método `driver.findElement`. Como existem muitas maneiras para encontrar um elemento na página (pelo id, nome, classe CSS etc.), o Selenium nos provê uma classe chamada `By` que tem um conjunto de métodos que nos ajudam a achar o elemento. Nesse caso, como queremos encontrar o elemento pelo seu nome, usamos `By.name("nome-aqui")`.

Tudo preenchido! Precisamos submeter o formulário! Podemos fazer isso de duas maneiras. A primeira delas é clicando no botão que temos na página. Ao olhar o código-fonte da página novamente, é possível perceber que o id do botão de Salvar é `btnSalvar`. Basta pegarmos esse elemento e clicar nele:

```
WebElement botaoSalvar = driver.findElement(By.id("btnSalvar"));  
botaoSalvar.click();
```

Uma outra alternativa mais simples ainda é “submeter” qualquer uma das

caixas de texto! O Selenium automaticamente procurará o *form* na qual a caixa de texto está contida e o submeterá! Ou seja:

```
nome.submit();  
// email.submit(); daria no mesmo!
```

Se tudo der certo, voltamos à listagem de usuários. Mas, desta vez, esperamos que o usuário Ronaldo esteja lá. Para terminar nosso teste, precisamos garantir de maneira automática que o usuário adicionado está lá. Para fazer esses tipos de verificação, utilizaremos um framework muito conhecido do mundo Java, que é o JUnit. O JUnit nos provê um conjunto de instruções para fazer essas comparações e ainda conta com um plugin que nos diz se os testes estão passando ou, caso contrário, quais testes estão falhando!

Para configurar o Eclipse no nosso projeto, clique com o botão direito do mouse sobre o projeto, e vá em `Build Path -> Add Libraries`. Adicione JUnit em sua versão 4.

Para garantir o usuário na listagem, precisamos procurar pelos textos “Ronaldo Luiz de Albuquerque” e “ronaldo2009@terra.com.br” na página atual. O Selenium nos dá o código-fonte HTML inteiro da página atual, através do método `driver.getPageSource()`. Basta verificarmos se existem o nome e e-mail do usuário lá:

```
boolean achouNome = driver.getPageSource().contains("Ronaldo Luiz de Albuq  
boolean achouEmail = driver.getPageSource().contains("ronaldo2009@terra.com
```

Sabemos que essas duas variáveis devem ser iguais a `true`. Vamos avisar isso ao JUnit através do método `assertTrue()` e, dessa forma, caso essas variáveis fiquem com `false`, o JUnit nos avisará:

```
assertTrue(achouNome);  
assertTrue(achouEmail);
```

Lembre-se que, para o método `assertTrue` funcionar, precisamos fazer o `import` estático do método `import static org.junit.Assert.assertTrue;`

Devemos agora encerrar o Selenium:

```
driver.close();
```

Por fim, para que o JUnit entenda que isso é um método de teste, precisamos mudar sua assinatura. Todo método do JUnit deve ser público, não retornar nada, e deve ser anotado com `@Test`. Veja:

```
@Test
public void deveAdicionarUmUsuario() {
    // ...
}
```

Observe que usamos o nome do método para explicar o que ele testa. Essa é uma boa prática.

Nosso método agora ficou assim:

```
import static org.junit.Assert.assertTrue;

import org.junit.Test;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;

public class UsuariosSystemTest {
    @Test
    public void deveAdicionarUmUsuario() {
        WebDriver driver = new FirefoxDriver();
        driver.get("http://localhost:8080/usuarios/new");

        WebElement nome =
            driver.findElement(By.name("usuario.nome"));
        WebElement email =
            driver.findElement(By.name("usuario.email"));

        nome.sendKeys("Ronaldo Luiz de Albuquerque");
        email.sendKeys("ronaldo2009@terra.com.br");
        nome.submit();

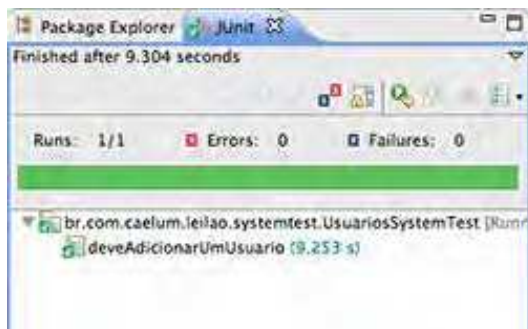
        boolean achouNome = driver.getPageSource()
            .contains("Ronaldo Luiz de Albuquerque");
        boolean achouEmail = driver.getPageSource()
```

```
        .contains("ronaldo2009@terra.com.br");

        assertTrue(achouNome);
        assertTrue(achouEmail);

        driver.close();
    }
}
```

Repare na anotação `@Test` antes do nome do método. Isso é obrigatório caso queiramos fazer uso do JUnit. Vamos agora executar o teste. Para isso, clique com o botão direito do mouse em cima do código-fonte da classe de teste e selecione `Run As -> Junit Test`, e espere o Selenium executar o teste! Ao final, você deve ver uma tela de confirmação do JUnit:



Pronto! Nosso primeiro teste para a aplicação de leilão está escrito.

O que acabamos de fazer é o que chamamos de **teste de sistema**. Ou seja, é um teste que exercita a aplicação do ponto de vista do usuário final, como um todo, sem conhecer seus detalhes internos. Se sua aplicação é web, então o teste de sistema é aquele que navegará pela interface web, interagir com ela, os dados serão persistidos no banco de dados, os serviços web serão consumidos etc.

A vantagem desse tipo de teste é que ele é muito parecido com o teste que o usuário faz. No teste de unidade, garantíamos que uma classe fun-

cionava muito bem. Mas nosso software em produção faz uso de diversas classes juntas. É aí que o teste de sistema entra; ele garante que o sistema funciona quando “tudo está ligado”.

## **5.2 NOVAMENTE, AS VANTAGENS DO TESTE AUTOMATIZADO**

São várias as vantagens de termos testes de sistema automatizado:

- O teste automatizado é muito mais rápido do que um ser humano.
- A partir do momento em que você escreveu o teste, você poderá executá-lo infinitas vezes a um custo baixíssimo.
- Mais produtividade, afinal, você gastará menos tempo testando (escrever um teste automatizado gasta menos tempo do que testar manualmente diversas vezes a mesma funcionalidade) e mais tempo desenvolvendo.
- Bugs encontrados mais rápido pois, já que sua bateria de testes roda rápido, você a executará a todo instante, encontrando possíveis partes do sistema que deixaram de funcionar devido a novas implementações.

O Selenium é uma excelente ferramenta para automatizar os testes. Sua API é bem clara e fácil de usar, além da grande quantidade de documentação que pode ser encontrada na internet.

### TESTES DE SISTEMA OU ACEITAÇÃO?

O termo **teste de aceitação** é bastante comum atualmente, dada a popularidade das metodologias ágeis. A diferença é bem sutil, pois eles tem uma intersecção.

Testes de sistema são testes *end-to-end*, ou seja, testam o seu sistema como uma caixa preta. Já testes de aceitação são usados para dar o “aceite” daquela funcionalidade (ela faz o que o usuário quer?), e a equipe os usa para dizer se terminou a implementação daquela funcionalidade.

Geralmente, testes de aceitação são automatizados usando testes de sistema, já que eles são o que melhor simulam o comportamento do usuário.

## 5.3 BOAS PRÁTICAS: PAGE OBJECTS

Já vimos que o Selenium facilita e muito nossa vida. Com o que já sabemos hoje, podemos escrever muitos métodos de teste e testar diferentes formulários web. Precisamos agora trabalhar para que nosso código de teste não se torne mais complicado do que deveria.

Veja o método de teste a seguir:

```
public class UsuariosSystemTest {

    private WebDriver driver;

    @Before
    public void inicializa() {
        driver = new FirefoxDriver();
    }

    @Test
    public void deveAdicionarUmUsuario() {
        driver.get("http://localhost:8080/usuarios/new");

        WebElement nome =
            driver.findElement(By.name("usuario.nome"));
    }
}
```

```
WebElement email =
    driver.findElement(By.name("usuario.email"));

nome.sendKeys("Ronaldo Luiz de Albuquerque");
email.sendKeys("ronaldo2009@terra.com.br");

nome.submit();

assertTrue(driver.getPageSource()
    .contains("Ronaldo Luiz de Albuquerque"));
assertTrue(driver.getPageSource()
    .contains("ronaldo2009@terra.com.br"));

}

@After
public void encerra() {
    driver.close();
}

}
```

O código de testes é simples de ler. Mas poderia ser melhor e mais fácil. E se conseguíssemos escrever desta forma?

```
@Test
public void deveAdicionarUmUsuario() {

    usuarios.novo()
        .cadastra("Ronaldo Luiz de Albuquerque", "ronaldo2009@terra.com.br");

    assertTrue(usuarios.existeNaListagem(
        "Ronaldo Luiz de Albuquerque", "ronaldo2009@terra.com.br"));

}
```

Veja só como o teste é bem mais legível! É muito mais fácil de ler e entender o que o teste faz. Agora precisamos implementar. Repare que a declaração da variável `usuarios` foi omitida do código. A ideia é que essa variável representa “a página de usuários”. Ela contém as seguintes ações: `novo()` para



ir para a página de novo usuário, e `existeNaListagem()`, que verifica se um usuário está lá.

Vamos escrever uma classe que representa a página de listagem de usuários e contém as operações descritas anteriormente. Para implementá-las, basta fazer uso do Selenium, igual fizemos nos nossos testes até então:

```
class UsuariosPage {

    public void visita() {
        driver.get("localhost:8080/usuarios");
    }

    public void novo() {
        // clica no link de novo usuário
        driver
            .findElement(By.linkText("Novo Usuário"))
            .click();
    }

    public boolean existeNaListagem(String nome, String email) {
        // verifica se ambos existem na listagem
        return driver.getPageSource().contains(nome) &&
            driver.getPageSource().contains(email);
    }
}
```

Ótimo! Veja que escrevemos basicamente o mesmo código que anteriormente, mas dessa vez os escondemos em uma classe específica. Só que esse código ainda não funciona. Precisamos do `driver` do Selenium. Mas, em vez de instanciar um driver dentro da classe, vamos receber esse driver pelo construtor. Dessa forma, ainda conseguimos fazer uso dos métodos `@Before` e `@After` do JUnit para abrir e fechar o driver e, quando tivermos mais classes iguais a essa (para cuidar das outras páginas do nosso sistema), para compartilhar o mesmo driver entre elas:

```
class UsuariosPage {
```

```
private WebDriver driver;

public UsuariosPage(WebDriver driver) {
    this.driver = driver;
}

public void visita() {
    driver.get("localhost:8080/usuarios");
}

public void novo() {
    // clica no link de novo usuário
    driver.findElement(By.linkText("Novo Usuário")).click();
}

public boolean existeNaListagem(String nome, String email) {
    // verifica se ambos existem na listagem
    return driver.getPageSource().contains(nome) &&
        driver.getPageSource().contains(email);
}
}
```

Excelente! Já conseguimos clicar no link de “Novo Usuário” e já conseguimos verificar se o usuário existe na página. Falta fazer agora o preenchimento do formulário. A pergunta é: onde devemos colocar esse comportamento? Na classe `UsuariosPage`? A grande ideia por trás do que estamos tentando fazer é criar uma classe para cada diferente página do nosso sistema! Dessa forma, cada classe ficará pequena, e esconderá todo o código responsável por usar a página. Ou seja, precisamos criar a classe `NovoUsuarioPage`.

Ela será muito parecida com nossa classe anterior. Ela também deverá receber o driver pelo construtor, e expor o método `cadastra()`, que preencherá o formulário e o submeterá:

```
class NovoUsuarioPage {

    private WebDriver driver;
```

```
public NovoUsuarioPage(WebDriver driver) {
    this.driver = driver;
}

public void cadastra(String nome, String email) {
    WebElement txtNome =
        driver.findElement(By.name("usuario.nome"));
    WebElement txtEmail =
        driver.findElement(By.name("usuario.email"));

    txtNome.sendKeys(nome);
    txtEmail.sendKeys(email);

    txtNome.submit();
}
}
```

Ótimo! Precisamos agora chegar nesse `NovoUsuarioPage`. Mas quando isso acontece? Quando clicamos no link “Novo Usuário”. Ou seja, o método `novo()`, depois de clicar no link, precisa retornar um `NovoUsuarioPage`:

```
public NovoUsuarioPage novo() {
    // clica no link de novo usuario
    driver.findElement(By.linkText("Novo Usuário")).click();
    // retorna a classe que representa a nova pagina
    return new NovoUsuarioPage(driver);
}
```

Agora, de volta ao nosso teste, temos o seguinte código:

```
public class UsuariosSystemTest {

    private WebDriver driver;
    private UsuariosPage usuarios;
```

```
@Before
public void inicializa() {
    this.driver = new FirefoxDriver();
    this.usuarios = new UsuariosPage(driver);
}

@Test
public void deveAdicionarUmUsuario() {

    usuarios.visita();
    usuarios.novo()
        .cadastra("Ronaldo Luiz de Albuquerque",
            "ronaldo2009@terra.com.br");

    assertTrue(usuarios.existeNaListagem(
        "Ronaldo Luiz de Albuquerque",
        "ronaldo2009@terra.com.br"));

}

@After
public void encerra() {
    driver.close();
}

}
```

Veja só como nossos testes ficaram mais claros! E o melhor, eles não conhecem a implementação por baixo de cada uma das páginas! Essa implementação está escondida em cada uma das classes específicas!

Sabemos que nosso HTML muda frequentemente. Se tivermos classes que representam as nossas várias páginas do sistema, no momento de uma mudança de HTML, basta que mudemos na classe correta, e os testes não serão afetados! Esse é o poder do encapsulamento, um dos grandes princípios da programação orientada a objetos, bem utilizada em nossos códigos de teste.

A ideia de escondermos a manipulação de cada uma das nossas páginas em classes específicas é inclusive um padrão de projetos. Esse padrão é con-

hecido por **Page Object**. Pense em escrever Page Objects em seus testes. Eles garantirão que seus testes serão de fácil manutenção por muito tempo.

## 5.4 TESTANDO FORMULÁRIOS COMPLEXOS

Vamos continuar testando nossa aplicação; agora é a vez do cadastro de leilão. Veja que essa tela é bem mais complexa: ela contém combos, checkboxes etc. Será que o Selenium consegue lidar com todos esses elementos?

A resposta é: sim! Vamos lá! Vamos escrever nosso Page Object que lida com a tela de Novo Leilão. Para adicionarmos um novo leilão, precisamos passar as seguintes informações: nome, valor inicial, usuário e marcar se o objeto é usado. A imagem a seguir mostra a tela de cadastro:



Para preencher uma caixa de texto, já sabemos como fazer:

```
WebElement nome = driver
    .findElement(By.name("leilao.nome"));
WebElement valor = driver
```

```
.findElement(By.name("leilao.valorInicial"));

nome.sendKeys("Dono do Produto");
valor.sendKeys("123,45");
```

Já o usuário é um *combobox*. Para selecionarmos uma opção no combo, devemos fazer uso da classe `Select`. Com esse objeto em mãos, podemos pedir para que ele selecione uma determinada opção. Por exemplo, se quisermos selecionar o usuário “João” (nome que aparece no combo), fazemos:

```
Select usuario = new Select(driver.findElement(By.name("leilao.usuario.id")));
usuario.selectByVisibleText("João");
```

Com o checkbox, é mais fácil. Basta clicarmos nele:

```
WebElement usado = driver.findElement(By.name("leilao.usado"));
usado.click();
```

Pronto! Agora sabemos como preencher nosso formulário por inteiro. Vamos escrever nosso Page Object:

```
public class NovoLeilaoPage {

    private WebDriver driver;

    public NovoLeilaoPage(WebDriver driver) {
        this.driver = driver;
    }

    public void preenche(String nome, double valor, String usuario, boolean usado) {

        WebElement txtNome =
            driver.findElement(By.name("leilao.nome"));
        WebElement txtValor =
            driver.findElement(By.name("leilao.valorInicial"));

        txtNome.sendKeys(nome);
        txtValor.sendKeys(String.valueOf(valor));

        WebElement combo =
```

```
        driver.findElement(By.name("leilao.usuario.id"));
        Select cbUsuario = new Select(combo);
        cbUsuario.selectByVisibleText(usuario);

        if(usado) {
            WebElement ckUsado =
                driver.findElement(By.name("leilao.usado"));
            ckUsado.click();
        }

        txtNome.submit();
    }
}
```

Excelente. Podemos usar a mesma estratégia do teste anterior: criar um Page Object para representar a listagem de leilões e chegar na tela de Novo Leilão por ela. Vamos também já colocar um método `existe()` para nos dizer se existe um produto nessa listagem:

```
class LeiloesPage {

    private WebDriver driver;

    public LeiloesPage(WebDriver driver) {
        this.driver = driver;
    }

    public void visita() {
        driver.get("http://localhost:8080/leiloes");
    }

    public NovoLeilaoPage novo() {
        // clica no link de novo leilão
        driver.findElement(By.linkText("Novo Leilão")).click();
        // retorna a classe que representa a nova página
        return new NovoLeilaoPage(driver);
    }
}
```

```
public boolean existe(String produto, double valor, String usuario,
    boolean usado) {

    return driver.getPageSource().contains(produto) &&
        driver.getPageSource().contains(String.valueOf(valor)) &&
        driver.getPageSource().contains(usado ? "Sim" : "Não");

}
}
```

Vamos agora ao nosso teste. Devemos cadastrar um leilão:

```
public class LeiloesSystemTest {

    private WebDriver driver;
    private LeiloesPage leiloes;

    @Before
    public void inicializa() {
        this.driver = new FirefoxDriver();
        leiloes = new LeiloesPage(driver);
    }

    @Test
    public void deveCadastrarUmLeilao() {

        leiloes.visita();
        NovoLeilaoPage novoLeilao = leiloes.novo();
        novoLeilao.preenche("Geladeira", 123, "Paulo Henrique", true);

        assertTrue(
            leiloes.existe("Geladeira", 123, "Paulo Henrique", true)
        );

    }
}
```

Mas o problema é: de onde virá esse usuário? Precisamos cadastrar o usuário “Paulo Henrique” antes de cadastrarmos um leilão para ele. Como



faremos isso? Será uma boa ideia já termos alguns dados populados, prontos para os testes?

Uma boa prática de testes é fazer com que a bateria de testes seja inteiramente responsável por montar o cenário necessário para o teste. Dessa forma, cada teste sabe qual cenário precisa montar. Ou seja, vamos fazer com que um usuário seja adicionado antes de adicionarmos um novo leilão. Para isso, usaremos o `LeiloesPage`:

```
public class LeiloesSystemTest {

    private WebDriver driver;
    private LeiloesPage leiloes;

    @Before
    public void inicializa() {
        this.driver = new FirefoxDriver();
        leiloes = new LeiloesPage(driver);

        UsuariosPage usuarios = new UsuariosPage(driver);
        usuarios.visita();
        usuarios.novo().cadastra(
            "Paulo Henrique",
            "paulo@henrique.com");
    }

    @Test
    public void deveCadastrarUmLeilao() {

        leiloes.visita();
        NovoLeilaoPage novoLeilao = leiloes.novo();
        novoLeilao.preenche("Geladeira", 123, "Paulo Henrique", true);

        assertTrue(
            leiloes.existe("Geladeira", 123, "Paulo Henrique", true)
        );
    }
}
```

Pronto. O teste passa. Veja como é fácil testar formulários complexos,

com diferentes tipos de entradas de dados. Além disso, lembre-se sempre de fazer com que seus testes sejam independentes, ou seja, eles devem ser responsáveis por todo o processo, desde a criação do cenário até a validação da saída.

## 5.5 CLASSES DE TESTE PAI

Teste é código. E por que não colocar todo código que é idêntico entre as várias classes de teste em um único lugar para facilitar reúso? Perceba que todo Page Object contém um `driver`, um método `visita()` e uma URL. Podemos isolar tudo isso em uma classe pai. Agora, se todas elas forem implementações dessas classes abstratas, todas elas terão os mesmos comportamentos e maneiras de funcionar:

```
abstract class PageObject {
    protected WebDriver driver;

    public PageObject(WebDriver driver) {
        this.driver = driver;
    }

    public abstract String url();

    public void visita() {
        driver.go( url() );
    }
}
```

O mesmo pode ser feito com as classes de teste. Todo teste começa com um `@Before` que abre o driver e um `@After` que o fecha. Podemos colocar esses métodos em uma classe pai, e o JUnit, inteligentemente os rodará mesmo que eles estejam na classe pai:

```
abstract class TesteDeSistema {
    @Before
    public void inicializa() {
        // inicializa webdriver
    }
}
```

```
@After
public void finaliza() {
    // finaliza webdriver
}
}
```

Reutilizar código de infraestrutura de testes é sempre uma boa ideia.

## 5.6 COMO LIMPAR O BANCO EM TESTES DE SISTEMA?

Quando tínhamos testes de unidade, limpar o cenário era fácil, afinal eram apenas objetos na memória. Depois, discutimos testes de integração, e ali ficou um pouco mais complicado, pois precisávamos limpar um banco de dados. Agora, no teste de sistema, é tudo ainda mais difícil. Não temos acesso “aos detalhes” do sistema, como código, banco de dados etc.

Mas, do mesmo jeito que fazíamos no teste de unidade, e favorecíamos a testabilidade na hora da criação da classe, precisamos criar maneiras da aplicação facilitar o teste. Já que fazer a aplicação estar em um estado inicial é essencial para o teste, ela deve então prover algum serviço para tal. Em aplicações web, uma simples URL, `/reseta-aplicacao`, que leva a aplicação ao seu estado inicial, é suficiente. Aí, basta o teste, antes de navegar pela aplicação, fazer uma requisição para esse endereço.

Mais para frente, discutirei sobre APIs para criação de cenários. Neste momento, perceba a aplicação deve facilitar a escrita de testes, mesmo que isso implique na produção de mais código e/ou serviços para tal.

## 5.7 REQUISIÇÕES AJAX

Vamos agora à página de detalhes de um leilão. Para isso, basta clicar em “Exibir” ao lado de qualquer leilão. Nessa página, o usuário pode ver os dados de um leilão e fazer lances nele. As imagens a seguir mostram um leilão sem nenhum lance, e com um lance adicionado:

Data	Usuário	Valor
------	---------	-------

Data	Usuário	Valor
30/05/2012	Maria da Silva	300.0

Veja o que acontece quando damos um lance. A página não “pisca”, mas é atualizada! Isso acontece porque nossa aplicação web fez uso do que chamamos de Ajax. Ou seja, a requisição aconteceu, mas ela foi por baixo dos panos, sem o usuário ver. Precisamos testar essa ação, e levar em conta que ela é executada via Ajax.

Nosso teste deve ser algo parecido com isso:

```
@Test
public void deveFazerUmLance() {

    leiloes.detatalhes(1);

    lances.lance("José Alberto", 150);

    assertTrue(lances.existeLance("José Alberto", 150));
}
```

Não há segredo no método `lance()` do `DetalhesDoLeilaoPage`. Precisamos apenas selecionar um usuário no combo e preencher um valor.

Nada de novo até então:

```
public class DetalhesDoLeilaoPage {

    private WebDriver driver;

    public DetalhesDoLeilaoPage(WebDriver driver) {
        this.driver = driver;
    }

    public void lance(String usuario, double valor) {
        WebElement txtValor =
            driver.findElement(By.name("lance.valor"));
        WebElement combo =
            driver.findElement(By.name("lance.usuario.id"));
        Select cbUsuario = new Select(combo);

        cbUsuario.selectByVisibleText(usuario);
        txtValor.sendKeys(String.valueOf(valor));

    }

}
```

Devemos agora submeter o formulário. Mas, dessa vez, não vamos submeter pela caixa de texto. Veja o código-fonte da página! O botão não submete o formulário, ele é um simples “button” do HTML. Precisamos clicar nele. Para isso, basta usar o método `click()`:

```
public class DetalhesDoLeilaoPage {

    private WebDriver driver;

    public DetalhesDoLeilaoPage(WebDriver driver) {
        this.driver = driver;
    }

    public void lance(String usuario, double valor) {
        WebElement txtValor =
```

```
        driver.findElement(By.name("lance.valor"));
        WebElement combo =
            driver.findElement(By.name("lance.usuario.id"));
        Select cbUsuario = new Select(combo);

        cbUsuario.selectByVisibleText(usuario);
        txtValor.sendKeys(String.valueOf(valor));

        driver.findElement(By.id("btnDarLance")).click();
    }
}
```

Agora vamos verificar se o novo lance efetivamente apareceu na tela, também da forma que já conhecemos:

```
public boolean existeLance(String usuario, double valor) {
    return driver.getPageSource().contains(usuario)
        && driver.getPageSource().contains(String.valueOf(valor));
}
```

Mas temos um problema. Uma requisição Ajax acontece por trás do panos, e pode levar alguns segundos para terminar. Se invocarmos o método `existeLance()` e a requisição ainda não tiver voltado, o método retornará falso! Precisamos fazer com que esse método aguarde até a requisição terminar.

Pediremos ao Selenium para que espere alguns segundos até que a requisição termine. Isso é conhecido por **explicit wait**. Para usá-lo, precisamos utilizar a classe `WebDriverWait`. No construtor, ela recebe o driver e a quantidade máxima de segundos a esperar. Em seguida, ela recebe a condição que faz esse tempo parar. No nosso caso, a condição é se o texto aparecer. Para isso, faremos uso de uma outra classe, a `ExpectedConditions`, e passaremos a expectativa correta:

```
Boolean temUsuario =
    new WebDriverWait(driver, 10)
        .until(ExpectedConditions
            .textToBePresentInElement(
```

```
        By.id("lancesDados"), usuario)
    );
```

Vamos colocar o *explicit wait* dentro do método `existeLance()`, completando o código para verificar também se existe o valor:

```
public boolean existeLance(String usuario, double valor) {
    Boolean temUsuario =
        new WebDriverWait(driver, 10)
            .until(ExpectedConditions
                .textToBePresentInElement(
                    By.id("lancesDados"), usuario)
                );

    if(temUsuario) return driver.getPageSource().contains(String.valueOf(valor));
    return false;
}
```

Agora, vamos ao teste. Repare que o cenário é maior: precisamos criar 2 usuários (um que é o dono do produto e outro para dar lance no produto) e um leilão. Vamos também limpar o cenário para facilitar nossa vida. Tudo isso no `@Before`:

```
public class LanceSystemTest {

    private WebDriver driver;
    private LeiloesPage leiloes;

    @Before
    public void inicializa() {
        this.driver = new FirefoxDriver();

        driver.get("http://localhost:8080/apenas-teste/limpa");

        UsuariosPage usuarios = new UsuariosPage(driver);
        usuarios.visita();
        usuarios.novo()
            .cadastra("Paulo Henrique", "paulo@henrique.com");
        usuarios.novo()
```

```
        .cadastra("José Alberto", "jose@alberto.com");

        leiloes = new LeiloesPage(driver);
        leiloes.visita();
        leiloes.novo().preenche("Geladeira", 100,
                                "Paulo Henrique", false);
    }

    @Test
    public void deveFazerUmLance() {

        DetalhesDoLeilaoPage lances = leiloes.detalhes(1);

        lances.lance("José Alberto", 150);

        assertTrue(lances.existeLance("José Alberto", 150));
    }
}
```

Falta só um detalhe: o método `detalhes(1)` do `LeiloesPage` não está implementado. Precisamos fazê-lo chegar à página de detalhes do produto para darmos os lances. Vamos pegar o detalhes do primeiro item da lista e pedir ao Selenium para achar todos os elementos da página com o texto “exibir”, e clicar no primeiro deles. Cuidado com maiúsculas e minúsculas! O link é “exibir” (tudo minúsculo). O Selenium é case-sensitive. Vamos lá:

```
public DetalhesDoLeilaoPage detalhes(int posicao) {
    List<WebElement> elementos =
        driver.findElements(By.linkText("exibir"));
    elementos.get(posicao - 1).click();

    return new DetalhesDoLeilaoPage(driver);
}
```

## 5.8 BUILDERS EM TESTES DE SISTEMA

Perceba que algumas páginas precisam de cenários complicados para começarem. Por exemplo, a página de teste de lances precisa de dois usuários



e um lance cadastrado. Para deixar nosso código de teste mais simples ainda, sem precisar escrever todo o código necessário para levantar cada um desses cenários, podemos escondê-lo em uma classe mais simples.

Por exemplo, veja a classe `CriadorDeCenario`:

```
class CriadorDeCenarios {

    private WebDriver driver;

    public CriadorDeCenarios(WebDriver driver) {
        this.driver = driver;
    }

    public CriadorDeCenarios umUsuario(String nome, String email) {
        UsuariosPage usuarios = new UsuariosPage(driver);
        usuarios.visita();
        usuarios.novo().cadastra(nome, email);

        return this;
    }

    public CriadorDeCenarios umLeilao(String usuario,
        String produto,
        double valor,
        boolean usado) {
        LeiloesPage leiloes = new LeiloesPage(driver);
        leiloes.visita();
        leiloes.novo().preenche(produto, valor, usuario, usado);

        return this;
    }
}
```

Com essa classe, podemos fazer com que os testes da classe `Lance` fiquem mais simples ainda de serem escritos.

```
public class LanceSystemTest {
    @Before
```

```
public void criaCenario() {  
    lances = new DetalhesDoLeilaoPage(driver);  
  
    new CriadorDeCenarios(driver)  
        .umUsuario("Paulo Henrique", "paulo@henrique.com")  
        .umUsuario("José Alberto", "jose@alberto.com")  
        .umLeilao("Paulo Henrique", "Geladeira", 100, false);  
}  
  
// código continua aqui...  
}
```

Isso nos garante que, caso um dia o cenário para criar um usuário mude, mudaremos apenas no `CriadorDeCenarios`.

Uma boa prática é sempre utilizar classes como essa quando precisamos montar um cenário que depende de outras páginas. Assim, o teste de uma página não precisa saber como a outra página funciona. É a ideia do encapsulamento (conceito da orientação a objetos) aplicado ao nosso código de testes.

### O MESMO TESTE EM VÁRIOS BROWSERS

Nada o impede de rodar a mesma bateria de testes de sistema para os mais diferentes browsers. Para isso, bastaria você alterar a linha que faz o `new FirefoxDriver()` para qualquer outro. Uma alternativa seria ler isso de uma variável de sistema, por exemplo. Ou, se preferir, você pode pagar serviços que fazem isso de maneira muito melhor, rodando seus testes na nuvem, e lhe devolvendo apenas os resultados, como é o caso do BrowserStack.

## 5.9 API PARA CRIAÇÃO DE CENÁRIOS

Continuando a discussão que começamos quando falamos de limpar a aplicação, você não deve ter medo de criar código para facilitar a testabilidade da

sua aplicação como um todo. Isso pode significar inclusive o desenvolvimento de serviços web que criam cenários para suas aplicações.

Você já percebeu ao longo deste livro que criar cenários é geralmente a parte mais trabalhosa do teste de qualquer nível. Criar entidades com valores predefinidos, e depois na integração, persisti-los, não é fácil. Em um teste de sistema, isso é exponencial: você precisa criar 2, 3, 4 cenários maiores, para conseguir testar aquela funcionalidade que faz uso de todos os cenários juntos.

Portanto, não é má ideia pensar em serviços web que montem os cenários que são requeridos pelo teste. Serviços web podem ter o formato ideal e fazer o que você achar melhor. Por exemplo, a aplicação pode prover serviços web que recebem as entidades serializadas (em XML, JSON, ou qualquer outro formato) e os insira no banco. Dessa forma, os testes montam os objetos (por meio de Test Data Builders), serializa-os e envia para a aplicação web. Outro caso seria a aplicação ter serviços já específicos, do tipo “cria notas fiscais de pessoas físicas”, que já instanciariam e persistiriam tudo o que for necessário.

Novamente, não há regra pra isso. Faça com que o código seja fácil de ser mantido e fácil de ser consumido pelos seus testes. Não pense que é besteira investir nessa infraestrutura. Ela lhe economizará muito tempo no futuro.

## CAPÍTULO 6

# Testes de serviços web

Serviços web estão por toda a parte, já que hoje a quantidade de aplicações, dos mais diferentes domínios e tecnologias, precisam se comunicar. Por exemplo, uma loja virtual precisa consultar os Correios para saber o valor de determinado frete, ou uma aplicação de compra de passagem aérea que fala com a aplicação de reserva de carros para fazer um pacote turístico.

Existem diferentes maneiras para se distribuir sistemas e fazê-los conversarem entre si. As maneiras mais conhecidas são SOAP e REST. Neste livro, lidaremos com serviços REST, já que esta tem sido a preferência de uma boa parte da indústria ultimamente. Aplicações REST fazem uso de requisições e respostas HTTP simples, geralmente trafegando dados em XML ou JSON.

Imagine uma aplicação que está preparada para responder em XML e JSON. Veja um retorno de exemplo:

```
<list>
```

```
<usuario>
  <id>1</id>
  <nome>Mauricio Aniche</nome>
  <email>mauricio.aniche@caelum.com.br</email>
</usuario>
<usuario>
  <id>2</id>
  <nome>Guilherme Silveira</nome>
  <email>guilherme.silveira@caelum.com.br</email>
</usuario>
</list>
```

Excelente. É assim que um serviço web em REST funciona: fazemos uma requisição e ele nos devolve uma resposta. Precisamos agora testar que, sempre que invocarmos esse serviço web, ele nos devolverá usuários. Para escrever esse teste, faremos uso do framework chamado **Rest-Assured**. Ele, junto com o JUnit, nos ajudará a escrever testes que consomem serviços web. Ele já tem um monte de métodos que nos ajudam a fazer requisições, ler respostas etc. O Rest-Assured pode ser encontrado aqui: <https://code.google.com/p/rest-assured/> ]<https://code.google.com/p/rest-assured/>.

## 6.1 USANDO O REST-ASSURED

Vamos começar nosso primeiro teste. O que ele fará é justamente uma requisição do tipo `GET` para o servidor, e garantirá que a lista com 2 usuários foi recuperada. Escrever testes já não é mais segredo nessa altura. Usaremos JUnit como sempre. Mas, aqui, utilizaremos a API do Rest-Assured para fazer essa requisição. A API do framework é fluente, e faz muito uso de métodos estáticos. Vamos importá-los todos desde já:

```
import static com.jayway.restassured.RestAssured.*;
import static com.jayway.restassured.matcher.RestAssuredMatchers.*;
import static org.hamcrest.Matchers.*;
```

O primeiro método que vamos aprender é o `get(URL)`. Ele faz uma requisição do tipo `GET` para a URL. Em seguida, precisamos dizer a ele que queremos tratar a resposta como `XML`. Veja só como fica a linha:

```
public class UsuariosWSTest {

    @Test
    public void deveRetornarListaDeUsuarios() {
        XmlPath path = get("/usuarios?_format=xml")
            .andReturn().xmlPath();
    }
}
```

Com esse objeto `XmlPath`, podemos agora pegar os dados desse XML. Por exemplo, sabemos que essa URL nos devolve 2 usuários. Vamos recuperá-los. Para isso, usaremos o método `getObject()`, que recebe um caminho dentro do XML, e a classe que ele deve desserializar:

```
@Test
public void deveRetornarListaDeUsuarios() {
    XmlPath path = get("/usuarios?_format=xml")
        .andReturn().xmlPath();
    Usuario usuario1 = path.getObject("list.usuario[0]",
        Usuario.class);
    Usuario usuario2 = path.getObject("list.usuario[1]",
        Usuario.class);
}
```

Com esses dois objetos em mãos, basta agora fazermos asserções neles:

```
@Test
public void deveRetornarListaDeUsuarios() {
    XmlPath path = get("/usuarios?_format=xml")
        .andReturn().xmlPath();
    Usuario usuario1 = path.getObject("list.usuario[0]",
        Usuario.class);
    Usuario usuario2 = path.getObject("list.usuario[1]",
        Usuario.class);

    Usuario esperado1 = new Usuario(1L,
        "Mauricio Aniche", "mauricio.aniche@caelum.com.br");
    Usuario esperado2 = new Usuario(2L,
        "Guilherme Silveira", "guilherme.silveira@caelum.com.br");
}
```

```
    assertEquals(esperado1, usuario1);
    assertEquals(esperado2, usuario2);
}
```

Pronto. Nosso teste passa. Veja que, com o uso do Rest-Assured, não gastamos tempo algum fazendo requisições, ou mesmo parseando as respostas. Preocupamo-nos apenas com o comportamento esperado.

Como falamos anteriormente, podemos passar a informação cuja resposta esperamos em XML pelo próprio header HTTP. Isso também é fácil com Rest-Assured. Veja o código a seguir, que faz uso dos métodos `given()` e `header()`:

```
@Test
public void deveRetornarListaDeUsuarios() {
    XmlPath path = given()
        .header("Accept", "application/xml")
        .get("/usuarios")
        .andReturn().xmlPath();

    Usuario usuario1 = path.getObject("list.usuario[0]",
        Usuario.class);
    Usuario usuario2 = path.getObject("list.usuario[1]",
        Usuario.class);

    Usuario esperado1 = new Usuario(1L,
        "Mauricio Aniche", "mauricio.aniche@caelum.com.br");
    Usuario esperado2 = new Usuario(2L,
        "Guilherme Silveira", "guilherme.silveira@caelum.com.br");

    assertEquals(esperado1, usuario1);
    assertEquals(esperado2, usuario2);
}
```

Nosso teste continua passando. Esses são os primeiros passos com o Rest-Assured.

### DEVO USAR FRAMEWORKS ESPECÍFICOS PARA MEU TESTE?

Qualquer framework que nos ajude a escrever testes e esconda qualquer complexidade deve ser analisado. Se fôssemos escrever o mesmo teste, mas sem o uso do Rest-Assured, precisaríamos gastar muitas linhas de código para fazer uma requisição, depois receber a resposta, parsear o XML etc. Com ele, a tarefa ficou fácil.

Daqui para frente, veremos mais APIs dele, que facilitarão ainda mais a escrita desses testes.

## 6.2 TESTANDO JSONs

Vamos agora testar o serviço que nos devolve um usuário, de acordo com o seu ID. Dessa vez, o retorno será em JSON. A URL para pegarmos o usuário 1, por exemplo, [http://localhost:8080/usuarios/show?usuario.id=1&\\_format=json](http://localhost:8080/usuarios/show?usuario.id=1&_format=json).

Vamos testar esse serviço agora. O teste é parecido com o anterior, mas dessa vez precisamos mudar duas coisas:

- Passar um parâmetro pela *querystring* (`usuario.id`).
- Tratar o retorno que, dessa vez, vem em JSON.

O tratamento em JSON é bem simples. O Rest-Assured abstrai bem isso para nós. Lembra do `XmlPath`? Pois bem, ele tem também o `JsonPath`, que é idêntico. A sintaxe é a mesma. Veja só como faríamos se o teste anterior fosse em JSON:

```
@Test
public void deveRetornarListaDeUsuarios() {
    JsonPath path = given()
        .header("Accept", "application/xml")
        .get("/usuarios")
        .andReturn().jsonPath();

    List<Usuario> usuarios = path
        .getList("list.usuario", Usuario.class);
}
```



```

Usuario esperado1 = new Usuario(1L,
    "Mauricio Aniche", "mauricio.aniche@caelum.com.br");
Usuario esperado2 = new Usuario(2L,
    "Guilherme Silveira", "guilherme.silveira@caelum.com.br");

assertEquals(esperado1, usuarios.get(0));
assertEquals(esperado2, usuarios.get(1));

}

```

Repare que só mudamos de um para outro!

Para passar o parâmetro também é bem simples. Podemos usar o método `parameter()`, que recebe o nome do parâmetro, bem como o conteúdo a ser enviado. Veja o novo teste completo:

```

@Test
public void deveRetornarUsuarioPeloId() {
    JsonPath path = given()
        .parameter("usuario.id", 1)
        .header("Accept", "application/json")
        .get("/usuarios/show")
        .andReturn().jsonPath();

    Usuario usuario = path.getObject("usuario", Usuario.class);
    Usuario esperado = new Usuario(1L,
        "Mauricio Aniche", "mauricio.aniche@caelum.com.br");

    assertEquals(esperado, usuario);
}

```

Note que estamos passando o `ID=1`, e `JSON` no `Accept`. Ao rodar o teste, ele passa.

Mesmo se seu `JSON` não voltar um objeto que possa ser desserializado, a API de Path (tanto de `XML` quanto de `JSON`) possibilita que você apenas navegue por ele. Por exemplo, se quiséssemos pegar apenas a String contendo o nome “Mauricio Aniche”, poderíamos fazer:

```
path.getString("usuario.nome")
```

Veja os métodos disponíveis: `getBoolean()`, `getDouble()`, entre outros. É importante conhecê-los para que você faça bom uso da biblioteca.

### DEVO DEPENDER DE CENÁRIOS PRÉ-PRONTOS?

Até aqui, estamos escrevendo asserções baseadas em cenários que já estão pré-prontos. O ideal é que você não dependa de cenários preexistentes no serviço terceiro. Se possível, seu teste deve ser responsável por criar o cenário, e destruí-lo ao fim, se necessário. Essa é uma discussão recorrente em todos os nossos cursos de teste.

Em último caso, se você não tiver possibilidade de criar o cenário (às vezes, a API externa não lhe dá essa possibilidade), use o cenário provido. Caso contrário, tente controlar o cenário você mesmo. Isso vai diminuir a fragilidade do seu teste.

Perceba que essa discussão também vale para o capítulo anterior, sobre testes de sistema.

## 6.3 ENVIANDO DADOS PARA O WEBSERVICE

Até o momento, só consultamos dados. Vamos agora consumir *webservices* que recebem dados. Vamos adicionar um usuário. O serviço está disponível em <http://localhost:8080/usuarios>. Para inserir um usuário, precisamos fazer um `POST` com um `XML` (ou `JSON`) de um usuário; o formato é o mesmo da nossa entidade `Usuario`.

Vamos começar nosso teste criando o usuário que será adicionado:

```
@Test
public void deveAdicionarUmUsuario() {
    Usuario joao = new Usuario("Joao da Silva", "joao@dasilva.com");
}
```

Agora usaremos a simples API do Rest-Assured para fazer o `POST`. Essa requisição é mais complicada. Precisamos:

- Dizer que a requisição enviará em XML (Content-Type no Header);
- Dizer que o retorno deve ser em XML também (Accept);
- O corpo da requisição deve ser o objeto “joao” serializado;
- O retorno do serviço web deve ser 200.

Vamos começar configurando os dados da requisição. Veja que invocamos o método `header()`, `contentType` e `body`. Os três são autoexplicativos. Mas repare que o `body` sabe que o objeto deve ser serializado em XML, justamente por causa do valor passado para o `contentType`:

```
@Test
public void deveAdicionarUmUsuario() {
    Usuario joao = new Usuario("Joao da Silva", "joao@dasilva.com");

    given()
        .header("Accept", "application/xml")
        .contentType("application/xml")
        .body(joao);
}
```

Em seguida, precisamos fazer o `post`, e receber os dados de volta como XML (da maneira com que já estamos acostumados). Com isso, já conseguimos inclusive fazer o `assert`:

```
@Test
public void deveAdicionarUmUsuario() {
    Usuario joao = new Usuario("Joao da Silva", "joao@dasilva.com");

    XmlPath retorno =
        given()
            .header("Accept", "application/xml")
            .contentType("application/xml")
            .body(joao)
        .when()
            .post("/usuarios")
        .andReturn()
```

```
        .xmlPath();

    Usuario resposta = retorno.getObject("usuario", Usuario.class);

    assertEquals("Joao da Silva", resposta.getNome());
    assertEquals("joao@dasilva.com", resposta.getEmail());
}
```

Por fim, é legal também garantir que o código de retorno do HTTP seja 200. Para isso, usaremos o método `expect()`, que faz a asserção que queremos:

```
@Test
public void deveAdicionarUmUsuario() {
    Usuario joao = new Usuario("Joao da Silva", "joao@dasilva.com");

    XmlPath retorno =
        given()
            .header("Accept", "application/xml")
            .contentType("application/xml")
            .body(joao)
            .expect()
                .statusCode(200)
            .when()
                .post("/usuarios")
            .andReturn()
                .xmlPath();

    Usuario resposta = retorno.getObject("usuario", Usuario.class);

    assertEquals("Joao da Silva", resposta.getNome());
    assertEquals("joao@dasilva.com", resposta.getEmail());
}
```

Excelente. Nosso teste agora passa. Observe novamente que, com o uso do framework, nossos testes são por demais simples de serem escritos.

Veja que aqui estamos fazendo uma inserção. Ou seja, um novo elemento

está sendo inserido no serviço externo. Você, desenvolvedor de testes, precisa ficar atento a isso, pois isso pode mudar o resultado de outros testes. Imagine que você tenha um teste que garanta que o serviço que retorna a quantidade de usuários funcione; se você adicionar um novo usuário, o teste vai quebrar, pois o número de usuários cresceu.

Essa discussão aconteceu no exercício do capítulo anterior. Precisamos sempre tomar cuidado com cenários em serviços externos. Se você criou um usuário, você precisa deletá-lo depois. Para isso, você pode usar algum serviço de deleção, que é disponibilizado pelo serviço web.

## Configurando o Rest-Assured

Você reparou que até agora temos passado endereços relativos, mas ele sabe que nossa aplicação está em <http://localhost:8080>? Pois bem, isso é a configuração padrão dele. Você pode mudá-la. Por exemplo, se quiséssemos mudar a URL base, faríamos:

```
RestAssured.baseURI = "http://www.meuendereco.com.br";  
RestAssured.port = 80;
```

Geralmente colocamos esse tipo de configuração em algum lugar centralizado. Por exemplo, em algum `@Before` ou `@BeforeClass` da bateria de testes. Assim, fica fácil mudar essa configuração, caso o endereço do serviço mude.

## 6.4 OUTROS RECURSOS DO REST-ASSURED

Além de fazer requisições e tratar bem as respostas de diferentes tipos, como XML e JSON, o Rest-Assured ainda nos dá outros recursos.

Por exemplo, podemos garantir que um *cookie* foi gerado. Para isso, basta usarmos o `expect()`, que já conhecemos. A URL `/cookie/teste` nos gera um cookie com o nome “rest-assured”, com o valor “funciona”. Vamos validá-lo:

```
@Test  
public void deveGerarUmCookie() {  
    expect()
```

```
        .cookie("rest-assured", "funciona")
    .when()
        .get("/cookie/teste");
}
```

Veja só como o teste fica curto.

Podemos também garantir que um determinado `header` chegou. Por exemplo, a mesma URL cria um header chamado “`novoHeader`”. Validar é idêntico:

```
@Test
public void deveGerarUmHeader() {
    expect()
        .header("novo-header", "abc")
    .when()
        .get("/cookie/teste");
}
```

O Rest-Assured pode fazer muitos outros tipos de asserções, dependendo da sua necessidade. Uma boa dica é consultar o manual em <https://code.google.com/p/rest-assured/wiki/Usage>.

## DESAFIOS EM TESTES DE SERVIÇOS WEB

Você deve ter percebido que quanto mais perto do mundo real, mais difícil o teste é, por diferentes motivos: montar cenários, invocar comportamentos, esperar o sistema (e seus N componentes) responderem corretamente etc.

Em serviços web, montar cenários é particularmente complicado. Analogamente à sugestão dos testes de sistema, se o serviço web é desenvolvido pela sua equipe, você pode ter uma API que monta cenários e reseta a aplicação para um estado inicial. Se o serviço não é seu, então precisará contar com a existência de algum tipo de *sandbox* do serviço terceiro para que você consiga testar. Em último caso, você também pode criar simuladores do serviço web do terceiro. Esses simuladores apenas recebem requisições e devolvem respostas no mesmo formato que o serviço original.



## CAPÍTULO 7

# E agora?

Espero que, ao longo deste livro, eu tenha lhe passado a importância de se automatizar testes, e o quanto isso é tangível. Espero também ter mostrado o lado prático e que você tenha entendido as ideias por trás de cada trecho de código apresentado.

Ainda há muito o que estudar, claro. Mas acredito que eu tenha conseguido dar uma visão geral sobre a prática de testes automatizados na indústria.

### **7.1 NÃO HÁ MAIS NENHUM TIPO DE TESTE?**

Claro que há. Podemos falar de testes de carga e estresse, que validam se a aplicação está pronta para suportar 300 usuários simultâneos, respondendo todas as requisições com tempo máximo de 1 segundo. Esse tipo de restrição é importante em sistemas financeiros, e você precisa de teste para isso.



Podemos pensar também em testes de usabilidade, com os quais garantimos que nossa interface está modelada de maneira a maximizar a experiência do usuário dentro do software.

Em aplicações web, podemos pensar em testes específicos para Javascript. Javascript, apesar de todo o preconceito existente por trás, é uma linguagem moderna, e que merece atenção. Se sua aplicação faz uso intenso de Javascript, e você tem regras de negócio importantes, elas precisam ser testadas de maneira automatizada. Aqui entra a parte importante: com tudo o que você viu aqui, é capaz de escrever esse tipo de teste. Qual o caminho? Escolher um framework de testes automatizados na linguagem, e sair escrevendo os testes (pensando nos cenários, ações e validações) para os diferentes trechos do seu código.

A partir do momento que você entendeu a ideia, perceberá que teste é tudo igual. Se você percebeu isso, e consegue hoje olhar para qualquer parte do seu sistema e pensar no teste para ela, então tenho certeza de que cumpri minha missão.

## **7.2 NÃO PRECISO NUNCA DE TESTES MANUAIS?**

Testes manuais ainda continuam sendo importantes, por diferentes motivos. O primeiro deles é para testes exploratórios: ninguém melhor do que um ser humano para olhar para um software e achar uma maneira de quebrá-lo. Ou, às vezes, você tem um trecho do sistema que ainda não achou como automatizar; nesse caso, não tem jeito, o teste manual é melhor do que nada.

Lembre-se de usar um ser humano para situações em que precisamos de um ser humano. Para situações onde precisamos de alguém pensando e refletindo sobre o que está vendo. Se você tem uma situação dessas, faça teste manual. Caso contrário, deixe a máquina fazer o trabalho por você.

## **7.3 TESTES DE SISTEMA O TEMPO TODO?**

Se o teste de sistema é tão mais parecido com o comportamento do usuário final, por que não só escrever testes desse nível? A pergunta faz total sentido e é bastante recorrente no dia a dia.

O problema com os testes de sistema é que eles são muito mais difíceis de serem mantidos. Eles são frágeis por natureza, afinal uma mudança pequena em qualquer ponto pode fazê-lo quebrar. Ele depende de muita coisa. Eles também são muito mais demorados do que testes de unidade, e a partir do momento em que você tem 100 ou 200 testes como esses, o feedback passa a levar horas em vez de segundos.

Você deve balancear seus testes entre todos os níveis. Favoreça testes de unidade para garantir que suas regras de negócio funcionem de maneira correta. Regras de negócio, no fim, são um conjunto de `ifs` e `fors`, e você consegue testá-las de maneira rápida e fácil.

Deixe o teste de sistema para garantir que aquelas partes quentes e importantes do seu sistema não parem nunca de funcionar. Ou seja, evite ao máximo escrever testes de sistema para partes menos importantes do sistema, do ponto de vista do negócio. Mas, por exemplo, teste seu sistema de pagamento com todos os tipos de teste que puder.

Testar não é ter simplesmente ter testes automatizados. É maximizar o feedback que essa bateria pode lhe dar. E para isso, você precisa balancear entre todos os diferentes níveis de teste.

## 7.4 ONDE POSSO FALAR MAIS SOBRE O ASSUNTO?

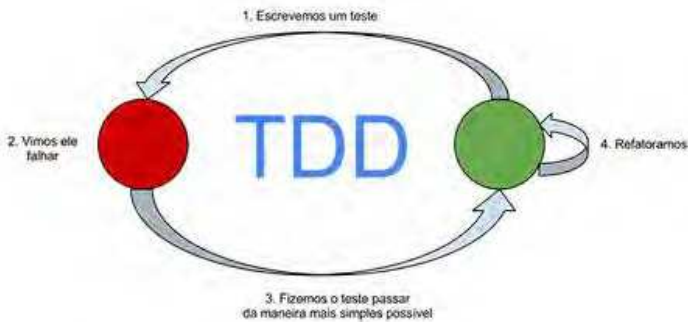
Você pode fazer parte da lista de discussão dos meus livros. Inicialmente, ela começou apenas como lista para meu primeiro livro sobre TDD, mas hoje as discussões abrangem tudo sobre testes automatizados.

Para isso, basta se cadastrar no Google Groups: <https://groups.google.com/forum/#!forum/tdd-no-mundo-real>. As discussões por lá são bastante interessantes, e tenho certeza de que conseguirá continuar seus estudos por lá.

Se você ainda achar que isso não é suficiente, você pode assistir à formação de testes online do Alura (<http://www.alura.com.br/cursos-online-testes>), no qual eu fui o instrutor. Lá, você terá a chance de reproduzir cada trecho de código mostrado aqui.

## 7.5 OBRIGADO!

Ufa, acabou. Espero que tenha gostado do livro. E lembre-se: **teste o tempo todo!**



Esse ciclo de desenvolvimento, onde escrevemos um teste antes do código, é conhecido por **Test-Driven Development**. A popularidade da prática de TDD tem crescido cada vez mais entre os desenvolvedores, uma vez que ela traz diversas vantagens:

- Se sempre escrevermos o teste antes, garantimos que todo nosso código já “nasce” testado;
- Temos segurança para refatorar nosso código, afinal sempre refatoraremos com uma bateria de testes que garante que não quebraremos o comportamento já existente;
- Como o teste é a primeira classe que usa o seu código, você naturalmente tende a escrever código mais fácil de ser usado e, por consequência, mais fácil de ser mantido.
- Efeitos no design. É comum percebermos que o projeto de classes de sistemas feitos com TDD é melhor do que sistemas feitos sem TDD.

## 2.4 EFEITOS NO DESIGN DE CLASSES

Muitos desenvolvedores famosos, como Kent Beck, Martin Fowler e Michael Feathers, dizem que a prática de TDD faz com que seu design de classes melhore. A grande pergunta é: como?

Para isso, faremos uso do **Selenium**. O Selenium é um framework que facilita e muito a vida do desenvolvedor que quer escrever um teste automatizado. A primeira coisa que uma pessoa faria para testar a aplicação seria abrir o browser. Com Selenium, precisamos apenas da linha a seguir:

```
WebDriver driver = new FirefoxDriver();
```

Nesse caso, estamos abrindo o Firefox! Em seguida, entraríamos em algum site. Vamos entrar no Google, por exemplo, usando o método `get()`:

```
driver.get("http://www.google.com.br/");
```

Para buscar o termo “Caelum”, precisamos digitar no campo de texto. No caso do Google, o nome do campo é “q”.



Para descobrir, podemos fazer uso do Inspector, no Chrome (ou do Firebug no Firefox). Basta apertar `Ctrl + Shift + I` (ou `F12`), e a janela abrirá. Nela, selecionamos a lupa e clicamos no campo cujo nome queremos descobrir. Ele nos levará para o HTML, onde podemos ver `name="q"`.



Com Selenium, basta dizermos o nome do campo de texto, e enviarmos o texto:

```
WebElement campoDeTexto = driver.findElement(By.name("q"));
campoDeTexto.sendKeys("Caelum");
```

Agora, basta submetermos o form! Podemos fazer isso pelo próprio campoDeTexto:

```
campoDeTexto.submit();
```

Pronto! Juntando todo esse código em uma classe Java simples, temos:

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;

public class TesteAutomatizado {

    public static void main(String[] args) {
        // abre firefox
        WebDriver driver = new FirefoxDriver();

        // acessa o site do google
```