Teste de Software
Aula 01

JUnit

Sumário

- Situação-exemplo: Sistema de Leilão
- Implementando uma nova funcionalidade
- Criando um Teste Unitário
- Classes de Equivalência
- Refatoramento
- Cláusulas @Before e @After
- Padrão Test Data Builder
- Testando Exceções
- Usando o Hamcrest

Situação-exemplo: sistema de Leilão

Alguns exemplos próximos:

a) coliseumleiloes.com.br
b) vizeuonline.com.br

Situação-exemplo: sistema de Leilão

Funcionalidades básicas:

a) Encontrar o maior lance
b) Encontrar o menor lance
c) Encontrar os três maiores lances

Algoritmo para encontrar o maior lance

```
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;
    }
```

Por que isso?

Exercício: implemente o que for necessário para executar esse código.

```
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());
    }
}
```

Exercício: implemente o que for necessário para executar esse código.

Implementando uma nova funcionalidade

Objetivo: encontrar **o menor lance** de todos.

Exercício: altere o método avalia() para que ele encontre também o menor lance.

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

Criando o teste unitário

Exercício: crie a classe de teste unitário referente à classe Avaliador.

1. Lembre-se de **testar vários cenários** (lembra da aula anterior?)
2. Lembre-se das convenções na escrita de testes com JUnit:
   - Nome da classe de test: **AvaliadorTest**
   - O nome dos métodos deve descrever o cenário de teste realizado (ex: public void lancesEmOrdemCrescente())
   - assertEquals(esperado, calculado)

Classes de equivalência p/ cenários de teste

Ordem Crescente

(100, 200, 300),
(340, 320, 560), ...

Ordem Decrescente

(300, 200, 100),
(740, 520, 260), ...

Ordem Aleatória

(170, 120, 250, 180),
(340, 420, 160, 90),
...

- Geralmente, só fazemos um método de teste para cada classe de equivalência
- Classes de testes também precisam de manutenção

Refatorando o código de teste!

Podemos melhorar a qualidade do código do nosso teste.

Já reparou quantas vezes você escreveu Assert.assertEquals()?

```
import org.junit.Assert;

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

Refatorando o código de teste!

Substitua pelo import do método estático!

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

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

O seu cliente deseja uma nova funcionalidade

O sistema de Leilão deve agora ser capaz de mostrar **os três maiores lances**!

Como você resolveria isso?

Exercício: crie um outro método que retorne uma lista com os três maiores lances em ordem decrescente.
Dica: lembre das aulas de Estrutura de Dados

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

public void avalia(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 pegaOsMaioresNo(Leilao leilao) {
    maiores = new ArrayList<Lance>(leilao.getLances());
    Collections.sort(maiores, 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> getTresMaiores() {
    return this.maiores;
```

Testando a nova funcionalidade...

```
@Test
public void deveEncontrarOsTresMaioresLances() {

 ...

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

Exercício: complete esse método. Insira pelo menos quatro lances. Execute o teste!

Deu bug?! Encontre-o e remova-o!

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!

Exercício: corrija o algoritmo que retorna os três maiores lances.

Testando a nova funcionalidade...novamente

```
@Test
public void deveEncontrarOsTresMaioresLances() {

 ...

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

Isso é suficiente??

Exercício: melhore esse teste.

Testando casos especiais

“Quando lidamos com listas, por exemplo, é sempre interessante tratarmos o caso da lista cheia, da lista com apenas um elemento, da lista vazia.” (ANICHE, 2015, p. 24)

Outro exemplo: **if(salario>=2000)**

- Cenário A: salário menor que 2000
- Cenário B: salário maior que 2000
- Cenário C: salário igual a 2000

Dica

“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.” (ANICHE, 2015, p. 26)

Melhorando o código de teste

Teste é código. Código mal escrito é difícil de ser mantido.
Observe que temos a seguinte linha em **todos** os métodos dessa classe:

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

E se no futuro mudarmos o construtor, passando algum parâmetro? Precisaríamos alterar em todos os métodos!

Melhorando o código de teste

```
public class AvaliadorTest {

    private Avaliador leiloeiro;

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

Melhorando o código de teste

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

    // invocando método auxiliar
    criaAvaliador();
    leiloeiro.avalia(leilao);
```

Melhorando o código de teste: cláusula @Before

É possível programar o JUnit para que ele invoque um método auxiliar antes de iniciar os métodos de test:

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

Melhorando o código de teste: cláusula @Before

Dessa forma, podemos retirar a invocação do método `criaAvaliador()` de **todos os métodos de teste**, já que o próprio JUnit irá fazer isso.

Se sua classe de teste possui 5 métodos de teste (@Test) então o JUnit executará o método `criaAvaliador()` 5 vezes: uma vez antes de cada método.

Melhorando o código de teste: Test Data Builders

- Builder: construtor

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.

Test Data Builders

Nosso objetivo é ter um código mais simples e enxuto:

```
@Test
    public void deveEncontrarOsTresMaioresLances() {
        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();
```

Test Data Builders

Escrever a classe `CriadorDeLeiloes` é razoavelmente simples.

O segredo é possibilitar que invoquemos um método atrás do outro.

Como?

R= basta retornarmos o `this` em todos os métodos!

Test Data Builders: definição

Trata-se de 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.” (ANICHE, 2015, p. 31)

```
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;
}
```

Cláusula @After

Métodos anotados com **@After** são executados após a execução do método de teste.

Utilizamos quando os testes consomem **recursos** que precisam ser **finalizados**.

Exemplos: conexão com o banco de dados, arquivo, socket, etc.”

Cláusulas `@BeforeClass` e `@AfterClass`

Método anotado com **`@BeforeClass`** são executados apenas uma vez, antes de todos os métodos de teste.

Método anotado com **`@AfterClass`** é executado uma vez, após a execução do último método de teste da classe.

Testando Exceções

O que aconteceria com nosso código se o `leilao` passado não recebesse nenhum lance?

```
maiorDeTodos = Double.NEGATIVE_INFINITY;
```

Isso não faria sentido!

Melhor utilizar uma **exception** para esse caso!

```
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"
            );
```

Como testar esse trecho de código??

Testando Exceções (atributo `expected`)


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

    leiloeiro.avalia(leilao);
}
```

Melhorando a legibilidade dos testes

Qual das linhas abaixo é mais intuitiva?

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

“garanta que o menor lance é igual a 250.0”

Projeto **Hamcrest**: hamcrest.org/JavaHamcrest

Usando o Hamcrest

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

@Test
public void deveEntenderLancesEmOrdemCrescente() {
    ...
    assertThat(leiloeiro.getMenorLance(), equalTo(250.0));
    assertThat(leiloeiro.getMaiorLance(), equalTo(400.0));
```

```
        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);
        }
}
```

Estamos usando quatro asserts! E veja que comparamos atributos, não os objetos em si. Isso geralmente **não é bom**.

Usando o Hamcrest (outro exemplo)


```
@Test
public void deveEncontrarOsTresMaioresLances() {
    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.getTresMaiores();
...
```

Usando o Hamcrest

Como ficaria usando o Hamcrest:

```
assertThat(maiores, hasItems(
        new Lance(maria, 400),
        new Lance(joao, 300),
        new Lance(maria, 200)
));
```

Matchers:

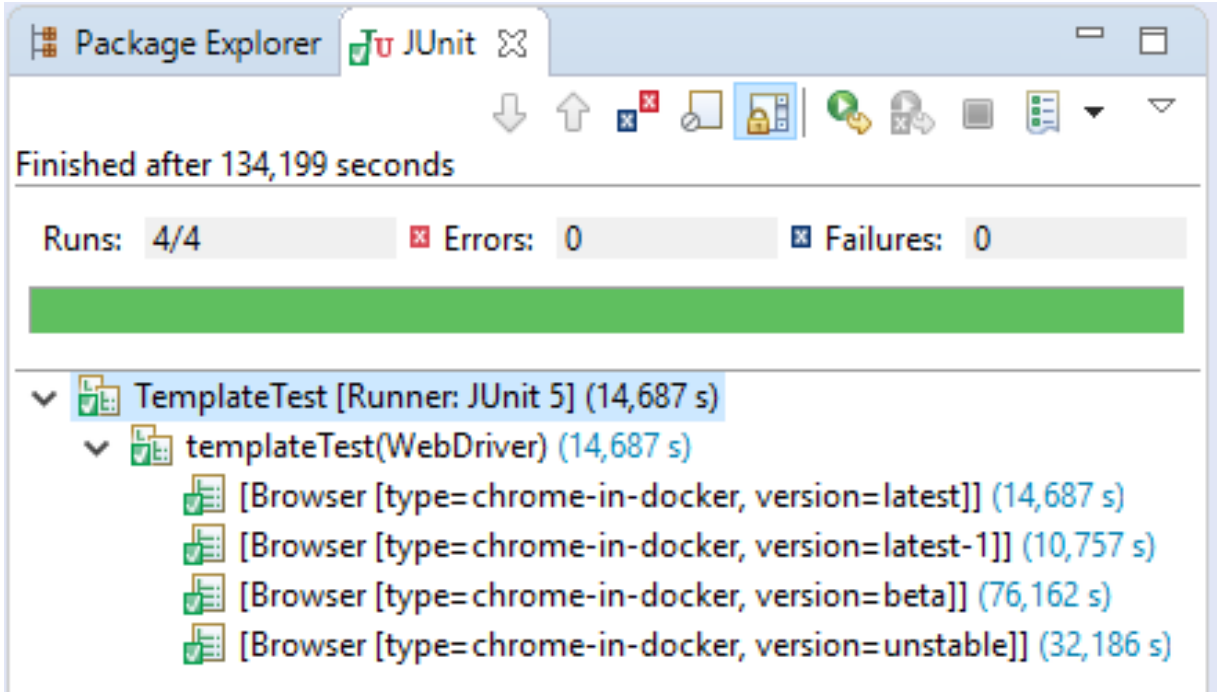
- assertThat
- hasItems
- equalTo
- ...

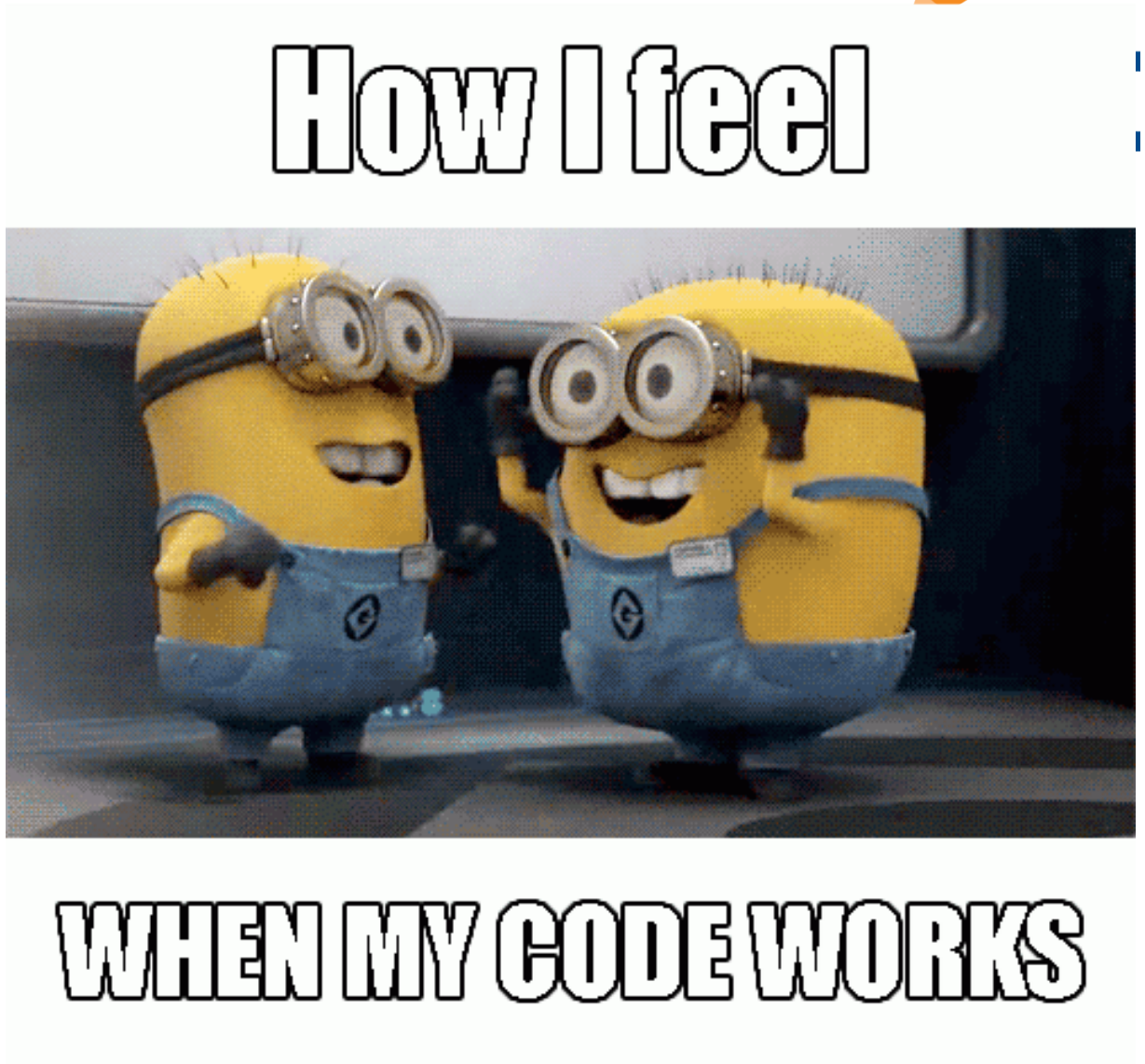
100% de cobertura de testes?

Cobertura de código: métrica de software que indica a porcentagem de código que está coberta por pelo menos um teste.

Alguns trechos de códigos não precisam ser testados:

Ex: Getters, setters.


Package Explorer
JUnit
Finished after 134,199 seconds
Runs: 4/4
Errors: 0
Failures: 0
TemplateTest [Runner: JUnit 5] (14,687 s)
templateTest(WebDriver) (14,687 s)
[Browser [type=chrome-in-docker, version=latest]] (14,687 s)
[Browser [type=chrome-in-docker, version=latest-1]] (10,757 s)
[Browser [type=chrome-in-docker, version=beta]] (76,162 s)
[Browser [type=chrome-in-docker, version=unstable]] (32,186 s)


How I feel
WHEN MY CODE WORKS

Referências

Mauricio Aniche. Test-Driven Development: Teste e Design no Mundo Real. Casa do Código, 2014.