**Exemplos de teste:**

**Comparando Doubles:**

|  |
| --- |
| **import** org.junit.Assert;  **import** org.junit.Test;  **import** br.com.caelum.leilao.dominio.Lance;  **import** br.com.caelum.leilao.dominio.Leilao;  **import** br.com.caelum.leilao.dominio.Usuario;  **import** br.com.caelum.leilao.servico.Avaliador;  **public** **class** AvaliadorTest {    @Test  **public** **void** deveEntenderLancesEmOrdemCrescente() {//o método precisa ser public, não pode ser static e não retorna nada  // cenario: 3 lances em ordem crescente  Usuario joao = **new** Usuario("João");  Usuario jose = **new** Usuario("José");  Usuario maria = **new** Usuario("Maria");  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: acao  Avaliador leiloeiro = **new** Avaliador();  leiloeiro.avalia(leilao);    //parte 3: validação  // exibindo a saida - a classe Assert tem vários métodos que nos ajudam a conferir o resultado calculado  **double** maiorEsperado = 400;  **double** menorEsperado = 250;  /\*\*  \* AssertEquals  \* primeiro o valor esperado, depois o método que vai ser testado e trazer o valor para comparação e depois um delta,  \* devido o double ter problemas para arredondar, esta é a margem que ele pode deixar "passar"  \*/  Assert.*assertEquals*(maiorEsperado, leiloeiro.getMaiorLance(), 0.00001);  Assert.*assertEquals*(menorEsperado, leiloeiro.getMenorLance(), 0.00001);  }  } |

**Dificuldades nos testes de unidade**

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, e 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?

**TDD – Test Drive Development**

A ideia focal é escrever o teste antes do código do produção com a validação da regra de negócio para ver ele falhar, depois escrever o código em produção da maneira mais simples possível para ver ele passar no teste, depois que passou fica livre para você refatorar o código e melhorar o código, pois caso erre em algo o teste falhará.

O ideal é garantir que toda regra de negócio seja testada, mesmo que simples.

**Entendendo o @Before**

O método Before é executado antes de cada teste da classe.

**Conhecendo o @After**

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, e etc.

(Apesar desses testes não serem mais considerados testes de unidade, afinal eles falam com outros sistemas, desenvolvedores utilizam JUnit para escrever testes de integração.

**@BeforeClass e @AfterClass**

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.

**Vantagens de testes de qualidade**

Nosso 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. Todas elas são maneiras para fazer com que nosso código de testes evolua mais facilmente.

**Melhorando a legibilidade**

Vamos agora continuar a melhorar nosso código de teste. Nossos testes já estão bem expressivos, mas algumas coisas ainda não são naturais. Uma delas é 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 abaixo 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.avalia(leilao);

assertThat(leiloeiro.getMenorLance(), equalTo(250.0));

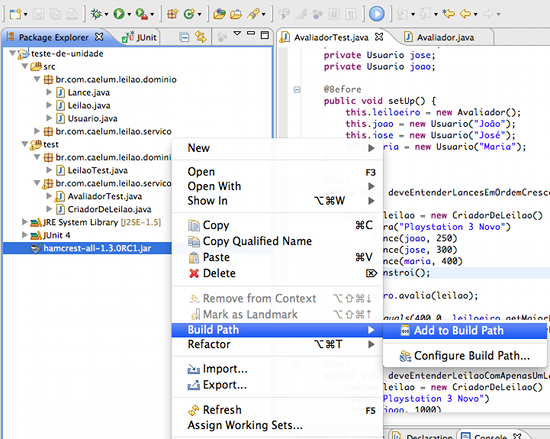
assertEquals(400.0, leiloeiro.getMaiorLance(), 0.00001);

}

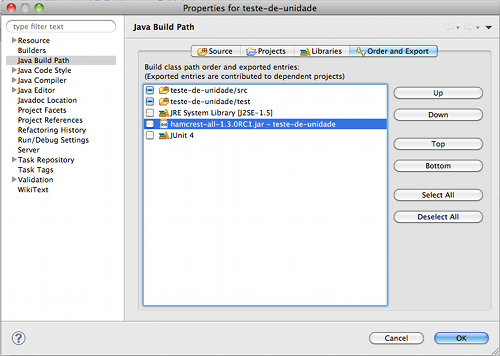
}

Veja que o primeiro assert é muito mais legível. Se lermos essa linha como uma frase em inglês, temos garantia 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! Baixe a biblioteca, clicando [aqui](http://code.google.com/p/hamcrest/downloads/detail?name=hamcrest-all-1.3.0RC1.jar). Com o .jar em mãos, coloque-o no projeto e adicione-o ao Build Path (clicando com botão direito em cima do .jar, e selecionando Build Path -> Add to Build Path), como demonstrado na figura abaixo:



Além disso, precisamos falar para o Eclipse que a biblioteca do Hamcrest deve ser exportada antes do JUnit. Vamos configurar isso também no JUnit. Clique com o botão direito do mouse sobre o projeto, e selecione Build Path -> Configure Build Path. Em seguida, selecione a aba Order and Export. Coloque o .jar do Hamcrest em cima do JUnit, clicando no botão Up, como mostrado na figura abaixo:



Agora, altere o método de teste na classe AvaliadorTest para que o mesmo use as asserções do Hamcrest. Repare nos imports:

import static org.junit.Assert.assertEquals;

import static org.hamcrest.MatcherAssert.assertThat;

import static org.hamcrest.Matchers.\*;

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

@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();

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 esse asserts para algo muito mais expressivo:

assertThat(maiores, hasItems(

new Lance(maria, 400),

new Lance(joao, 300),

new Lance(maria, 200)

));

Veja que estamos conferindo se a lista maiores contém os 3 lances esperados! Para que o matcher (o nome na qual esses métodos estáticos hasItems, equalTo, etc, são chamados) funcione, precisamos implementar o método equals() na classe Lance:

@Override

public boolean equals(Object obj) {

if (this == obj)

return true;

if (obj == null)

return false;

if (getClass() != obj.getClass())

return false;

Lance other = (Lance) obj;

if (usuario == null) {

if (other.usuario != null)

return false;

} else if (!usuario.equals(other.usuario))

return false;

if (Double.doubleToLongBits(valor) != Double

.doubleToLongBits(other.valor))

return false;

return true;

}

O Hamcrest possui muitos outros matchers e você pode conferi-los na documentação do projeto, clicando [aqui](http://code.google.com/p/hamcrest/wiki/Tutorial).

Lembre-se sempre de deixar seu teste o mais legível possível! O Hamcrest é uma alternativa!

[**DIS**](https://cursos.alura.com.br/forum/curso-tdd/exercicio-testando-excecoes/973/novo)