01

Testando o que realmente é necessário

Transcrição

No fim da aula passada, 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. 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, e sim criamos um novo.

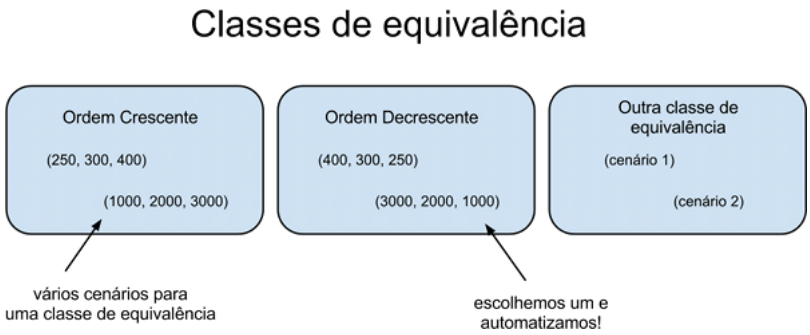
```
class AvaliadorTest {

    @Test
    public void deveEntenderLancesEmOrdemCrescente() {
      // codigo omitido
    }

    @Test
    public void deveEntenderLancesEmOrdemCrescenteComOutrosValores() {
        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, 1000.0));
        leilao.propoe(new Lance(jose, 2000.0));
        leilao.propoe(new Lance(maria, 3000.0));

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

        Assert.assertEquals(3000.0, leiloeiro.getMaiorLance(), 0.0001);
        Assert.assertEquals(1000.0, 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!
}
```

Infelizmente testar todas as combinações é impossível! E se tentarmos fazer isso (e escrevermos muitos testes, como no exemplo acima), 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 abaixo 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.

```
class AvaliadorTest {

    @Test
    public void deveEntenderLancesEmOrdemCrescente() {
      // codigo omitido
    }

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

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

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

        Assert.assertEquals(1000.0, leiloeiro.getMaiorLance(), 0.0001);
        Assert.assertEquals(1000.0, leiloeiro.getMenorLance(), 0.0001);
    }
}
```

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

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

class AvaliadorTest {

    // outros testes ainda estao aqui...

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

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

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

        // veja que não precisamos mais da palavra Assert!
        assertEquals(1000.0, leiloeiro.getMaiorLance(), 0.0001);
        assertEquals(1000.0, 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!

Precisamos continuar a escrever testes para as classes de equivalência que levantamos! Mas essa será sua tarefa no exercício!

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

        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;
    }

    public double getMaiorLance() {
        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 deveEncontrarOsTresMaioresLances() {
        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.avalia(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 abaixo 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 o tamanho dela precisamos verificar o conteúdo interno dela. Então 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 deveEncontrarOsTresMaioresLances() {
        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.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);
    }
}
```

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!? Mãos à obra!