Teste de unidade - parte 1

Site: Moodle institucional da UTFPR
Curso: CETEJ158A - Teste de Software - J29 (2023_03) J30A (2024_01)
Livro: Teste de unidade - parte 1

Impresso por: MARLENE VASCONCELOS MORAES DE OLIVEIRA
Data: quinta-feira, 25 jul. 2024, 05:35

Descrição

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?

Índice

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.

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.avalia(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.avalia(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.

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 você 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) {

// cenario: 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.avalia(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.avalia(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.

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 as IDEs.

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:

```
public class NewClass {

}

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.avalia(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 o resultado dado pelo JUnite, 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 {

    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.avalia(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 na IDE com o resultado do teste.

Nosso teste não passa. Veja que uma mensagem de erro foi 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 ficará 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?

5. Convenções na escrita de testes

Como você percebeu, teste é código, e código pode ser escrito de muitas for- mas 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 desenvolve- dor 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 co- mum. 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.curso estão em br.com.curso 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.

5.1. 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. Falaremos mais sobre isso à frente.

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.

7. Testando o que realmente é necessário

No fim do seção Olá mundo, JUnit!, 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.avalia(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!
}
```

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 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.avalia(leilao);

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

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

    }

}
```

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.avalia(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!

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 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;
    }

    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 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;

public class AvaliadorTest {

    @Test
// outros testes aqui
    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, 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!?

10.1. 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.

11. Referência Bibliográfica

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