Teste de unidade - parte 2

Site:	Moodle institucional da UTFPR	Impresso por:	MARLENE VASCONCELOS MORAES DE OLIVEIRA
Curso:	CETEJ158A - Teste de Software - J29 (2023_03) J30A (2024_01)	Data:	domingo, 25 ago. 2024, 09:15
Livro:	Teste de unidade - parte 2		

Descrição

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? 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.

Índice

1. 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.

2. 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.

3. 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.

4. 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.avalia(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.

```java
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: validacão
        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 deveEncontrarOsTresMaioresLances() {
// 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.

5. 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:

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

    @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();

        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);
    }
}
```

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.

6. @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.

7. 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.

8. 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.

9. 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() lancará uma exceção. Além disso, como fazemos o assert? Não existe um assertException() ou algo do tipo:

```
public class AvaliadorTest {

    @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:

```
public class AvaliadorTest {

    @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:

```
public class AvaliadorTest {

    @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.

10. 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.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 garanta que o menor lance é igual a 250.0. Muito mais legível!

Para conseguirmos escrever asserts como esse, podemos fazer uso do pro- jeto Hamcrest. Ele contém um monte de instruções como essas, que simples- mente 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:

```
class AvaliadorTest {

@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 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.

11. Referência Bibliográfica

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