Praticando Test-Driven Development (TDD)

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, 09:13
Livro:	Praticando Test-Driven Development (TDD)		

Descrição

Será que conseguimos pensar em uma outra maneira de desenvolver e escrever nossos testes, que não a que estamos acostumados?

Índice

1. Testes, do jeito que você já sabe

Apesar de ser simples, no contexto de um sistema de leilão, essa classe é de extrema importância, portanto, merece ser testada.

```java
public class Leilao {

    private String descricao;
    private List<Lance> lances;

    public Leilao(String descricao) {
        this.descricao = descricao;
        this.lances = new ArrayList<Lance>();
    }

    public void propoe(Lance lance) {
        lances.add(lance);
    }

    public String getDescricao() {
        return descricao;
    }

    public List<Lance> getLances() {
        return Collections.unmodifiableList(lances);
    }
}
```

O método propoe(), em especial, é essencial para o leilão e provavelmente sofrerá mudanças no decorrer do projeto. Por isso, ela precisa ser testada para termos certeza de que ela está funcionando conforme se espera. Inicialmente, vamos analisar se um determinado lance que foi proposto ficará armazenado no leilão. Para isso, temos dois casos a serem averiguados: a realização de apenas um lance e a de mais de um lance.

O código para realizar tal teste não tem muito segredo. Começaremos instanciando um leilão e serão propostos alguns lances nele:

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

public class LeilaoTest {

    @Test
    public void deveReceberUmLance() {
        Leilao leilao = new Leilao("Macbook Pro 15");
        assertEquals(0, leilao.getLances().size());

        leilao.propoe(new Lance(new Usuario("Steve Jobs"), 2000));
        assertEquals(1, leilao.getLances().size());
        assertEquals(2000, leilao.getLances().get(0).getValor(), 0.00001);
    }

    @Test
    public void deveReceberVariosLances() {
        Leilao leilao = new Leilao("Macbook Pro 15");
        leilao.propoe(new Lance(new Usuario("Steve Jobs"), 2000));
        leilao.propoe(new Lance(new Usuario("Steve Wozniak"), 3000));
        assertEquals(2, leilao.getLances().size());
        assertEquals(2000, leilao.getLances().get(0).getValor(), 0.00001);
        assertEquals(3000, leilao.getLances().get(1).getValor(), 0.00001);
    }
}
```

Agora implementaremos duas novas regras de negócio no processo de lances em um leilão:

- Uma pessoa não pode propor dois lances em sequência;
- Uma pessoa não pode dar mais do que cinco lances no mesmo leilão.

2. Mudando a maneira de desenvolver

Contudo, dessa vez, vamos fazer diferente. Estamos muito acostumados a implementar o código de produção e testá-lo ao final. Mas será que essa é a única maneira de desenvolver um projeto? Vamos tentar inverter e começar pelos testes! Além disso, vamos tentar ao máximo ser o mais simples possível, ou seja, pensar no cenário mais simples naquele momento e implementar sempre o código mais simples que resolva o problema.

Vamos começar pelo cenário. Um novo leilão cujo mesmo usuário dê dois lances seguidos e, por isso, o último lance deve ser ignorado:

```
    @Test
    public void naoDeveAceitarDoisLancesSeguidosDoMesmoUsuario() {

        Leilao leilao = new Leilao("Macbook Pro 15");
        Usuario steveJobs = new Usuario("Steve Jobs");
        leilao.propoe(new Lance(steveJobs, 2000));
        leilao.propoe(new Lance(steveJobs, 3000));
        assertEquals(1, leilao.getLances().size());
        assertEquals(2000,
                leilao.getLances().get(0).getValor(), 0.00001);
    }
```

Ótimo, teste escrito. Ao rodar o teste, ele falha. Mas tudo bem, já estávamos esperando por isso! Precisamos fazê-lo passar agora, da maneira mais simples possível. Vamos modificar o método propoe(). Agora ele, antes de adicionar o lance, verificará se o último lance da lista não pertence ao usuário que está dando o lance naquele momento. Veja que fazemos uso do método equals() na classe Usuario.

```
    public void propoe(Lance lance) {

        if (lances.isEmpty() ||
        !lances.get(lances.size() - 1)
                .getUsuario()
                .equals(lance.getUsuario())
        lances.add(lance);

    }
```

Se rodarmos o teste agora, ele passa! Nosso código funciona! Mas veja que o código que produzimos não está muito claro. O if em particular está confuso. Agora é uma boa hora para melhorar isso, afinal temos certeza de que o código atual funciona! Ou seja, se melhorarmos o código e rodarmos o teste novamente, ele **deverá continuar verde**.

Vamos, por exemplo, extrair o código responsável por pegar o último elemento da lista em um método privado:

```
    public void propoe(Lance lance) {
        if (lances.isEmpty() || !ultimoLanceDado().getUsuario().equals(lance.getUsuario())) {
            lances.add(lance);
        }
    }

    private Lance ultimoLanceDado() {
        return lances.get(lances.size() - 1);
    }
```

Perfeito. Vamos para a próxima regra de negócio: um usuário só pode dar no máximo 5 lances para um mesmo leilão. De novo, começaremos pelo teste. Vamos criar um leilão e fazer um usuário dar 5 lances nele. Repare que, devido à regra anterior, precisaremos intercalá-los, já que o mesmo usuário não pode fazer dois lances em sequência:

```java
@Test
    public void naoDeveAceitarMaisDoQue5LancesDeUmMesmoUsuario() {
        Leilao leilao = new Leilao("Macbook Pro 15");
        Usuario steveJobs = new Usuario("Steve Jobs");
        Usuario billGates = new Usuario("Bill Gates");

        leilao.propoe(new Lance(steveJobs, 2000));
        leilao.propoe(new Lance(billGates, 3000));
        leilao.propoe(new Lance(steveJobs, 4000));
        leilao.propoe(new Lance(billGates, 5000));
        leilao.propoe(new Lance(steveJobs, 6000));
        leilao.propoe(new Lance(billGates, 7000));
        leilao.propoe(new Lance(steveJobs, 8000));
        leilao.propoe(new Lance(billGates, 9000));
        leilao.propoe(new Lance(steveJobs, 10000));
        leilao.propoe(new Lance(billGates, 11000));

// deve ser ignorado
        leilao.propoe(new Lance(steveJobs, 12000));

        assertEquals(10, leilao.getLances().size());

        int ultimo = leilao.getLances().size() - 1;
        Lance ultimoLance = leilao.getLances().get(ultimo);

        assertEquals(11000.0, ultimoLance.getValor(), 0.00001);
    }
```

Ao rodarmos, o teste falhará! Excelente, vamos agora fazê-lo passar escrevendo o código mais simples:

```java
    public void propoe(Lance lance) {
        int total = 0;
        for (Lance l : lances) {
            if (l.getUsuario().equals(lance.getUsuario())) {
                total++;
            }
        }
        if (lances.isEmpty() || (!ultimoLanceDado().getUsuario().equals(lance.getUsuario()) && total < 5)) {
            lances.add(lance);
        }
    }
```

Pronto! O teste passa! Mas esse código está claro o suficiente? Podemos melhorar! De novo, uma ótima hora para refatorarmos nosso código. Vamos extrair a lógica de contar o número de lances de um usuário em um método privado:

```java
    public void propoe(Lance lance) {
        if (lances.isEmpty() || (!ultimoLanceDado().getUsuario().equals(lance.getUsuario())
                && qtdDelancesDo(lance.getUsuario()) < 5)) {
            lances.add(lance);
        }

    private int qtdDelancesDo(Usuario usuario) {
        int total = 0;
        for (Lance lance : lances) {
            if (lance.getUsuario().equals(usuario)) {
                total++;
            }
        }
        return total;
    }
```

Rodamos o teste, e tudo continua passando. Podemos melhorar ainda mais o código, então vamos continuar refatorando. Dessa vez, vamos extrair aquela segunda condição do if para que a leitura fique mais clara:

```java
    public void propoe(Lance lance) {
        if (lances.isEmpty() || podeDarLance(lance.getUsuario())) {
            lances.add(lance);
        }
    }

    private boolean podeDarLance(Usuario usuario) {
        return !ultimoLanceDado().getUsuario().equals(usuario)
                && qtdDelancesDo(usuario) < 5;
    }
```

Pronto! Os testes continuam passando! Poderíamos continuar es- crevendo testes antes do código, mas vamos agora pensar um pouco sobre o que acabamos de fazer.

3. Test-Driven Development

Em primeiro lugar, escrevemos um teste; rodamos e o vimos falhar; em seguida, escrevemos o código mais simples para passar o teste; rodamos novamente, e dessa vez ele passou; por fim, refatoramos nosso código para que ele ficasse melhor e mais claro.

Esse ciclo de desenvolvimento, onde escrevemos um teste antes do código, é conhecido por Test-Driven Development. A popularidade da prática de TDD tem crescido cada vez mais entre os desenvolvedores, uma vez que ela traz diversas vantagens:

- Se sempre escrevermos o teste antes, garantimos que todo nosso código já “nasce” testado;
- Temos segurança para refatorar nosso código, afinal sempre refatoraremos com uma bateria de testes que garante que não quebraremos o comportamento já existente;
- Como o teste é a primeira classe que usa o seu código, você natural- mente tende a escrever código mais fácil de ser usado e, por consequência, mais fácil de ser mantido.
- Efeitos no design. É comum percebermos que o projeto de classes de sistemas feitos com TDD é melhor do que sistemas feitos sem TDD.

4. Efeitos no design de classes

Muitos desenvolvedores famosos, como Kent Beck, Martin Fowler e Michael Feathers, dizem que a prática de TDD faz com que seu design de classes melhore. A grande pergunta é: como?

Em alto nível, a ideia é que, quando você começa pelo teste, você escreve naturalmente um código que é mais fácil de ser testado. E um código mais fácil de ser testado apresenta algumas características interessantes:

- Ele é mais coeso. Afinal, um código que faz muita coisa é mais difícil de ser testado.
- Ele é menos acoplado. Pois testar código acoplado é mais difícil.
- A interface pública é simples. Você não quer invocar 10 métodos para conseguir testar o comportamento.
- As precondicões são mais simples. Você também não quer montar imensos cenários para testar o método.

Por esses motivos, ao escrever classes que favoreçam a testabilidade, você naturalmente está criando classes mais simples e mais bem desenhadas.

4.1. Devo testar métodos privados?

A resposta é direta: não. Se você sente vontade de testar aquele método privado isolado do resto, é o teste lhe dizendo que esse código está no lugar errado. Nesses casos, extraia esse trecho de código para uma classe específica, e teste-a naturalmente. Lembre-se: se está difícil testar, é porque você pode fazer melhor.

5. Baby steps

Baby steps é o nome que damos à ideia de sempre tentarmos escrever o código mais simples que faz o teste passar, e começar pelo cenário de teste mais simples naquele momento. Dar esses passos pequenos pode ser muito benéfico para a sua implementação. Começar pelo teste mais simples nos possibilita evoluir o código aos poucos (geralmente gostamos de começar pelo caso mais difícil, o que pode dificultar).

Além disso, ao implementar códigos simples, aumentamos a facilidade de manutenção do nosso código. Afinal, código simples é muito mais fácil de se manter do que códigos complexos. Muitas vezes nós, programadores, escrevemos códigos complicados desnecessariamente. TDD nos lembra o tempo todo de ser simples.

Muitas pessoas, no entanto, dizem que tomar passos de bebê o tempo todo pode ser contraproducente. Segundo o próprio autor da prática, Kent Beck, você deve tomar passos pequenos sempre que sua “confiança sobre o código” estiver baixa. Se você está trabalhando em um trecho de código e está bastante confiante sobre ele, você pode dar passos um pouco maiores. Mas cuidado: passos grandes não devem ser a regra, e sim a exceção.

5.1. Cuidado com trechos simples

Sempre que escrevemos um trecho de código, achamos que ele é simples. Afinal, nós o escrevemos, ele está inteiro na nossa cabeça. Não há dúvidas. Mas quantas vezes não voltamos a um código que escrevemos 1 mês atrás e não entendemos nada? A única maneira de trabalhar com qualidade e segurança em códigos assim é tendo uma bateria de testes que garanta qualquer mudança feita.

Portanto, não se deixe enganar. Se o método contém uma regra de negócio, teste-o. Você agradecerá no futuro.

6. Devo ver o teste falhar?

Devemos ver o teste falhar, pois é uma das maneiras que temos para garantir que nosso teste foi implementado corretamente. Afinal, um teste automatizado é código, e podemos implementá-lo incorretamente.

Ao ver que o teste que esperamos falhar realmente falha, temos a primeira garantia de que o implementamos de maneira correta. Imagine se o teste que esperamos que falhe na prática não falha. O que aconteceu?

Para completar o “teste” do teste, podemos escrever o código de produção mais simples que o faz passar. Dessa forma, garantimos que o teste fica vermelho ou verde quando deve.

7. TDD 100% do tempo?

Não. Como toda prática de engenharia de software, ela deve ser usada no mo- mento certo. TDD faz muito sentido ao implementar novas funcionalidades, ao corrigir bugs, ao trabalhar em códigos complexos etc.

Mas às vezes não precisamos seguir o fluxo ideal da prática de TDD. Por exemplo, às vezes queremos só escrever um conjunto de testes que faltaram para determinada funcionalidade. Nesse momento, não faríamos TDD, mas sim escreveríamos testes.

Em códigos extremamente simples, talvez a prática de TDD não seja necessária. Mas lembre-se: cuidado para não achar que “tudo é simples”, e nunca praticar TDD.

8. Referência Bibliográfica

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