01

Cuidando dos seus testes

Transcrição

Nesse momento, já temos um número considerável de testes em nosso sistema. E o melhor, nossa bateria de testes só tende a crescer! Com isso, teremos mais segurança e qualidade na manutenção e evolução do nosso código.

Só que antes de continuarmos a escrever testes, precisamos nos atentar em um ponto importante: teste é código. E código mal escrito é difícil de ser mantido, atrapalhando o desenvolvimento! Com isso em mente, vamos verificar a nossa bateria de testes até o momento. Olhe a classe `AvaliadorTest`, por exemplo, em 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 omitido

        // 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.

Veja que, 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.

```
public class AvaliadorTest {

    private Avaliador leiloeiro;

    @Before
    public void criaAvaliador() {
        this.leiloeiro = new Avaliador();
    }
```

Ao rodarmos essa bateria de testes, o JUnit executará o método `criaAvaliador()` 3 vezes: uma vez antes de cada método de teste!

Veja que todos os nossos testes criam usuários, então podemos fazê-lo no método `criaAvaliador()`, para que os nossos métodos de teste em si, que são o que realmente importam na bateria de testes, fiquem cada vez mais simples. Vamos pegar o conjunto de usuários do primeiro teste, por exemplo, mover para este método e colocá-los como atributo da classe, assim todos os métodos os enxergarão:

```
@Before
public void criaAvaliador() {
    this.leiloeiro = new Avaliador();
    this.joao = new Usuario("João");
    this.jose = new Usuario("José");
    this.maria = new Usuario("Maria");
}
```

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`.

Podemos melhorar ainda mais nosso código de teste. Veja que criar um `Leilao` não é uma tarefa fácil nem simples de ler. E veja 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:

```
@Test
public void deveEncontrarOsTresMaioresLances() {
    Leilao leilao2 = 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);
}
```

Veja como esse código é mais fácil de ler, e mais enxuto que o anterior! E escrever a classe `CriadorDeLeilao` é razoavelmente simples! O único segredo talvez seja possibilitar com 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;
    }
}
```

Com isso, podemos descartar a versão anterior e utilizar o `CriadorDeLeilao()`, voltar `Leilao leilao2` para `Leilao leilao` e então rodar o teste novamente.

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.

Cuide dos seus códigos de teste. Refatore-os constantemente. Lembre-se: uma bateria de testes mal escrita pode deixar de ajudar e começar a te atrapalhar!