02

Testes de unidade com JUnit

Download

Caso queira começar o treinamento a partir desse vídeo, pode baixar o projeto aqui (https://s3.amazonaws.com/caelum-online-public/auron/auron-stage5.zip). Só baixe este arquivo se não tiver feito os exercícios dos capítulos anteriores.

A classe Sorteador

No capítulo anterior preparamos o formulário do sorteio. Agora vamos implementar a lógica e garantir pelos testes que ela funciona. Lembrando que no nosso modelo, temos 3 classes: `Sorteio`, `Participante` e `Par`, este último, representa o relacionamento entre um sorteio e o participante.

Dentro do método `sortear()` da classe `SorteioBean` vamos precisar de uma lista de participantes. Mais para frente carregaremos essa lista do banco, mas isso não é nossa preocupação nesse momento. Nossa tarefa agora é gerar os pares para o sorteio a ser cadastrado.

Como a lógica é um pouco mais elaborada, vamos criar uma classe dedicada e com essa responsabilidade. Vamos chamá-la de `Sorteador`. O construtor receberá o `sorteio`, que já existe como atributo na classe `SorteioBean`, e a lista de participantes. O método que realmente irá gerar os pares chamaremos de `sortear`. Por fim nossa classe ficará com a seguir, mas ainda sem a implementação do método sortear.

```
package br.com.caelum.auron.model;

import java.util.List;

public class Sorteador {

    private List<Participante> participantes;
    private int totalDeParticipantes;

    public Sorteador(Sorteio sorteio, List<Participante> participantes) {
        this.participantes = participantes;
        this.sorteio = sorteio;
    }

    public void sortear() {
        //aqui vamos gerar os pares
    }
}
```

Primeiro teste com JUnit

Para ter certeza que essa lógica funciona, vamos criar um teste primeiro. O objetivo é garantir que a classe sorteará corretamente os pares para um sorteio.

Vamos criar um *Test Case* para classe `Sorteador` pelo Eclipse. O *Test Case* se chamará `SorteadorTest` e ficará no mesmo pacote da classe `Sorteador`. Uma coisa importante é que a pasta onde ficará o teste será `src/test/java` e não

`src/main/java`. Desta forma o teste fica separado da classe a ser testada. Não esqueça definir a *Class under Test* que é o nosso `Sorteador`. No formulário, já vamos marcar o método `setup` para ser criado automaticamente. Pronto, já temos uma classe teste preparada e pré-configurada:

```
package br.com.caelum.auron.modelo;

import static org.junit.Assert.fail;

public class SorteadorTest {

    @Before
    public void setup() {
    }

    @Test
    public void test() {
        fail("Not yet implemented");
    }
}
```

Para gerarmos os pares vamos precisar de uma lista de participantes, por isso iremos ter três atributos do tipo `Participante` na classe do teste. A criação dos participantes é feita no método `setup` pois este método é executado antes de cada teste. Assim cada teste terá a sua nova lista de participantes. Isso é importante pois os cenários dos testes sempre devem ser executados isoladamente. Vamos guardar os participantes em uma lista. Criaremos um atributo do tipo `java.util.List` e adicionaremos os participantes na lista dentro do método `setup`:

```
public class SorteadorTest {

    private Participante p1;
    private Participante p2;
    private Participante p3;

    private List<Participante> participantes;

    @Before
    public void setup() {
        p1 = new Participante();
        p1.setNome("Leonardo");
        p2 = new Participante();
        p2.setNome("Nico");
        p3 = new Participante();
        p3.setNome("Fábio");

        participantes = Arrays.asList(p1, p2, p3);

    }

    //...
}
```

O primeiro caso de teste será bem simples e verificará se a quantidade de pares sorteados é igual a quantidade dos participantes. O nome de método será `aQuantidadeDeParesEParticipantesDeveSerAMesma()`. É fundamental que o nome

do teste descreva bem o que estamos testando. Assim a intenção do teste fica documentada além de facilitar o entendimento do mesmo quando falha:

```
@Test
public void aQuantidadeDeParesEParticipantesDeveSerAMesma() {

}
```

Para descobrir a quantidade de pares vamos usar o sorteio e para isso será necessário inicializa o objeto `Sorteio`. Faremos isso de forma semelhante ao que fizemos nos atributos anteriores:

```
public class SorteadorTest {

    //outros atributos
    private Sorteio sorteio;

    @Before
    public void setup() {
        //criação dos participantes
        sorteio = new Sorteio();
    }

    //...
}
```

Uma vez criado podemos chamar `sorteio.getPares().size()` e guardar o retorno em uma variável auxiliar.

```
@Test
public void aQuantidadeDeParesEParticipantesDeveSerAMesma() {
    int quantidadeDePares = sorteio.getPares().size();

}
```

Ao Invés de descobrir a quantidade pela lista de pares, podíamos perguntar diretamente para o sorteio e assim melhorar a legibilidade do teste. Um código fácil de ser entendido é o que sempre buscamos em um projeto de software. Vamos então mudar para: `sorteio.getQuantidadeDePares()`.

```
@Test
public void aQuantidadeDeParesEParticipantesDeveSerAMesma() {
    int quantidadeDePares = sorteio.getQuantidadeDePares();
}
```

Como não temos esse método na classe `Sorteio` o Eclipse reclama. Podemos pedir o auxílio dele na criação do método e na implementação chamaremos o método `size()` da coleção de pares como já vimos antes.

```
public class Sorteador {

       // atributos e construtor omitidos

    public void sortear() {
```

```
        int totalDeParticipantes = participantes.size();

    }
}
```

Como já mencionamos, o número de pares formados deve ser equivalente ao de participantes. Antes de fazer a verificação, vamos chamar o método `sortear()` de um objeto `Sorteador` como a seguir:

```
@Test
public void aQuantidadeDeParesEParticipantesDeveSerAMesma() {
    int quantidadeDePares = sorteio.getQuantidadeDePares();
    int quantidadeDeParticipantes = participantes.size();

    Sorteador sorteador = new Sorteador(sorteio, participantes);
    sorteador.sortear();
}
```

Depois de executarmos a lógica, podemos verificar o resultado. Para isso, usaremos a classe `Assert` do JUnit. Ela possui métodos auxiliares para verificar a igualdade.

```
@Test
public void aQuantidadeDeParesEParticipantesDeveSerAMesma() {
    int quantidadeDePares = sorteio.getQuantidadeDePares();
    int quantidadeDeParticipantes = participantes.size();

    Sorteador sorteador = new Sorteador(sorteio, participantes);
    sorteador.sortear();

    Assert.assertTrue(quantidadeDePares == quantidadeDeParticipantes);
}
```

Pronto, o primeiro teste foi criado. Podemos executá-lo dentro do Eclipse indo no menu *Run As -> JUnit Test*. Após execução é exibido o relatório do JUnit em vermelho. Claro que o teste falharia, pois a lógica do método `sortear` ainda não foi implementada por completo.

Lógica para sortear pares

Antes de implementar a lógica vamos entender como ela deveria funcionar. Tendo 3 participantes, por exemplo, Leo, Nico e Fábio devemos criar 3 pares. Vamos simular que Leo tenha tirado Nico, ou seja, o primeiro e segundo participante são um par. O segundo par será o segundo e terceiro, ou seja, Nico tirou o Fábio. Já o último par é composto do terceiro e primeiro participante, Fábio e Leo respectivamente. Para ficar mais claro temos os seguintes pares: (1º-> 2º) , (2º -> 3º) e (3º -> 1º).

Repare que sempre usaremos o participante atual e o próximo para definir um par, exceto quando chegamos ao último. Aí o próximo participante é o primeiro da lista.

Vamos começar a implementar essa lógica e guardar o índice do participante atual em uma variável auxiliar. Para iterar com a lista de participantes vamos guardar o total de participante dentro de mais uma variável auxiliar. O laço será feito através de um while: *enquanto o índice atual é menor do que o total continuamos executando*:

```
public void sortear() {
    int indiceAtual = 0;
    int totalDeParticipantes = participantes.size();

    while(indiceAtual < totalDeParticipantes) {
        //continuaremos aqui
    }
}
```

Em cada iteração vamos criar um `Par` que recebe no construtor o participante atual (*índice atual*) e o próximo (*índice atual + 1*), além do sorteio. O participante atual representa o amigo e o próximo representa o amigo oculto.

Para fechar o relacionamento bidirecional e manter a consistência passaremos o par criado ao sorteio. No fim do laço vamos incrementar o índice atual:

```
public void sortear() {
    int indiceAtual = 0;
    int totalDeParticipantes = participantes.size();

    while(indiceAtual < totalDeParticipantes) {
        Par par = new Par(participantes.get(indiceAtual), participantes.get(indiceAtual + 1), s
        sorteio.adicionaPar(par);

        indiceAtual++;
    }
}
```

Já escrevemos uma boa parte da lógica, vamos testá-lo para vermos se estamos com o código do método `sortear` completo. Ao executar o teste um `ArrayIndexOutOfBoundException` é lançado. Esta exceção aconteceu pois não tratamos aquele caso especial quando é preciso gerar o último par da lista, pegando o último e primeiro participantes. Da maneira como desenvolvemos o `Sorteador` quando o último par está sendo formado é buscado um elemento maior do que o tamanho da lista, ou seja, a instrução `participantes.get(indiceAtual + 1)` falha!

Logo, voltando ao método `sortear()` é preciso verificar em cada iteração se chegamos ao fim da lista. Vamos então adicionar no início do laço um *if* que verifica se o *índice atual é igual ao total de participantes – 1*. Se essa condição for verdadeira devemos criar um novo par baseado no último e primeiro elemento da lista:

```
public void sortear() {
    int indiceAtual = 0;
    int totalDeParticipantes = participantes.size();

    while(indiceAtual < totalDeParticipantes) {

        if(indiceAtual == totalDeParticipantes -1) {
            Par par = new Par(participantes.get(indiceAtual), participantes.get(0), sorteio);
            sorteio.adicionaPar(par);
        }

        Par par = new Par(participantes.get(indiceAtual), participantes.get(indiceAtual + 1), s
        sorteio.adicionaPar(par);

        indiceAtual++;
```

```
    }
}
```

Um teste pode falhar

Aparentemente o nosso código está correto, hora de verificar pelo teste. No entanto, ao executar, permanece a exceção. Faltou mais uma coisa! Quando geramos o último par dentro do if não devemos continuar com a iteração e sim sair do laço. Para tal basta colocar o comando *break* no fim do *if*.

```
public void sortear() {
    int indiceAtual = 0;
    int totalDeParticipantes = participantes.size();

    while(indiceAtual < totalDeParticipantes) {

        if(indiceAtual == totalDeParticipantes -1) {
            Par par = new Par(participantes.get(indiceAtual), participantes.get(0), sorteio);
            sorteio.adicionaPar(par);
                        break;
        }

        ....
    }
}
```

Vamos testar mais uma vez. Repare que este ciclo é o jogo tradicional no Test Driven Development, o TDD, implementar uma parte da lógica e testar, sempre caminhando em pequenos passos que podem ser verificados pelos testes. Procurar o feedback do teste o mais rápido possível é fundamental no TDD.

O resultado do JUnit mudou mais ainda não é o verde que desejamos. Repare que o JUnit relata que agora aconteceu uma falha e não um erro. Isso significa que o teste executou sem exceção mas a verificação final falhou.

Aconteceu algo comum no desenvolvimento de testes. Um teste pode falhar porque a lógica está errada, mas também porque o teste foi escrito errado! Talvez você já tenha percebido o problema no teste ou tenha estranhado quando o escrevemos.

Repare que bem no início do teste chamamos o método `getQuantidadeDePares()` da classe `Sorteio`, ou seja antes de sortear os pares!

O sorteio só recebe os pares através da nossa lógica dentro do método `sortear()`. Devemos chamar o método `getQuantidadeDePares()` após a execução da lógica:

```
@Test
public void aQuantidadeDeParesEParticipantesDeveSerAMesma() {

    int quantidadeDeParticipantes = participantes.size();

    Sorteador sorteador = new Sorteador(sorteio, participantes);
    sorteador.sortear();

    //só depois de chamado sortear
```

```
        int quantidadeDePares = sorteio.getQuantidadeDePares();
        Assert.assertEquals(quantidadeDePares, quantidadeDeParticipantes);
    }
```

Vamos executar de novo o teste. Finalmente, o relatório mostra o tão desejado verde. No próximo capítulo vamos escrever mais testes, mas primeiro precisamos passar pelos exercícios.