02

Mais testes

Download

Caso queira começar o treinamento a partir desse vídeo, pode baixar o projeto aqui (https://s3.amazonaws.com/caelum-online-public/auron/auron-stage6.zip). **Só baixe este arquivo se não tiver feito os exercícios dos capítulos anteriores.**

No último capítulo começamos a escrever testes, mas estamos longe de garantir que a nossa lógica realmente funcione como esperado. Nesse capítulo criaremos mais testes para verificar a funcionalidade do nosso *sorteador*.

Para que a lógica do nosso sorteador funcione, precisamos ter no mínimo dois participantes. Para verificarmos esta condição, criaremos um novo teste chamado `naoDeveAceitarUmaListaComMenosDeDoisParticipantes()`. Lembrando que o nome do teste deve deixar bem clara sua intenção e não é incomum nome de métodos extensos como o que acabamos de criar. Por fim, nossos testes servem como documentação da aplicação:

```
@Test
public void naoDeveAceitarUmaListaDeParticipantesComMenosDeDoisParticipantes() {

}
```

Neste método criaremos um sorteador como já fizemos antes. Em seguida, chamaremos o método `sortear()`. Agora vem a pergunta: qual será o comportamento do método ao receber uma lista com menos de dois participantes?

Queremos que o método `sortear()` lance uma `Exception`. Deixaremos isso claro através do atributo `expected` da anotação `@Test`, indicando ao JUnit que esperamos o lançamento de uma exceção. Este atributo recebe o `class` da exceção. Vamos chamá-la de `SorteioException`.

```
@Test(expected=SorteioException.class)
public void naoDeveAceitarUmaListaDeParticipantesComMenosDeDoisParticipantes() throws SorteioEx
    Sorteador sorteador = new Sorteador(sorteio, new ArrayList());
    sorteador.sortear();
}
```

Também devemos adicionar um `throws` indicando que esse método lançará uma `SorteioException`, porém ainda não temos essa classe. Portanto, vamos cria-la e fazer com que herde de `Exception`.

Repare que estamos criando uma exceção do tipo *checked*. Chamaremos também o construtor da classe mãe através da palavra-chave `super` para que possamos passar uma mensagem para nossa *exception*.

```
public class SorteioException extends Exception {
    public SorteioException() {
        super("Por favor, insira dois ou mais participantes na lista.");
    }
}
```

Vamos rodar nossos testes. Podemos ver que este último que escrevemos não passa, pois ainda não lançamos a exceção dentro do método `sortear()`.

Repare que nossa lógica exige pelo menos dois participantes para que o sorteio seja realizado. Portanto, ao verificarmos que nossa lista não possui esta quantidade, lançaremos uma `SorteioException`.

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

        if(totalDeParticipantes < 2)
             throw new SorteioException();

...
}
```

Para isso é necessário também colocarmos um `throws` no método.

```
public void sortear() throws SorteioException {
       ...
}
```

Ao adicionarmos a exceção *checked* na assinatura do método, nosso projeto para de compilar. Vamos primeiro na classe `SorteidorBean`. No método que usa o `Sorteador` faremos um `try-catch` podendo até mesmo usar o auxílio do Eclipse.

```
@Named
@RequestScoped
public class SorteioBean {
        ...
    public void sortear() {
        List<Participante> participantes = new ArrayList<>();

        try {
            Sorteador sorteador = new Sorteador(sorteio, participantes);
            sorteador.sortear();
        } catch (SorteioException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

Não se preocupe quanto a mensagem pois mais a frente iremos colocá-la mais amigável ao usuário. Outro problema de compilação é no primeiro teste, `aQuantidadeDeParesEParticipantesDeveSerAMesma()` que fizemos no capítulo anterior. Como também chamamos o método `sortear()` colocaremos um throws no cenário de teste, porque não estamos interessados no tratamento de erro:

```
@Test
public void aQuantidadeDeParesEParticipantesDeveSerAMesma() throws SorteioException {
        int quantidadeDeParticipantes = participantes.size();
```

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

        int quantidadeDePares = sorteio.getQuantidadeDePares();

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

Se tudo está compilando, não podemos esquecer de rodar novamente nossos testes e verificar se todos passam. Como todos os testes passam, então podemos continuar.

Nosso próximo teste será parecido com o que já fizemos. Verificaremos se uma determinada lista passada para o construtor é nula. Para isso, vamos criar um cenário de teste e chamá-lo de `naoDeveAceitarUmaListaDeParticipantesNula()`.

Também devemos indicar ao JUnit que esperamos o lançamento de `SorteioException`. Para isso, usaremos o atributo `expected` da anotação `@Test` como fizemos anteriormente. Como este método lançará uma `SorteioException`, colocaremos um `throws` em sua assinatura. A implementação do teste é bem simples, criamos um `Sorteio` que recebe uma lista de participante nula.

```
@Test(expected=SorteioException.class)
public void naoDeveAceitarUmaListaDeParticipantesNula() throws SorteioException {
          Sorteador sorteador = new Sorteador(sorteio, null);
}
```

Ao rodarmos este teste, perceberemos que ele falhará. Pois a exceção que o JUnit esperava não foi lançada já que não fizemos essa condição no nosso construtor. Portanto, vamos lançar uma `SorteioException` no construtor da classe `Sorteador` caso a lista de participantes passada como argumento seja *nula*:

```
public Sorteador(Sorteio sorteio, List<Participante> participantes) throws SorteioException {
        if(participantes == null)
            throw new SorteioException();

        this.sorteio = sorteio;
        this.participantes = participantes;
        totalDeParticipantes = participantes.size();
    }
```

Só que temos um problema: a mensagem padrão da nossa `exception` foi definida dentro dela, e não pode ser alterada:

```
public class SorteioException extends Exception {
    public SorteioException() {
        super("Por favor, insira dois ou mais participantes na lista.");
    }
}
```

Certamente não é isso o que queremos. Vamos melhorar a classe `SorteioException` criando um novo construtor que receberá a mensagem que será associada à exceção.

```
public class SorteioException extends Exception {
    public SorteioException(String mensagem) {
        super(mensagem);
    }
}
```

Agora, podemos passar a mensagem através deste construtor:

```
public Sorteador(Sorteio sorteio, List<Participante> participantes) throws SorteioException {
        if(participantes == null)
            throw new SorteioException("Por favor, insira uma lista de participantes!");

        this.sorteio = sorteio;
        this.participantes = participantes;
        totalDeParticipantes = participantes.size();
}
```

Não podemos esquecer de passar a mensagem também no método `sortear()` da classe `Sorteador`, já que agora o construtor do `SorteioException` espera receber uma String:

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

        if(totalDeParticipantes < 2)
             throw new SorteioException("Por favor, insira dois ou mais participantes na lista.'
        ....
}
```

Ao executar todos os testes percebemos que não temos nenhum problema mais.

Outras regras de negócio

Lembrando da nossa regra de negócios: sabemos que cada participante **terá** um amigo oculto e **será** amigo oculto de outro participante. Ou seja, um participante não pode ter dois amigos ocultos e nem ser amigo oculto de dois participantes.

Garantiremos que não possuímos essa falha em nosso algoritmo implementando mais um teste. Ele verificará se um participante é **amigo** em mais de um `Par`, caso isso ocorra, o teste deve falhar.

Para isso, criaremos o método `naoDeveRepetirUmAmigo()`.

```
@Test
public void naoDeveRepetirUmAmigo() throws SorteioException {

}
```

Vamos instanciar nossa classe `Sorteador` passando em seu construtor o atributo `sorteio` e nossa lista de `participantes`.

```
@Test
public void naoDeveRepetirUmAmigo() throws SorteioException {
      Sorteador sorteador = new Sorteador(sorteio, participantes);
      sorteador.sortear();
}
```

Precisamos ter em mãos, o conjunto de pares que foram criados pelo sorteador. Para isso, faremos:

```
@Test
public void naoDeveRepetirUmAmigo() throws SorteioException {
      Sorteador sorteador = new Sorteador(sorteio, participantes);
      sorteador.sortear();

      Set<Par> pares = sorteio.getPares();
}
```

Vamos usar a classe `Iterator` para pegarmos os pares desse conjunto. O `iterator` possui um método `next` que devolve sempre o próximo elemento. Como sabemos que devem existir 3 pares, vamos chamar 3 vezes `next` .

```
@Test
public void naoDeveRepetirUmAmigo() throws SorteioException {
    Sorteador sorteador = new Sorteador(sorteio, participantes);
    sorteador.sortear();

    Set<Par> pares = sorteio.getPares();
    Iterator<Par> it = pares.iterator();

    Par par = it.next();
    Par par2 = it.next();
    Par par3 = it.next();
}
```

Com cada par em mãos vamos recuperar o amigo desse par, para isso será necessário gerar os getters e setters da classe `Par`

Por fim, podemos fazer a verificação do teste e afirmar que nenhum amigo deve ser igual ao outro. Usaremos o método `assertFalse` para garantir que os amigos não se repetem, ou seja, são diferentes. Vamos verificar se o primeiro amigo é diferente do segundo, o segundo é diferente do terceiro e o terceiro diferente do primeiro.

```
@Test
public void naoDeveRepetirUmAmigo() throws SorteioException {
    Sorteador sorteador = new Sorteador(sorteio, participantes);
    sorteador.sortear();

    Set<Par> pares = sorteio.getPares();
    Iterator<Par> it = pares.iterator();

    Par par = it.next();
    Par par2 = it.next();
    Par par3 = it.next();
```

```
        Participante amigo1 = par.getAmigo();
    Participante amigo2 = par2.getAmigo();
    Participante amigo3 = par3.getAmigo();

    assertFalse(amigo1.equals(amigo2));
    assertFalse(amigo2.equals(amigo3));
    assertFalse(amigo3.equals(amigo1));
}
```

Ao rodar os testes, ele passa. Garantimos então que nenhum amigo se repete em um par.

Novamente analisando nosso algoritmo, percebemos que um participante não pode ser seu próprio amigo oculto. Garantiremos que isso não aconteça implementando o teste: `deveVerificarSeAmigoEDiferenteDoAmigoOculto()` :

```
@Test
public void deveVerificarSeAmigoEDiferenteDoAmigoOculto() throws SorteioException {
    Sorteador sorteador = new Sorteador(sorteio, participantes);
    sorteador.sortear();

    Set<Par> pares = sorteio.getPares();
    Iterator<Par> it = pares.iterator();

    Par par = it.next();
    Par par2 = it.next();
    Par par3 = it.next();

    Participante amigo1 = par.getAmigo();
    Participante amigoOculto1 = par.getAmigoOculto();

    Participante amigo2 = par2.getAmigo();
    Participante amigoOculto2 = par2.getAmigoOculto();

    Participante amigo3 = par3.getAmigo();
    Participante amigoOculto3 = par3.getAmigoOculto();

    assertFalse(amigo1.equals(amigoOculto1));
    assertFalse(amigo2.equals(amigoOculto2));
    assertFalse(amigo3.equals(amigoOculto3));
}
```

Vamos rodar novamente o teste. Ele também passa, então sabemos que nosso algoritmo sempre sorteia amigo e amigo oculto diferentes. Temos um caso específico do nosso algoritmo: quando o loop do método `sortear` chega ao último participante da lista. Quem será seu amigo oculto? O primeiro participante!

Ou seja, devemos garantir que o amigo oculto do último participante, é sempre o primeiro.

Para isso, iremos implementar o teste `deveVerificarSeOAmigoOcultoDoUltimoParEhOAmigoDoPrimeiroPar` .

```
@Test
public void deveVerificarSeOAmigoOcultoDoUltimoParEhOAmigoDoPrimeiroPar() throws SorteioExceptio
    Sorteador sorteador = new Sorteador(sorteio, participantes);
    sorteador.sortear();

    Set<Par> pares = sorteio.getPares();
```

```
        Iterator<Par> it = pares.iterator();
        Par par1 = it.next();
        it.next();
        Par par3 = it.next();

        Participante primeiro = par1.getAmigo();
        Participante ultimo = par3.getAmigoOculto();

        assertTrue(ultimo.equals(primeiro));

    }
```

Vamos rodar nossos testes e verificar se todos passam. No próximo capítulo, vamos utilizá-los para fazermos algumas *refatorações* em nosso sistema. E com isso, garantir que não quebraremos nosso código nosso código já escrito. Até lá!