02

Preparando o teste

Transcrição

Imagine que agora o nosso gestor quer que ao entrar na parte administrativa do nosso sistema, este consiga de alguma forma ver algo como um relatório que exibem o valor total de todos os produtos separados por tipo de preço (*Ebook*, *Impresso* e *Combo*).

O primeiro passo para conseguirmos fazer com que isso seja possível é pedir ao banco de dados a soma do valor de todos os produtos por tipo de preço. Na classe `ProdutoDAO` criaremos um novo método chamado `somaPrecosPorTipo` que retornará um objeto do tipo `BigDecimal` .

Este método deve receber o tipo de preço por parametro, realizar a consulta ao banco de dados e retornar o resultado, isso através do objeto `manager` , como já feito anteriormente.

```
public BigDecimal somaPrecosPorTipo(TipoPreco tipoPreco){
    TypedQuery<BigDecimal> query = manager.createQuery("select sum(preco.valor) from Produto p j
    query.setParameter("tipoPreco", tipoPreco);
    return query.getSingleResult();
}
```

Mas quem garante que este código funciona e que o resultado do mesmo esteja correto? No mínimo, teríamos que criar uma nova página de relatórios e verificar manualmente o resultado e assim validar se o código funciona como esperado ou não.

Para validar o funcionamento de determinado código, precisamos de testes. Existem duas formas básicas de se testar código, uma delas foi comentarada anteriormente e é conhecida como teste manual, onde precisamos de alguém para verificar manualmente os resultados gerados pelo código em algum lugar.

A segunda forma, mais prática é que sempre funcionará por meio de testes automatizados. Falando de forma mais objetiva, escreveremos um código que testa se o nosso código funciona corretamente usando a comparação de resultados esperado.

Observação: Na Alura, há diversos cursos de testes em diversas tecnologias, basta usar a busca e encontrará vários destes cursos ou basta [clicar aqui para ver a lista de cursos de testes (https://cursos.alura.com.br/search?query=testes)]

Projetos **Maven** sempre contam com um **Source Folder** especifico para testes, localizado em: `src/tests/java` . Para criação dos testes para nossa aplicação, dentro deste *Source Folder* criaremos as classes de testes usando os mesmos pacotes das classes de *produção*.

Criaremos então a classe `ProdutoDAOTest` que conterá os testes relacionados a classe `ProdutoDAO` . Esta classe estará no *source folder* de testes e terá o mesmo pacote da `ProdutoDAO` : `br.com.casadocodigo.loja.daos` .

Esta classe terá um método que testa se a soma dos preços dos produtos encontrados no banco é igual a um valor esperado por nós. O método responsável por esta verificação se chamará `deveSomarTodosOsPrecosPorTipoLivro` e não terá retorno.

Este método usará a classe `ProdutoDAO` para salvar uma determinada quantidade de produtos que criaremos para cada tipo de preço. A criação dos produtos será feita com a ajuda de uma classe auxiliadora chamada `ProdutoBuilder`, o código desta classe se encontra abaixo. Copiaremos esta classe para o pacote `br.com.casadocodigo.loja.builders` dentro do *source folder tests*.

```
package br.com.casadocodigo.loja.builders;

import java.math.BigDecimal;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.List;

import br.com.casadocodigo.loja.models.Preco;
import br.com.casadocodigo.loja.models.Produto;
import br.com.casadocodigo.loja.models.TipoPreco;

public class ProdutoBuilder {

    private List<Produto> produtos = new ArrayList<>();

    private ProdutoBuilder(Produto produto) {
        produtos.add(produto);

    }

    public static ProdutoBuilder newProduto(TipoPreco tipoPreco, BigDecimal valor) {
        Produto livro = create("livro 1", tipoPreco, valor);
        return new ProdutoBuilder(livro);
    }

    public static ProdutoBuilder newProduto() {
        Produto livro = create("livro 1", TipoPreco.COMBO, BigDecimal.TEN);
        return new ProdutoBuilder(livro);
    }

    private static Produto create(String nomeLivro, TipoPreco tipoPreco, BigDecimal valor) {
        Produto livro = new Produto();
        livro.setTitulo(nomeLivro);
        livro.setDataLancamento(Calendar.getInstance());
        livro.setPaginas(150);
        livro.setDescricao("Livro top sobre testes");
        Preco preco = new Preco();
        preco.setTipo(tipoPreco);
        preco.setValor(valor);
        livro.getPrecos().add(preco);
        return livro;
    }

    public ProdutoBuilder more(int number) {
        Produto base = produtos.get(0);
        Preco preco = base.getPrecos().get(0);
        for (int i = 0; i < number; i++) {
            produtos.add(create("Book " + i, preco.getTipo(), preco.getValor()));
        }
        return this;
    }
```

```
    public Produto buildOne() {
        return produtos.get(0);
    }

    public List<Produto> buildAll() {
        return produtos;
    }
}
```

Note que o há um método que se chama `newProduto` que recebe um objeto `TipoPreco` e o valor do produto em si. Outro método a ser utilizado é o método `more` que recebe um número que indica quandos produtos queremos criar e o método `buildAll` que nos retorna a lista de produtos criados.

Dessa forma podemos fazer algo como:

```
List<Produto> livrosImpressos = ProdutoBuilder.newProduto(TipoPreco.IMPRESSO, BigDecimal.TEN).mo
```

Assim teremos uma lista com três produtos do tipo impresso, com o preço 10 em mãos. Podemos fazer o mesmo processo para a criação dos produtos do tipo `EBOOK` da seguinte forma:

```
List<Produto> livrosEbook = ProdutoBuilder.newProduto(TipoPreco.EBOOK, BigDecimal.TEN).more(3).b
```

Agora, podemos usar um laço para percorrer cada uma das listas e salvar cada um dos produtos no banco de dados com o objeto da classe `ProdutoDAO`. Usando `streams` do Java 8, teremos o seguinte resultado.

```
public void deveSomarTodosOsPrecosPorTipoLivro() {
    ProdutoDAO produtoDAO = new ProdutoDAO();

    List<Produto> livrosImpressos = ProdutoBuilder.newProduto(TipoPreco.IMPRESSO, BigDecimal.TEN
    List<Produto> livrosEbook = ProdutoBuilder.newProduto(TipoPreco.EBOOK, BigDecimal.TEN).more

    livrosImpressos.stream().forEach(produtoDAO::gravar);
    livrosEbook.stream().forEach(produtoDAO::gravar);
}
```

Observação: Caso não tenha entendido bem o `forEach` feito através do `stream`, saiba que estes são recursos do Java 8 e que você pode aprende-los no curso de Java 8: Tire proveito dos novos recursos da linguagem (https://cursos.alura.com.br/course/java8-lambdas) disponível aqui na Alura.

Agora, para que possamos realmente testar a aplicação, precisamos adicionar uma nova dependência no arquivo `pom.xml`. Esta se trata do *framework* de testes **JUnit**.

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
```

```
    <scope>test</scope>
</dependency>
```

E ao final do métoto usar a classe `Assert` do *JUnit* para verificar se o valor retornado do banco de dados é igual ao valor da soma dos produtos que temos no código. Primeiro precisamos recuperar este valor, claro!

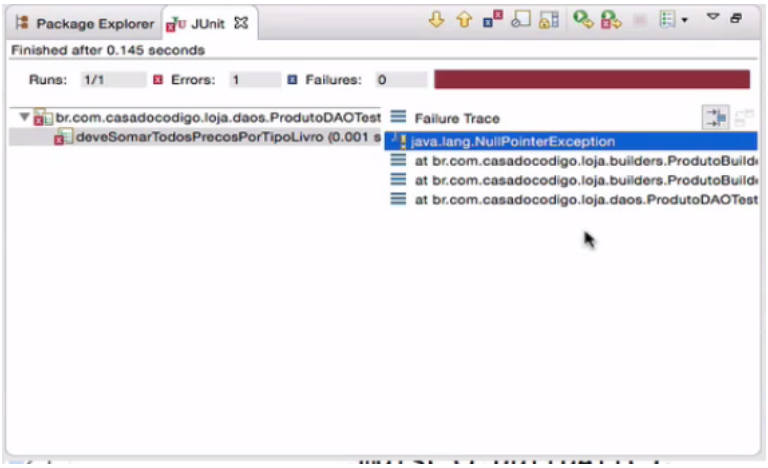
```
BigDecimal valor = produtoDAO.somaPrecosPorTipo(TipoPreco.EBOOK);
Assert.assertEquals(new BigDecimal(40).setScale(2), valor);
```

Note que estamos usando o método `somaPrecosPorTipo` da classe `ProdutoDAO` para recuperar o valor da soma dos preços dos produtos do tipo `EBOOK`. Comparando com o valor que esperamos de acordo com a lista de produtos que foi criada a partir do `ProdutoBuilder`. O `setScale(2)` simplesmente adiciona duas casas decimais ao valor `40`. Assim teremos:

```
@Test
public void deveSomarTodosOsPrecosPorTipoLivro() {
    ProdutoDAO produtoDAO = new ProdutoDAO();

    List<Produto> livrosImpressos = ProdutoBuilder.newProduto(TipoPreco.IMPRESSO, BigDecimal.TEI
    List<Produto> livrosEbook = ProdutoBuilder.newProduto(TipoPreco.EBOOK, BigDecimal.TEN).more

    livrosImpressos.stream().forEach(produtoDAO::gravar);
    livrosEbook.stream().forEach(produtoDAO::gravar);

    BigDecimal valor = produtoDAO.somaPrecosPorTipo(TipoPreco.EBOOK);
    Assert.assertEquals(new BigDecimal(40).setScale(2), valor);
}
```

Observação: Lembre-se de marcar o método com a anotação `@Test` para que o *JUnit* saiba que este método é um teste a ser executado.

Ao executarmos o códico com o *JUnit Test*, receberemos um erro mostrando que o teste falhou por causa de um `NullPointerException`.



Este erro acontece por que a classe `ProdutoDAO` usa o objeto `manager` para fazer consultas no banco de dados. Mas o objeto `manager` só é criado pelo *Spring* e dentro do contexto do *Spring*. Para solucionar o problema, teríamos que capturar o contexto do *Spring* de alguma forma, criar o objeto `manager` para só depois podermos realizar o teste. Complicado, certo? Vamos ver como faremos.