46529-06 - Projeto e Arquitetura de Software - Turma 127 - 2021/1 - Prof. Bernardo Copstein

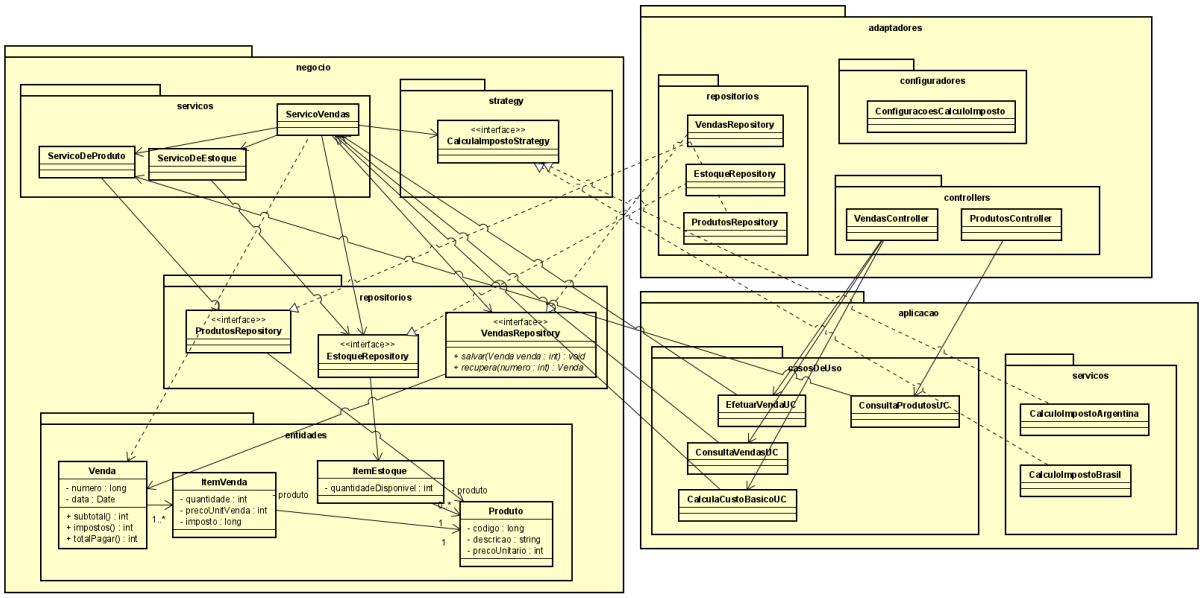
Alunos: André Marques, Fernado Elger, Gabriel Bilhar, Giovanni Frozza

Trabalho 1 - Sistema de Vendas

1. Objetivo

O objetivo do trabalho é evoluir um protótipo de backend para uma versão capaz de atender todos os requisitos solicitados, atendendo os princípios SOLID, a arquitetura CLEAN e utilizando padrões de projeto sempre que adequado.

2. Diagrama de classes



3. Padrões Utilizados

3.1 Builder

O padrão Builder é bem conhecido no Java, especialmente útil para você criar objetos com muitas opções de configurações, facilitando na visualização das classes pois o padrão Builder evita que as classes que representam os objetos fiquem muito extensas e difíceis de manter, assim ajudando em implementações futuras, não deixando o código ficar poluído e entendível. Além de suportar encadeamento em

seus métodos, o padrão Builder é reconhecido na classe que possui um único método de criação e vários métodos de configuração de objetos resultantes.

Segue um trecho da aplicação demonstrando a problemática da não utilização do padrão:

```
Produto produto1 = new Produto();
produto1.setCodigo(10L);
produto1.setDescricao("Geladeira");
produto1.setPrecoUnitario(2500.0);
this.servicoDeProduto.adicionaProduto(produto1);
Estoque estoque1 = new Estoque();
estoque1.setProduto(produto1);
estoque1.setQuantidadeDisponivel(10);
this.servicoDeEstoque.adiconaEstoqueProduto(estoque1);

Produto produto2 = new Produto();
produto2.setCodigo(20L);
produto2.setDescricao("Fogao");
produto2.setPrecoUnitario(1200.0);
this.servicoDeProduto.adicionaProduto(produto2);
Estoque estoque2 = new Estoque();
estoque2.setProduto(produto2);
estoque2.setQuantidadeDisponivel(0);
this.servicoDeEstoque.adiconaEstoqueProduto(estoque2);
```

O padrão Builder foi utilizado nas entidades do projeto, as quais possuíam diversos atributos e para facilitar na criação das mesmas, implantamos o padrão referido. Segue um exemplo do mesmo trecho da imagem anterior, porém com o builder implementado:

```
Produto produto1 = new Produto.Builder()
.codigo(10L)
.descricao("Geladeira")
.precoUnitario(2500.55)
.build();
this.servicoDeProduto.adicionaProduto(produto1);

Estoque estoque1 = new Estoque.Builder()
.produto(produto1)
.quantidadeDisponivel(10)
.build();
this.servicoDeEstoque.salvarEstoqueProduto(estoque1);

Produto produto2 = new Produto.Builder()
.codigo(20L)
.descricao("Fogao")
.precoUnitario(1200.00)
.build();
this.servicoDeProduto.adicionaProduto(produto2);
```

3.2 Strategy

Com o padrão Strategy conseguimos realizar as mesmas operações mas de maneiras diferentes, geralmente utilizamos ele quando as opções de herança e implementação de interface nas classes não nos ajudam o suficiente para resolver o problema, mas caso se utilize do Strategy para usar quando uma herança apenas seria necessária o projeto pode ficar mais complicado.

Utilizamos o padrão para resolver o problema de diferentes países terem cálculos de impostos diferentes. Assim, criamos uma interface para definir o método de cálculo e suas implementações, responsáveis por calcularem de fato o imposto com base nas regras impostas em cada país. Criamos uma classe de configuração que contém as implementações da interface e essa classe é responsável por definir qual das implementações deve ser utilizada. Conforme o enunciado do trabalho proposto dizia, cada país iria subir a aplicação em um servidor diferente, logo, antes da inicialização da aplicação, definimos uma variável de ambiente que fica responsável por definir qual implementação será considerada.

Interface do Strategy:

```
public interface ICalculoImpostoStrategy {

    Double calculaValorDoImposto(List<ProdutoDTO> itens, double subtotal);

}
```

Primeira implementação do Strategy:

```java
public class CalculoImpostoBrasil implements ICalculoImpostoStrategy {

    private final double VALOR_IMPOSTO_IVA = 0.12;

    private final double VALOR_LIMITE_COMPRA_PARA_IMPOSTO_IVA = 8000;

    private final double VALOR_IMPOSTO_IVA_COM_VALOR_LIMITE_EXCEDENTE = 0.2;

    @Override
    public Double calculaValorDoImposto(List<ProdutoDTO> itens, double subtotal) {
        double imposto;

        if (subtotal <= VALOR_LIMITE_COMPRA_PARA_IMPOSTO_IVA) {
            imposto = subtotal * VALOR_IMPOSTO_IVA;
        } else {
            imposto = subtotal * VALOR_IMPOSTO_IVA_COM_VALOR_LIMITE_EXCEDENTE;
        }

        return imposto;
    }
}
```

Segunda implementação do Strategy:

```java
public class CalculoImpostoArgentina implements ICalculoImpostoStrategy {

    private final double VALOR_IMPOSTO_IVA = 0.1;

    private final double IMPOSTO_IVA_PRODUTO_10L = 0.25;

    private final double IMPOSTO_IVA_PRODUTO_30L = 0.15;

    @Override
    public Double calculaValorDoImposto(List<ProdutoDTO> itens, double subtotal) {

        double imposto = 0;

        for (ProdutoDTO product : itens) {

            if (product.getCodigo().equals(10L)) {
                imposto += product.getPreco() * IMPOSTO_IVA_PRODUTO_10L;
            } else if (product.getCodigo().equals(30L)) {
                imposto += product.getPreco() * IMPOSTO_IVA_PRODUTO_30L;
            } else {
                imposto += product.getPreco() * VALOR_IMPOSTO_IVA;
            }
        }

        return imposto;
    }
}
```

Configuração do Strategy:

```java
@Configuration
public class ConfiguracoesCalculoImposto {

    @Bean
    @ConditionalOnProperty(name = "REGRA.IMPOSTO", havingValue = "brasil", matchIfMissing = true)
    public ICalculoImpostoStrategy opcaoBrasil() {
        return new CalculoImpostoBrasil();
    }

    @Bean
    @ConditionalOnProperty(name = "REGRA.IMPOSTO", havingValue = "argentina")
    public ICalculoImpostoStrategy opcaoArgentina() { return new CalculoImpostoArgentina(); }

}
```

3.3 Factory

O padrão Factory encapsula a criação de objetos deixando as subclasses decidirem qual objeto elas querem criar.

Implementamos o Factory juntamente com o Strategy. Utilizamos para resolver o problema de impor restrições na venda. Restrições diferentes deveriam ser consideradas com base na hora atual. Com isso, criamos uma classe que devolvia uma instância do Strategy, e essa instância devolvida dependia da hora atual.

Definição do Factory:

```java
@Component
public class RestricoesFactory {

    private final LocalTime HORARIO_FECHAMENTO = LocalTime.of( hour: 19, minute: 0, second: 0);

    public IRestricoesStrategy restricoes() {
        if (LocalTime.now().isAfter(HORARIO_FECHAMENTO)) {
            return new RestricoesNivelBaixo();
        } else {
            return null;
        }
    }

}
```