Roteiro SOLID 01

Os roteiros a serem desenvolvidos visam trazer a percepção da evolução no processo de desenvolvimento de software. Por isso, iremos criar um projeto com vários pacotes, onde cada pacote representa a evolução da implementação deste projeto.

Neste caso iremos explorar um dos princípios do SOLID :

S - Single Responsibility Principle (Princípio da Responsabilidade Única)

Uma classe deve ter apenas uma razão para mudar, ou seja, deve ter apenas uma responsabilidade.

Início do projeto – Pacote : roteiro1.parte1

1 – Criar um projeto no NetBeans chamado **SOLIDroteiros**

2 – Dentro do projeto criar um pacote chamado **roteiro1.parte1**

Cenário :

Iremos montar a simulação de um sistema simples para gerenciamento de pedidos em um restaurante. Este sistema precisa realizar três funções principais :

- Conexão com o banco de dados

- Processamento de pedidos (regra de negócio)

- Cálculo de preços (integração com as regras de negócio)

3 – Inicialmente a primeira classe a ser criada será a de conexão com o Banco MySql. A ideia é simular uma conexão com o banco de dados MySql. Esta classe deve ser criada no pacote **roteiro1.parte1** e deve se chamar **MysqlConnection.** Nesta classe teremos apenas um método chamado **connect()**

```
package roteiro1.parte1;

public class MysqlConnection {

    public void connect() {
        System.out.println("Conectando com o MySQL");
    }
}
```

4 – Devemos criar agora PedidoService, que deverá conter as funções necessárias para a realização de um pedido no restaurante conforme o código a seguir.

```
package roteiro1.parte1;

import java.util.List;

public class PedidoService {
    private MysqlConnection connection;
    private List<String> itens;

   public PedidoService(List<String> itens) {
      this.connection = new MysqlConnection();
      this.itens = itens;
   }

   public void processarPedido() {
      this.connection.connect();
      double total = calcularTotal();
      System.out.println("Pedido processado. Valor total: R$ " + total);
   }

   private double calcularTotal() {
      double total = 0;
      for (String item : itens) {
         if (item.equals("Pizza")) total += 30.0;
         else if (item.equals("Bebida")) total += 10.0;
      }
      return total;
   }
```

5 - Neste mesmo pacote crie a classe a seguir para que possamos testar a criação de um pedido

```
package roteiro1.parte1;

import java.util.Arrays;
import java.util.List;

public class TestePedido {

   public static void main(String[] args) {
            List<String> lista = new ArrayList<>();
            lista.add("Pizza");
            lista.add("Bebida");

            PedidoService pedidoService = new PedidoService(lista);
            pedidoService.processarPedido();
```

Análise Crítica

Podemos observar que nesta modelagem a classe **PedidoService viola o princípio de Responsabilidade Única.** Ela centraliza responsabilidades que deveriam ser separadas.:

- Gerenciamento de conexão com o banco de dados.
    - Mesmo sendo uma simples instância de conexão com o Banco de dados no construtor da classe, não é considerada uma boa prática: `this.connection = new MysqlConnection();`
    - Essa prática caracteriza um **forte acoplamento** entre uma classe de negócio e uma classe de conexão com o banco.
- Lógica de processamento de pedidos
    - A princípio o método processarPedido() parece estar adequado, mas faz chamada aos cálculos com regras específicas que estão na mesma classe
- Cálculo de preços
    - Apesar de ser um simples cálculo para totalizar o pedido, em que este totalizador poderia ficar na mesma classe, neste caso não é considerado uma boa prática, pois contém uma regra de negócio associado ao cálculo e isso pode trazer problemas futuros com a evolução e manutenção do projeto.

OBS.: Coesão x Acoplamento

- **Coesão**
    - Representa o grau de harmonia e contexto de uma classe
    - Está relacionado com a responsabilidade única
- **Acoplamento**
    - Representa o grau de dependência entre classes
    - Está relacionado com o impacto que uma mudança tem em outros trechos de código

Pacote : roteiro1.parte2

1 – No mesmo projeto crie o pacote roteiro1.parte2

2 – Copie todas as classes criadas na parte1 para o novo pacote

3 – Vamos refatorar a classe **PedidoService** para tentar reduzir o forte acoplamento com a classe de conexão com o banco de dados. Faça o seguinte ajuste no construtor da classe.

```
public PedidoService(MysqlConnection connection, List<String> itens) {
    this.connection = connection;
    this.itens = itens;
}
```

4 – Agora vamos fazer os ajustes necessários na classe **TestePedido**

```
package roteiro1.parte1;

import java.util.Arrays;
import java.util.List;

public class TestePedido {

  public static void main(String[] args) {

        MysqlConnection connection = new MysqlConnection();

        List<String> lista = new ArrayList<>();
        lista.add("Pizza");
        lista.add("Bebida");

        PedidoService pedidoService = new PedidoService(connection,
lista);
        pedidoService.processarPedido();
```

Análise Crítica

A técnica adotada nesta refatoração chama-se **Injeção de Dependência.**

Essa é uma boa prática adotada em modelagem orientada a objetos (OO), pois promove um design mais flexível, modular e de fácil manutenção.

Conceitualmente, injeção de dependência é um padrão de design onde um objeto (classe) **não cria suas próprias dependências diretamente**, mas as recebe de uma fonte externa, geralmente por meio de um **construtor**, um **método setter** ou uma **fábrica (padrão de projeto que será discutido no futuro)**.

Resultado - Promove a Responsabilidade Única:

Ao injetar dependências, a classe principal não é mais responsável por criar ou gerenciar essas dependências. Isso alinha-se ao Princípio da Responsabilidade Única, uma das boas práticas do SOLID.

OBS.:

Uma outra refatoração que ainda poderia ser feita neste caso seria o uso de uma interface na injeção de dependência, pois tornaria modelagem ainda mais flexível na relação : Classe de negócio x Classe de conexão de Banco de Dados. (NÃO É NECESSÁRIA ESSA IMPLEMENTAÇÃO NO MOMENTO).

Pacote : roteiro1.parte3

1 – No mesmo projeto crie o pacote roteiro1.parte3

2 – Copie todas as classes criadas na parte2 para o novo pacote

3 – Novamente vamos refatorar a classe **PedidoService** para tentar isolar o cálculo totalizador do pedido, assim como as regras de negócio envolvidas. Para isso vamos criar uma nova classe chamada **CalculadoraPreco** e fazer os ajustes necessários na classe **PedidoService**

```
package roteiro1.parte3;

import java.util.List;

public class CalculadoraPreco {

   public double calcularTotal(List<String> itens) {

      double total = 0;

      for (String item : itens) {

         if (item.equals("Pizza")) total += 30.0;

         else if (item.equals("Bebida")) total += 10.0;

      }

      return total;

   }
```

```
package roteiro1.parte1;

import java.util.List;

public class PedidoService {
    private MysqlConnection connection;
    private CalculadoraPreco calculadora;
    private List<String> itens;

   public PedidoService(MysqlConnection connection, List<String> itens) {
      this.connection = connection;
      this.calculadora = new CalculadoraPreco();
      this.itens = itens;
   }

   public void processarPedido() {
      this.connection.connect();
      double total = this.calculadora.calcularTotal(this.itens);
      System.out.println("Pedido processado. Valor total: R$ " + total);
   }

   private double calcularTotal() {
      double total = 0;
      for (String item : itens) {
         if (item.equals("Pizza")) total += 30.0;
         else if (item.equals("Bebida")) total += 10.0;
      }
```

4 – Realize os testes de execução na classe **TestePedido**

Análise Crítica

Depois de 2 refatorações importantes, temos agora a classe **PedidoService** com a única responsabilidade de processamentos dos pedidos.

Questão

Agora com as classes bem divididas ainda existem possíveis refatorações ?

Pacote : roteiro1.parte4

E

Pacote : roteiro1.parte5

Nesta etapa do roteiro crie 2 pacotes conforme as instruções abaixo, pois faremos uma nova simulação.

Mudança no cenário original :

Suponha que o cliente resolveu criar uma campanha de desconto promocional. A regra da campanha é a seguinte :

- **Se o pedido tiver apenas Pizza, aplica-se um desconto de 10% no total do pedido.**
- **Se o pedido tiver Pizza e Bebida, aplica-se um desconto de 15% no total do pedido.**

1 – No mesmo projeto crie o pacote **roteiro1.parte4** e **roteiro1.parte5**

2 – Tente implementar no pacote **roteiro1.parte4** a mudança necessária utilizando como referência a modelagem presente na **parte1**

3 – Tente implementar no pacote **roteiro1.parte5** a mudança necessária utilizando como referência a modelagem presente na **parte3**

Análise Crítica

Descreva aqui as suas percepções para as duas soluções desenvolvidas.