Roteiro SOLID 04

Os roteiros a serem desenvolvidos visam trazer a percepção da evolução no processo de desenvolvimento de software. Por isso, daremos sequência ao projeto criando vários pacotes, onde cada pacote representa a evolução da implementação deste projeto.

Neste caso iremos explorar um dos princípios do SOLID :

I - Interface Segregation Principle (Princípio da Segregação de Interfaces)

Uma classe não deve ser forçada a implementar interfaces que ela não utiliza. Prefira interfaces menores e específicas.

Compreender e aplicar o Princípio da Segregação de Interfaces (ISP), garante que interfaces sejam específicas para cada tipo de cliente, evitando métodos desnecessários e reduzindo o acoplamento do sistema.

O princípio da Segregação de Interfaces segue as seguintes premissas:

- Crie interfaces granulares a específicas para os seus clientes.
- Clientes não devem ser forçados a depender de interfaces que eles não usam.

Pacote : roteiro4.parte1

1 – Dê sequência ao mesmo projeto no NetBeans chamado **SOLIDroteiros**

2 – Dentro do projeto criar um pacote chamado **roteiro4.parte1**

Cenário :

Vamos modelar um sistema de pagamento digital, que deve oferecer diferentes formas de pagamento, incluindo cartão de crédito, boleto e criptomoedas.

O sistema deve oferecer:

- **Processamento de pagamento**
- **Geração de fatura (quando necessário)**
- **Validação de saldo (para pagamentos via criptomoeda e cartão de crédito)**

3 – Diante deste cenário devemos criar a interface **MetodoPagamento** conforme o código abaixo.

Esta interface deverá servir de base para cada forma de pagamento que deve ser implementada.

```
package roteiro4.parte1;

public interface MetodoPagamento {

   void processarPagamento(double valor);
   void gerarFatura();
   void validarSaldo();

}
```

4 – Na sequência devemos criar as classes que representam as formas de pagamento e que implementam a interface **MetodoPagamento**. São elas : **CartaoCredito**, **BoletoBancario**, **Criptomoeda**.

```
package roteiro4.parte1;

public class CartaoCredito implements MetodoPagamento {
   @Override
   public void processarPagamento(double valor) {
      System.out.println("Pagamento de R$ " + valor + " processado no cartão de
crédito.");
   }

   @Override
   public void gerarFatura() {
      System.out.println("Fatura gerada para o cartão de crédito.");
   }

   @Override
   public void validarSaldo() {
      System.out.println("Validando saldo disponível no cartão de crédito.");
```

```
package roteiro4.parte1;

public class BoletoBancario implements MetodoPagamento {
   @Override
   public void processarPagamento(double valor) {
      System.out.println("Boleto gerado para pagamento de R$ " + valor);
   }

   @Override
   public void gerarFatura() {
      System.out.println("Fatura gerada para o boleto bancário.");
   }

   @Override
   public void validarSaldo() {
      throw new UnsupportedOperationException("Boletos não precisam validar
saldo!");
```

```
package roteiro4.parte1;

public class Criptomoeda implements MetodoPagamento {
    @Override
    public void processarPagamento(double valor) {
        System.out.println("Pagamento de R$ " + valor + " realizado com
criptomoeda.");
    }

    @Override
    public void gerarFatura() {
        throw new UnsupportedOperationException("Criptomoedas não geram
fatura!");
    }

    @Override
    public void validarSaldo() {
```

5 – Agora crie a classe **TestePagamento**, conforme o código abaixo para que façamos os testes.

```
package roteiro4.parte1;

public class TestePagamento {

    public static void main(String[] args) {

        System.out.println("\n PAGAMENTO CARTAO DE CREDITO");
        MetodoPagamento cartao = new CartaoCredito();
        cartao.processarPagamento(100.00);


        System.out.println("\n PAGAMENTO BOLETO BANCARIO");
        MetodoPagamento boleto = new BoletoBancario();
        boleto.processarPagamento(200.00);


        System.out.println("\n PAGAMENTO CRIPTO");
        MetodoPagamento cripto = new Criptomoeda();
        cripto.processarPagamento(300.00);


    }
}
```

6 – Reavalie o **TestePagamento** adicionando os trechos de código abaixo

```
package roteiro4.parte1;

public class TestePagamento {

    public static void main(String[] args) {

        System.out.println("\n PAGAMENTO CARTAO DE CREDITO");
        MetodoPagamento cartao = new CartaoCredito();
        cartao.gerarFatura();
        cartao.processarPagamento(100.00);


        System.out.println("\n PAGAMENTO BOLETO BANCARIO");
        MetodoPagamento boleto = new BoletoBancario();
        boleto.gerarFatura();
        boleto.processarPagamento(200.00);


        System.out.println("\n PAGAMENTO CRIPTO");
        MetodoPagamento cripto = new Criptomoeda();
        cripto.gerarFatura();
        cripto.processarPagamento(300.00);


    `
```

Análise Crítica

Podemos observar que nesta modelagem existe a **violação** **do princípio da Segregação de Interfaces.**

- BoletoBancario não precisa validar saldo, mas é obrigado a implementar validarSaldo(), gerando uma exceção.
- Criptomoeda não gera fatura, mas é forçada a implementar gerarFatura(), gerando outra exceção.
- Cada classe implementa métodos que não fazem sentido para seu contexto, o que aumenta o acoplamento e a complexidade desnecessária.

Seguindo com a análise, responda as seguintes questões :

Mesmo com problemas na modelagem, existe alguma solução de contorno ? Caso não tenha condições de alterar o modelo no momento ?

Consegue identificar se esta modelagem viola algum outro princípio do SOLID ?

Pacote : roteiro4.parte2

1 – No mesmo projeto crie o pacote roteiro4.parte2

2 – Na ideia de tentar seguir o princípio da Segregação de Interfaces. Devemos criar interfaces específicas e que seja possível aplicar em cada forma de pagamento (quando necessário). São Elas : **MetodoPagamento**, **MetodoPagamentoComFatura**, **MetodoPagamentoComSaldo**.

```
package roteiro4.parte2;

// Interface para métodos de pagamento genéricos
public interface MetodoPagamento {
    void processarPagamento(double valor);
}
```

```
package roteiro4.parte2;

// Interface específica para pagamentos que precisam gerar fatura
public interface MetodoPagamentoComFatura {
    void gerarFatura();
}
```

```
package roteiro4.parte2;

// Interface específica para pagamentos que precisam validar saldo
public interface MetodoPagamentoComSaldo {
    void validarSaldo();
}
```

3 – Agora as classes **CartaoCredito**, **BoletoBancario** e **Criptomoeda** devem implementar as interfaces adequadas ao seu contexto.

```
package roteiro4.parte2;

public class CartaoCredito implements MetodoPagamento, MetodoPagamentoComFatura,
MetodoPagamentoComSaldo {
    @Override
    public void processarPagamento(double valor) {
        System.out.println("Pagamento de R$ " + valor + " processado no cartão de crédito.");
    }

    @Override
    public void gerarFatura() {
        System.out.println("Fatura gerada para o cartão de crédito.");
    }

    @Override
    public void validarSaldo() {
        System.out.println("Validando saldo disponível no cartão de crédito.");
    }
```

```
package roteiro3.parte2;

public class BoletoBancario implements MetodoPagamento,
MetodoPagamentoComFatura {
    @Override
    public void processarPagamento(double valor) {
        System.out.println("Boleto gerado para pagamento de R$ " + valor);
    }

    @Override
    public void gerarFatura() {
        System.out.println("Fatura gerada para o boleto bancário.");
    }
```

```
package roteiro4.parte2;

public class Criptomoeda implements MetodoPagamento,
MetodoPagamentoComSaldo {
    @Override
    public void processarPagamento(double valor) {
        System.out.println("Pagamento de R$ " + valor + " realizado com criptomoeda.");
    }

    @Override
    public void validarSaldo() {
        System.out.println("Validando saldo disponível na carteira de criptomoeda.");
    }
```

4 – Agora crie a classe **TestePagamento**, conforme o código abaixo para que façamos os testes.

```
package roteiro4.parte2;

public class TestePagamento {
    public static void main(String[] args) {

        System.out.println("\n PAGAMENTO CARTAO DE CREDITO");
        MetodoPagamento cartao = new CartaoCredito();
        cartao.processarPagamento(100.00);

        System.out.println("\n PAGAMENTO BOLETO BANCARIO");
        MetodoPagamento boleto = new BoletoBancario();
        boleto.processarPagamento(200.00);

        System.out.println("\n PAGAMENTO CRIPTO");
        MetodoPagamento cripto = new Criptomoeda();
        cripto.processarPagamento(300.00);
    }
}
```

5 – Reavalie o **TestePagamento** adicionando os trechos de código abaixo

```
package roteiro4.parte2;

public class TestePagamento {
   public static void main(String[] args) {

      System.out.println("\n PAGAMENTO CARTAO DE CREDITO");
      MetodoPagamento cartao = new CartaoCredito();
      cartao.gerarFatura();
      cartao.processarPagamento(100.00);

      System.out.println("\n PAGAMENTO BOLETO BANCARIO");
      MetodoPagamento boleto = new BoletoBancario();
      boleto.processarPagamento(200.00);

      System.out.println("\n PAGAMENTO CRIPTO");
      MetodoPagamento cripto = new Criptomoeda();
      cripto.processarPagamento(300.00);
   }
}
```

6 – Possivelmente não conseguiu utilizar o método gerarFatura na questão anterior.

Segue abaixo uma possível solução de contorno, mas tente responder porque isso aconteceu.

Adicione o trecho de código abaixo para utilizar o método gerarFatura no cartão de crédito.

```
if (cartao instanceof CartaoCredito){
    ((MetodoPagamentoComFatura)cartao).gerarFatura();
}
```

Análise Crítica

A modelagem agora parece atender ao **princípio da Segregação de Interfaces.** Mas, aparentemente gerou um **sutil violação do princípio da Substituição de Liskov**.

- As interfaces foram granuladas conforme as suas especificidades, atendendo ao princípio da Segregação de Interfaces.
- Os objetos **cartao**, **boteto** e **cripto** são declarados do mesmo tipo (entidade **MetodoPagamento**), e são substituíveis como forma de pagamento. Atendendo ao princípio da Substituição de Liskov.
    - São substituíveis porque podemos alternar as formas de pagamento sem necessariamente “quebrar” o código. Mas, neste caso em específico podemos ter uma ligeira inconsistência com a regra de negócio.

- Um cartão de crédito pode gerar fatura, e neste caso não conseguimos usar o método **gerarFatura()** de forma explicita.

Siga para Parte 3

Mudança no cenário :

Agora, nosso sistema precisa suportar reembolsos, mas nem todos os métodos de pagamento permitem reembolsos da mesma forma. Vejam :

- **Cartão de Crédito : Pode ser estornado automaticamente**
- **Boleto Bancário : O reembolso deve ser feito manualmente via transferência bancária**
- **Criptomoeda : Não permite reembolso diretamente.**

A dúvida que surge é:

- **Criamos uma nova interface para reembolsos?**
- **Ou criamos uma classe abstrata para os métodos de pagamento?**

A primeira opção pode ser questionável, já que teríamos tantas interfaces no modelo :

- **1 - MetodoPagamento**
- **2 - MetodoPagamentoComFatura**
- **3 - MetodoPagamentoComSaldo**
- **Adicionado 4 - PagamentoReembolsavel**

Administrar 4 Interfaces na modelagem pode ser confuso !

Assim, vamos tentar uma solução híbrida, utilizando Classe Abstrata + Interface

Siga para Parte 3

Pacote : roteiro4.parte3

1 – No mesmo projeto crie o pacote roteiro4.parte3

2 – Vamos inicialmente criar a classe abstrata Pagamento. Nesta classe vamos tentar definir todos os métodos que existem em comum entre as formas de pagamento. Obs.: Mesmo que exista alguma exceção em algum método.

processarPagamento() – é comum a todas as formas de pagamento e deve ser implementado **nas 3 formas de pagamento**.

gerarFatura() – é comum apenas em **cartão de crédito** e **boleto bancário**. Portanto, devem ser sobrescrito nas respectivas subclasses.

validarSaldo() – é comum apenas em **cartão de crédito** e **criptomoeda**. Portanto, devem ser sobrescrito nas respectivas subclasses.

```
package roteiro4.parte3;

public abstract class Pagamento {

   public abstract void processarPagamento(double valor);

   public void gerarFatura() {
      throw new UnsupportedOperationException("Este método de pagamento não suporta
fatura.");
   }

   public void validarSaldo() {
      throw new UnsupportedOperationException("Este método de pagamento não suporta
```

3 – Neste modelo criaremos apenas uma interface para tratar a questão do reembolso devido a mudança no cenário.

```
package roteiro4.parte3;

public interface PagamentoReembolsavel {

   void processarReembolso(double valor);
}
```

4 – Precisamos agora fazer os ajustes necessários nas subclasses **CartaoCredito**, **BoletoBancario** e **Criptomoeda**.

CartaoCredito – com os 4 métodos. Um deles proveniente da Interface **PagamentoReembolsavel**.

```
package roteiro4.parte3;

public class CartaoCredito extends Pagamento implements PagamentoReembolsavel {
    @Override
    public void processarPagamento(double valor) {
        System.out.println("Pagamento de R$ " + valor + " processado no cartão de crédito.");
    }

    @Override
    public void gerarFatura() {
        System.out.println("Fatura gerada para o cartão de crédito.");
    }

    @Override
    public void validarSaldo() {
        System.out.println("Validando saldo disponível no cartão de crédito.");
    }

    @Override
    public void processarReembolso(double valor) {
        System.out.println("Estorno de R$ " + valor + " realizado no cartão de crédito.");
    }
```

BoletoBancario – com os 3 métodos. Dois métodos provenientes da superclasse **Pagamento**, e outro proveniente da Interface **PagamentoReembolsavel**.

```
package roteiro4.parte3;

public class BoletoBancario extends Pagamento implements PagamentoReembolsavel {
    @Override
    public void processarPagamento(double valor) {
        System.out.println("Boleto gerado para pagamento de R$ " + valor);
    }

    @Override
    public void gerarFatura() {
        System.out.println("Fatura gerada para o boleto bancário.");
    }

    @Override
    public void processarReembolso(double valor) {
        System.out.println("Reembolso de R$ " + valor + " deve ser feito via transferência bancária.");
    }
}
```

Criptomoeda – com os 2 métodos. Um método proveniente da superclasse **Pagamento.**

```
package roteiro4.parte3;

public class Criptomoeda extends Pagamento {
    @Override
    public void processarPagamento(double valor) {
        System.out.println("Pagamento de R$ " + valor + " realizado com criptomoeda.");
    }

    @Override
    public void validarSaldo() {
        System.out.println("Validando saldo disponível na carteira de criptomoeda.");
    }
`
```

5 – Vamos agora aos testes na classe **TestePagamento**, conforme o código abaixo

```
package roteiro4.parte33;

public class TestePagamento {
    public static void main(String[] args) {

        System.out.println("\n PAGAMENTO CARTAO DE CREDITO");
        Pagamento cartao = new CartaoCredito();
        cartao.gerarFatura();
        cartao.processarPagamento(100.00);
        if (cartao instanceof PagamentoReembolsavel) {
            ((PagamentoReembolsavel) cartao).processarReembolso(50.00);
        }

        System.out.println("\n PAGAMENTO BOLETO BANCARIO");
        Pagamento boleto = new BoletoBancario();
        boleto.gerarFatura();
        boleto.processarPagamento(200.00);
        if (boleto instanceof PagamentoReembolsavel) {
            ((PagamentoReembolsavel) boleto).processarReembolso(100.00);
        }


        System.out.println("\n PAGAMENTO CRIPTO");
        Pagamento cripto = new Criptomoeda();
        try{
            cripto.gerarFatura();
        } catch (UnsupportedOperationException e){
            System.out.println("ERRO - Gerar Fatura : " + e.getMessage());
        }
        cripto.processarPagamento(300.00);
        if (cripto instanceof PagamentoReembolsavel) {
            ((PagamentoReembolsavel) cripto).processarReembolso(100.00);
        }

    `
```

Análise Crítica

Nesta modelagem optamos por uma modelagem hibrida trabalhando **Classe Abstrata + Interface**.

Podemos identificar várias **vantagens** nesta modelagem

- Evita a violação do Princípio da Substituição de Liskov
    - Pagamento é substituível sem quebras de comportamento.
    - Um código que espera Pagamento pode receber qualquer subclasse sem problemas.
- Melhora a reutilização de código
    - Métodos comuns (processarPagamento, gerarFatura, validarSaldo) ficam centralizados na classe abstrata.
- Aplicamos o princípio da Separação de Interfaces
    - A interface PagamentoReembolsavel adiciona uma funcionalidade apenas onde faz sentido, sem forçar toas as subclasses a implementá-lo.

Mas também identificamos uma **desvantagem**

Observe que ao tentar usar o método gerarFatura com o objeto cripto tivemos que fazer uso de um tratamento de exceção. Claro que isso estaria ferindo uma regra de negócio, e se isso ocorrer em uma situação real, seria por falha de quem está programando. O problema é que se o programador cometer esse erro, teríamos uma falha crítica no sistema. E neste caso a modelagem não está ajudando muito a prevenir esta falha.

- O Uso Excessivo de Exceções é uma Má Prática
    - Exceções devem ser usadas para erros inesperados, não para controlar fluxo de execução.
    - Lançar exceções impede que a substituição de objetos aconteça de forma fluida.
    - Se o desenvolvedor precisa ficar verificando se um método pode ser chamado para evitar exceções, isso indica um problema de design.

Pacote : roteiro4.parte4

1 – No mesmo projeto crie o pacote roteiro3.parte4

2 – Copie todas as classes da **parte3** para o novo pacote.

3 – Vamos insistir um pouco mais nesta solução da parte 3 e tentar fazer alguma melhoria. Um recurso interessante em Java é utilizar um método com **"código morto"**. Ou seja, um método que ainda não tem implementação. Vamos tentar utilizar este recurso para evitar o lançamento de exceções que quebrem o código.

Para isso precisamos tornar o método **gerarFatura()** abstrato (retirando o lançamento de exceção)

```
package roteiro4.parte4;

public abstract class Pagamento {

    public abstract void processarPagamento(double valor);

    public abstract void gerarFatura();

    public void validarSaldo() {
        throw new UnsupportedOperationException("Este método de pagamento não suporta
validação de saldo.");
    }
}
```

Esta ação irá forçar a implementação nas subclasses. Nas classes que não precisam gerar fatura teremos a implementação com "código morto" (Método vazio). Veja abaixo a classe **Criptomoeda**

```
package roteiro4.parte4;

public class Criptomoeda extends Pagamento {
    @Override
    public void processarPagamento(double valor) {
        System.out.println("Pagamento de R$ " + valor + " realizado com criptomoeda.");
    }

    @Override
    public void validarSaldo() {
        System.out.println("Validando saldo disponível na carteira de criptomoeda.");
    }

    @Override
    public void gerarFatura() {

    }
}
```

5 – Vamos agora aos testes na classe **TestePagamento**, conforme o código abaixo. Agora sem a necessidade de tratamento de exceção.

```
package roteiro4.parte33;

public class TestePagamento {
    public static void main(String[] args) {

        System.out.println("\n PAGAMENTO CARTAO DE CREDITO");
        Pagamento cartao = new CartaoCredito();
        cartao.gerarFatura();
        cartao.processarPagamento(100.00);
        if (cartao instanceof PagamentoReembolsavel) {
            ((PagamentoReembolsavel) cartao).processarReembolso(50.00);
        }

        System.out.println("\n PAGAMENTO BOLETO BANCARIO");
        Pagamento boleto = new BoletoBancario();
        boleto.gerarFatura();
        boleto.processarPagamento(200.00);
        if (boleto instanceof PagamentoReembolsavel) {
            ((PagamentoReembolsavel) boleto).processarReembolso(100.00);
        }


        System.out.println("\n PAGAMENTO CRIPTO");
        Pagamento cripto = new Criptomoeda();
        cripto.gerarFatura();
        try{
            cripto.gerarFatura();
        } catch (UnsupportedOperationException e){
            System.out.println("ERRO - Gerar Fatura : " + e.getMessage());
        }
        cripto.processarPagamento(300.00);
        if (cripto instanceof PagamentoReembolsavel) {
            ((PagamentoReembolsavel) cripto).processarReembolso(100.00);
        }
```

Análise Crítica

Este recurso com o “Código Morto” não é visto pela comunidade como um **código limpo**. Trata-se de um recurso interessante quando o desenvolvedor pretende de fato implementar alguma funcionalidade, mas que momentaneamente está incompleto.

Por que isso pode ser um problema?

- Viola o Princípio da Responsabilidade Única (SRP - Single Responsibility Principle)
    - Criptomoeda tem um método que não faz parte de sua responsabilidade, apenas para atender a um contrato da classe base.
- Pode confundir outros desenvolvedores

- Se alguém vê cripto.gerarFatura(), pode pensar que esse método faz algo útil, quando na verdade não faz nada.

Pacote : roteiro4.parte5

1 – No mesmo projeto crie o pacote roteiro3.parte5

2 – Copie todas as classes da **parte4** para o novo pacote.

3 – Vamos refatorar com mais uma melhoria, invertendo a lógica para o método **gerarFatura()** na classe **Pagamento**

Na classe abstrata informamos que o método de gerar fatura não se aplica, e sobrescrevemos este método apenas nas classes que de fato geram fatura.

```
package roteiro4.parte4;

public abstract class Pagamento {

    public abstract void processarPagamento(double valor);

    public void gerarFatura(){
        System.out.println("Este método não se aplica a esta forma de pagamento.");
    }

    public void validarSaldo() {
        throw new UnsupportedOperationException("Este método de pagamento não suporta validação
de saldo.");
    }
```

A classe Criptomoeda agora não precisa implementar este método

```
package roteiro4.parte4;

public class Criptomoeda extends Pagamento {
    @Override
    public void processarPagamento(double valor) {
        System.out.println("Pagamento de R$ " + valor + " realizado com criptomoeda.");
    }

    @Override
    public void validarSaldo() {
        System.out.println("Validando saldo disponível na carteira de criptomoeda.");
    }

}
```

4 – Faça agora os testes necessários na classe **TestePagamento.**

Análise Crítica

Faça agora a análise completa da evolução deste roteiro.

A solução hibrida (Classe Abstrata + Interface) pareceu interessante ?

Foi possível conciliar os 4 princípios do SOLID vistos até o momento, nesta solução ?

- Princípio da Responsabilidade Única
- Princípio do Aberto/Fechado
- Princípio da Substituição de Liskov
- Princípio da Segregação de Interfaces