Roteiro SOLID 03

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 :

L - Liskov Substitution Principle (Princípio da Substituição de Liskov)

Objetos de uma classe derivada devem poder substituir objetos da classe base sem alterar o comportamento esperado do programa.

O Princípio de Substituição de Liskov está diretamente ligado ao princípio Aberto/Fechado (Roteiro 02), e foi criado por Barbara Liskov, uma grande cientista da computação amaricana.

Este princípio tem como objetivo nos alertar quanto a utilização da herança, que é um poderoso mecanismo e deve ser utilizado com extrema parcimônia. O princípio é baseado na seguinte premissa:

"As classes derivadas devem ser substituíveis por suas classes bases."

Ou seja, você deve ser capaz de substituir uma classe base por qualquer uma de suas classes derivadas sem que isso quebre ou altere o comportamento esperado do programa. Isso significa que qualquer função que use uma classe base deve funcionar igualmente bem, se passar uma classe derivada no lugar.

Uma boa aplicação do princípio de Substituição de Liskov garante por consequência o princípio Aberto/Fechado, já que um bom design respeita a extensão sem modificar o comportamento existente.

Pacote : roteiro3.parte1

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

2 – Dentro do projeto criar um pacote chamado **roteiro3.parte1**

Cenário :

Iremos montar a simulação de um sistema simples para gerenciamento de transportes urbanos que lida com diferentes tipos de veículos. O sistema precisa realizar as seguintes funções principais :

- **Cálculo de tarifa de viagem, conforme o tipo de veículo utilizado e suas características**

3 – Diante deste cenário a primeira classe a ser criada é **VeiculoTransporte** conforme o código abaixo.

Esta classe servirá de base para todos os tipos de transporte utilizando herança (recurso da OO). Neste momento iremos nos preocupar apenas com o cálculo da tarifa.

```
package roteiro3.parte1;

public class VeiculoTransporte {
   protected double tarifaBase;

   public VeiculoTransporte(double tarifaBase) {
      this.tarifaBase = tarifaBase;
   }

   public double calcularTarifa() {
      return tarifaBase;
   }
```

4 – Na sequência devemos criar as subclasses de **VeiculoTransporte**. São elas : **Onibus**, **Bicicleta**.

```
package roteiro3.parte1;

public class Onibus extends VeiculoTransporte {

   public Onibus(double tarifaBase) {
      super(tarifaBase);
   }

   @Override
   public double calcularTarifa() {
      return tarifaBase * 1.2; // Aumenta 20% para longas distâncias
   }
```

```
package roteiro3.parte1;

public class Bicicleta extends VeiculoTransporte {

   public Bicicleta() {
      super(0); // Bicicletas são gratuitas
   }

   @Override
   public double calcularTarifa() {
      throw new UnsupportedOperationException("Bicicletas não têm tarifa!");
   }
```

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

```
package roteiro3.parte1;

public class TesteTransporte {

    public static void main(String[] args) {
        VeiculoTransporte onibus = new Onibus(5.00);
        System.out.println("Tarifa Ônibus: R$ " + onibus.calcularTarifa());
    }
}
```

5 – Em seguida modifique a classe **TesteTransporte** adicionando um novo meio de transporte como teste.

```
package roteiro3.parte1;

public class TesteTransporte {

    public static void main(String[] args) {
        VeiculoTransporte onibus = new Onibus(5.00);
        System.out.println("Tarifa Ônibus: R$ " + onibus.calcularTarifa());

        VeiculoTransporte bicicleta = new Bicicleta();
        System.out.println("Tarifa Bicicleta: R$ " + bicicleta.calcularTarifa()); // ⚠ Vai gerar erro!
    }
```

Análise Crítica

Podemos observar que nesta modelagem existe a **violação do princípio da Substituição de Liskov.** Tentamos utilizar 2 meios de transporte que herdam da mesma entidade (**VeiculoTransporte**) e um erro foi gerado.

- Bicicleta herda de VeiculoTransporte, mas o método calcularTarifa() **não faz sentido** para ela.
- Como resultado, Bicicleta **lança uma exceção**, quebrando o contrato da classe base.
- O código agora **precisa tratar Bicicleta como uma exceção**, em vez de tratá-la como qualquer outro veículo.

Como solução de contorno adote a seguinte modificação no código da classe **TesteTransporte** :

```
package roteiro3.parte1;

public class TesteTransporte {

    public static void main(String[] args) {
        VeiculoTransporte onibus = new Onibus(5.00);
        System.out.println("Tarifa Ônibus: R$ " + onibus.calcularTarifa());

        try {
            VeiculoTransporte bicicleta = new Bicicleta();
            System.out.println("Tarifa Bicicleta: R$ " + bicicleta.calcularTarifa());
        } catch (UnsupportedOperationException e) {
            System.out.println("Bicicletas não têm tarifa, então ignoramos esse cálculo.");
```

Agora, a execução do programa **não vai quebrar**, mas a modelagem **continua errada**. O tratamento da exceção está apenas **disfarçando** o problema.

➡ Um código bem modelado não deveria precisar de um tratamento de exceção para corrigir um problema previsível.

Pacote : roteiro3.parte2

1 – No mesmo projeto crie o pacote roteiro3.parte2

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

3 – A primeira refatoração a fazer deve ser na classe **VeiculoTransporte**. Devemos torná-la uma classe abstrata. Assim, as instâncias de objetos só poderão serão feitas nas subclasses (Onibus e Bicicleta). Além disso, o método calcularTarifa() também deve ser abstrato. Esta ação obrigará que o cálculo da tarifa seja feito da forma correto nas subclasses.

```
package roteiro3.parte2;

public abstract class VeiculoTransporte {

    public abstract double calcularTarifa();

}
```

4 – Faça os ajustes necessário nas classes **Onibus** e **Bicicleta**, conforme o código abaixo.

```
package roteiro3.parte2;

public class Onibus extends VeiculoTransporte {
    private double tarifaBase;

    public Onibus(double tarifaBase) {
        this.tarifaBase = tarifaBase;
    }

    @Override
    public double calcularTarifa() {
        return tarifaBase * 1.2;
    }
```

```
package roteiro3.parte2;

public class Bicicleta extends VeiculoTransporte {
    @Override
    public double calcularTarifa() {
        return 0; // Agora Bicicleta pode ser tratada como um veículo, sem
lançar exceções
    }
}
```

5 – Faça um novo teste agora com a classe **TesteTransporte**, conforme o código abaixo

```
package roteiro3.parte2;

public class TesteTransporte {

   public static void main(String[] args) {
      VeiculoTransporte veiculo1 = new Onibus(5.00);
      VeiculoTransporte veiculo2 = new Bicicleta(); // Agora é um VeiculoTransporte válido

      System.out.println("Tarifa Ônibus: R$ " + veiculo1.calcularTarifa());
      System.out.println("Tarifa Bicicleta: R$ " + veiculo2.calcularTarifa()); // Agora não quebra
o código
   }
}
```

Análise Crítica

A modelagem agora parece atender ao **princípio da Substituição de Liskov.** Mas, o exemplo apresentado parece muito simplório para um sistema real. Mesmo utilizando este exemplo simples e didático, o que aconteceria em uma possível evolução deste cenário ?

Siga para Parte 3

Mudança no cenário :

Precisamos que o nosso sistema tenha um controle para consumo de combustível utilizado nas viagens, conforme a distância percorrida.

Trataremos essa solução com 2 abordagens diferentes nos pacotes : ***roteiro3.parte3*** *e* ***roteiro3.parte4****. Assim poderá fazer uma análise crítica da melhor solução.*

Pacote : roteiro3.parte3 (Primeira Solução)

1 – No mesmo projeto crie o pacote roteiro3.parte3

2 – Copie todas as classes da **parte2** para o novo pacote.

3 – Nesta evolução teremos que ter o controle de distância percorrida. Este atributo é inerente aos dois tipos de veículos então devemos fazer a refatoração na classe **VeiculoTransporte.** Além disso, de forma semelhante a modelagem da parte 2 adicionamos o método abstrato **calcularTotalCombustivel**

```
package roteiro3.parte3;

public abstract class VeiculoTransporte {

    private double distancia;

    public VeiculoTransporte(double distancia)
    {
        this.distancia = distancia;
    }

    public double getDistancia() {
        return distancia;
    }

    public abstract double calcularTarifa();

    public abstract double calcularTotalCombustivel(double
```

4 – Precisamos agora refatorar as classes **Onibus** e **Bicicleta**, promovendo os ajustes nece

```
package roteiro3.parte3;

public class Onibus extends VeiculoTransporte {
    private double tarifaBase;
    private double consumoPorKm;

    public Onibus(double tarifaBase, double distancia) {
        super(distancia);
        this.tarifaBase = tarifaBase;
    }

    @Override
    public double calcularTarifa() {
        return tarifaBase * 1.2;
    }

    @Override
    public double calcularTotalCombustivel(double consumoPorKm) {
        this.consumoPorKm = consumoPorKm;
        return this.consumoPorKm * this.getDistancia();
    }
```

```
package roteiro3.parte3;

public class Bicicleta extends VeiculoTransporte {

    public Bicicleta (double distancia)
    {
        super(distancia);
    }

    @Override
    public double calcularTarifa() {
        return 0; // Agora Bicicleta pode ser tratada como um veículo, sem lançar exceções
    }

    @Override
    public double calcularTotalCombustivel(double consumoPorKm) {
        return 0;
    }
```

5 – Faça um novo teste agora com a classe **TesteTransporte**, conforme o código abaixo

```
package roteiro3.parte3;

public class TesteTransporte {

   public static void main(String[] args) {
      VeiculoTransporte veiculo1 = new Onibus(5.00, 10.0);
      VeiculoTransporte veiculo2 = new Bicicleta(10.0); // Agora é um VeiculoTransporte válido

      System.out.println("Tarifa Ônibus: R$ " + veiculo1.calcularTarifa());
      System.out.println("Distância Ônibus: " + veiculo1.getDistancia());
      System.out.println("Consumo Total Ônibus: " + veiculo1.calcularTotalCombustivel(12));

      System.out.println("Tarifa Bicicleta: R$ " + veiculo2.calcularTarifa());
      System.out.println("Distância Bicicleta: " + veiculo1.getDistancia());
      System.out.println("Consumo Total Bicicleta: " + veiculo2.calcularTotalCombustivel(12));


   }
```

Pacote : roteiro3.parte4 (Segunda Solução)

1 – No mesmo projeto crie o pacote roteiro3.parte4

2 – Copie todas as classes da **parte2** para o novo pacote.

3 – Nesta evolução teremos que ter o controle de distância percorrida. Este atributo é inerente aos dois tipos de veículos então devemos fazer a refatoração na classe **VeiculoTransporte.**

```
package roteiro3.parte4;

public abstract class VeiculoTransporte {
    private double distancia;

    public VeiculoTransporte(double distancia)
    {
        this.distancia = distancia;
    }

    public double getDistancia() {
        return distancia;
    }

    public abstract double calcularTarifa();
```

4 – Observe que ainda não tratamos a questão do cálculo do combustível. Para isso, devemos criar a interface **VeiculoMotorizado**, conforme segue abaixo

```
package roteiro3.parte4;

public interface VeiculoMotorizado {

    public double calcularTotalCombustivel(double consumoPorKm);

}
```

5 – Precisamos agora refatorar as classes **Onibus** e **Bicicleta**, promovendo os ajustes nece

```
package roteiro3.parte4;

public class Onibus extends VeiculoTransporte implements VeiculoMotorizado{
    private double tarifaBase;
    private double consumoPorKm;

    public Onibus(double tarifaBase, double distancia) {
        super(distancia);
        this.tarifaBase = tarifaBase;
    }

    @Override
    public double calcularTarifa() {
        return tarifaBase * 1.2;
    }

    @Override
    public double calcularTotalCombustivel(double consumoPorKm) {
        this.consumoPorKm = consumoPorKm;
        return this.consumoPorKm * super.getDistancia();
    }
```

```
package roteiro3.parte4;

public class Bicicleta extends VeiculoTransporte {

    public Bicicleta (double distancia)
    {
        super(distancia);
    }

    @Override
    public double calcularTarifa() {
        return 0; // Agora Bicicleta pode ser tratada como um veículo, sem lançar exceções
    }

}
```

6 – Faça um novo teste agora com a classe **TesteTransporte**, conforme o código abaixo

```
package roteiro3.parte3;

public class TesteTransporte {

   public static void main(String[] args) {
      VeiculoTransporte veiculo1 = new Onibus(5.00, 10.0);
      VeiculoTransporte veiculo2 = new Bicicleta(10.0); // Agora é um VeiculoTransporte válido

      System.out.println("Tarifa Ônibus: R$ " + veiculo1.calcularTarifa());
      System.out.println("Distância Ônibus: " + veiculo1.getDistancia());
      System.out.println("Tarifa Bicicleta: R$ " + veiculo2.calcularTarifa());
      System.out.println("Distância Bicicleta: " + veiculo1.getDistancia());

      System.out.println("CALCULO DE CONSUMO DA VIAGEM");
      if (veiculo1 instanceof Onibus){
         System.out.println("Consumo Veiculo 1: " +
((Onibus)veiculo1).calcularTotalCombustivel(12));
      }
      else
      {
         System.out.println("Veiculo 1: Não é um veículo motorizado");
      }

      if (veiculo2 instanceof Onibus){
         System.out.println("Consumo Veiculo 2: " +
((Onibus)veiculo2).calcularTotalCombustivel(12));
      }
      else
      {
         System.out.println("Veiculo 2: Não é um veículo motorizado");
```

Análise Crítica

Faça a sua análise crítica comparando as soluções adotadas na **parte3** e **parte4** deste roteiro.

As duas soluções são aceitáveis ?

Uma das soluções é a mais adequada ? Qual e ? Por quê ?

Observe que a solução adotada na parte 4, foi necessário inserir um trecho de código para testar a instância do objeto. Por que isso foi necessário ?

O princípio da Substituição de Liskov continua sendo obedecido ?

Consegue observar a aplicação de algum outro princípio SOLID na análise destas soluções ?