



**UNIVERSIDADE DE BRASÍLIA - UnB**  
**FACULDADE UnB GAMA - FGA**

**TRABALHO PRÁTICO DE ORIENTAÇÃO POR OBJETOS**  
**SISTEMA DE MOBILIDADE URBANA (RIDE-SHARING)**

HUGO SOUSA ROSA - 251035274  
LUIZ FELIPE MENDES CORDEIRO GOMES - 251012322  
JOSUÉ XAVIER CARNEIRO - 241011949

Brasília - DF  
Dezembro - 2025

## 1. INTRODUÇÃO

O relatório apresenta uma análise completa do projeto de um sistema de mobilidade (ride-sharing), detalhando a implementação do sistema de corridas, relações entre classes, herança, polimorfismo, associações e também justificando as exceções customizadas do código.

## 2. EXPLICAÇÃO DO CÓDIGO

O código modela um sistema semelhante ao de aplicativos de compartilhamento de corridas (assim como Uber e 99). A seguir vamos descrever o funcionamento do projeto:

### 1. Classe Categoria e suas implementações (CategoriaComum e CategoriaLuxo):

Categoria é uma interface que define dois métodos: `calcularPreco(double distanciaKm)` e `getNome()`.

As classes `CategoriaComum` e `CategoriaLuxo` implementam a interface `Categoria` e adicionam regras específicas em cada uma delas para o cálculo de preço da categoria em função da distância e também retornam o nome da categoria.

A classe `Categoria` e as suas implementações permite que o sistema trate as categorias da corrida de forma polimórfica.

### 2. Classe Corrida:

Representa a corrida que foi solicitada por um passageiro.

A classe mantém referências a `Passageiro`, `Motorista`, `Categoria`, `MetodoPagamento`, e outros atributos essenciais para a classe e possui métodos para gerenciar o início e fim da corrida, atribuir motorista a corrida, cancelar e processar o pagamento.

A classe também aplica verificações através de exceções customizadas como `"EstadoInvalidoDaCorridaException"`, `"SaldoInsuficienteException"` e `"PagamentoRecusadoException"`.

### 3. Classe Usuario e herança:

A classe `Usuario` é abstrata e contém informações comuns para as classes `Motorista` e `Passageiro` que herdam dela.

Algumas das informações presentes na classe `Usuario` são nome, cpf, senha, email, telefone e métodos para a avaliação do usuário. As classes `Motorista` e `Passageiro` possuem informações e implementações particulares para cada uma.

#### - Classe Motorista:

Contém informações, como cnh, status e veículo e métodos que alteram o status de disponibilidade do motorista. A classe permite a avaliação dos passageiros e usa verificações que lançam a exceção “MotoristaInvalidoException” para casos em que o motorista não está disponível para fazer a corrida.

#### - **Classe Passageiro:**

Possui uma lista de métodos de pagamento e pode chamar as corridas. Realiza a verificação se o passageiro adicionou método de pagamento antes de chamar a corrida e pode lançar a exceção “PassageiroPendenteException” caso não passe pela verificação. A classe também permite avaliar o motorista.

#### **4. Classe Veiculo:**

Representa o veículo do motorista e inclui atributos de placa, modelo, cor, ano e categoria.

#### **5. Classes de métodos de pagamento (Dinheiro, Pix, CartaoCredito)**

As classes Dinheiro, Pix e CartaoCredito implementam a interface MetodoPagamento que possui o metodo processarPagamento. Cada classe implementa o metodo processarPagamento de uma maneira diferente e lança SaldoInsuficienteException ou PagamentoRecusadoException quando não passar por alguma verificação.

#### **6. Enums StatusMotorista e StatusCorrida:**

Definem os estados possíveis para os motoristas e para as corridas.

### **3. ASSOCIAÇÕES, HERANÇAS E POLIMORFISMOS APLICADOS**

O projeto aplica diferentes tipos de relacionamentos:

#### **Associações:**

##### **Corrida e Passageiro:**

A classe possui um atributo passageiro, indicando que toda corrida possui um passageiro específico.

##### **Corrida e Motorista:**

A classe possui um atributo motorista, indicando que toda corrida possui um motorista designado

##### **Corrida e Categoria:**

A classe contém um atributo categoria, usado para determinar o tipo da corrida e calcular valores.

**Corrida e MetodoPagamento:**

Corrida possui um atributo metodoPagamento, o que representa uma associação com a interface MetodoPagamento (que sera implementado pelas classes Pix, Dinheiro e CartaoCredito).

**Corrida e StatusCorrida:**

A classe possui um atributo status que representa uma associação com o enum StatusCorrida e define o status da corrida.

**Passageiro e MetodoPagamento:**

A classe possui um atributo List<MetodoPagamento>, representando uma associação 0..\* entre Passageiro e seus métodos de pagamento.

**Motorista e Veiculo:**

A classe possui um atributo veiculo, indicando uma associação com a classe Veiculo.

**Motorista e StatusMotorista:**

A classe possui um atributo status que representa uma associação com o enum StatusMotorista e define o status do motorista.

**Veiculo e Categoria:**

A classe possui um atributo categoria, o que indica uma associação com a classe Categoria.

**Associações na classe MainInterativa:**

A classe MainInterativa possui atributos passageiros, corridas, e motoristas, o que indica associação com Passageiro, Motorista e Corrida.

Exemplo no código:

```
public Corrida(Passageiro passageiro, String partida, String destino, double distancia, Categoria categoria) {  
    this.passageiro = passageiro;  
    this.categoria = categoria;  
    this.distancia = distancia;  
    this.partida = partida;  
    this.destino = destino;  
    this.precoFinal = categoria.calcularPreco(distancia);  
    this.status = StatusCorrida.SOLICITADA;  
}
```

**Dependências:****MetodoPagamento e Exceções:**

Métodos de pagamento lançam SaldoInsuficienteException e PagamentoRecusadoException, caracterizando dependência.

**Corrida e Exceções:**

A classe Corrida lança e utiliza EstadoInvalidoDaCorridaException, caracterizando dependência.

### **Pix / Dinheiro / CartaoCredito e Exceções:**

Essas classes lançam exceções durante o processamento de pagamento, caracterizando dependência.

### **Motorista e Passageiro:**

O Motorista possui o método avaliarPassageiro que utiliza informações de Passageiro, caracterizando dependência.

Exemplo no código:

```
public interface MetodoPagamento {  
    public void processarPagamento(double valor, Passageiro passageiro) throws SaldoInsuficienteException, PagamentoRecusadoException;  
  
    public String getTipoPagamento();  
}
```

## **Implementação de Interface:**

### **Pix e MetodoPagamento:**

Pix implementa os métodos definidos pela interface.

### **Dinheiro e MetodoPagamento:**

Dinheiro implementa os métodos definidos pela interface.

### **CartaoCredito e MetodoPagamento:**

CartaoCredito implementa os métodos definidos pela interface.

### **CategoriaLuxo e Categoria:**

CategoriaLuxo implementa os métodos definidos pela interface para o cálculo do preço.

### **CategoriaComum e Categoria:**

CategoriaComum também implementa os métodos definidos pela interface para o cálculo do preço.

Exemplo no código:

```
public class Pix implements MetodoPagamento{  
  
    private String chavePix;
```

## **Herança:**

### **Motorista e Usuario**

A classe Motorista estende Usuario, herdando atributos e comportamentos da classe abstrata.

### **Passageiro e Usuario**

A classe Passageiro também estende Usuario, herdando atributos e comportamentos da classe abstrata.

Exemplo no código:

```
public class Passageiro extends Usuario{
    private List<MetodoPagamento> metodosPagamento = n

    public Passageiro(String nome, String cpf, String
        super(nome, cpf, senha, email, telefone);
```

## Polimorfismos:

### Polimorfismo de Inclusão:

Motorista e Passageiro são tratados como Usuario.

Pix, Dinheiro e CartaoCredito são tratados como MetodoPagamento.

CategoriaLuxo e CategoriaComum é tratada como Categoria.

Exemplo no código:

```
public class Passageiro extends Usuario{
    private List<MetodoPagamento> metodosPagamento = new ArrayList<>();

    public Passageiro(String nome, String cpf, String senha, String email, String telefone) {
        super(nome, cpf, senha, email, telefone);
    }
}
```

```
public class CartaoCredito implements MetodoPagamento{
    private String numeroCartao, bandeira, validade, cvv;

    public CartaoCredito(String numeroCartao, String bandeira, String cvv, String validade) {
        this.numeroCartao = numeroCartao;
        this.bandeira = bandeira;
        this.cvv = cvv;
        this.validade = validade;
    }
}
```

```
public class CategoriaLuxo implements Categoria {
    private static double tarifaBase = 9.0;
    private static double precoKm = 2.2;
```

### Polimorfismo por Coerção:

Conversões automáticas de int para double no cálculo de avaliações. Ex: Ao passar um int para o atributo distanciaKm ele vai converter para um double.

Exemplo no código:

```
public interface Categoria {

    double calcularPreco(double distanciaKm);

    String getNome();
}
```

### Polimorfismo Paramétrico:

Uso de listas com generics: List<MetodoPagamento>, List<Motorista> etc.

Exemplo no código:

```
public class Passageiro extends Usuario{
    private List<MetodoPagamento> metodosPagamento = new ArrayList<>();
}
```

### Polimorfismo por Sobrescrita:

processarPagamento() é implementado de três maneiras diferentes nas classes CartaoCredito, Pix e Dinheiro..

calcularPreco() implementado com valores diferentes em cada categoria (CategoriaLuxo e CategoriaComum).

Exemplo no código:

```
@Override
public void processarPagamento(double valor, Passageiro passageiro)
    throws SaldoInsuficienteException, PagamentoRecusadoException {
    if (numeroCartao == null || numeroCartao.isEmpty() || cvv == null || cvv.i
        throw new PagamentoRecusadoException("Dados do cartão inválidos.");
    }
    System.out.println("Pagamento via Cartão de Crédito de R$ " + valor + " pr
}
```

```
@Override
public void processarPagamento(double valor, Passageiro passageiro)
    throws SaldoInsuficienteException, PagamentoRecusadoException {
    if (chavePix == null || chavePix.isEmpty()) {
        throw new PagamentoRecusadoException("Chave Pix inválida.");
    }
    System.out.println("Pagamento via Pix de R$ " + valor + " processado com sucesso");
}
```

### Polimorfismo por Sobrecarga:

É possível identificar os polimorfismos por sobrecarga nas classes Corrida, Motorista, Passageiro, Usuario e Veiculo, onde existe métodos de mesmo nome mas com assinaturas diferentes.

Exemplo no código:

```
public Corrida() {}

public Corrida(Passageiro passageiro, String partida, String destino, double distancia, Cat
    this.passageiro = passageiro;
    this_categoria = categoria;
```

## 4. JUSTIFICATIVA PARA AS EXCEÇÕES CUSTOMIZADAS

As exceções foram criadas para reforçar regras necessárias para que o sistema de corridas funcione corretamente e evitar que o programa quebre em casos inválidos.

### Exceções customizadas:

“**EstadoInvalidoDaCorridaException**”: protege operações de corrida fora do estado correto.

“**MotoristaInvalidoException**”: impede motoristas sem veículo de ficarem online.

“**NenhumMotoristaDisponivelException**”: indicaria ausência de motoristas disponíveis.

“**PagamentoRecusadoException**”: falha no processamento de pagamento.

“**PassageiroPendenteException**”: passageiro sem método de pagamento cadastrado.

“**SaldoInsuficienteException**”: saldo insuficiente do passageiro

Exemplo no código:

```
public void atribuirMotorista(List<Motorista> motoristas) throws EstadoInvalidoDaCorridaException {
    if (this.status != StatusCorrida.SOLICITADA) {
        throw new EstadoInvalidoDaCorridaException("Só é possível atribuir motorista a uma corrida SOLICITADA.");
    }
}
```



## 5. UML

