# *Projeto OO*

## 1. Introdução:

O projeto tem como principal objetivo o desenvolvimento de um protótipo de sistema para um aplicativo de corridas, como a Uber, a 99 e o Cabify. O sistema deve ser feito através da linguagem de programação “Java” utilizando os conhecimentos adquiridos durante as aulas de Orientação a Objeto.

### 2.1. Objetivo Geral:

Desenvolver um protótipo de um sistema para gerenciamento de corridas que aplique os conhecimentos da Programação Orientada a Objetos adquiridos em sala de aula, garantindo a integridade dos dados e a extensibilidade do sistema.

### 2.2. Objetivos Específicos:

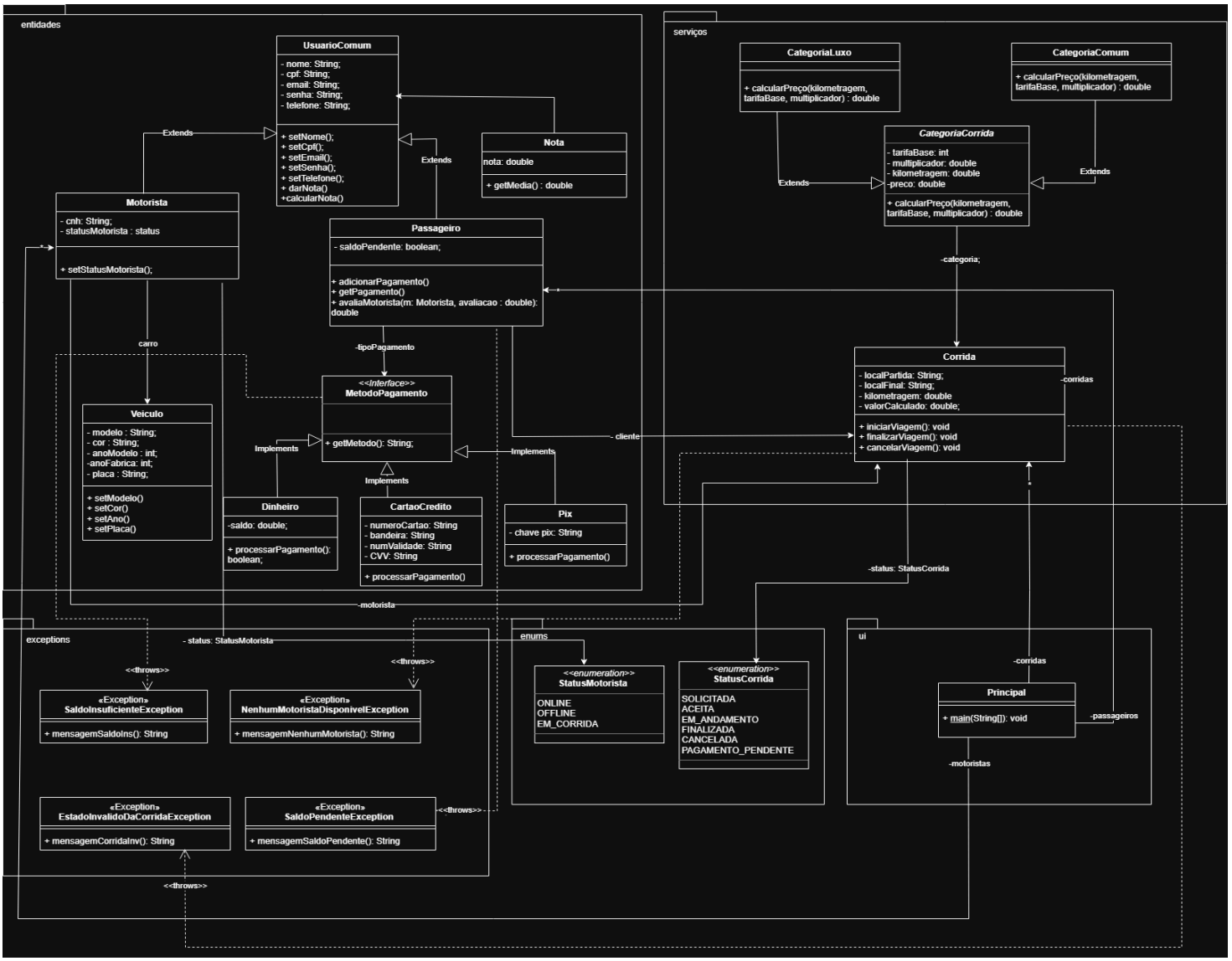
Para alcançar o objetivo geral, foram estabelecidas as seguintes metas técnicas:

- **Aplicação de Herança e Polimorfismo:** Implementar hierarquias de classes (ex: `UsuarioComum`) e interfaces (ex: `MetodoPagamento`) para reduzir a duplicidade de código e permitir a flexibilidade de algoritmos.
- **Garantia de Encapsulamento:** Proteger o estado interno dos objetos (como saldo e notas) através de modificadores de acesso e métodos assessores, prevenindo inconsistências.
- **Tratamento de Exceções:** Desenvolver uma arquitetura de tratamento de erros robusta, criando exceções personalizadas (como `SaldoInsuficienteException` e `NenhumMotoristaDisponivelException`) para gerenciar fluxos alternativos e falhas de negócio.
- **Modularidade:** Organizar o código em pacotes lógicos (`entidades`, `servicos`, `exceptions`, `ui`), facilitando a manutenção e a leitura do projeto.

# *UML*

**O projeto foi estruturado seguindo o padrão de separação de responsabilidades:**

- **src.entidades:** Contém os modelos de dados principais (`Motorista`, `Passageiro`, `Veiculo`, `Nota`) e a abstração de pagamento (`MetodoPagamento`).
- **src.servicos:** Contém a lógica de negócio principal, como a herança de categorias (`CategoriaCorrida`) e o processamento da `Corrida`.
- **src.exceptions:** Armazena as classes de erro personalizadas para controle de fluxo.
- **src.enums:** Define os estados constantes do sistema (`StatusCorrida`, `StatusMotorista`).
- **src.ui:** Contém a classe `Principal`, responsável pela interface de interação com o usuário.



# Explicação dos componentes do código

**Herança:**

A herança foi utilizada para promover o reaproveitamento do código.

**Hierarquia de Usuários (Reutilização):** As entidades `Motorista` e `Passageiro` compartilhavam um conjunto significativo de atributos (nome, CPF, email, credenciais). Para evitar a redundância de código, foi implementada a superclasse `UsuarioComum`. As subclasses estendem esta base, herdando comportamentos comuns e encapsulando apenas suas regras de negócio específicas (ex: validação de CNH para motoristas e gestão de pagamentos para passageiros).

**Extensão de Funcionalidades:** A herança também foi fundamental na criação das exceções personalizadas. Ao estender a classe `Exception` do Java (`extends Exception`), as novas classes (como `SaldoInsuficienteException`) herdam toda a infraestrutura de rastreamento de pilha.

```java
public class Passageiro extends UsuarioComum {
    private boolean saldoPendente = false;
    private MetodoPagamento tipoPagamento;

    private List<Double> notas = new ArrayList<>();

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

    public void verificarSePodeViajar() throws SaldoPendenteException {
        if (this.saldoPendente) {
            throw new SaldoPendenteException("O passageiro " + getNome() + " possui débitos pendentes.");
        }
    }

    public boolean isSaldoPendente() { return saldoPendente; }
    public void setSaldoPendente(boolean saldoPendente) { this.saldoPendente = saldoPendente; }
    public void setPagamento(MetodoPagamento tipoPagamento) { this.tipoPagamento = tipoPagamento; }
    public MetodoPagamento getPagamento() { return this.tipoPagamento; }

    public void darNota(double nota) {
        this.notas.add(nota);
    }

    public double calcularNota() {
        if(notas.isEmpty()) return 5.0;
        double soma = 0.0;
        for (Double n : notas) soma += n;
```

**Polimorfismo:**

O polimorfismo permitiu que o sistema tratasse objetos de tipos diferentes de maneira uniforme;

**Polimorfismo de Interface:** A interface `MetodoPagamento` define um contrato escrito para transações financeiras. O sistema de cobrança foi programado para interagir apenas com esta interface, desconhecendo se a instância concreta é `Pix`, `Dinheiro` ou `CartaoCredito`. Isso permite a adição de novos meios de pagamento sem a necessidade de alterar ou recompilar a classe `Passageiro` ou `Corrida`.

**Polimorfismo de Inclusão (Sobrescrita):** Na hierarquia de serviços, a classe abstrata `CategoriaCorrida` define a assinatura do método `calcularPreco()`. As subclasses `CategoriaComum` e `CategoriaLuxo` sobrescrevem (*override*) este método para aplicar suas próprias regras de precificação. A decisão sobre qual cálculo executar ocorre em tempo de execução (*Late Binding*), dependendo do objeto instanciado associado à corrida.

```java
public class CartaoCredito implements MetodoPagamento {
    private String numeroCartao, bandeira, numValidade, CVV;

    public CartaoCredito(String numeroCartao, String bandeira, String numValidade, String CVV) {
        this.numeroCartao = numeroCartao;
        this.bandeira = bandeira;
        this.numValidade = numValidade;
        this.CVV = CVV;
    }

    @Override
    public void processarPagamento(double valor) throws SaldoInsuficienteException {
        if (valor <= 0) {
            throw new SaldoInsuficienteException("Valor inválido para cartão.");
        }

        System.out.println("Pagamento de R$ " + valor + " processado no Crédito (" + bandeira + ")... Aprovado!");
    }

    @Override
    public String getMetodo() {
        return "Cartão de Crédito";
    }

    @Override
    public String toString() {
        return "Cartão de Crédito (" + this.bandeira + ")";
    }
}
```

**Associações:**

As associações foram fundamentais para modelar a interação entre os objetos, transformando classes isoladas em um sistema harmônico.

**Associação:** A classe `Motorista` possui uma associação forte com `Veiculo` (Composição/Agregação). A classe `Corrida` atua como entidade associativa, vinculando um `Passageiro` a um `Motorista` e uma `CategoriaCorrida`.

**Associação Direta :** A classe `Motorista` mantém uma associação direta com a classe `Veiculo` através do atributo `private Veiculo carro`. Isso reflete a regra de negócio de que, para estar "Online", o motorista necessita de um veículo vinculado, permitindo que o sistema acesse os dados do carro (modelo, placa) diretamente através do objeto motorista.

**Associação via Interface:** A classe `Passageiro` se associa à interface `MetodoPagamento`, e não a uma classe concreta (como `Pix`). Essa associação fraca permite que o tipo de pagamento seja trocado em tempo de execução (Dependency Injection via *setter*), sem que a classe `Passageiro` precise ser modificada.

```
public class Corrida {
    private String localPartida, localFinal;
    private double kilometragem;
    private StatusCorrida status;
    private Passageiro p;
    private Motorista motorista;
    private double valorCalculado;
    private CategoriaCorrida categoria;
```

# Exceções customizadas

**Criamos um pacote de dedicado Exceptions para garantir um tratamento preciso nos erros.**

- **SaldoInsuficienteException:** Necessário para interromper o fluxo quando um passageiro tenta pagar via cartão de crédito e não possui saldo suficiente. A principal diferença para um “erro genérico” é que ele permite que a interface sugira a troca do método de pagamento.

- **NenhumMotoristaDisponivelException:** Está exceção melhora a experiência do usuário, pois quando não há motoristas disponíveis ela permite que o sistema retorne uma mensagem alertando o passageiro.

- **EstadoInvalidoDaCorridaException:** Protege o sistema de realizar operações ilógicas, como finalizar uma corrida que já foi finalizada ou calcular o preço de uma corrida não iniciada. Garantindo a integridade do protótipo.

- **SaldoPendenteException:** Garante a proteção de receita da plataforma, está exceção impede que usuários inadimplentes consumam novos serviços.