UNIVERSIDADE DE BRASÍLIA

FACULDADE DE CIÊNCIAS E TECNOLOGIA EM ENGENHARIA

Augusto Garcia Medeiros – 232000697

Danielly Reis dos Santos – 211066258

Júlia Pêgo de Meneses – 231037745

SISTEMA DE MOBILIDADE URBANA: APLICAÇÃO PRÁTICA DE CONCEITOS DE ORIENTAÇÃO A OBJETOS

Brasília

2025

1. Introdução

1.1 Objetivo

Este relatório tem como objetivo apresentar o desenvolvimento de um protótipo de Sistema de Mobilidade Urbana, utilizando a linguagem de programação Java, seguindo os pilares da Programação Orientada a Objetos (Encapsulamento, Herança, Polimorfismo e Abstração).

O sistema simula o funcionamento de um aplicativo de transporte e permite:

- **Cadastro e Gestão de Usuários:** Passageiros e Motoristas herdam de uma base comum (Usuário), com a validação de documentos (CNH e Veículo) para motoristas.
- **Gerenciamento do Ciclo de Vida da Corrida:** Controle de estados (Aceita, Solicitada, Em Andamento, Finalizada) garantindo a integridade do sistema através de validações.
- **Processamento de Pagamentos:** Implementação de métodos de pagamento(Cartão de Débito, Crédito, PIX e Carteira Digital) através de interfaces.
- **Sistema de Avaliação:** Registro de feedback entre motoristas e passageiros, com cálculo automático da média das notas.
- **Cálculo de Tarifas:** Aplicação de polimorfismo para calcular o valor da corrida com base no tipo de veículo.
- **Tratamento de Exceções:** Implementação de uma hierarquia de exceções personalizadas para lidar com os possíveis erros do sistema.

1.2 Tecnologias Utilizadas

- **Linguagem:** Java.
- **Paradigma:** Programação Orientada a Objetos (POO).
- **Bibliotecas Principais:** java.util, java.time.
- **Ferramentas de Versionamento:** Git e GitHub
- **Ambiente de Desenvolvimento:** Eclipse e VSCode

2. Diagrama de Classes UML

Link para acessar o diagrama completo:

https://drive.google.com/drive/folders/1zr-K7Nwc4cQYaJ6275S0fnpjNRm685WN?usp=sharing

3. Conceitos da POO Aplicados

3.1 Encapsulamento

Função: Proteger os dados, garantir acesso controlado e facilitar a manutenção do código.

Exemplo no Sistema:

1)

// Na classe abstrata "Usuário", utilizamos o encapsulamento "private" para proteger dados sensíveis como a senha e o CPF, permitindo a leitura apenas via métodos públicos.

```
public abstract class Usuario {
    private String senha; // Atributo privado, invisível para outras classes
    private String CPF;

    // Método Get utilizado para controlar o acesso à senha
    public String getSenha() {
        return senha;
    }
```

```
}
```

2)

// Na classe “Motorista”, o encapsulamento protege a “CNH”, permitindo apenas métodos da própria classe ou o getter manipular o objeto.

```
public class Motorista extends Usuario {
    private CNH cnh; // Atributo privado específico do motorista

    // O acesso externo é feito apenas pelo método público
    public CNH getCnh() {
        return cnh;
    }
}
```

3)

// Na classe “Usuário”, a “mediaNota” não possui um método “set”, ela é calculada internamente toda vez que uma avaliação é recebida, mantendo a segurança do sistema.

```
public abstract class Usuario {
    private double mediaNota; // Atributo privado
    private List<Integer> avaliacoes = new ArrayList<>();

        public void receberAvaliacao(int nota) {
        this.avaliacoes.add(nota);
        this.atualizarMedia(); // Método privado que recalcula o valor
    }
}
```

3.2 Herança

Função: Reutilizar Código e criar hierarquias lógicas.

Exemplo no sistema:

1)

// Neste sistema, a classe “Motorista” e “Passageiro” herdam da classe “Usuário”, ou seja, ambos já são criados com todos os dados pessoas e sistema de avaliações que foram implementados na classe “Usuário”.

public class Passageiro extends Usuario
public class Motorista extends Usuario

2)

// Além de herdar os atributos e métodos da classe mãe, as classes filhas também podem adicionar seus próprios atributos e métodos, como a classe “Motorista” adiciona “CNH” e “Veiculo”.

```
public class Motorista extends Usuario {

   // Classe Motorista herda nome,cpf etc e adiciona o que é exclusivo dele:
   private CNH cnh;
   private Veiculo veiculo;
}
```

3)

// A herança também é utilizada no Tratamento de Exceções. Criamos uma classe base para as exceções “MobilidadeUrbanaException” e todas as outras herdam dela, isso permite capturar erros específico ou tratar falhas genéricas num único bloco

```
   // A classe SaldoInsuficiente "é um" tipo de erro de Mobilidade Urbana
public class SaldoInsuficienteException extends MobilidadeUrbanaException {
   public SaldoInsuficienteException(String msg) {
      super(msg); // Reaproveita o construtor da mãe
   }
}
```

*3.*3 Polimorfismo

Função: Permite que um mesmo método ou classe possa assumir diferentes formas ou comportamentos, dependendo do objeto que o chama.

Exemplo no sistema:

1) Sobrescrita (Override): Ocorre quando uma classe concreta implementa um método que foi definido em uma interface ou superclasse, alterando seu comportamento.

// Na interface MetodoPagamento, o método "processarPagamento" é sobrescrito de formas diferentes, o "CartaoDeCredito" simula uma autorização externa, enquanto a "CarteiraApp" verifica o saldo interno e lança exceções distintas.

```
// Na classe CartaoDeCredito
@Override
public void processarPagamento(double valor) throws
PagamentoRecusadoException {
   if (!autorizar()) throw new PagamentoRecusadoException(...);
// Lança PagamentoRecusado
}

// Na classe CarteiraApp
@Override
public void processarPagamento(double valor) throws
SaldoInsuficienteException {
   if (this.saldo < valor) throw new SaldoInsuficienteException(...);
   this.saldo -= valor;
// Lança SaldoInsuficiente
}
```

2) Sobrecarga (Overload): É a capacidade de ter, na mesma classe, diversos métodos com o mesmo nome, diferenciado por lista de parâmetros diferentes).

// Na classe "CentralDeCorridas", aplicamos sobrecarga no método "solicitarCorrida", que possui duas versões: uma simplificada com apenas passageiro, origem e destino, e outra completa que permite especificar também o método de pagamento e tipo de veículo. Ambas têm o mesmo nome, mas listas de parâmetros diferentes, caracterizando a sobrecarga.

// O método de cima utiliza uma configuração default, enquanto a debaixo permite ao usuário escolher a preferência.

```
public Corrida solicitarCorrida(Passageiro p, String origem, String destino) /

public Corrida solicitarCorrida(Passageiro p, String origem, String
destino,MetodoPagamento mp, TipoVeiculo tv)
```

3) **Paramétrico:** Ocorre quando utilizamos classes ou interfaces genéricas que podem operar com qualquer tipo de dado, especificado apenas no momento da declaração (uso de Generics).

// Na classe "Usuário", temos uma lista parametrizada para inteiros.

private List<Integer> avaliacoes = new ArrayList<>()

// Na classe “CentralDeCorridas” temos uma lista parametrizada para Motoristas.

private List<Motorista> motoristasCadastrados = new ArrayList<>();

3.4 Exceções

Função: Permitem criar erros específicos do domínio da aplicação, facilitando o tratamento de situações excepcionais de forma mais clara e organizada.

Exemplo no sistema:

1)

// A classe “MobilidadeUrbanaException” é a exceção base personalizada do sistema, que estende “Exception”, tornando-se uma exceção verificada, o que obriga seu tratamento ou declaração no código. Ela serve como superclasse para todas as outras exceções específicas do domínio (como “MotoristaInvalidoException”), criando uma hierarquia organizada.

```
public class MobilidadeUrbanaException extends Exception{
    private static final long serialVersionUID = 1L;

   public MobilidadeUrbanaException(String mensagem) {
        super(mensagem);
    }
}
```

2)

//Com base na “MobilidadeUrbanaException”, a classe “MotoristaInvalidoException” é uma exceção especializada que herda da exceção base do sistema. Ela representa um erro específico, identifica situações onde o motorista não atende aos requisitos para executar uma corrida (CNH inválida, veículo inválido).

```
public class MotoristaInvalidoException
extends MobilidadeUrbanaException{
    private static final long serialVersionUID = 1L;

   publicMotoristaInvalidoException(String mensagem) {
        super(mensagem);
```

```
    }
}
```

Exceções customizadas tratam erros de negócio como situações normais do sistema, não apenas falhas técnicas, tornando o código mais claro e fácil de entender.

4. Associações entre as Classes

Tipo	**Exemplo**	**Descrição**
Herança	Passageiro extends Usuario	Passageiro é um tipo especializado de Usuário
Agregação	Passageiro contém List<MetodoPagamento>	Passageiro têm uma lista de métodos de pagamento
Associação	Corrida liga Passageiro e Motorista	a Corrida conecta temporariamente as duas classes

Cardinalidades:

- **Passageiro 1 → N Corrida** (um Passageiro pode solicitar N corridas)
- **Motorista 1 → 1 Veículo** (um Motorista pode ter apenas um Veículo)

5. Exceções Personalizadas

Exceção	**Explicação**	**Exemplo no código**
SaldoInsuficienteException	Saldo insuficiente para pagar o valor da corrida	CarteiraApp (processarPagamento)
PagamentoRecusadoExcepion	Simula o pagamento recusado por operadoras de cartões de crédito e débito	CartaoDeCredito (processarPagamento)
NenhumMotoristaDisponivelException	Impede que uma corrida seja criada sem um motorista disponível com os requisitos solicitados	CentralDeCorridas(buscarMotoristaDisponivel)
EstadoInvalidoDaCorridaException	Garante que o fluxo da corrida seja respeitado corretamente	Corrida(iniciar, finalizar e cancelar)
PassageiroPendenteExcepion	Bloqueia o passageiro caso tenha pendências de corridas anteriores	CentralDeCorridas (solicitarCorrida)

MotoristaInvalidoException	Motorista com CNH vencida ou sem veículo válido é impedido de ficar online	Motorista (ficarOnline)