



UnB - FCTE

Faculdade de Ciências e Tecnologias em Engenharia



**conhecimento em movimento
sociedade em transformação**

Sistema de Mobilidade Urbana (Ride-Sharing)

Ana Paula Jardim Rezende Vilela - 241010880

Daniel Almeida Frota - 251041010

Déborah da Silva Fragoso - 251036333

Gabriela Cristina de Moraes Barbosa - 232023691

Brasília - DF
Novembro de 2025

1. Introdução

1.1 Objetivo

Este relatório tem como objetivo apresentar e explicar o desenvolvimento de um Sistema de Mobilidade Urbana (Ride-Sharing), implementado em Java, detalhando os fundamentos da Programação Orientada a Objetos aplicados neste projeto. O sistema conta com:

- Cadastro de usuários (Passageiros e Motoristas);
- Suporte de diferentes formas de pagamento (PIX, Dinheiro e Cartão);
- Cadastro de veículo e CNH;
- Status de disponibilidade (Online, Offline e Em Corrida);
- Categorias com especificação diferente (Comum e Luxo);
- Cancelamento da viagem;
- Coleta de feedback.

1.2 Tecnologias Utilizadas

- Linguagem de programação: Java;
- Plataforma: Eclipse IDE;
- Paradigma: Orientação a Objetos;
- Bibliotecas: java.util e tratamento de exceções customizado.

2. Diagrama de Classes UML

2.1 Links do Diagrama

Drive:

https://app.diagrams.net/#G10bXSddA04tACgj74d0stwqWljYjbQIAA#%7B%20pageId%22%3A%22sEJqAz_hMhrWfwqWvW5w%22%7D

Acessar Draw.io pelo Drive.

3. Aplicações da Programação Orientada a Objetos

3.1 Associações

Existem três tipos principais de associação em Orientação a Objetos: Associação Simples, Agregação e Composição. Todos eles foram aplicados no projeto no intuito polir o código e apresentar o sistema de modo satisfatório. A Associação Simples é quando um objeto utiliza outro, ou seus recursos, no código tem-se como exemplo a associação de uso no histórico de corridas do passageiro, em que o objeto historicoCorridas é usado pelo Passageiro p para armazenar a quantidade de corridas feitas por ele. Os objetos são independentes, o historicoCorridas não depende de “p” para existir, até pode ser utilizado por um Motorista m numa mesma relação.

A Agregação é quando tem uma relação de “posse” com outro, quando um objeto possui outro. Um exemplo no código é quando o Motorista m possui um Veiculo veiculo. Neste caso os objetos ainda são independentes, veiculo pode existir sem “m”, e vice-versa.

Finalmente, na Composição um objeto faz parte de outro de forma essencial, se o objeto inteiro deixar de existir, as outras partes morrem junto. Um exemplo de composição no projeto aparece na classe Corrida, que cria internamente objetos como Scanner e Random. Esses objetos são parte essencial do comportamento da corrida e só existem dentro dela; quando a instância de Corrida deixa de existir, eles deixam de existir também.

Mais exemplos de associações encontrados no projeto:

Tipo	Exemplo	Descrição
Herança	Passageiro extends Usuario	Especialização de Usuários. O passageiro adquire atributos comuns da classe base Usuario (nome, CPF, etc.).
Agregação	Motorista contém Veiculo	Relação de posse (<i>has-a</i>). O motorista possui um veículo, mas o Veiculo existe independentemente do motorista.
Composição	Corrida cria Random e Scanner	Relação de posse forte. Os objetos utilitários são essenciais para a Corrida e moram junto com a sua instância.
Associação	Corrida liga Passageiro e Motorista	Relação central do sistema. A Corrida conecta o usuário que solicita o serviço ao usuário que o executa.
Associação (Uso)	Passageiro utiliza Carteira	Relação onde o Passageiro precisa de um método de pagamento para operar no sistema.

3.2 Encapsulamento

O Encapsulamento é um princípio em OO que permite esconder detalhes internos de uma classe e controlar o acesso aos seus atributos e métodos. Isso garante acesso controlado e protege os dados, evitando acessos diretos indevidos e trazem praticidade, sendo de fácil manutenção.

Exemplos de encapsulamento apresentado no projeto:

1 -

```
public class Login{  
    private Passageiro passageiroLogado;  
    private String email, senha;  
    public Passageiro getPassageiroLogado() {return this.passageiroLogado}  
}
```

Neste primeiro exemplo, o encapsulamento está presente para impedir acesso externo aos atributos, como as strings *email* e *senha*, com modificador de acesso *private*. Enquanto os métodos costumam ter acesso público para permitir o uso fácil e controlado em outras classes.

2 -

```
public class Motorista extends Usuario{  
    private double estrelas = 5;  
    public void setEstrelas(double novaNota) {void}  
}
```

Já neste exemplo, os modificadores de acesso permitem que apenas pelo método *setEstrelas*, que é *public* o valor de *estrelas* seja alterado, evitando alterações indevidas.

3 -

```
public class Motorista extends Usuario{  
    private double estrelas = 5;  
    public double getEstrelas() {return estrelas;}  
}
```

Assim como o exemplo anterior, o *private* protege *estrelas* e para ter acesso ao seu valor por fora da classe, somente chamando o método *getEstrelas*.

3.3 Herança

A Herança é um mecanismo em OO que permite que uma classe derive de outra, reutilizando seus atributos e comportamentos, de fato “herdando” as “características” da “classe-mãe”, chamada superclasse. A classe que deriva da superclasse é a subclasse. Permite a reutilização de código (evitando que seja duplicado), além de organizar a hierarquia, polindo e otimizando o projeto em questão. É caracterizado por extends numa seta de ponta triangular aberta em UML, e no início da criação da classe como é possível ver no exemplo:

```
public class Motorista extends Usuario{  
    public Veiculo getVeiculo() {return this.veiculo;}  
}
```

Aqui a classe Motorista herda atributos de Usuario como nome, email, senha etc., mas diferente da outra subclasse de Usuario (a classe Passageiro), para o sistema, é necessário que um Motorista m tenha um Veiculo v, portanto um atributo exclusivo para m é adicionado: o getVeiculo.

3.4 Polimorfismo

O Polimorfismo é um princípio em OO que permite que objetos de diferentes classes respondam de maneira distinta a uma mesma operação. Ou seja, um mesmo método pode apresentar comportamentos diferentes dependendo do objeto que o utiliza. Existem diferentes tipos de Polimorfismo em OO, como a Sobrescrita de Métodos, que é a mais comum, os outros tipos são: Sobrecarga de Métodos, Polimorfismo por Inclusão e Polimorfismo por Coerção. Esse princípio garante um código mais limpo, legível e flexível, já que permite a reutilização de código de um jeito muito simples. Alguns exemplos de Polimorfismo no sistema:

1. Polimorfismo por Sobrescrita (*Override*)

I) Sobrescrita do Método de Pagamento (`processarPagamento`)

O método `cadMtdPag()` é definido na classe abstrata `Carteira` e implementado de forma diferente em cada subclasse (`Dinheiro`, `PIX`, `Cartao`). Essa sobrescrita permite que o sistema chame o método genérico, enquanto o objeto concreto executa a lógica específica.

Implementação na subclasse `Dinheiro`:

```
@Override
```

```
public void processarPagamento(float valorCorrida) throws SaldoInsuficienteException {  
    if (this.carteiraFisica < valorCorrida) {  
        throw new SaldoInsuficienteException("Saldo insuficiente.");  
    }  
    this.carteiraFisica -= valorCorrida;  
}
```

Implementação na subclasse `PIX` (simulação):

```
@Override
```

```
public void processarPagamento(float valorCorrida) throws PagamentoRecusadoException {  
    // Simula a falha de 20% no PIX  
    if (Math.random() < 0.2) {  
        throw new PagamentoRecusadoException("PIX recusado.");  
    }  
    // Processamento bem-sucedido  
}
```

2. Polimorfismo por Sobrecarga (*Overload*).

I) Sobrecarga de Construtores em `Motorista`

A implementação de construtores alternativos é um tipo de polimorfismo por sobrecarga, pois temos assinaturas diferentes para métodos de mesmo nome (Motorista). Isso permite a criação de objetos de formas distintas: para cadastro interativo (sem argumentos) ou para inicialização de dados (com todos os argumentos).

```
public class Motorista extends Usuario {  
  
    public Motorista() {  
        // Construtor padrão para o cadastro interativo  
    }  
  
    public Motorista(String nome, String email, String status, Veiculo veiculo) {  
        // Construtor sobreescrito para a inicialização de presets  
        this.setNome(nome);  
        // ... inicialização dos demais atributos  
    }  
}
```

3. Polimorfismo Paramétrico (Generics)

Para a gestão das coleções de objetos do sistema, optou-se pela utilização de Generics (Tipos Genéricos) do Java. Essa abordagem permite o Polimorfismo Paramétrico, onde uma única classe de estrutura de dados (`ListadeCadastros<T>`) pode manipular diferentes tipos de objetos (Motorista, Passageiro, etc.) sem a necessidade de reescrever a lógica de inserção, remoção ou busca para cada entidade. Além da reutilização de código, o uso de Generics garante a Segurança de Tipos (Type Safety) em tempo de compilação, evitando erros de casting (conversão) indevidos.

3.5 Exceções

Exceções são eventos que ocorrem durante a execução de um programa e interrompem o fluxo normal das instruções. Elas são usadas para sinalizar e tratar erros de forma controlada. No caso deste projeto, além das Exceções usuais do Java, foram utilizadas Exceções personalizadas.

Exceção Customizada	Quando Ocorre	Onde o Código Dispara (Lançamento)
NenhumMotoristaDisponivelException	A busca por um motorista "ONLINE" não retorna resultados compatíveis (lista de disponíveis vazia).	Disparada dentro de <code>Corrida.atribuirMotorista()</code> .
EstadoInvalidoDaCorridaException	Um motorista tenta realizar uma ação inconsistente com o estado atual da corrida (violação do ciclo de vida).	Disparada em métodos de ciclo de vida da <code>Corrida</code> (ex: <code>iniciarViagem()</code> , <code>cancelarViagem()</code>).
SaldoInsuficienteException	O valor da corrida excede o saldo disponível na carteiraFisica.	Disparada em <code>passageiro.Dinheiro.processarPagamento()</code> .
PagamentoRecusadoException	Simulação de recusa pela operadora de cartão de crédito ou falha na chave PIX.	Disparada em <code>passageiro.PIX</code> ou <code>passageiro.Cartao</code> (no método <code>processarPagamento</code>).