Ana Clara Teixeira Guimarães - 251005883
Iara Severino Souza - 251035292
Letícia da Silva Pereira - 251040256

CRIAÇÃO DE UM SISTEMA DE MOBILIDADE URBANA

ORIENTAÇÃO À OBJETOS

Prof. André Luiz Peron Martins Lanna

BRASÍLIA

NOVEMBRO/2025

Introdução

A proposta do projeto final é modelar e desenvolver um protótipo de sistema para um aplicativo de compartilhamento de corridas, semelhante a plataformas como Uber, 99 ou Cabify. Este ecossistema digital conecta dois tipos principais de usuários: passageiros, que convocam transporte, e motoristas, que oferecem o serviço. O documento descreve como essas técnicas foram incorporadas ao código, apresenta o modelo UML e as principais decisões de projeto, demostrando como os conteúdos estudados em sala de aula na disciplina de Orientação a Objetos foram aplicados ao desenvolvimento do projeto.

Desenvolvimento

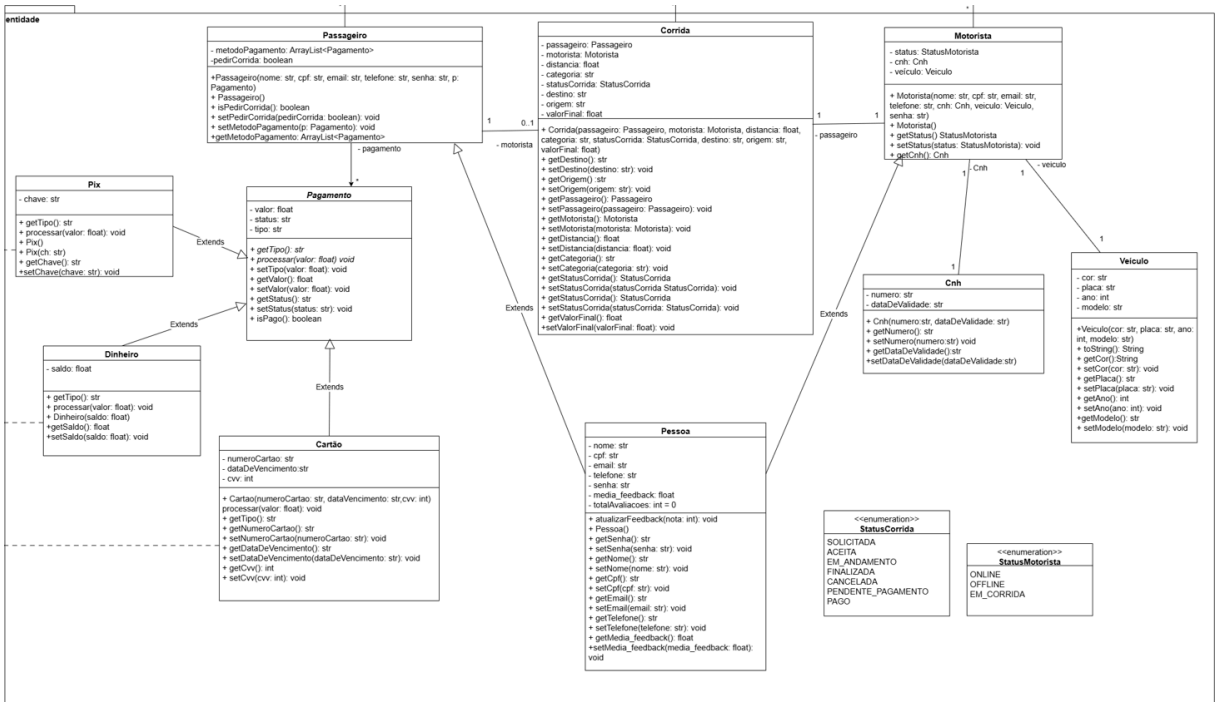
1. Modularização do Projeto

A arquitetura do sistema foi planejada para refletir com os princípios da Programação Orientada a Objetos estudados ao longo da disciplina e, ao mesmo tempo, garantir uma organização modular, clara. Numa visão geral do projeto, o sistema foi estruturado a partir de quatro camadas principais: **principal**, **entidades, serviços e exceções.**

No centro do sistema estão as **entidades**, que representam os elementos fundamentais do domínio: pessoas, motoristas, passageiros, veículos, cnh, corrida, pagamento e formas de pagamento. A lógica essencial do sistema - como solicitar corridas, gerenciar o estado de uma viagem, realizar o pagamento e registrar feedback - é executada pelos serviços.

O **pacote entidades** reúne todas as classes que representam os elementos principais do sistema. Ele é o núcleo estrutural do projeto, porque guarda tudo o que existe no aplicativo: pessoa, motoristas, passageiros, veículos, corridas e formas de pagamento.
Dentro deste pacote estão:

- As classes principais: Pessoa, Passageiro, Motorista, Veículo e Corrida. Elas representam os elementos mais importantes do sistema.
- A hierarquia de pagamento, onde existe uma classe principal (Pagamento) e várias subclasses que representam os tipos de pagamento possivel (Cartao, Dinheiro e Pix).
- Enums, que são listas de valores fixos usados no sistema, como os estados da corrida(StatusCorrida) e da situação do motorista(StatusMotorista).

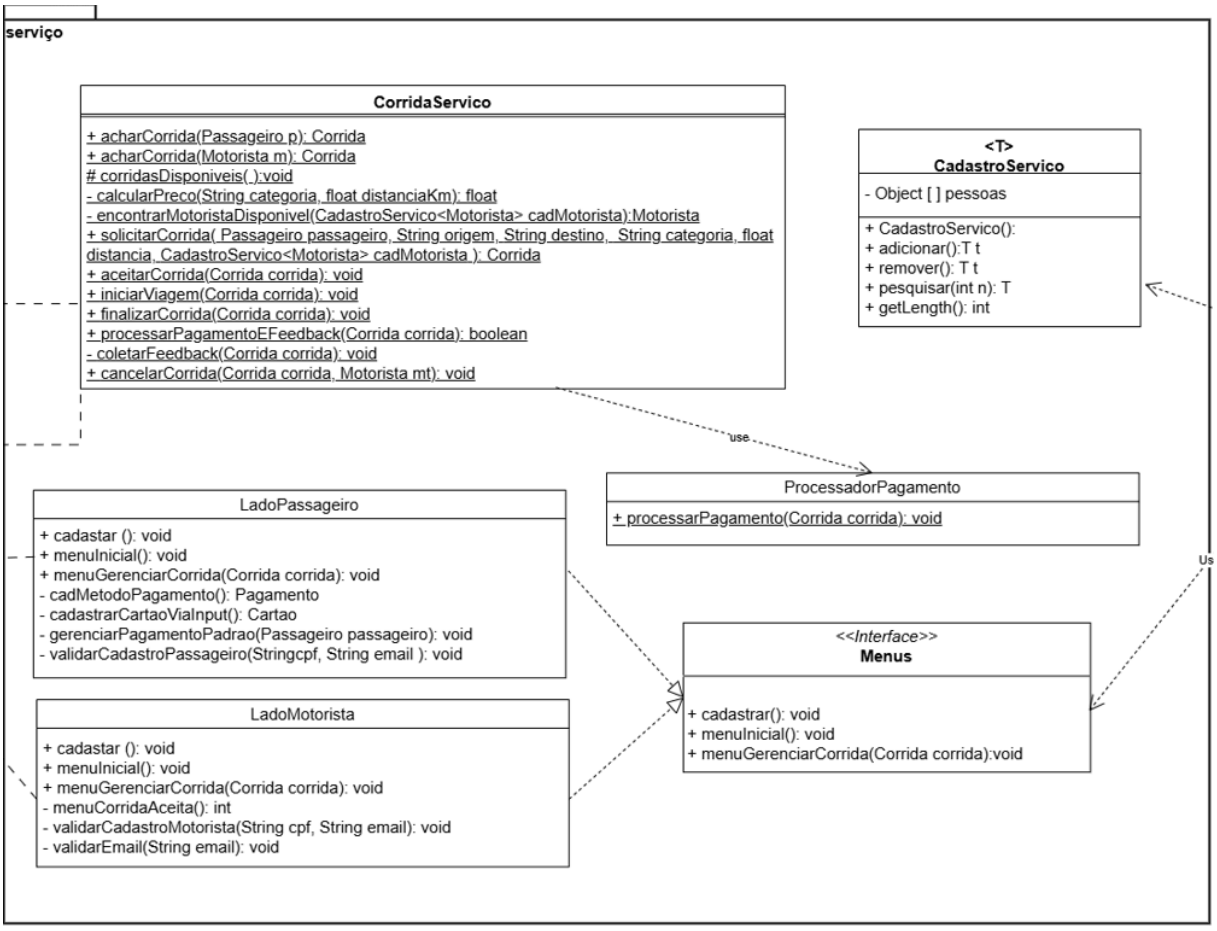


O **pacote serviços** reúne as classes que fazem o sistema funcionar na prática. Enquanto o pacote entidades descreve “o que tem”, o pacote serviços descreve o que acontece no aplicativo. É onde os processos mais importantes se encontram.

Dentro deste pacote estão:

- CadastroServico
- CorridaServico
- LadoMotorista
- LadoPassageiro
- Menus
- ProcessadorPagamento

A ideia desse pacote é centralizar operações, evitando que entidades armazenem responsabilidades que não fazem parte da sua natureza.

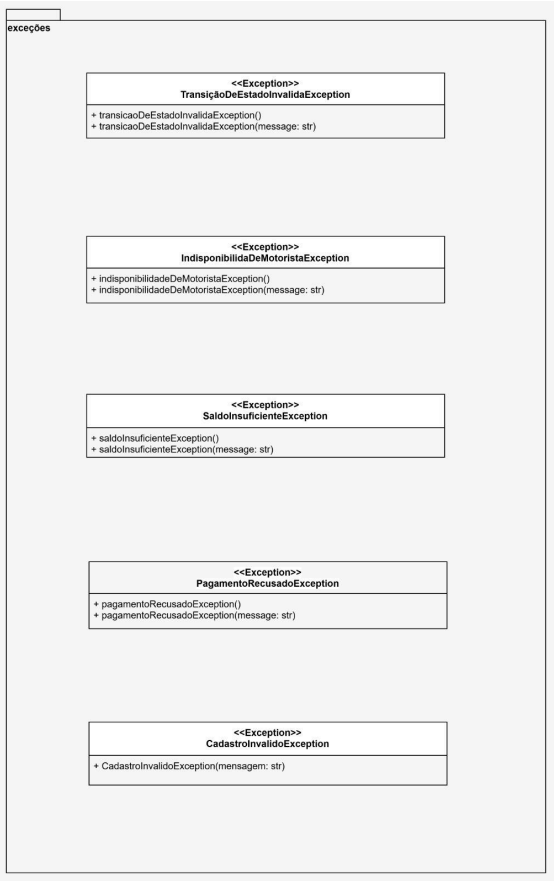


O **pacote exceções** guarda todas as exceções personalizadas criadas para o sistema. Essas exceções representam erros importantes que podem acontecer durante o uso do aplicativo e ajudam a deixar o sistema mais seguro e organizado. Com elas, o

programa consegue identificar e tratar situações anormais sem travar, como falta de saldo, falha no cartão ou tentativas de ações proibidas.
As exceções implementadas incluem:

- SaldoInsuficienteException
- PagamentoRecusadoException
- IndisponibilidadeDeMotoristaException
- TransicaoDeEstadoInvalidaException
- CadastroInvalidoException

Esse pacote separa o fluxo normal do sistema do fluxo de erros.



2. Criação e Orientação a objetos

As classes descrevem as características e comportamentos de algo no sistema, enquanto objetos são instâncias reais criadas a partir dessas classes. No projeto de mobilidade urbana, esse conceito aparece de forma direta: criamos classes como **Pessoa, Passageiro, Motorista, Veículo e Corrida.**

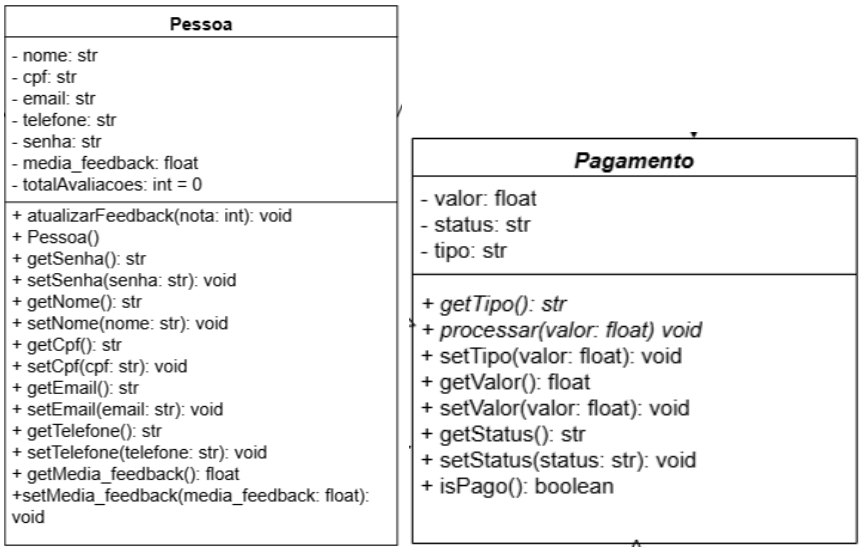
A partir delas, o sistema cria objetos reais, como um passageiro específico ou uma corrida solicitada, permitindo que o aplicativo funcione com dados e comportamentos definidos. Essa estrutura mostra, na prática, como os conceitos básicos de orientação a objetos servem de base para organizar e representar todo o funcionamento do sistema.

2.1. Abstração

No projeto, as classes abstratas — aquelas que servem como modelos gerais para outras classes mais específicas — são as classes: **Pessoa e Pagamento**. A classe Pessoa reúne atributos e comportamentos comuns a qualquer pessoa do sistema(Ex.: Nome, email, telefone,etc.), enquanto as subclasses Passageiro e Motorista adicionam funções específicas de cada tipo(Ex.: CNH e Veículo em Motorista). Da mesma forma, o Pagamento foi representado por uma classe abstrata, que define o comportamento geral de um pagamento, mas deixa que cada tipo específico(como cartão, dinheiro ou PIX) implemente sua própria lógica.

Assim, a abstração organiza melhor o sistema, evita repetição de código e permite adicionar novas funcionalidades (como novos tipos de pagamento ou novos perfis de usuários) sem modificar o que já existe (facilita a manutenção).

No UML:



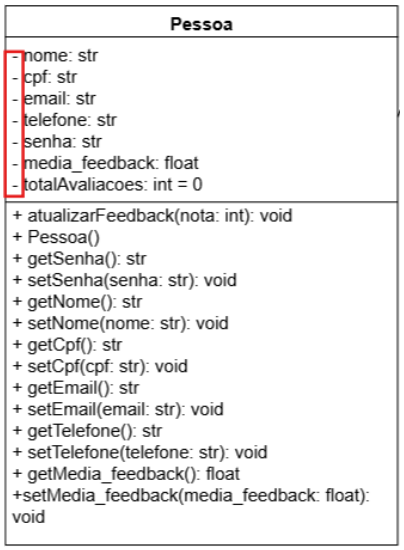
No código:

```
public abstract class Pessoa {
```

```
public abstract class Pagamento {
```

2.2. Encapsulamento

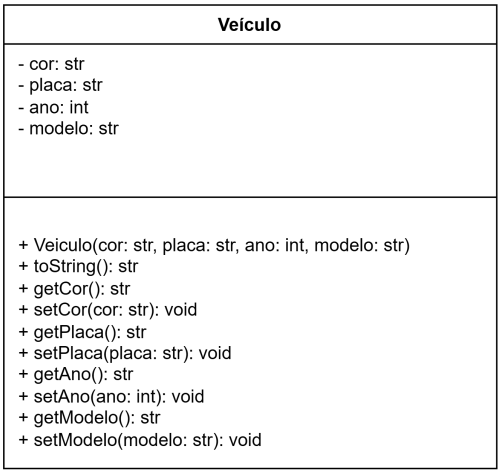
No projeto, o encapsulamento aparece de forma direta no uso de atributos privados e métodos públicos getters e setters. Todas as classes do pacote entidades — como Pessoa, Passageiro, Motorista, Veiculo e Corrida — escondem seus dados internos, permitindo acesso apenas pelas operações definidas pela própria classe. Isso garante segurança e evita que o sistema entre em estados inválidos.



Um exemplo importante de encapsulamento está no cálculo da média de avaliações (**media_feedback**). A classe responsável **não expõe diretamente a lista de notas,** mas possui um método próprio (**atualizarfeedback**) que atualiza a média de forma controlada. Assim, nenhum outro trecho do sistema consegue alterar a média diretamente, garantindo segurança e sigilo.

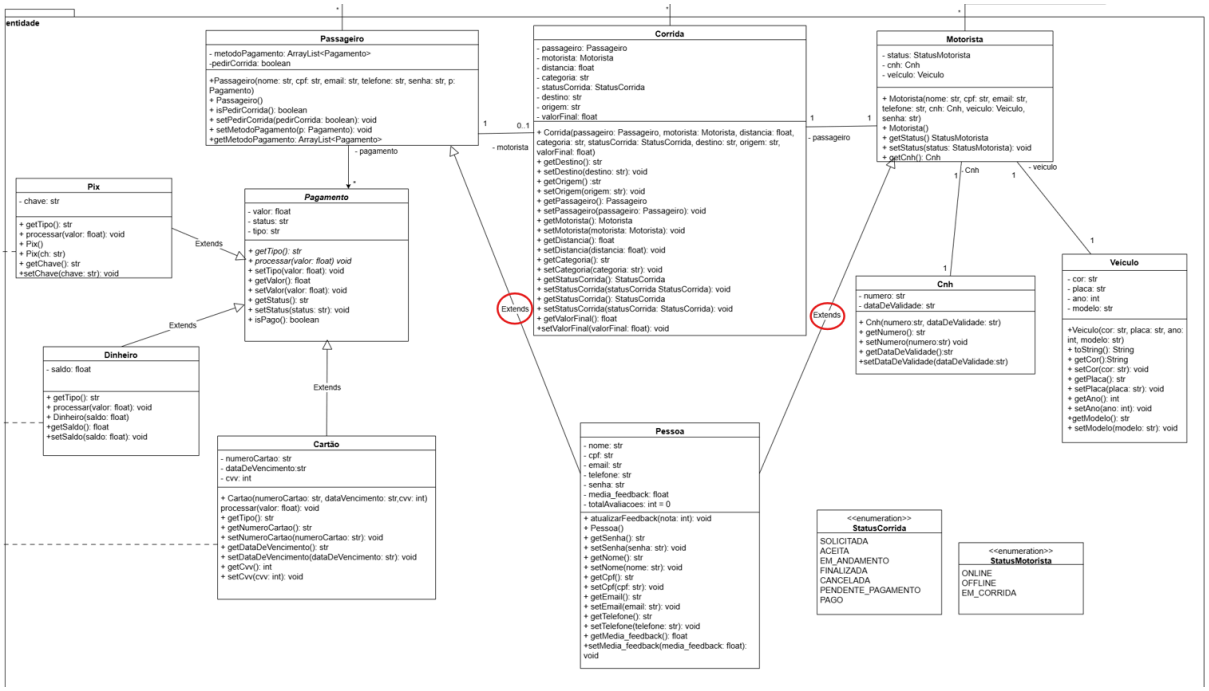
```
10      private float media_feedback;
```

Outro conceito importante é a **retenção de estado** (quando o objeto precisa manter informações ao longo do tempo). Isso é evidente na classe **Veiculo**, que guarda seu estado de características (placa, modelo, cor, ano). Esses dados permanecem armazenados enquanto o objeto existir, permitindo que o sistema valide se o motorista está apto a operar. O mesmo ocorre na classe Corrida, que retém seu estado atual (solicitada, aceita, em andamento, finalizada, etc.), garantindo que cada ação só possa ser executada no momento certo.

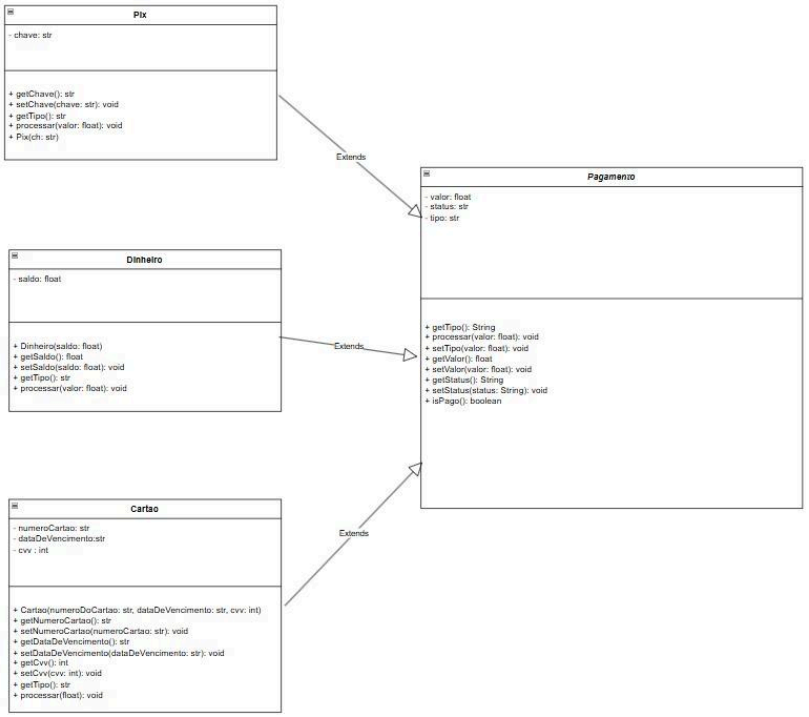


2.3. Heranças

No projeto, esse conceito aparece principalmente na relação entre **Pessoa, Passageiro e Motorista.** Como ambos compartilham informações básicas (nome, CPF, e-mail, telefone, senha e feedback), eles herdam esses elementos diretamente da classe Pessoa, evitando repetição de código e mantendo os atributos comuns.



Já a classe **Pagamento** tem um tipo de **herança hierárquica**, já que pagamento possui várias subclasses. Nessa classe, os diferentes tipos — **cartão, pix e dinheiro** — herdam e implementam a estrutura geral definida em Pagamento. A utilização dessas heranças evita a repetição do código e facilita a manutenção do programa.

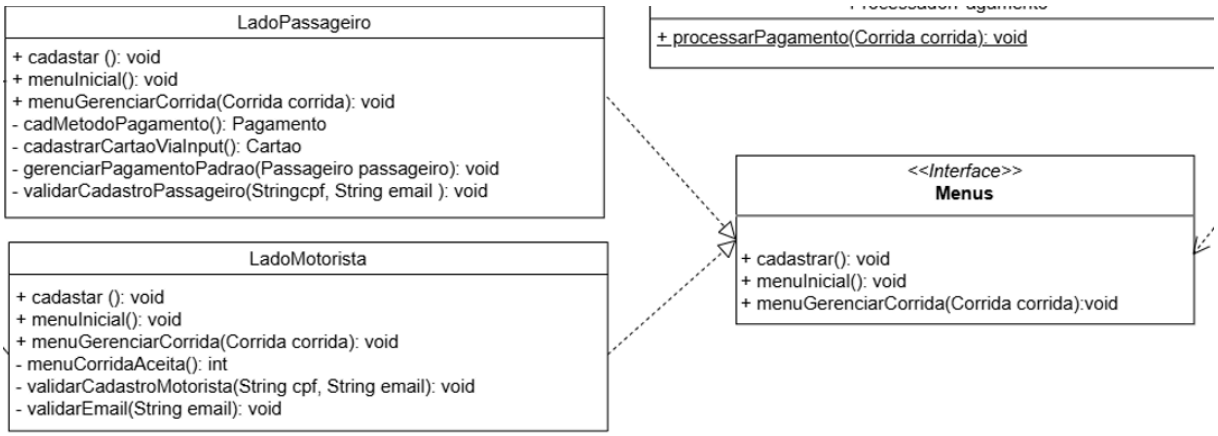


Uma interface é um tipo de estrutura que define um conjunto de métodos, mas não implementa o comportamento deles. Ela funciona assim: se uma classe herda uma interface, ela é obrigada a fornecer o código para esses métodos.

Em resumo, essa estrutura permite:

• padronizar o comportamento de diferentes classes e permite o polimorfismo;

No nosso código, a **interface Menus** é uma superclasse, enquanto **LadoPassageiro e LadoMotorista** são subclasses que implementam os métodos da interface de maneira distinta. Ou seja, o menu do passageiro apresenta apenas funções voltadas para esse tipo de usuário e o menu do motorista possui opções específicas desse papel.




```
16    public class LadoMotorista implements Menus {
33    public class LadoPassageiro implements Menus {
```

Outro exemplo de herança presente no projeto aparece no tratamento de exceções. Todas as exceções criadas — como SaldoInsuficienteException, PagamentoRecusadoException, IndisponibilidadeDeMotoristaException e TransincaoDeEstadoInvalidaException — herdam diretamente da classe Exception.

2.4. Polimorfismos

O projeto utiliza vários tipos de polimorfismo ensinados na disciplina, cada um aplicado em um ponto diferente do sistema. Eles aparecem principalmente em:

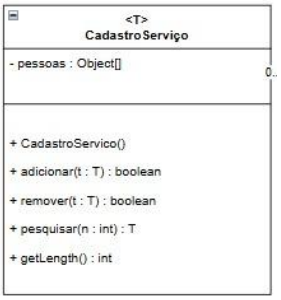
1. **Polimorfismo por Sobrescrita (Override)**
   Ocorre quando uma subclasse modifica o comportamento de um método herdado.
   a) Menus + LadoMotorista e LadoPassageiro:
   As classes LadoMotorista e LadoPassageiro herdam da interface Menus e sobrescrevem os métodos definidos nela, como:
   - cadastrar();
   - menuInicial();
   - menuGerenciarCorrida();

   Cada uma implementa o comportamento de acordo com o seu tipo de usuário.

   b) Pagamento + processar
   As classes Cartao, Dinheiro e Pix sobrescrevem o método processar para aplicar as suas exceções.

   c) Pagamento + getTipo();
   As classes Cartao, Dinheiro e Pix sobrescrevem o método getTipo(), cada uma retorna seu nome.

2. **Polimorfismo Paramétrico (Generics)**
   Ocorre quando um código funciona para vários tipos sem precisar ser reescrito.
   No projeto, isso aparece no:
   CadastroServico<T>
   Esse serviço usa generics para cadastrar qualquer tipo de usuário: Motorista ou Passageiro.
   Exemplo:



3. **Sobrecarga de Métodos (Overload)**
   Ocorre quando métodos têm o mesmo nome, mas assinaturas diferentes.

No projeto, isso aparece em três lugares importantes:

a) CorridaServico

acharCorrida(Passageiro p)

acharCorrida(Motorista m)

Ambos se chamam “achar corrida”, mas fazem coisas diferentes.

b) Principal (login)

login(LadoMotorista lm)

login(LadoPassageiro lp)

Ambos tratam o login, mas cada um recebe um tipo diferente.

Isso é útil porque o sistema usa a mesma ideia de operação, mas aplicada a atores distintos.

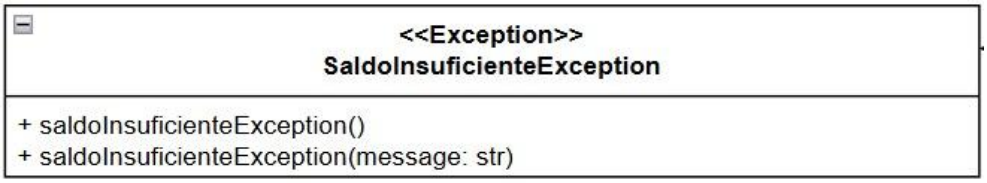
```
134    private static void login(LadoMotorista lm)
174    private static void login(LadoPassageiro lp)
```

c) Exceções

Todas, menos a exceção CadastroInvalidoException, tem dois métodos: um para lançar mensagem para o usuário e o outro que retorna a mensagem para a classe mãe.

saldoInsuficienteException()

saldoInsuficienteException(message: str)



4. **Polimorfismo por Coerção**

Acontece quando o sistema interpreta um tipo como outro de forma automática ou forçada. No projeto, isso aparece, por exemplo:

Verificar o método de pagamento selecionado

A coerção serve para **converter a referência de um tipo genérico (Pagamento) para um tipo específico(o escolhido pela pessoa) (Dinheiro)**.

- Cartao
- Pix
- Dinheiro

Quando é necessário acessar comportamentos específicos, é feita a coerção:

```
for (Pagamento p : mp) {
    if (p.getTipo().equals("DINHEIRO")) {
        dinheiro = (Dinheiro) p; // Coerção
        break;
    }
}
```

2.5. Associação entre Classes e Enums

A associação representa um relacionamento entre duas classes, mostrando como os objetos delas interagem dentro do sistema. No projeto de mobilidade urbana, as **associações foram usadas para expressar as ligações entre os elementos** e garantir que cada classe acesse somente o que faz sentido em seu contexto.

Os principais exemplos de associações são:

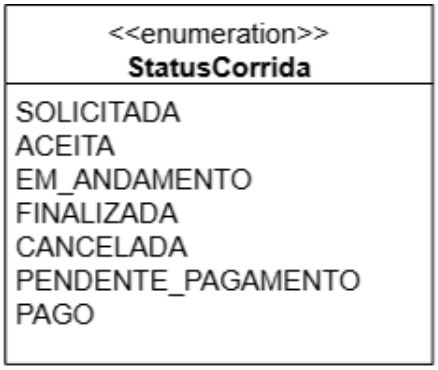
- **Passageiro → Pagamento:** Um passageiro pode ter vários meios de pagamento. Essa é uma associação do tipo 1 para muitos (1..*), representando que cada passageiro armazena uma lista de métodos de pagamento.
- **Motorista → Veículo:** Cada motorista possui um único veículo ativo, o que forma uma associação 1 para 1 (1..1). Isso reflete a realidade dos aplicativos de mobilidade, já que um motorista dirige apenas um carro por vez.
- **Corrida → Passageiro:** Toda corrida é sempre criada por um passageiro específico, formando uma associação 1 → 1.
- **Corrida → Motorista:** A corrida inicialmente não tem motorista, mas após a aceitação, ela passa a estar associada a um motorista. Por isso, esse relacionamento pode ser visto como 0..1 para 1.
- **Pessoa → media_feedback:** Cada usuário (passageiro ou motorista) pode receber muitas avaliações, formando uma associação 1 → 0... Já cada feedback pertence exatamente a um avaliador e um avaliado.

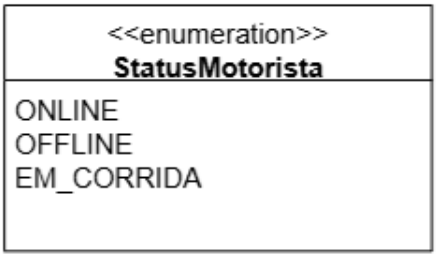
O conceito de **multiplicidade** indica quantos objetos de uma classe podem se relacionar com objetos de outra classe. No projeto, a multiplicidade ajuda a representar corretamente a lógica de um aplicativo de mobilidade. Por exemplo, um Passageiro pode ter vários métodos de pagamento (1..*), enquanto uma Corrida sempre pertence a um único passageiro (1..1). Já o Motorista possui apenas um veículo ativo (1..1), e uma Corrida só terá motorista após a aceitação (0..1).

Os **enums (tipos enumerados)** são usados para representar conjuntos de valores pré-definidos. Eles são úteis quando algo deve ter apenas algumas opções possíveis. No projeto, os enums ajudam a organizar e padronizar informações que não mudam.

Foram definidos dois enums principais:

- **StatusCorrida** – controla o estado da corrida (**SOLICITADA, ACEITA, EM_ANDAMENTO, FINALIZADA, CANCELADA, PENDENTE_PAGAMENTO, PAGA**)
- **StatusMotorista** – indica a situação atual do motorista (**ONLINE, OFFLINE, EM_CORRIDA**).





2.6. Tratamento de Exceções

1. **Exceções de Pagamento**

As exceções relacionadas ao pagamento foram tratadas diretamente nos métodos **processar()** de cada forma de pagamento, pois é nesse ponto que ocorre a validação específica de cada método:

2. **Cartão:** A recusa pode acontecer por número inválido, CVV incorreto ou dados incompletos. Por isso, o método processar() da classe Cartao lança **PagamentoRecusadoException** sempre que a operadora (simulada) detecta um erro.
3. **Dinheiro:** As operações em dinheiro utilizam um saldo interno. Para evitar que o sistema finalize uma corrida sem que o passageiro tenha fundos suficientes, o método processar() lança **SaldoInsuficienteException**.
4. **Pix:** O pagamento depende da chave informada. Se a chave for vazia, inválida ou muito curta, o pagamento é imediatamente interrompido com **PagamentoRecusadoException**, pois a transação não pode prosseguir.

Após a validação, as exceções são tratadas no serviço central:

- **PagamentoRecusadoException** é lançada pelas subclasses de pagamento e tratada em **CorridaServico**, que então decide como a corrida deve continuar (por exemplo, colocando-a em *pendente de pagamento*).
- **SaldoInsuficienteException** é lançada apenas no pagamento em dinheiro e tratada também em **CorridaServico**, enquanto o **LadoPassageiro** gerencia o resultado.

Dessa forma, o método processar() interrompe imediatamente quando há erro, garante que o pagamento não avance sem validações e notifica o restante do sistema da maneira correta — usando exceções.

5. **Exceção de Motorista Indisponível**

Essa exceção é lançada dentro do serviço central de operações, na classe **CorridaServico**, especificamente no método **encontrarMotoristaDisponivel()**. Ela ocorre quando o sistema tenta criar uma corrida, mas **não existe nenhum motorista online e livre para aceitar a solicitação**.

Como essa falha precisa ser comunicada diretamente ao usuário, ela é **tratada na interface do Menus**, dentro da classe **LadoPassageiro**. A interface apresenta a mensagem de erro e impede que o passageiro prossiga, simulando o comportamento real de aplicativos de mobilidade.

6. **Exceções de Transição de Estado**

Essa exceção também é lançada em **CorridaServico**, sempre que uma operação tenta alterar o estado da corrida para algo que **não é permitido pelo ciclo de vida definido** (por exemplo: tentar iniciar uma corrida que ainda não foi aceita, ou finalizar uma corrida que não está em andamento).

Como tanto motorista quanto passageiro podem tentar executar ações inválidas, essa exceção é **tratada na interface Menus**, **LadoMotorista** e **LadoPassageiro**. Assim, qualquer tentativa de ação fora da ordem correta resulta em uma mensagem clara ao usuário.

3. Testes e Cenários

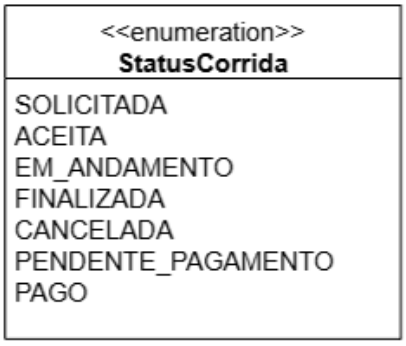
3.1. Ciclo de Vida da Corrida

O **ciclo de vida da corrida** representa todas as etapas pelas quais uma corrida passa *desde o momento em que é solicitada até sua conclusão*. No projeto, esse fluxo foi modelado com o enum **StatusCorrida** e controlado por métodos específicos dentro da classe Corrida. Cada etapa só pode acontecer se o objeto estiver em um estado válido, garantindo consistência.

No UML e no código:

```
package entidade;


public enum StatusCorrida {
    SOLICITADA,
    ACEITA,
    FINALIZADA,
    EM_ANDAMENTO,
    CANCELADA,
    PENDENTE_PAGAMENTO,
    PAGA;
}
```



O ciclo funciona da seguinte forma:

1. **SOLICITADA**
   > A corrida é criada pelo passageiro usando o método **solicitarCorrida()**;
   > Nesse estado, ainda não existe motorista atribuído.
   > É a única fase em que o passageiro pode cancelar a corrida (Se for feito, é pelo método **cancelarCorrida();**).
2. **ACEITA**
   > Um motorista disponível aceita atender a solicitação por meio do método **aceitarCorrida();**
   > A partir daqui, o motorista fica vinculado à corrida.
   > A corrida ainda não começou — ela está aguardando o motorista chegar ao local.
3. **EM_ANDAMENTO**
   > O motorista inicia a viagem usando o método **iniciarViagem();**
   > Esse método só pode ser executado se o estado anterior for ACEITA; caso contrário, o sistema lança **TransicaoDeEstadoInvalidaException**.
4. **FINALIZADA**
   > A corrida é concluída com o método **finalizarCorrida().**
   > Nessa etapa, o sistema calcula o valor final e tenta processar o pagamento.
   - Se o pagamento é bem-sucedido → o status é definido como **FINALIZADA** e **PAGA**.
   - Se o pagamento falha → a corrida passa para **PENDENTE_PAGAMENTO**.

5. **PENDENTE_PAGAMENTO**
   > Esse estado indica que a corrida terminou, mas o pagamento não foi concluído.
   > Aqui, o sistema lança uma das exceções:
   - **SaldoInsuficienteException** — não há saldo suficiente para pagar a corrida.
   - **PagamentoRecusadoException** — o provedor recusou o pagamento (cartão inválido, CVV incorreto, Pix inválido etc.).
6. **CANCELADA**
   > A corrida pode ser cancelada **apenas enquanto estiver em SOLICITADA**.
   > Tentativas de cancelar em qualquer outro estado geram **TransicaoDeEstadoInvalidaException**, garantindo conformidade com a realidade.
7. **ERRO NO CADASTRO:** Quando no momento de cadastro a pessoa: coloca um email sem “@” e “.” ou repete um email já cadastrado ou CPF sem ter 11 números. O sistema lança CadratroInvalidoExcepiton e impede inconformidades com a realidade.

Conclusão

O desenvolvimento deste projeto permitiu aplicar de forma prática todos os principais conceitos estudados na disciplina de Orientação a Objetos. A partir da construção de um sistema completo de mobilidade urbana, foi possível utilizar herança, polimorfismo, abstração, encapsulamento, modularização e tratamento de exceções em um contexto realista e coerente.

O UML final resume toda essa estrutura, evidenciando como as classes se relacionam e como os princípios da orientação a objetos foram aplicados de forma integrada e consistente.

>> Diagrama UML do Projeto

UMLPOO.drawio (1).pdf

LINK:
https://drive.google.com/file/d/134wRTsXONXB5wKPfXcB0s0dRZJ0w1I9J/view?usp=drive_link