Universidade de Brasília
Disciplina: Orientação a objetos
Segundo Semestre de 2025

Matheus Costa Cordeiro de Oliveira (251013698)

Murilo Paulino (251013787)

Matheus Ferraz (251037617)

Rafael Lima Sant'Ana (251035659)

Documentação Técnica: Sistema de Ride-Sharing

1. Visão Geral da Arquitetura

O sistema foi desenvolvido em **Java** seguindo estritamente o paradigma de **Orientação a Objetos (POO)**. A arquitetura adota uma abordagem em camadas, promovendo a separação de responsabilidades entre a modelagem de domínio (Entidades), a lógica de negócios (Serviços) e a interface de consumo (Aplicação/Main). O tratamento de exceções foi customizado para refletir regras de negócio específicas.

2. Detalhamento das Camadas e Componentes

2.1. Camada de Domínio (Pacote `entidades`)

Esta camada encapsula as regras de negócio fundamentais e o estado da aplicação.

- **Hierarquia de Usuários (`Usuario`, `Motorista`, `Passageiro`):**
  - A classe `Usuario` atua como superclasse, centralizando atributos comuns (identificação e autenticação) e o histórico de avaliações.
  - `Passageiro` estende `Usuario`, especializando-se com a gestão de métodos de pagamento e controle de pendências financeiras.
  - `Motorista` estende `Usuario`, agregando objetos de valor como `CNH` e `Veiculo`, além de gerenciar seu próprio estado operacional (`StatusMotorista`).
- **Núcleo Operacional (`Corrida`):**
  - A classe `Corrida` funciona como a entidade central de agregação. Ela orquestra a interação entre um `Passageiro`, um `Motorista`, uma `CategoriaServico` e um `MetodoDePagamento`.
  - Responsável por manter o estado transacional do serviço através do enumerador `StatusCorrida`.
- **Sistema de Precificação (`CategoriaServico`, `Comum`, `Luxo`):**
  - Utiliza uma classe abstrata `CategoriaServico` para definir o contrato de precificação.

    - As subclasses (Comum, Luxo) implementam a lógica de cálculo específica (polimorfismo), alterando a tarifa base e multiplicadores sem modificar o código cliente.
- **Sistema Financeiro (MetodoDePagamento, Pix, CartaoCredito, Dinheiro):**
    - Define uma interface MetodoDePagamento para padronizar o processamento de transações.
    - As implementações concretas encapsulam regras específicas, como validação de saldo (Dinheiro) ou simulação de latência e falhas de operadora (Pix, CartaoCredito).
- **Objetos de Valor e Auxiliares:**
    - Avaliacao: Registra o feedback entre as partes, associando avaliador, avaliado e nota.
    - CNH e Veiculo: Representam documentos e ativos imutáveis no contexto de uma corrida, garantindo a integridade dos dados do motorista.

2.2. Camada de Serviços (Pacote servicos)

Responsável pela orquestração das entidades e execução dos casos de uso.

- **GerenciadorUsuario:**
    - Atua como um repositório em memória (*In-Memory Repository*).
    - Gerencia o ciclo de vida dos objetos Passageiro e Motorista, além de prover métodos de busca para alocação de motoristas disponíveis.
- **GerenciadorDeCorridas:**
    - Implementa o padrão *Facade* para as operações de corrida.
    - Centraliza a lógica complexa: solicitação de corrida, cálculo de rota/distância (simulado), transição de estados da corrida e processamento de pagamentos.
    - Garante a consistência das regras de negócio lançando exceções como NenhumMotoristaDisponivelException ou EstadoInvalidoDaCorridaException.

2.3. Camada de Apresentação e Controle (Main)

- **Main:**
    - Ponto de entrada da aplicação.
    - Responsável pela injeção de dependências (instanciação dos gerenciadores).
    - Executa o fluxo de simulação via console, controlando a interação com o usuário e a thread de execução para simular o tempo de deslocamento.

---

3. Aplicação dos Pilares da Orientação a Objetos

A robustez do sistema é garantida pela aplicação correta dos quatro pilares fundamentais da POO:

3.1. Encapsulamento

O estado interno dos objetos é protegido contra acesso indevido. Todos os atributos são declarados como `private` ou `protected`, sendo expostos apenas via métodos acessores (*getters/setters*) ou, preferencialmente, através de métodos de negócio (ex: o saldo da classe `Dinheiro` só é alterado via método `processarPagamento`, garantindo consistência).

3.2. Herança

Utilizada para promover a reutilização de código e estabelecer hierarquias lógicas. A generalização da classe `Usuario` evita a duplicação de atributos como `nome`, `cpf` e `senha` nas classes `Motorista` e `Passageiro`. Da mesma forma, `NegocioException` serve como base para todas as exceções da aplicação, facilitando o tratamento de erros (`try/catch`).

3.3. Polimorfismo

Implementado amplamente para garantir extensibilidade e desacoplamento:

- **Interfaces:** A classe `GerenciadorDeCorridas` interage com a interface `MetodoDePagamento`, desconhecendo a implementação concreta (Pix, Dinheiro, etc.). Isso permite a adição de novos meios de pagamento sem refatoração do núcleo do sistema.
- **Sobrescrita (Override):** As classes `Comum` e `Luxo` sobrescrevem o método `calcularPreco` da classe pai, permitindo que a `Corrida` calcule o valor final independentemente da categoria instanciada.

3.4. Abstração

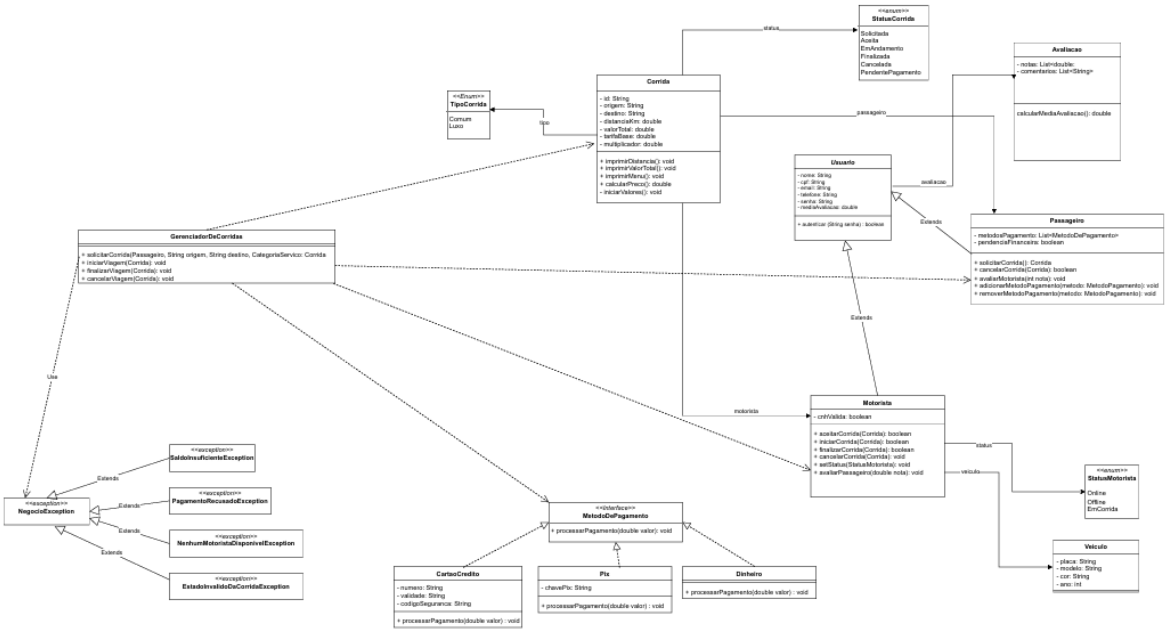
O sistema oculta a complexidade de implementação através de classes abstratas e interfaces.

- **Exemplo:** A classe `CategoriaServico` define *o que* deve ser feito (calcular preço), mas deixa para as subclasses definirem *como* fazer. O consumidor da classe não precisa saber a fórmula matemática exata, apenas que o método retornará o valor correto.

---

4. Fluxo de Interação e Dados

1. **Inicialização:** O `Main` instancia o `GerenciadorUsuario` e popula o sistema com motoristas e um passageiro.
2. **Solicitação:** O `Passageiro` solicita uma corrida via `GerenciadorDeCorridas`. O sistema valida pendências financeiras e busca um `Motorista` com status `Online`.
3. **Execução:** Ao encontrar um motorista, a entidade `Corrida` é criada. O status do motorista muda para `EmCorrida`.
4. **Conclusão:** Após a simulação do trajeto, o método de pagamento selecionado é processado polimorficamente. Se aprovado, a corrida é finalizada, o motorista retorna ao status `Online` e uma `Avaliacao` é gerada.


«enum»
StatusCorrida
Solicitada
Aceita
EmAndamento
Finalizada
Cancelada
PendentePagamento
status
Avaliacao
- notas: List<double>
- comentarios: List<String>
calcularMediaAvaliacao(): double
«Enum»
TipoCorrida
Comum
Luxo
tipo
Corrida
- id: String
- origem: String
- destino: String
- distanciaKm: double
- valorTotal: double
- tarifaBase: double
- multiplicador: double
+ imprimirDistancia(): void
+ imprimirValorTotal(): void
+ imprimirMenu(): void
+ calcularPreco(): double
- iniciarValores(): void
passageiro
avaliacao
Usuario
- nome: String
- cpf: String
- email: String
- telefone: String
- senha: String
- mediaAvaliacao: double
+ autenticar (String senha): boolean
Estende
Passageiro
- metodosPagamento: List<MetodoDePagamento>
- pendenciaFinanceira: boolean
+ solicitarCorrida(): Corrida
+ cancelarCorrida(Corrida): boolean
+ avaliarMotorista(int nota): void
+ adicionarMetodoPagamento(metodo: MetodoPagamento): void
+ removerMetodoPagamento(metodo: MetodoPagamento): void
GerenciadorDeCorridas
+ solicitarCorrida(Passageiro, String origem, String destino, CategoriaServico: Corrida
+ iniciarViagem(Corrida): void
+ finalizarViagem(Corrida): void
+ cancelarViagem(Corrida): void
Usa
Estende
motorista
Motorista
- cnhValida: boolean
+ aceitaCorrida(Corrida): boolean
+ iniciarCorrida(Corrida): boolean
+ finalizarCorrida(Corrida): boolean
+ cancelarCorrida(Corrida): void
+ setStatus(StatusMotorista): void
+ avaliarPassageiro(double nota): void
status
veiculo
«enum»
StatusMotorista
Online
Offline
EmCorrida
Veiculo
- placa: String
- modelo: String
- cor: String
- ano: int
«exception»
SaldoInsuficienteException
«exception»
PagamentoRecusadoException
«exception»
NenhumMotoristaDisponivelException
«exception»
EstadoInvalidoDaCorridaException
«exception»
NegocioException
Estende
Estende
Estende
Estende
«interface»
MetodoDePagamento
+ processarPagamento(double valor): void
CartaoCredito
- numero: String
- validade: String
- codigoSeguranca: String
+ processarPagamento(double valor) : void
Pix
- chavePix: String
+ processarPagamento(double valor) : void
Dinheiro
+ processarPagamento(double valor) : void