Aluno: Caio Andrade Carneiro de Aguiar **Professor:**André Luiz Peron Martins Lanna

Matrícula:251019388

1. Resumo do Sistema

O projeto consiste em um protótipo funcional de um sistema de mobilidade urbana, desenvolvido em Java, que simula o ciclo completo de solicitação e atendimento de corridas. O sistema foi arquitetado visando a modularidade e a segurança de dados, dividindo as responsabilidades em cinco pacotes principais: `entidades`, `servicos`, `app` , `excecoes` e `enums`.

O fluxo principal permite que usuários (Passageiros e Motoristas) realizem cadastro e login. Passageiros podem solicitar corridas escolhendo entre categorias de serviço (Comum ou Luxo), visualizar estimativas de preço e efetuar pagamentos (Pix, Cartão ou Dinheiro). Motoristas gerenciam sua disponibilidade (Online/Offline), aceitam corridas compatíveis com a categoria do seu veículo e conduzem o ciclo de vida da viagem.

2. Explicação dos Conceitos de OO Aplicados

2.1. Herança

- **Classe Base `Usuario`:** Centraliza atributos comuns (`nome`, `cpf`, `email`, `telefone`, `senha`) e comportamentos compartilhados, como validação de credenciais (`validarSenha`) e gestão de histórico de avaliações (`adicionarAvaliacao`).
- **Subclasses `Motorista` e `Passageiro`:** Estendem `Usuario`, herdando suas características e adicionando especificidades. O `Motorista` possui CNH, Veículo e Status, enquanto o `Passageiro` possui métodos de pagamento e controle de bloqueio financeiro.

2.2. Polimorfismo

O sistema explora três tipos de polimorfismo:

1. **Interfaces:** Através da interface `MetodoPagamento`, o sistema permite que diferentes classes (`PagamentoPix`, `PagamentoCartao`, `PagamentoDinheiro`) implementem o método `processarPagamento()` de formas distintas. O sistema chama o método na interface sem precisar conhecer a implementação concreta, permitindo a troca dinâmica da forma de pagamento pelo passageiro.

2. **Generics:** Foi implementada a classe `Repositorio<T>`, que atua como um banco de dados em memória genérico. Ela é utilizada para armazenar `Motorista`, `Passageiro` e `Corrida`, garantindo a tipagem forte e evitando a duplicação de código de armazenamento para cada entidade.
3. **Sobrescrita (Override):** Os métodos `toString()` foram sobrescritos em todas as entidades para fornecer representações textuais adequadas de cada objeto nos menus do console.

2.3. Associações entre Objetos

As relações entre as classes foram modeladas para refletir a realidade do negócio:

- **Composição (`Motorista` possui `Veiculo`):** Foi modelado de forma que o Veículo é parte essencial do cadastro do Motorista no contexto do aplicativo. O motorista detém a referência direta ao seu veículo de trabalho.
- **Agregação (`Passageiro` usa `MetodoPagamento`):** O passageiro possui uma referência para um metodo de pagamento, mas esse metodo pode ser substituído (agregada) a qualquer momento, tendo um ciclo de vida independente do passageiro.
- **Associação Simples (`Corrida`):** A entidade `Corrida` atua como o elo central, associando-se a um `Passageiro` (solicitante) e, posteriormente, a um `Motorista` (condutor), além de se associar a Enums de estado (`StatusCorrida`).

3. Justificativa para as Exceções Customizadas

SaldoInsuficienteException: Garante que o fluxo de pagamento em dinheiro/saldo seja interrompido de forma controlada se o usuário não tiver fundos, obrigando o sistema a tratar a falha (bloqueando o usuário).

PagamentoRecusadoException: Modela a falha de comunicação com operadoras de cartão. É essencial para distinguir um erro técnico de pagamento de um erro de lógica interna.

NenhumMotoristaDisponivelException: Permite que o sistema informe ao usuário que a solicitação falhou por falta de oferta, sem confundir com erros de sistema, permitindo uma nova tentativa posterior.

PassageiroPendenteException: Implementa a regra de negócio de bloqueio. Impede proativamente que um usuário inadimplente gere novos custos ao sistema.

UsuarioJaCadastradoException: Garante a integridade dos dados, impedindo duplicidade de e-mails no "banco de dados" em memória.

DistanciaInvalidaException: Validação de entrada que impede a criação de corridas com distâncias negativas ou zeradas, protegendo o cálculo financeiro.

NotaInvalidaException: Garante que as avaliações fiquem estritamente dentro da regra de negócio (1 a 5 estrelas), encapsulando essa regra dentro da entidade Usuario.

EstadoInvalidoDaCorridaException: Indica uma falha grave na máquina de estados (ex: tentar finalizar uma corrida que nem começou).

4. Diagrama de Classes