# Projeto Arquitetural: UberPB

## 1. Introdução e Objetivos

### 1.1. Visão Geral do Sistema

O "UberPB" é uma aplicação de simulação de transporte de passageiros e motoristas. O sistema opera inteiramente através de uma interface de linha de comando (terminal), sem uma interface gráfica de usuário. Ele gerencia entidades fundamentais como Passageiro, Motorista, Gerente, Veículo e Corrida.

### 1.2. Requisitos Arquiteturais (Drivers)

A arquitetura foi guiada pelos seguintes objetivos principais:

* **Estrutura:** O código é estruturado seguindo um padrão em camadas (Model-View-Service) para garantir que as responsabilidades (dados, apresentação e controle) sejam bem definidas e que o sistema seja fácil de entender e modificar.
* **Simplicidade de Implantação:** O sistema é auto suficiente, não exigindo a configuração de um banco de dados externo. A persistência de dados é feita em arquivos locais.
* **Clareza no Tratamento de Erros:** O sistema utiliza exceções customizadas para lidar com erros de negócio de forma explícita e controlada.
* **Modularidade:** O uso de interfaces como Repository e uma implementação genérica (JsonRepository) permite que a lógica de acesso a dados seja consistente, abstraindo a leitura e escrita dos arquivos JSON.

## 2. Visão Lógica: Arquitetura em Camadas(Model-View-Service)

O sistema é estruturado em torno de um padrão de camadas que separa a representação da informação (Modelo) da interação do usuário com ela (Visão).

### 2.1. Detalhamento dos Componentes

* **View (Visão):**
  + **Responsabilidade:** Apresentar os dados ao usuário e capturar suas interações. A View não contém lógica de negócio.
  + **Implementação:** A View do sistema é a interface de Linha de Comando (Terminal). A classe Main.java serve como ponto de entrada, que inicializa a View (chamando View.iniciar()). A classe MenuPrincipalView é a principal responsável por "desenhar" o menu inicial. A responsabilidade pela interface é distribuída por todo o pacote org.example.view, com classes dedicadas para cada tela (ex: LoginView, SolicitarCorridaView, MenuPassageiroView).
* **Controle de Fluxo (na Camada View):**
  + **Responsabilidade:** Atuar como intermediário entre a interação do usuário (View) e a lógica de negócio (Model).
  + **Implementação:** O controle de fluxo da aplicação é gerenciado pelas próprias classes da camada View. Elas recebem a entrada do usuário (via Scanner em ViewUtils), instanciam e invocam os métodos apropriados na camada de Service (ex: SolicitarCorridaView chama CorridaService.solicitarCorrida(...)) e navegam para a próxima tela com base na resposta (ex: LoginView chama MenuPassageiroView.exibir(passageiroLogado)).
* **Model (Modelo):**
  + **Responsabilidade:** O coração da aplicação, representando os dados, as regras de negócio e a lógica de persistência.
  + **Implementação:** O Model é uma camada abrangente composta por três subcomponentes:
    - **Entidades ( /model/entity ):** Representam a estrutura dos dados. São as classes como Usuario (classe base), Motorista, Passageiro, Corrida e Veiculo. Enums como StatusCorrida e CategoriaVeiculo são usados para representar estados e tipos fixos.
    - **Serviços ( /model/service ):** Contêm a lógica de negócio (o "cérebro"). Classes como CorridaService, PassageiroService e MotoristaService executam as regras de negócio, validações e cálculos.
    - **Repositórios ( /model/repository ):** Lidam com a persistência dos dados, abstraindo a fonte de dados. A classe genérica JsonRepository<T> é responsável por ler e escrever em arquivos JSON usando a biblioteca Gson. Implementações específicas como MotoristaRepository e CorridaRepository a utilizam para fornecer métodos de acesso aos dados para a camada de serviço.

## 3. Visão de Dados

### 3.1. Mecanismo de Persistência

O sistema utiliza arquivos de texto no formato JSON como seu banco de dados, com cada tipo de entidade principal armazenada em seu próprio arquivo. A biblioteca **Google Gson** é utilizada para a serialização e desserialização dos objetos Java.

## 4. Visão de Implantação (Deployment)

* **Ambiente de Execução:** Java Virtual Machine (JVM).
* **Empacotamento:** A aplicação é empacotada como um arquivo JAR executável.
* **Dependências Externas:**
  + **Runtime (JAR Principal):** A única dependência de *runtime* da aplicação principal é a biblioteca com.google.code.gson, utilizada para a persistência de dados em JSON.
  + **Testes:** A suíte de testes de unidade utiliza o framework org.junit.jupiter (JUnit 5).
  + **Visualização (Simulação):** A funcionalidade de simulação de viagem (SimuladorViagem.java) gera um arquivo de dados que é consumido por um Mapa.html.

## 5. Funcionalidades implementadas

### 5.1. Cadastro e autenticação

#### RF01: Permitir cadastro de passageiros e motoristas.

**Implementação:** O cadastro é feito por meio de fluxos dedicados na camada de View (CadastroPassageiroView.java, CadastroMotoristaView.java). As regras de negócio (como validação de e-mail e CPF duplicados) são tratadas pelos respectivos serviços (PassageiroService.java, MotoristaService.java).

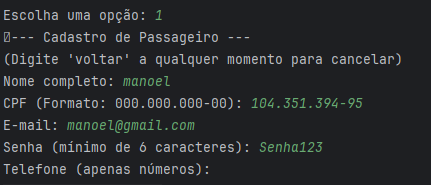


Figura 1

#### RF02: Validar características do veículo para definir a categoria.

**Implementação:** Durante o cadastro do motorista, o sistema exige um segundo passo (CadastroVeiculoView.java), onde os dados do veículo são coletados. O VeiculoService.java contém a lógica determinarCategoria que, com base no ano, assentos e outros atributos, define a categoria do veículo (UberX, Comfort, etc.).

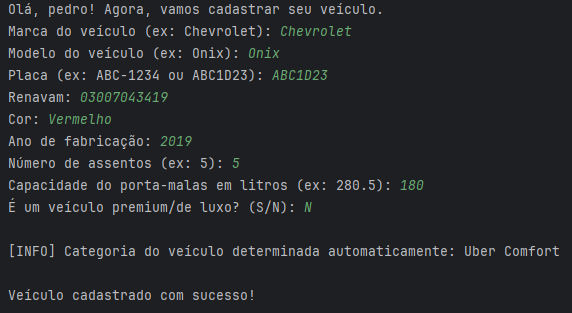


Figura 2

#### RF03: Permitir login com e-mail.

**Implementação:** A LoginView.java captura e-mail e senha e os repassa aos serviços PassageiroService e MotoristaService para autenticação.

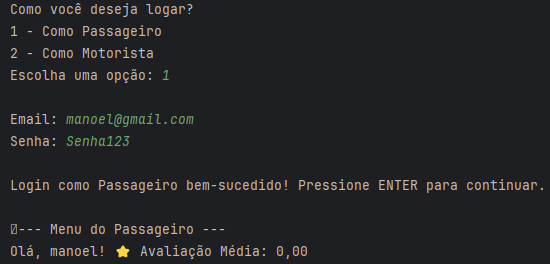


Figura 3

### 5.2. Solicitação de corrida

#### RF04, RF06: Inserir origem, destino e selecionar categoria.

**Implementação:** A SolicitarCorridaView.java gerencia este fluxo completo, permitindo ao passageiro selecionar uma origem e destino da lista (ou cadastrar um novo) e, em seguida, selecionar a categoria de veículo desejada..

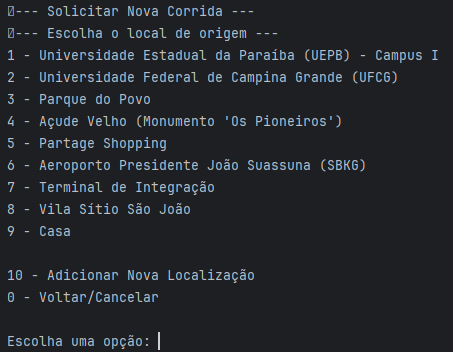


Figura 4

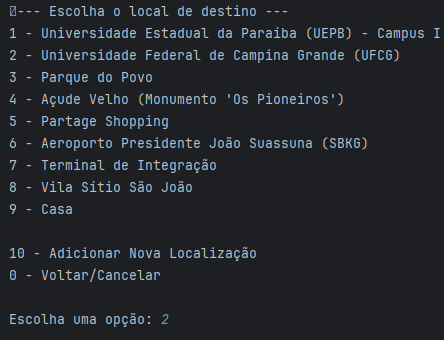


Figura 5

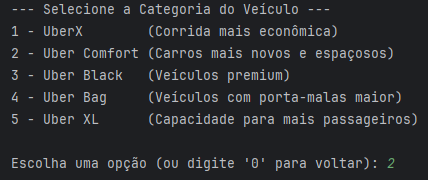


Figura 6

#### RF05, RF07: Calcular estimativa de preço e notificar motoristas.

**Implementação:** O CorridaService.java centraliza esta lógica. O método calcularPrecoEstimado usa a fórmula (de Localizacao.java) e fatores de categoria/demanda. Após a confirmação, o método solicitarCorrida inicia o processo de busca, chamando encontrarProximoMotoristaDisponivel para notificar os motoristas elegíveis.



Figura 7

### 5.3. Aceite Corrida

#### RF08: Motorista poder aceitar ou recusar a corrida.

**Implementação:** O motorista, através da CorridaView.java, visualiza as corridas notificadas. Ele pode aceitar (chamando CorridaService.aceitarCorrida) ou recusar (chamando MotoristaService.negarCorrida).

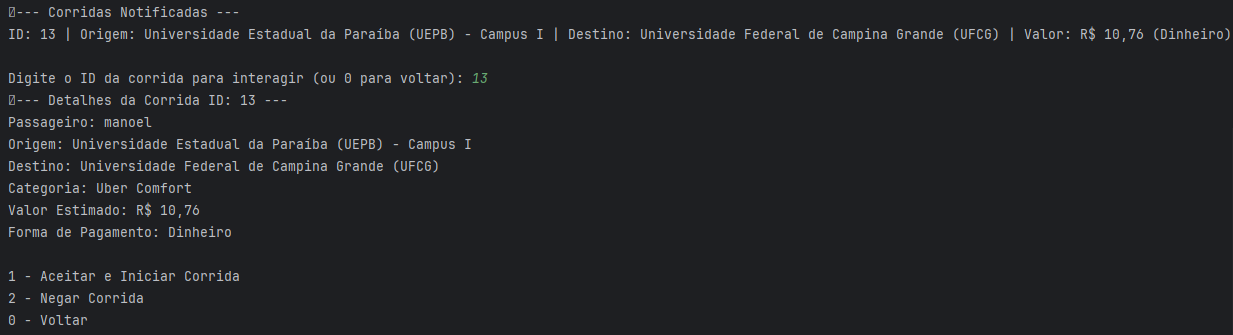


Figura 8

#### RF09: Atribuir corrida ao motorista mais próximo/melhor avaliado.

**Implementação:** A lógica de atribuição está em CorridaService.encontrarProximoMotoristaDisponivel. O método filtra motoristas (disponíveis, na categoria correta, que não rejeitaram) e os ordena usando um comparador que prioriza avaliação (para categorias premium) ou distância (para UberX).

### 5.4. Acompanhamento da Corrida

#### RF10, RF11, RF12: Acompanhar localização e rota.

**Implementação:** O passageiro pode monitorar o status na AcompanharCorridasView.java. Quando a corrida é aceita, o SimuladorViagem.java é acionado, escrevendo coordenadas em dados\_viagem.js. O Mapa.html (aberto pela MapaView.java) lê esse arquivo e exibe a simulação da rota (em linha reta) no navegador usando Leaflet. Estas funcionalidades foram concluídas nas Sprints 2 e 3.

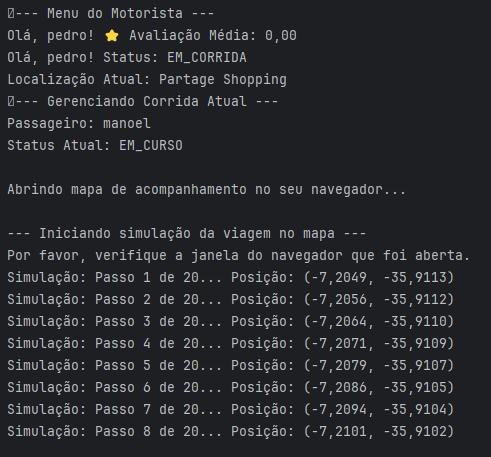


Figura 9

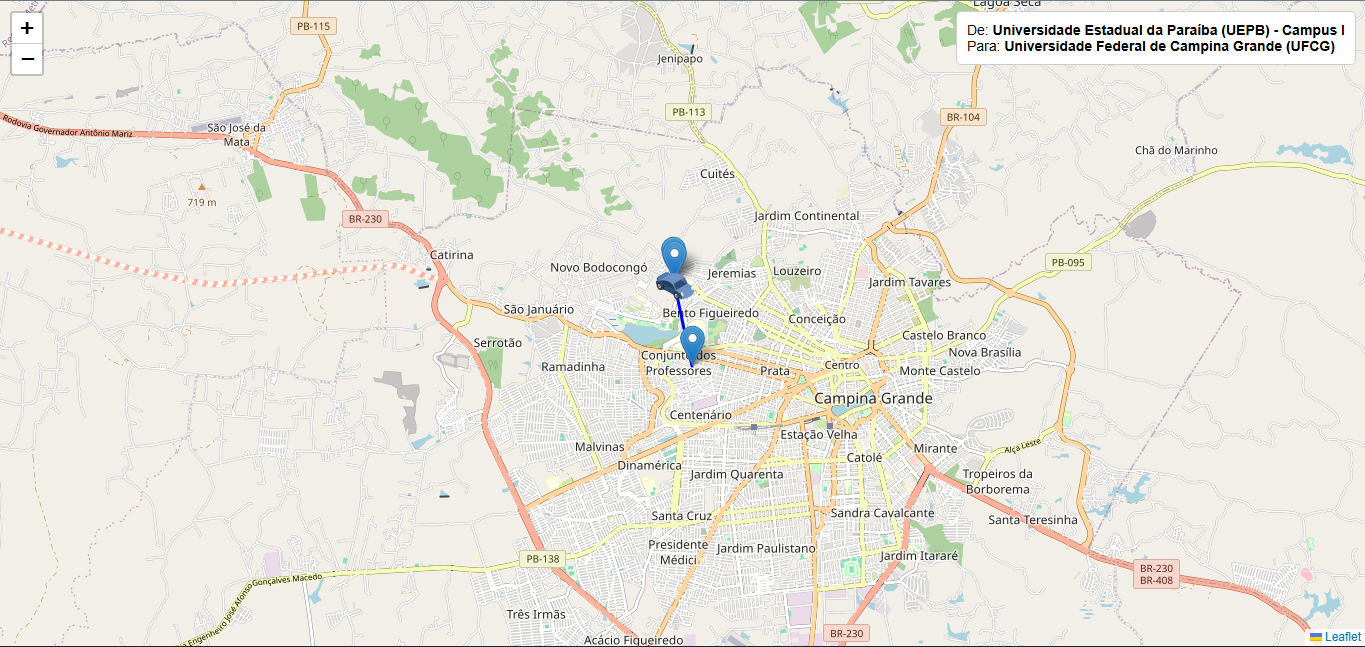


Figura 10

### 5.5. Pagamentos

#### RF13: Permitir pagamento via cartão, PIX, PayPal ou dinheiro.

**Implementação**: A PagamentoView.java gerencia a seleção do método de pagamento. A enum FormaPagamento.java confirma que todas as quatro formas (Cartão, PIX, PayPal, Dinheiro) estão implementadas.

#### RF14: Calcular automaticamente o valor da corrida.

**Implementação:** O CorridaService.calcularPrecoEstimado é usado para definir o valor da corrida no momento da solicitação.

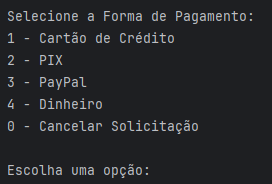


Figura 11

#### RF15: Emitir recibo eletrônico ao final da corrida.

**Implementação:** O recibo é implementado como parte do histórico. O HistoricoCorridasView.java exibe os detalhes completos da viagem (origem, destino, valor, motorista, etc.) após sua conclusão.

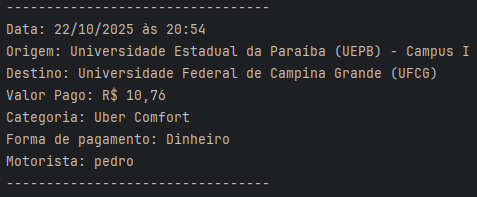


Figura 12

### 5.6. Avaliação

#### RF16: Permitir que passageiros e motoristas se avaliem mutuamente.

**Implementação:** Após a corrida, ambos os usuários são notificados. Os fluxos AvaliacaoView.java e AvaliacaoPendentesView.java gerenciam a coleta da nota (1-5). A nota é salva no objeto Corrida e a média do usuário é atualizada via recalcularMediaAvaliacao nos respectivos serviços.

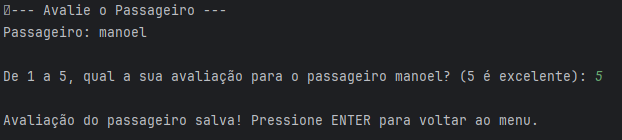


Figura 13

#### RF17: Utilizar a média das avaliações para priorizar motoristas.

**Implementação:** Conforme descrito no RF09, o CorridaService.java já utiliza a mediaAvaliacao do motorista como um dos critérios de ordenação ao buscar o melhor motorista para a corrida.

### 5.7. Histórico

#### RF18: Manter histórico de corridas filtrável por categoria de carro.

**Implementação:** Corridas finalizadas são movidas para a lista historicoCorridas na entidade Passageiro. O HistoricoCorridasView.java exibe essa lista e implementa uma lógica de filtro que permite ao usuário visualizar apenas corridas de uma categoria específica (ex: "UberX").

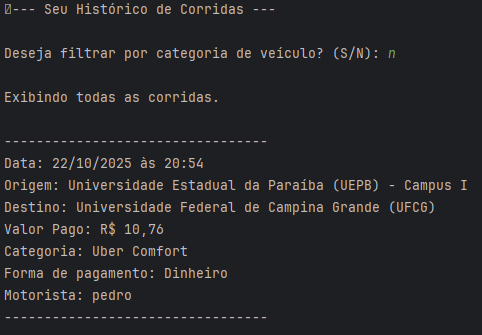


Figura 14

## 6. Relatório de teste

### Sprint 1

Durante a Sprint 1, foram realizados os testes iniciais do sistema, com foco nas funcionalidades de **cadastro e autenticação de usuários**, além da **solicitação de corrida**.

No primeiro conjunto de testes, avaliou-se o **cadastro de novos usuários**, verificando se o sistema aceitava apenas dados válidos e se armazenava corretamente as informações. Foram testados diferentes cenários, incluindo campos em branco, senhas curtas e e-mails duplicados. No primeiro teste, o sistema apresentou uma falha, permitindo o cadastro com senha de apenas três caracteres. O erro foi corrigido com a adição de uma validação mínima de comprimento de senha. Após a correção, os testes foram reexecutados e aprovados.

Em seguida, testou-se o **módulo de autenticação**, confirmando que apenas usuários registrados conseguiam acessar o sistema. Inicialmente, ocorreu um problema no processo de comparação da senha criptografada, o que impedia o login de novos usuários. A falha foi rastreada até a função de hash e devidamente ajustada. Após a correção, todos os testes de login e logout foram bem-sucedidos.

Por fim, foram realizados testes na **funcionalidade de solicitação de corrida**, simulando um passageiro solicitando uma viagem até determinado destino. O teste verificou se a corrida era criada com status inicial **SOLICITADA** e se os dados de origem, destino e categoria do veículo eram armazenados corretamente. Na primeira execução, o status da corrida não estava sendo atualizado no repositório. O problema foi corrigido e os testes seguintes confirmaram o funcionamento esperado.

Os resultados da Sprint 1 garantiram que o sistema possuía um fluxo básico de cadastro, login e solicitação de corrida funcionando corretamente, servindo de base sólida para as próximas etapas de desenvolvimento.

### Sprint 2

Na Sprint 2, o foco foi validar o fluxo operacional da corrida, desde o **aceite por um motorista** até o **acompanhamento e finalização da viagem**.

Nos **testes de aceite**, simulou-se o cenário em que um motorista disponível aceita uma corrida. O sistema atualizou corretamente o status para **ACEITA** e vinculou o ID do motorista à corrida. Em seguida, foi testado o caso de **dois motoristas tentando aceitar a mesma corrida**. Inicialmente, o segundo motorista também conseguia aceitar, o que configurava um erro de concorrência. A falha foi corrigida com a atualização imediata do status no repositório após o primeiro aceite. Após o ajuste, o segundo motorista passou a receber uma resposta negativa, e o teste foi aprovado.

Já nos **testes de acompanhamento da corrida**, verificou-se o comportamento do sistema durante o início, andamento e término da viagem. Ao iniciar a corrida, o status mudou corretamente para **EM\_CURSO**, e o horário de início foi registrado. No encerramento, o sistema atualizou o status para **FINALIZADA** e registrou o horário de término. O primeiro teste apresentou uma falha, pois o motorista não voltava ao status **DISPONÍVEL** após o término. O método finalizarCorrida() foi revisado e corrigido, garantindo o retorno automático à disponibilidade. Após a correção, o teste foi reexecutado e aprovado.

Esses testes também confirmaram que as informações de histórico do passageiro e do motorista eram atualizadas corretamente. Todos os casos foram finalizados com sucesso, garantindo um fluxo completo e estável entre solicitação, aceite e finalização de corrida.

### Sprint 3

Na Sprint 3, foram implementadas e testadas funcionalidades relacionadas à **etapa financeira da corrida**, abrangendo o **cálculo do valor**, a **emissão do recibo eletrônico** e o **processo de pagamento**.

O primeiro teste verificou o **cálculo do valor da corrida**, considerando distância, tempo estimado e categoria do veículo. Foram realizadas simulações com diferentes distâncias e categorias (UberX, Comfort e XL). Em um primeiro momento, observou-se uma diferença no cálculo do valor para corridas curtas, devido a um arredondamento incorreto na conversão da distância. Após corrigir o método de cálculo, os resultados passaram a refletir os valores esperados e o teste foi aprovado.

Em seguida, foi testada a **emissão do recibo eletrônico**, gerado automaticamente após a finalização da corrida. O teste confirmou a criação do recibo com todos os dados da viagem — incluindo valores, horários, motorista e passageiro. No primeiro teste, o recibo estava sendo salvo com campos nulos em corridas recentes. A causa foi um atraso na gravação da corrida antes da emissão do recibo. O problema foi solucionado implementando uma verificação de sincronização, garantindo que o recibo só fosse gerado após o salvamento completo da corrida.

Por fim, foram realizados os testes da **funcionalidade de pagamento**, simulando tanto pagamentos bem-sucedidos quanto falhos. Inicialmente, uma corrida paga não estava sendo marcada corretamente como “paga” no repositório. Após ajustar a atualização do campo de status do pagamento, o teste foi repetido com sucesso.

Com isso, a Sprint 3 foi concluída com todas as funcionalidades financeiras testadas e funcionando corretamente, garantindo transparência, precisão e rastreabilidade no processo de pagamento e emissão de recibos.

### Sprint 4

A Sprint 4 concentrou-se nas funcionalidades de **histórico de corridas** e **avaliações mútuas entre passageiros e motoristas**, voltadas à experiência do usuário e à confiabilidade do sistema.

Nos **testes do sistema de históricos**, foi verificado se todas as corridas concluídas eram registradas corretamente para passageiro e motorista. Foram simuladas múltiplas corridas seguidas e, após cada finalização, confirmou-se que os dados eram adicionados aos respectivos históricos. O primeiro teste apresentou uma falha, onde corridas canceladas também estavam sendo exibidas. O problema foi corrigido adicionando um filtro que inclui apenas corridas com status **FINALIZADA**. Após o ajuste, todos os testes passaram, garantindo a precisão das informações exibidas no histórico.

Em seguida, os **testes do sistema de avaliações** verificaram se passageiros e motoristas conseguiam avaliar-se mutuamente ao final da corrida, e se as notas eram corretamente salvas e utilizadas na média geral. Durante o primeiro teste, observou-se que as avaliações dos motoristas não estavam sendo persistidas no banco de dados. O erro foi rastreado até a função de atualização do perfil e corrigido. Após a correção, novas execuções confirmaram que as notas estavam sendo gravadas corretamente e refletidas nas médias individuais.

Com esses resultados, a Sprint 4 foi encerrada com êxito, garantindo o pleno funcionamento do histórico de corridas e do sistema de avaliações, consolidando a base de confiança e transparência entre usuários e motoristas.