UEPB
Universidade Estadual da Paraíba
Departamento de Computação – DC
Engenharia de Software II
Professor: Dra. Sabrina de Figueirêdo Souto

ASCENDINO MARTINS DE AZEVEDO NETO

JOHNATAN RODRIGUES DOS SANTOS

LARYSSA FINIZOLA COSTA DA SILVA

RELATÓRIO - RELEASE 01

CAMPINA GRANDE – PB

24 de setembro de 2025

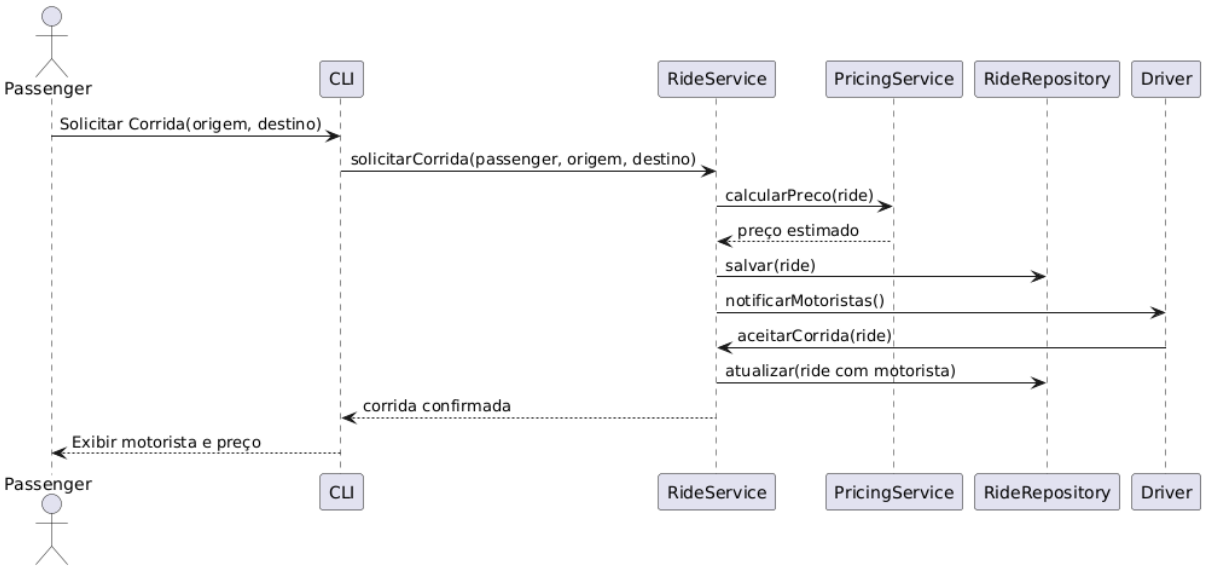
1. INTRODUÇÃO

A Release 1 do sistema UberPB marca a primeira entrega funcional do projeto, resultado das atividades iniciais de modelagem, implementação e testes. Nesta etapa, o foco principal foi a construção da base estrutural do software, com a definição das classes centrais, serviços e repositórios responsáveis por sustentar as operações do sistema. Foram implementados os fluxos essenciais de cadastro e autenticação de usuários, além das primeiras funcionalidades de solicitação, aceite e acompanhamento de corridas, assegurando que passageiros e motoristas pudessem interagir de maneira integrada.

2. DESCRIÇÃO DA ARQUITETURA

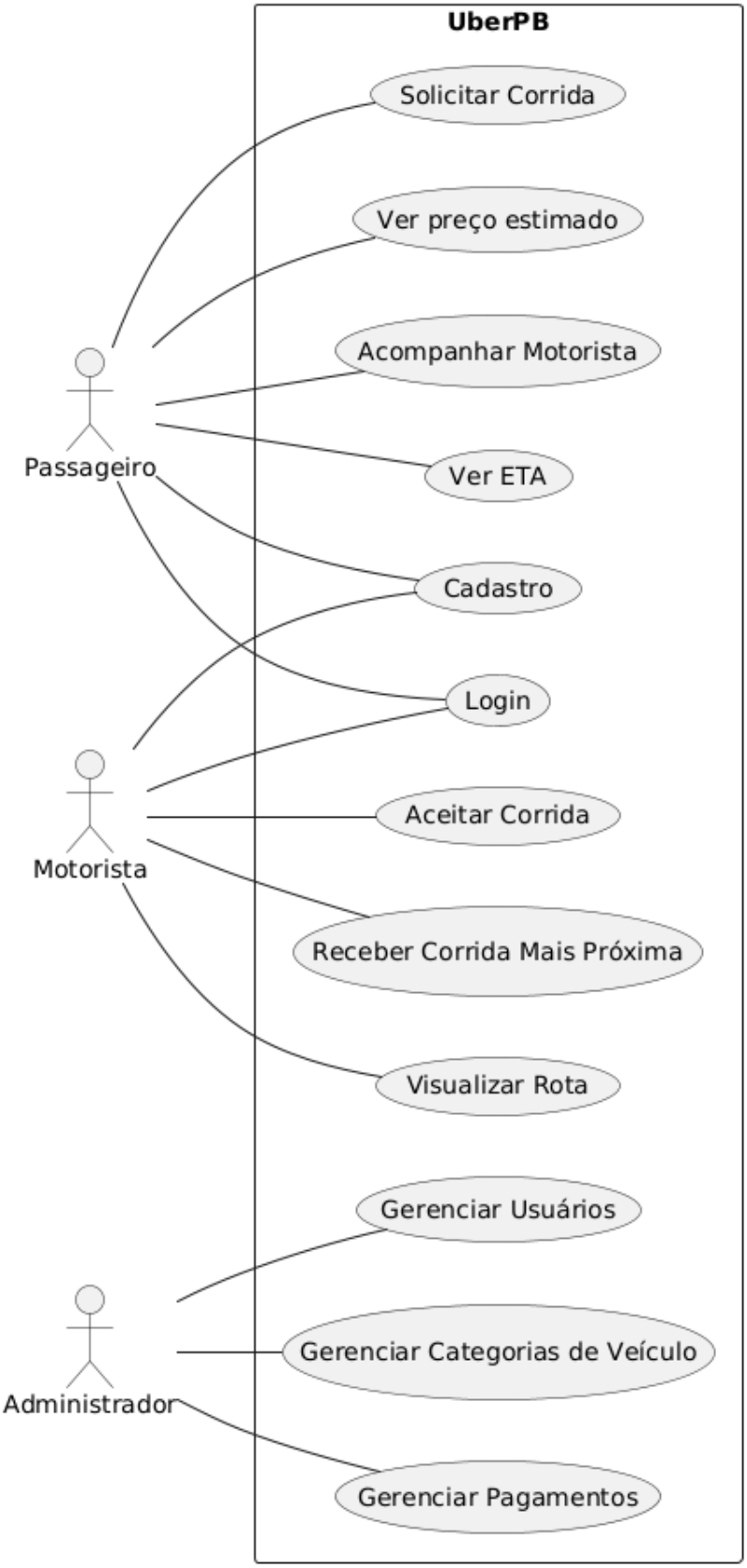
O UberPB adota uma arquitetura em camadas para organizar melhor suas responsabilidades. A camada de modelo (model) define as entidades centrais, como usuários, veículos e corridas. A camada de repositórios (repo) cuida da persistência, isolando a lógica de armazenamento. A camada de serviços (service) implementa as regras de negócio, como cadastro, autenticação, cálculo de preços e gerenciamento de corridas. Já a camada de utilidades (util) fornece funções auxiliares, como validações e cálculo de distâncias, enquanto a camada de interface (cli) serve de ponto de contato com o usuário. Por fim, a camada de testes (test) assegura a qualidade do sistema com JUnit.

Figura 01: Diagrama de Sequência.



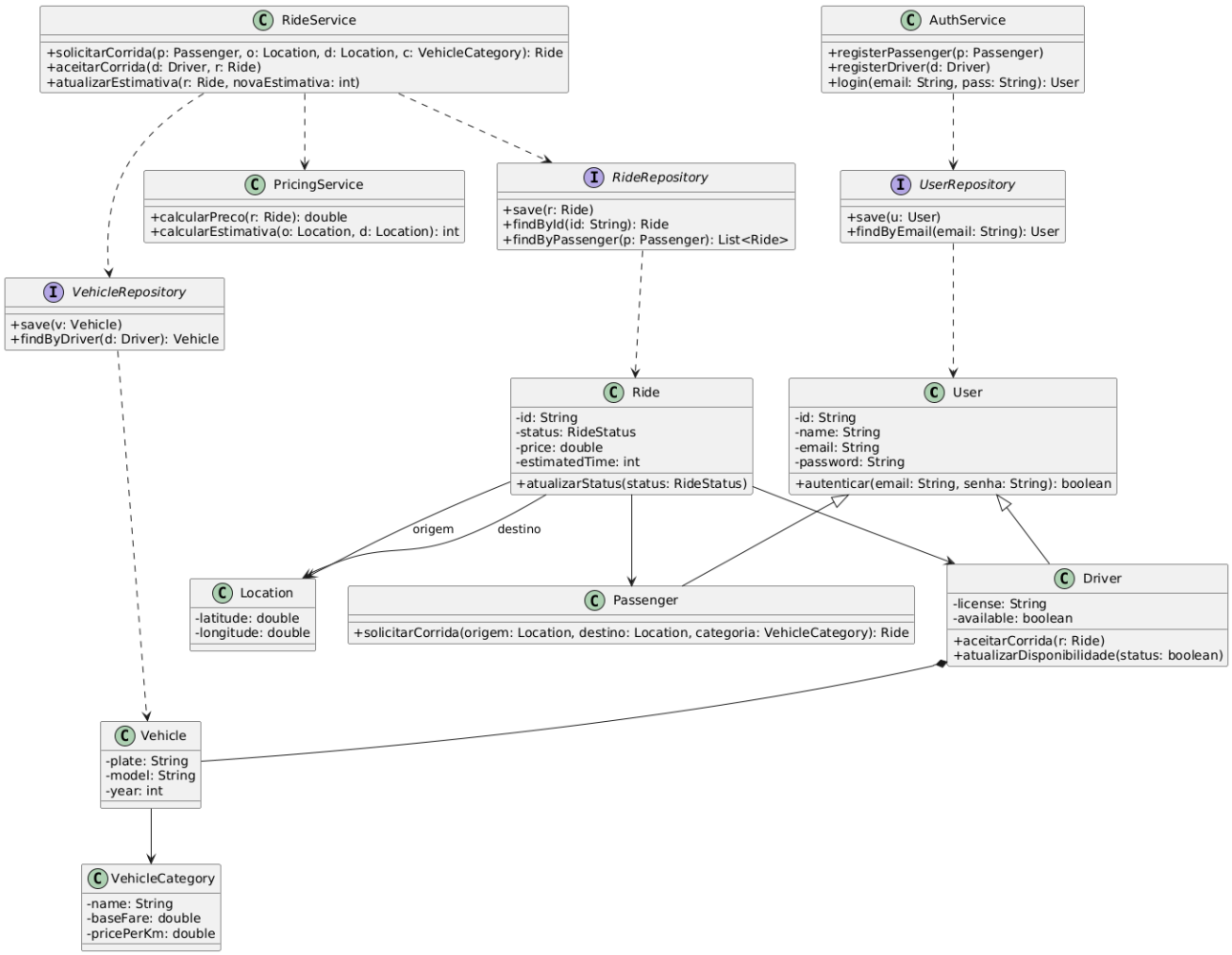
Fonte: Autoria Própria, 2025

Figura 02: Diagrama de Caso de Uso.



Fonte: Autoria própria, 2025.

Figura 03: Diagrama de Classes.



Fonte: Autoria Própria, 2025.

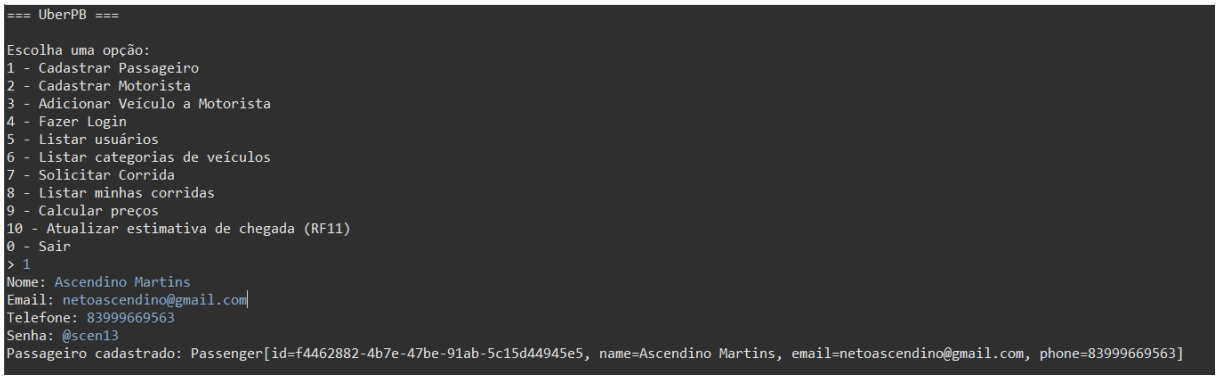
3. FUNCIONALIDADES DESENVOLVIDAS

3.1 Cadastro e Autenticação

O módulo de cadastro e autenticação foi pensado como a porta de entrada para os três tipos de usuários do sistema: passageiros, motoristas e administradores. Para isso, foi implementada a classe User, que abstrai os atributos comuns a qualquer perfil, como nome, e-mail e senha, permitindo que as especializações de usuário fossem criadas de maneira organizada. A partir dessa estrutura, surgem as classes Passenger e Driver, responsáveis por estender o comportamento básico e incluir informações específicas. No caso dos motoristas, além dos dados pessoais, também são registrados o veículo, a categoria habilitada a situação do cadastro e a categoria do veículo.

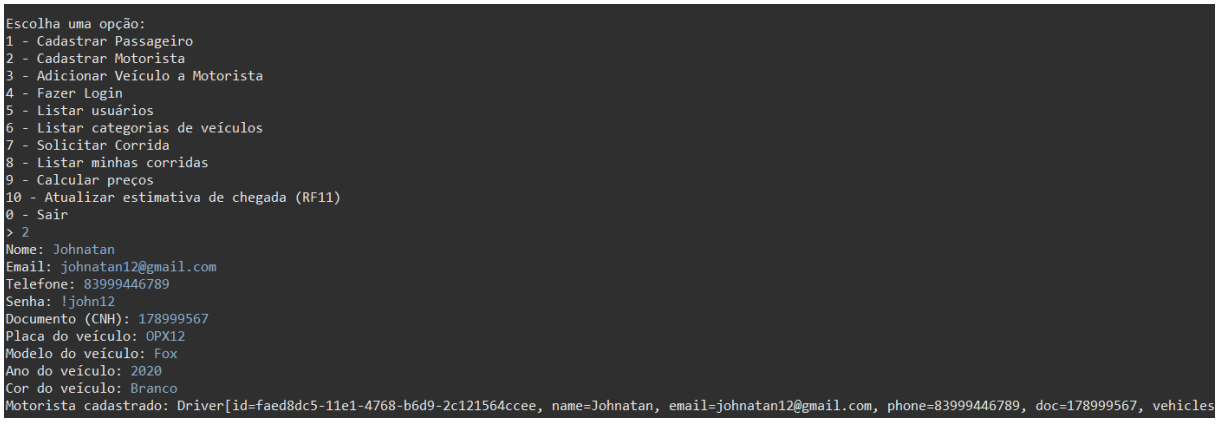
O processo de autenticação é conduzido pelo AuthService, que centraliza operações como login e criação de novos usuários. Esse serviço verifica se as credenciais fornecidas estão corretas e, em caso de sucesso, concede acesso ao sistema. O armazenamento das informações cadastradas é garantido pelo UserRepository, que utiliza persistência em arquivos locais, assegurando que as informações não sejam perdidas entre execuções do programa. Outro aspecto importante é a etapa de verificação de documentos, implementada pelas classes DocumentValidator e Validator, que realizam checagens como a validade da CNH e os dados do veículo, garantindo que motoristas apenas sejam aprovados após o cumprimento das regras estabelecidas. Dessa forma, o módulo de cadastro e autenticação não só registra os perfis de maneira eficiente, mas também adiciona uma camada de segurança e confiabilidade ao sistema.

Figura 04: Cadastro de Passageiro.



Fonte: Eclipse IDE.

Figura 05: Cadastro de Motorista.



Fonte: Eclipse IDE.

Figura 06: Categorias de Veículos.

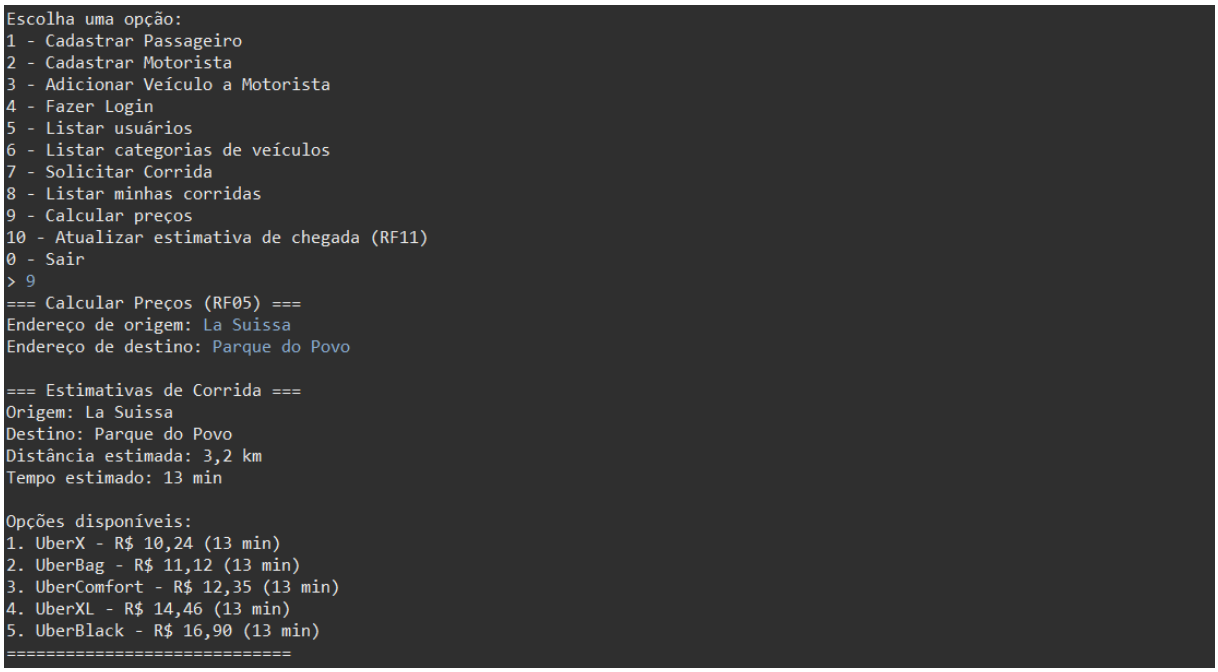
Fonte: Eclipse IDE.

3.2 Solicitação de Corrida

A solicitação de corridas é um dos núcleos centrais do sistema UberPB. Esse processo é modelado pela classe Ride, que funciona como entidade principal de uma viagem, armazenando informações como origem, destino, passageiro, motorista e status da corrida. O gerenciamento dessa entidade é feito pelo RideService, que contém métodos responsáveis por criar novas corridas, atualizar seu estado e disponibilizar os dados aos usuários. Quando um passageiro solicita uma corrida, ele informa origem, destino e categoria do veículo desejado. A partir disso, o sistema inicia uma série de cálculos e verificações para tornar a experiência mais próxima de um sistema real de mobilidade.

O cálculo do preço e do tempo estimados é realizado pelo PricingService, que utiliza as definições de tarifas contidas nas classes Tariff e VehicleCategory. O sistema diferencia categorias de veículos, aplicando preços mais altos em opções premium e valores reduzidos em modalidades básicas. Para fornecer resultados consistentes, foi implementada a classe utilitária DistanceCalculator, que calcula distâncias aproximadas entre os pontos informados, baseando-se em simulações. Esse cálculo é usado tanto para estimar o preço final quanto o tempo de chegada previsto. Além disso, os resultados do cálculo são encapsulados na classe PricingInfo, que guarda os detalhes da estimativa e permite consultas rápidas. Assim, o processo de solicitação de corrida garante clareza para o passageiro, viabiliza a escolha da categoria e mantém a consistência dos dados em todo o fluxo.

Figura 07: Cálculo de valores para corrida.



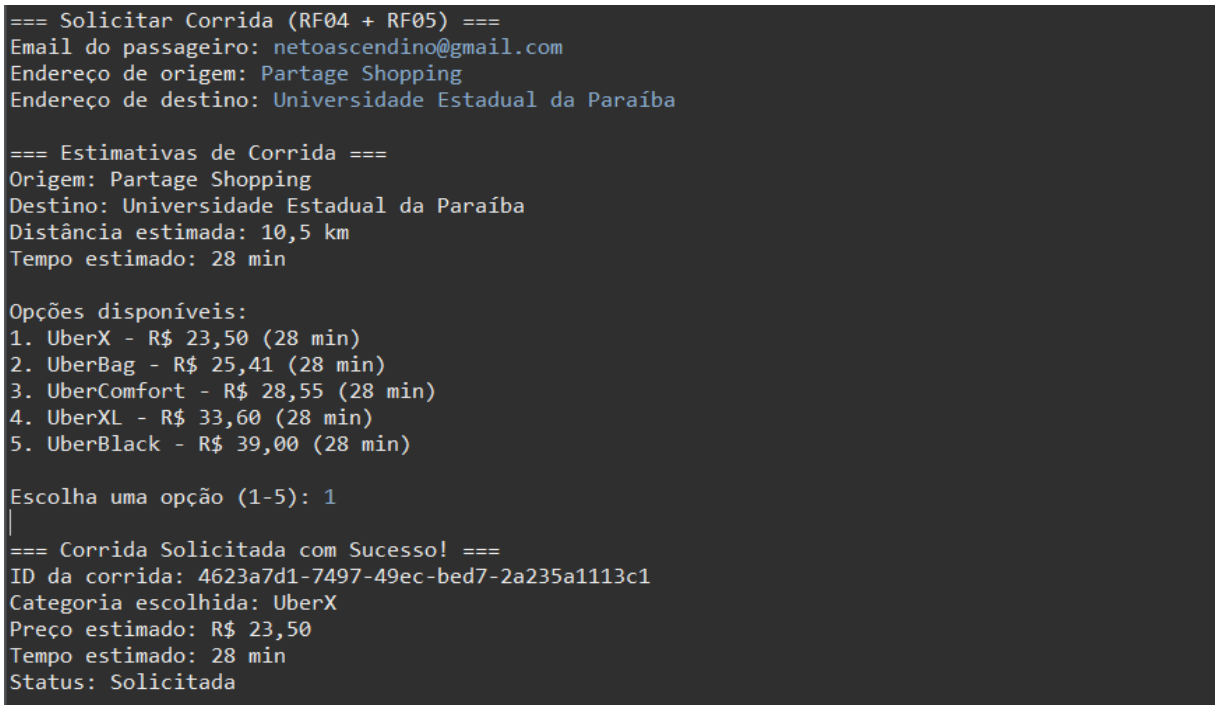
Fonte: Eclipse IDE.

3.3 Aceite da Corrida

Após a solicitação, o sistema precisa gerenciar a distribuição da corrida entre motoristas disponíveis. Esse processo é conduzido pelo RideService, que notifica motoristas da categoria escolhida e os permite aceitar ou recusar a viagem. Quando um motorista aceita, o sistema associa automaticamente o passageiro ao profissional mais próximo, registrando o vínculo no objeto Ride. Caso o motorista recuse, a corrida é redirecionada para outro motorista habilitado, de modo a garantir que o passageiro seja atendido. Essa lógica traz para o projeto uma dinâmica similar à dos sistemas reais de transporte por aplicativo, ainda que em versão simplificada.

A consistência desse processo é assegurada pelo RideRepository, que mantém os dados das corridas persistidos em arquivos locais. Cada mudança de estado — seja a criação, aceite, início ou conclusão — é registrada, de forma que o histórico da corrida possa ser recuperado a qualquer momento. O status da viagem pode ser atualizado para valores como pendente, em andamento ou concluída, permitindo que motoristas e passageiros tenham uma visão compartilhada do progresso. Com isso, o módulo de aceite não apenas garante o funcionamento do sistema de distribuição de corridas, como também assegura transparência e rastreabilidade no fluxo das operações

Figura 08: Simulação de corrida.



Fonte: Eclipse IDE.

3.4 Acompanhamento da Corrida

O acompanhamento da corrida foi desenvolvido para oferecer aos usuários a possibilidade de visualizar o andamento da viagem em tempo real, ainda que em ambiente simulado. Esse processo também é controlado pelo RideService, que atualiza periodicamente a posição do motorista durante a corrida. A atualização da localização é feita em conjunto com a classe DistanceCalculator, que recalcula a cada etapa a distância restante até o destino. Com base nesse valor, o sistema gera uma nova estimativa de tempo de chegada, que pode ser consultada pelo passageiro durante o deslocamento.

O motorista, por sua vez, recebe uma rota otimizada, representada como uma simulação de navegação. Esse recurso reforça a similaridade com sistemas reais de GPS utilizados em aplicativos de transporte. Além disso, o objeto PricingInfo pode ser consultado ao longo da corrida, disponibilizando os cálculos de preço e tempo previstos. Ao final da viagem, o status da corrida é atualizado para concluída, e o registro é salvo no RideRepository, criando um histórico permanente tanto para motoristas quanto para passageiros. Esse histórico é fundamental para oferecer rastreabilidade e confiabilidade ao sistema, assegurando que todas as corridas possam ser revisadas posteriormente.

Figura 09:Histórico de corrida.

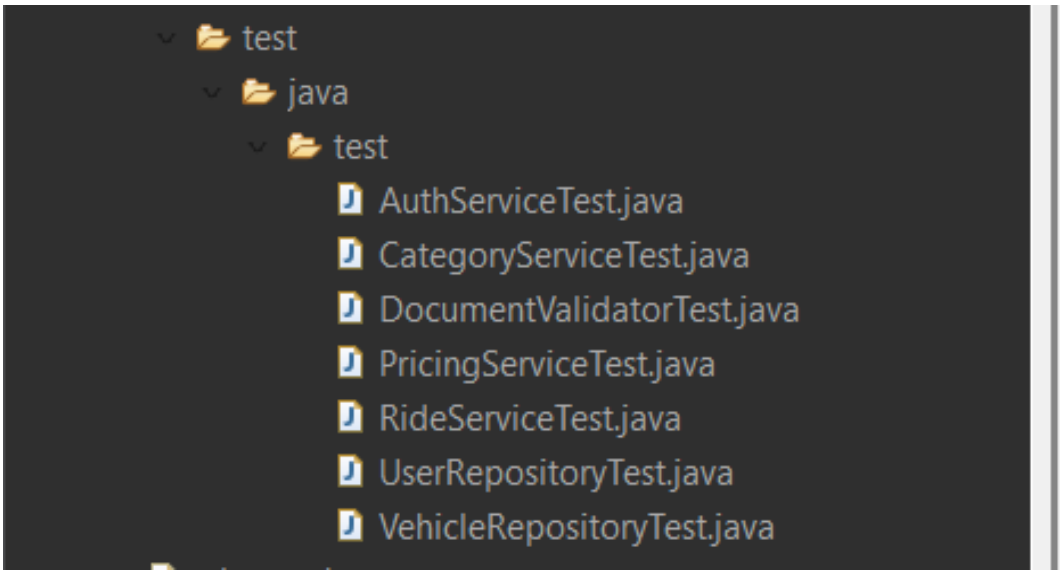
Fonte: Eclipse IDE.

3. TESTES E COBERTURA DE CÓDIGO

Com o objetivo de garantir robustez e qualidade no desenvolvimento, o projeto UberPB foi acompanhado por uma bateria de testes unitários, todos implementados com o framework JUnit. Os testes foram organizados em classes específicas para cada módulo do sistema. O AuthServiceTest valida fluxos de cadastro e login, verificando casos de sucesso e falha na autenticação. O CategoryServiceTest assegura que as categorias de veículos sejam registradas e consultadas corretamente. Já o DocumentValidatorTest foi desenvolvido para garantir que apenas motoristas com documentos válidos sejam aprovados no sistema. O PricingServiceTest, por sua vez, confere se os cálculos de preço e tempo estão consistentes com as regras estabelecidas, enquanto o RideServiceTest cobre fluxos mais complexos, como solicitação, aceite, andamento e finalização de corridas.

Além dos serviços, a camada de persistência também recebeu testes específicos, representados pelo UserRepositoryTest e VehicleRepositoryTest, que validam a gravação e leitura de dados em arquivos locais. Para acompanhar a evolução da cobertura de testes, foi utilizado o Emma Plugin, integrado ao Eclipse. Essa ferramenta gera relatórios detalhados, permitindo identificar quais classes, métodos e linhas foram exercitados durante a execução da suíte de testes. A meta estabelecida para o projeto foi de pelo menos 80% de cobertura, e os resultados alcançados confirmam que os principais fluxos foram amplamente validados. Dessa forma, os testes asseguram não apenas a funcionalidade correta do sistema, mas também sua estabilidade em diferentes cenários, promovendo confiança e qualidade no código entregue.

Figura 10: Testes utilizados durante a Release.



Fonte: Eclipse IDE.

4. CONCLUSÃO

O período de desenvolvimento da Release 1 foi dedicado à implementação do requisito 1 até o requisito 12. A equipe, composta por Ascendino Martins de Azevedo Neto, Johnatan Rodrigues dos Santos E Laryssa Finizola Costa da Silva trabalhou em duas sprints consecutivas (Sprint 1: 26/08 – 08/09 e Sprint 2: 09/09 – 24/09), utilizando Java como linguagem de programação, arquitetura MVC e persistência de dados em arquivos locais e testes em JUnit.

Nesta release, o sistema UberPB consolidou sua arquitetura em camadas bem definidas, contemplando modelos, repositórios e serviços, o que garante separação de responsabilidades, maior clareza e facilidade de manutenção. Foram implementados os requisitos funcionais principais, como cadastro e autenticação de usuários, solicitação de corridas, atribuição a motoristas e cálculo de preço e tempo estimados.

A modelagem, representada nos diagramas de caso de uso, classes e sequência, reforça a visão integrada do sistema e sua aderência aos padrões de projeto, além de destacar a comunicação entre entidades e serviços. Com isso, o sistema atinge um nível satisfatório de funcionalidade e qualidade de código, servindo como base sólida para futuras iterações, onde poderão ser incluídas melhorias de usabilidade, otimização de algoritmos e novas funcionalidades.