Universidade Estadual do Piauí - UESPI
Curso de Bacharelado em Ciência da Computação
Disciplina de Tópicos Especiais em Computação II

Trabalho Final

Objetivo

Desenvolver **testes unitários e/ou de integração** para **um caso de uso** do sistema de corridas por aplicativo descrito no projeto anexo. O trabalho deve aplicar os conceitos de **Clean Code**, **Clean Architecture**, **SOLID**, e **TDD**. O foco será nas camadas de **domínio (entities)** e **aplicação (use cases)**.

Descrição do Trabalho

1. Cada equipe (de até 3 integrantes) deve escolher **um caso de uso** listado no escopo do projeto.
2. A implementação será restrita às camadas de domínio e aplicação, simulando dependências externas com **test doubles**.
3. O trabalho deve ser desenvolvido seguindo os princípios de **TDD**, começando pelos testes.
4. O código e comentários devem ser escritos **em inglês**.
5. Além do código, a entrega deve incluir um diagrama representando das **camadas de domínio e aplicação**, mostrando as relações de dependência entre classes, incluindo os componentes principais (entidades, interfaces, use cases).

Casos de Uso Disponíveis

Os casos de uso disponíveis para escolha são:

1. **Signup (Criação de Conta)**
2. **GetAccount (Obter Conta)**
3. **RequestRide (Solicitar Corrida)**
4. **AcceptRide (Aceitar Corrida)**
5. **StartRide (Iniciar Corrida)**
6. **FinishRide (Finalizar Corrida)**

7. **GetRide (Obter Corrida)**
8. **GetRides (Obter as Corridas)**
9. **UpdatePosition (Atualizar Posição)**

Requisitos do Trabalho

1. **Clean Code, Clean Architecture e SOLID:**
   - Código limpo, legível e fácil de entender.
   - Organização em camadas conforme o padrão Clean Architecture.
   - Aplicação rigorosa dos princípios SOLID para garantir:
     - *Single Responsibility Principle (SRP)*: Cada classe ou função deve ter apenas uma razão para mudar.
     - *Open/Closed Principle (OCP)*: Componentes devem ser abertos para extensão, mas fechados para modificação.
     - *Liskov Substitution Principle (LSP)*: Subtipos devem ser substituíveis por seus tipos base sem alterar o comportamento esperado.
     - *Interface Segregation Principle (ISP)*: Interfaces devem ser específicas e adaptadas às necessidades dos clientes.
     - *Dependency Inversion Principle (DIP)*: Depender de abstrações, não de implementações.
2. **TDD:**
   - Testes devem ser escritos antes do código funcional.
   - Utilizar Jest para implementar os testes.
   - Outros pacotes auxiliares de *mocking* também podem ser utilizados.
3. **Test Doubles:**
   - Empregar *mocks* para simular dependências externas, como repositórios, serviços, *adapters*, *gateways*, etc.
4. **TypeScript:**
   - Código implementado em TypeScript, configurado com `strict: true`.
   - Um *template* de projeto TypeScript já configurado pode ser baixado no link:
     - https://github.com/uespi-phb/setups.git
5. **Representação das Camadas:**
   - Representar as camadas de domínio e aplicação e as relações de dependência entre classes, incluindo os componentes principais (entidades, interfaces, use cases).

Entrega

- O projeto poderá ser desenvolvido por uma equipe de até 3 (três) pessoas.
- O trabalho deve ser disponibilizado em um repositório público no GitHub.
- A entrega do trabalho deverá ser feita por meio do sistema SIGAA, informando o link do repositório GitHub referente ao projeto.
- Prazo de entrega: 12/01/2025.

1. **Código-fonte:**
   - Código-fonte TypeScript contendo os testes e implementação do caso de uso escolhido.
2. **Diagrama de Dependências:**
   - Arquivo PDF com a representação gráfica das camadas de domínio e aplicação e as relações de dependência entre classes, incluindo os componentes principais (entidades, interfaces, use cases).
3. **Arquivo README:**
   - Arquivo README, em formato Markdown, na raiz do projeto contendo:
     - Nomes dos integrantes da equipe.
     - Instruções claras sobre como compilar e executar os testes.

Critérios de Avaliação

1. **Diagrama de Dependências (20%):**
   - **Clareza e Coerência:**
     - O diagrama deve representar de forma clara e coerente as relações entre as camadas de domínio e aplicação, evidenciando as interações entre entidades, interfaces e use cases.
   - **Desacoplamento e SOLID:**
     - Verificar se as classes respeitam os princípios **SOLID**, com especial atenção para:
       - Separação de responsabilidades (SRP).
       - Uso de abstrações em vez de implementações diretas (DIP).
     - Relações de dependência bem definidas, sem violações arquiteturais, como dependências cíclicas ou invasão de camadas.
2. **Adesão aos princípios de Clean Architecture e Clean Code (40%):**
   - **Organização Arquitetural:**
     - O código deve estar adequadamente organizado em camadas conforme o padrão **Clean Architecture**.

    - As camadas devem ser independentes, garantindo que a lógica de domínio não dependa de detalhes de implementação externos.
- **Qualidade do Código:**
    - O código deve ser limpo e legível, observando os princípios do Clean Code, com nomes significativos, eliminação de duplicações e boas práticas de formatação.

3. **Uso correto de TDD e testes bem implementados (40%):**
    - **Processo de Desenvolvimento:**
        - Avaliação baseada nos *commits* realizados no repositório, verificando se os testes foram criados antes da implementação funcional.
        - Cada commit deve ser pequeno e descrever claramente a mudança realizada, demonstrando a progressão incremental do desenvolvimento.
    - **Qualidade dos Testes:**
        - Cobertura de casos relevantes, incluindo cenários positivos e negativos.
        - Uso de **test doubles** (*mocks*) para simular dependências externas de forma adequada.
        - Testes devem ser claros, legíveis e organizados, com nomes que descrevam corretamente os cenários testados.

Universidade Estadual do Piauí - UESPI
Curso de Bacharelado em Ciência da Computação
Disciplina de Tópicos Especiais em Computação II

Sistema Backend de Corridas por Aplicativo
Descrição Geral do Projeto

Contexto

Sistema backend, baseado em REST API, de um serviço de corridas por aplicativo.

Créditos

A descrição desse sistema foi idealizada por Rodrigo Branas em seu curso Clean Code e Clean Architecture.

Escopo

1. Criação da Conta

- Tudo começa com a CRIAÇÃO DA CONTA, que pode ser feita pelo PASSAGEIRO ou pelo MOTORISTA ou ambos.
- É necessário informar o NOME, EMAIL, CPF, SENHA e, no caso de motorista, a PLACA DO CARRO.
- Não será permitido que o mesmo email tenha mais de uma conta, não existe restrição em relação ao CPF.

2. Corrida

- Após a CRIAÇÃO DA CONTA, um PASSAGEIRO pode SOLICITAR UMA CORRIDA informando a origem e o destino, que são coordenadas com latitude e longitude.
- Ao solicitar, o passageiro não é cobrado e nem o valor da corrida é exibido já que ele será calculado apenas depois que a corrida for finalizada.
- A CORRIDA ficará com o status de SOLICITADA e deve ser retornado o IDENTIFICADOR da corrida.
- Caso o PASSAGEIRO já tenha uma corrida não CONCLUÍDA, não deve ser possível SOLICITAR UMA CORRIDA.
- Um MOTORISTA pode ACEITAR A CORRIDA que terá seu status modificado para ACEITA e o PASSAGEIRO fica no aguardo do MOTORISTA que aceitou a corrida.

- Após ACEITAR A CORRIDA o MOTORISTA não poderá aceitar qualquer outra corrida
- Não deve ser possível ACEITAR UMA CORRIDA que não estiver com o status SOLICITADA.
- Após o PASSAGEIRO entrar no carro o MOTORISTA deve INICIAR A CORRIDA, que terá seu status modificado para EM ANDAMENTO.
- Durante a CORRIDA, a cada minuto, é feita uma ATUALIZAÇÃO DA POSIÇÃO, registrando cada latitude e longitude do trajeto da CORRIDA.
- Chegando ao destino o MOTORISTA deve ENCERRAR A CORRIDA, que agora terá o status de CONCLUÍDA. Com a lista de POSIÇÕES o valor da corrida deve ser calculado por meio da soma da distância entre cada uma delas, seguindo a seguinte tabela de preço:

Horário	**Tarifa**
Dia normal entre 8 e 22 horas	R$2,10 por km
Dia normal entre 22 e 8 horas	R$3,90 por km
Domingo entre 8 e 22 horas	R$2,90 por km
Domingo entre 22 e 8 horas	R$5,00 por km

- Dessa forma a tarifa da CORRIDA é mais justa tanto para o PASSAGEIRO quanto para o MOTORISTA em caso de trânsito intenso, desvios ou corridas mais longas que tenham mudança da tarifa de preço.
- O ciclo de vida da CORRIDA é SOLICITADA -> ACEITA -> EM ANDAMENTO -> CONCLUÍDA, podendo o PASSAGEIRO ou o MOTORISTA CANCELAR A CORRIDA, desde que ela não esteja CONCLUÍDA ou CANCELADA.
- Após o cálculo da tarifa o PAGAMENTO DEVE SER PROCESSADO.
- Algumas ações serão necessárias como OBTER A CONTA, OBTER A CORRIDA, OBTER AS CORRIDAS.
- Outras também podem ser realizadas, mas são opcionais, como ESTIMAR O VALOR DA CORRIDA, AUTORIZAR O PAGAMENTO, CAPTURAR O PAGAMENTO, VERIFICAR CONTA, ATUALIZAR DADOS DA CONTA, AVALIAR A CORRIDA e ENVIAR COMPROVANTE DA CORRIDA.

Use Cases

1. Signup (CRIAÇÃO DA CONTA)
2. GetAccount (OBTER A CONTA)
3. RequestRide (SOLICITAR UMA CORRIDA)
4. AcceptRide (ACEITAR A CORRIDA)
5. StartRide (INICIAR A CORRIDA)

6. FinishRide (FINALIZAR A CORRIDA)
7. GetRide (OBTER A CORRIDA)
8. GetRides (OBTER AS CORRIDAS)
9. UpdatePosition (ATUALIZAR POSIÇÃO)
10. ProcessPayment (PROCESSAR PAGAMENTO)

Modelo de Dados

A seguir é apresentada uma sugestão de modelo de dados para PostgreSQL.

```
drop schema rideapi cascade;
create schema rideapi;

create table rideapi.account (
    account_id uuid,
    name text,
    email text,
    cpf text,
    car_plate text,
    is_passenger boolean,
    is_driver boolean,
    password text,
    password_algorithm text
);

create table rideapi.ride (
    ride_id uuid,
    passenger_id uuid,
    driver_id uuid,
    status text,
    fare numeric,
    distance numeric,
    from_lat numeric,
    from_long numeric,
    to_lat numeric,
    to_long numeric,
    date timestamp
);

create table rideapi.position (
    position_id uuid primary key,
    ride_id uuid,
    lat numeric,
    long numeric,
    date timestamp
);
```

DEPENDENCY DIAGRAM EXAMPLE

Authentication Use Case

Dependency Diagram