1 Introdução

Este trabalho aborda a concepção e implementação de um módulo de vendas. Destaca-se o encapsulamento da lógica em casos de uso, independente da infraestrutura, e a aplicação de estratégias flexíveis de cálculos de descontos. A separação eficiente de responsabilidades, entre consultas e vendas, utilizando objetos de transferência de dados, é enfatizada. A persistência de dados é realizada por meio do Java Persistence API (JPA), com o Sistema de Gerenciamento de Banco de Dados (SGBD) H2 em memória. O objetivo é criar um módulo de vendas sólido e fácil de manter, preparado para futuras evoluções do sistema, uma vez que as mudanças na implementação técnica não afetam diretamente a lógica de negócios.

2 Implementação

O desenvolvimento de uma aplicação eficiente e escalável requer uma aplicação de princípios sólidos de arquitetura e design. Nesse contexto, a estrutura da aplicação foi cuidadosamente planejada, seguindo uma abordagem voltada para casos de uso, onde a lógica do sistema foi encapsulada de forma clara e concisa. Essa abordagem permite que as funcionalidades sejam desenvolvidas e testadas de forma isolada, promovendo a modularidade e a reutilização de código.
Essa separação de responsabilidades contribui para uma estrutura mais organizada e coesa, tornando mais fácil compreender e dar manutenção do código. A separação de responsabilidades também foi interrompida por algumas entidades, como pedido e item, utilizando objetos de transferência de dados (DTOs) para promover a comunicação eficaz entre as diversas camadas da aplicação. Essa prática ajuda a evitar ocultações entre as entidades e simplificar o fluxo de dados ao longo das diferentes camadas da aplicação, lembrando que os endpoints foram testadas usando workspace Postman.

Figura 1: Postman

3 Diagrama de classes

No domínio, destacam-se classes como ForaDeValidadeException, Item, ItemDTO, Orçamento, Pedido, PedidoDTO, Produto, e ProdutoNaoExisteException. Estas classes refletem objetos e abordagens relevantes para o módulo de vendas. Item e Produto apresentam produtos e itens de um pedido, enquanto Pedido e Orçamento modelam as transações de venda. As abordagens, como ForaDeValidadeException e ProdutoNaoExisteException, oferecem um mecanismo para tratar erros específicos no domínio.
No âmbito dos casos de uso, classes como GalpaoRepoImp, JpaGalpaoRep, JpaOrcamentoImp, JpaPedidoImp, OrcamentoRepoImp, e PedidoRepoImp desempenham papéis cruciais. Essas classes representam implementações concretas de repositórios e serviços relacionados ao galpão, orçamento e pedidos. A utilização de interfaces (OrcamentoRepoImp e PedidoRepoImp) fornecem uma abstração que favorece a substituição fácil de implementações, seguindo o princípio da conversão de dependência.
Na camada de apresentação, destacam-se os controladores como ConsultasController, EstoqueController, VendasController, e ModuloVendasApplication. Esses controladores gerenciam a interação entre o usuário e o sistema, facilitando consultas, atualizações de estoque e transações de vendas. ModuloVendasApplication pode ser considerada uma classe principal, que coordena a aplicação como um todo.
A camada de teste é representada pelas classes: ModuloVendasApplication e OrcamentoRepoImpTest. Estas classes garantem que as implementações atendam aos requisitos especificados, validando a robustez e a correção do sistema.
Obs: O diagrama real vai estar no repositório no Git (pois não coube aqui).

Domínio:	Casos de Uso:	Presenter:	Teste:
ForaDeValidadeException Item ItemDTO Orçamento Pedido PedidoDTO Produto ProdutoNaoExisteException	GalpaoRepoImp JpaGalpaoRep JpaOrcamentoImp JpaPedidoImp OrcamentoRepoImp PedidoRepoImp	ConsultasController EstoqueController VendasController ModuloVendasAplication	ModuloVendasApplication OrcamentoRepoImpTest

Figura 2: Classes

4 Testes implementados

Testes unitários usando o framework JUnit e Mockito.
@ExtendWith(MockitoExtension.class): Essa anotação é usada para estender o suporte do JUnit com o Mockito, permitindo o uso de anotações do Mockito no teste.

- Classe ModuloVendasApplicationTests:

  Usa uma anotação @InjectMockspara injetar automaticamente as dependências simuladas na instância de "CalculaDescontoCase" quando o teste é concluído.

  O método setUp é anotado com @BeforeEach, que será executado antes de cada método de teste. O método de teste real é "calculaDesconto", nele, estão sendo configurados comportamentos esperados para os mocks "dezCompras" e "tresCompras". Esses mocks representam os

resultados esperados quando os métodos “calcula” são chamados com o argumento ”clienteName”. A cobertura de teste foi pensada visando calcular todas as possíveis compras, sendo possível calcular com compra superior e inferior a dez mil.

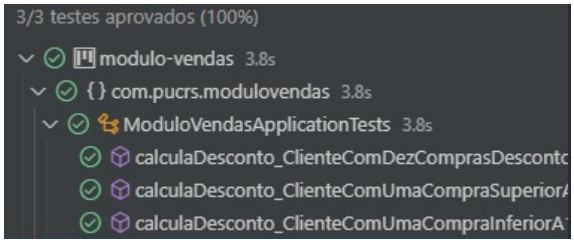

Figura 3: ModuloVendasApplicationTests

- Classe OrcamentoRepoImpTest:

  Ele cria mocks para uma interface “JpaOrcamentoImp” para isolar uma classe sob teste e verificar se os métodos dessa classe interagem corretamente com suas dependências simuladas. Os testes cobrem os métodos findByCod, findAll, persist, e findByNomeCliente, garantindo que a lógica da classe “OrcamentoRepoImp” funcione conforme o esperado.

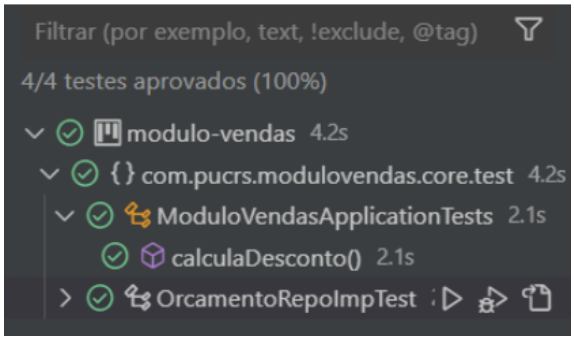

Figura 4: OrcamentoRepoImpTest

5 Requisitos

Injeção de Dependência:

Podemos observar o uso da anotação @Autowirednas classes CalculaDescontoCasee nos parâmetros (ConsultasControllere VendasController), essa anotação é típica do frameworks Spring, diminuindo a injeção de dependência.

Estratégia Padrão:

Uma classe de exemplo: CalculaDescontoCase implementa um padrão de Estratégia. Ela usa ”TresCompras”e ”DezCompras”para obter descontos com base no padrão definido pela ICalculaDesconto.

Padrão MVC (Model-View-Controller):

Dois exemplos: os drivers ”ConsultasController”e ”VendasController”seguem o padrão do projeto MVC. Eles atuam como controladores que interagem com o modelo e entidades como Orcamento e Pedido, e retornam respostas ao cliente.

Módulo estatístico com 3 modelos de análise:

- Top 3 produtos por preço:

```
Map<String, Double> top3ProdPorPreco = somatorioPorNome.entrySet().stream()
        .sorted(Map.Entry.<String, Double>comparingByValue().reversed())
        .limit(3)
        .collect(Collectors.toMap(
                Map.Entry::getKey,
                Map.Entry::getValue,
                (existing, replacement) -> existing,
                LinkedHashMap::new
        ));
return top3ProdPorPreco;
```

Figura 5: Por preço

- Top 3 nomes por somatório:

```
Map<String, Double> top3NomesPorSomatorio = somatorioPorNome.entrySet().stream()
        .sorted(Map.Entry.<String, Double>comparingByValue().reversed())
        .limit(3)
        .collect(Collectors.toMap(
                Map.Entry::getKey,
                Map.Entry::getValue,
                (existing, replacement) -> existing,
                LinkedHashMap::new
        ));
return top3NomesPorSomatorio;
```

Figura 6: Por somatório

- Top 3 compradores:

```
Map<String, Long> top3Compradores = contagemNomes.entrySet().stream()
        .sorted(Map.Entry.<String, Long>comparingByValue().reversed())
        .limit(3)
        .collect(Collectors.toMap(
                Map.Entry::getKey,
                Map.Entry::getValue,
                (existing, replacement) -> existing,
                LinkedHashMap::new
        ));
return top3Compradores;
```

Figura 7: Compradores