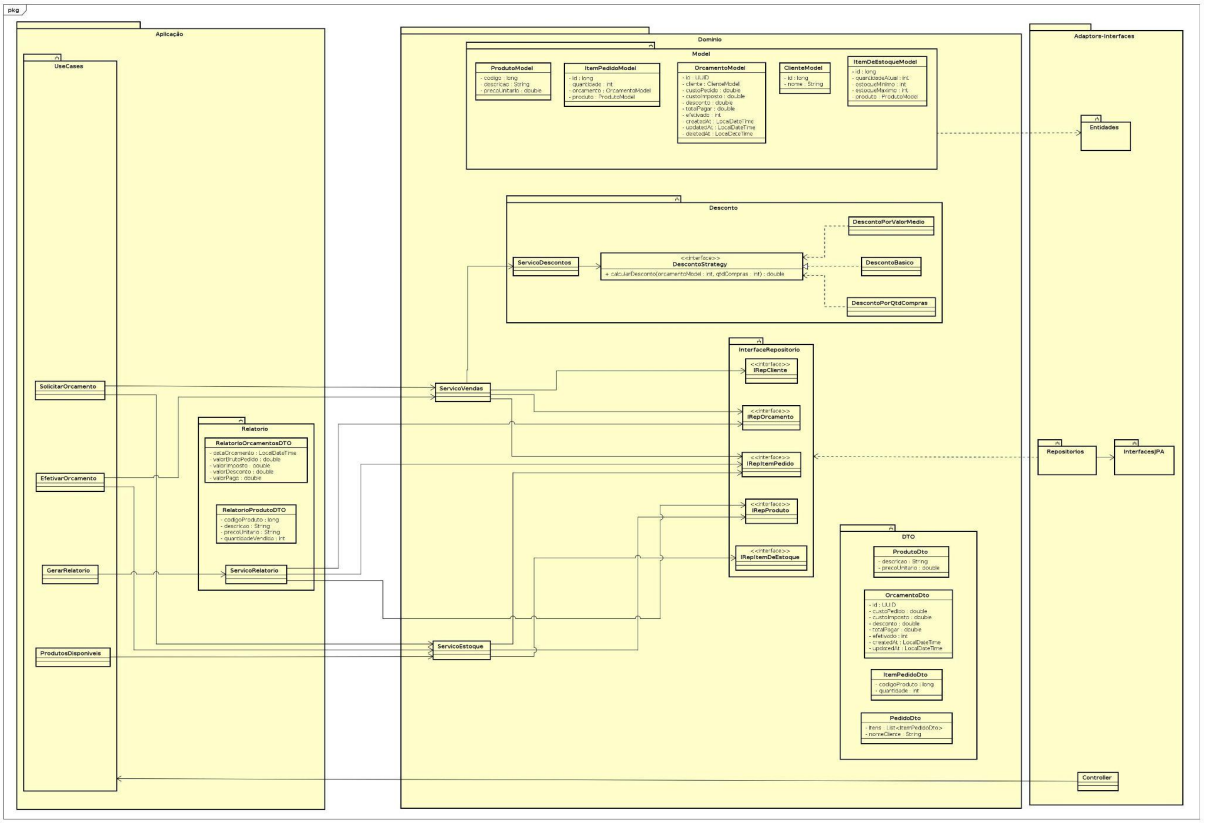
Trabalho Final de Fundamentos de Desenvolvimento de Software - PUCRS
Professor: Bernardo Copstein

Sistema de Vendas

Mateus F. Poletto

Diagrama de Classes:



Conforme solicitado pelos StakeHolders (Bernardo Copstein, Alexandre Augustini), foi construída a aplicação monolítica do Back-End de um sistema de vendas parecido com o sistema de vendas de uma loja de conveniência. A ferramenta foi construída utilizando os fundamentos da arquitetura limpa (Clean Architecture), o que ocasionou no acoplamento de alguns padrões de projeto os quais foram julgados necessários além dos princípios S.O.L.I.D.

Ainda na questão de organização do Software, foi dividido o Back-End em 3 camadas. A camada mais externa nomeada no diretório da aplicação é “adaptorsInterfaces” (interfaces e adaptadores), essa possui como função conectar a aplicação com o mundo externo e possuir os trechos do programa mais acoplados com as tecnologias específicas

trabalhadas, como no caso da biblioteca de conexão com a o banco de dados para persistência de informações, o JPA. Além do mais, possui a implementação dos repositórios das Entidades definidas pelos Modelos.

Indo em direção às camadas mais internas do sistema, chegamos a camada de nome “aplicacao” (aplicação), essa possui algumas regras extras do negócio não tão sérias quantas as encontradas na camada mais interna, porém fatores mais “simples” são encontrados aqui e vistos nos Casos de Uso presentes. Além do mais, trata-se de uma camada com funções extras da aplicação desenvolvida, como por exemplo a geração de relatórios.

Chegando ao “core” da app temos a camada de “dominio” (domínio), nessa camada encontramos as regras de negócio definidas, serviços principais, estratégias, os modelos que possuem a modelagem dos dados a serem trabalhados e as interfaces dos repositórios a serem implementados. Encontramos todas essas informações aqui e tentamos deixar as mesmas de forma mais independente possível com as tecnologias utilizadas para que no momento que desejamos mover nossa aplicação para outra tecnologia, ou até mesmo fazer grandes alterações no projeto, o processo não se torne caro e complicado.

Padrões de Projeto:

Repositório: Com a intenção de promover a separação de responsabilidades, facilitar futuras trocas de fonte de dados e lógica de controle das mesmas, além de facilitar testes, o padrão Repository foi amplamente utilizado, abrangendo e se relacionando com todos os Modelos existentes que dão a ideia de como será feita a persistência dos dados. Foram 5 modelos criados, dentre eles “Cliente”, “Orcamento”, “ItemPedido”, “Produto” e “ItemDeEstoque”, portanto 5 interfaces de repositórios que deixam em aberto a lógica a ser utilizado para a implementação escolhida na ocasião necessária, sendo elas: “IRepCliente”, “IRepOrcamento”, “IRepItemPedido”, “IRepProduto” e “IRepItemDeEstoque”. Dessa maneira, serviços não precisam se preocupar com detalhes específicos do repositório.

Injeção de Dependẽncias: Com intenção de promover o desacoplamento, facilitar o teste unitário e melhorar a legibilidade, foi utilizado esse padrão em todo o projeto, um exemplo prático disso é visto é em “ServicoEstoque”. Nessa classe a injeção de dependências é de responsabilidade do Framework, apenas definimos a interface que deve ser implementada pelas classes que podem ser utilizadas pelo serviço e então deixamos que a tecnologia cuide dessa responsabilidade para nós.

Princípio da Segregação de Interfaces (ISP) e Princípio da Inversão de Dependência (DIP): Com a intenção de facilitar manutenção, adição de novas tecnologias ou até alteração de tecnologias utilizadas foram utilizados o ISP e o DIP. Nesse projeto de vendas de uma loja, essas duas estratégias são vistas atuando de maneira conjunta nos repositórios, foram criadas várias interfaces, uma para cada modelo, criando interfaces pequenas e com utilidades específicas (ISP). Essas interfaces, foram injetadas nos serviços desse aplicativo, porém o serviço que está tendo essas interfaces injetadas está buscando

utilizar a implementação da mesma, o que ocorre ao definirmos a implementação Primária para o nosso projeto (DIP).

Princípio da Responsabilidade Única (SRP): Aplicado nas classes de Serviços do projeto, esse padrão foi utilizado para tornar o código mais fácil de entender, testar e manter, porque cada classe tem uma responsabilidade clara e bem definida. Isso é visto por exemplo na classe “ServicoEstoque” que sua função é apenas gerenciar o estoque, e suas função refletem isso. Por exemplo, a função “produtosDisponiveis()” retorna todos os produtos disponíveis e a função “verificaProdutos()” verifica se há quantidade suficiente de um produto no estoque.

Princípio Aberto/Fechado (OCP) (Strategy): No projeto, o OCP serviu de grande ajuda para fazer o módulo de descontos do projeto. Como o projeto possui 3 estratégias diferentes de uso de descontos e no início só possuía uma, foi utilizado o OCP para criar outras estratégias de desconto e utilizadas de maneira que foi necessário alterar código antigo, e sim apenas expandi-lo trazendo as estratégias novas de desconto. Nesse momento, o padrão Strategy facilitou o trabalho de aplicação do OCP. Pois assim foi possível encapsular as estratégias de desconto e fazer com que as mesmas variassem de forma independente, alterando a estratégia conforme o cliente se encaixava.