Relatorio do primeiro PBL de Algoritmos e Programação II

Pedro Henrique de A. Silva

[1] Universidade Estadual de Feira de Santana (UEFS)

ph.academic2002@gmail.com

1. Introdução

Após ser contratado, seu chefe lhe solicita para desenvolver um programa uma loja online, tal programa deve procurar seguir as boas práticas de SOLID e ser desenvolvido em Java. A loja precisa ser capaz de cadastrar clientes, que eles façam pedidos de produtos, que seja possível gerenciar o estoque e pagamentos, também deve ser capaz de cadastrar os Produtos com base nos seus tipos. O programa deve ser capaz de manter um controle preciso do estoque e seus pedidos, além de precisar ser fácil de usar e convidativo para os clientes, com funções como adicionar ao carrinho, exibir seu preço total e produtos. Com os principais pontos e necessidades postos em prática, partimos para a fundamentação teórica necessária para resolvermos o problema.

2. Fundamentação Teórica

Nessa sessão, será desenvolvido todos os conhecimentos que foram precisos adquirir e suas devidas fontes para conseguir chegar na resolução desse problema. A fundamentação teórica logo abaixo citado nas Seções 2, se referem principalmente a conceitos de Padrões de projeto e SOLID, a linguagem utilizada nesse problema que é a Java SE edition desenvolvido pela empresa [Oracle].

2.1. Principio da Responsabilidade Única

Esse principio prevê que uma classe deve procurar ter uma única responsabilidade, basicamente, cada classe deve ser especializada em uma tarefa e apenas nela, tornando simples entender o seu funcionamento.

2.2. Princípio Aberto/Fechado

O princípio Aberto e fechado prevê que seu código deve ser capaz de adicionar novas funcionalidades sem necessitar modificar o que já foi escrito, ou seja, aberto a extensão e fechado a modificação.

2.3. Princípio da Substituição de Liskov

Liskov dizia que os objetos de uma classe derivada devem poder substituir objetos da classe sem alterar a correta execução do código, basicamente, uma subclasse deve ser capaz de ser substituída pela super classe sem causar erros.

2.4. Princípio da Segregação de Interfaces

Esse principio indica que uma interface não deve forçar uma classe a implementar um método que ela não utiliza, basicamente indicando que essa interface está grande de mais e possui métodos desnecessários, podendo ser quebrado em outras duas ou mais provavelmente.

2.5. Princípio da Inversão de Dependência

Um programa deve depender de implementações abstratas, as abstrações não devem depender de detalhes, os detalhes devem depender das abstrações, indica basicamente a necessidade de utilizar de abstrações para facilitar a portabilidade do programa.

2.6. Singleton

Conforme aprendido em [Refactoring Guru 2024], Padrões de projeto servem para resolver problemas comuns e recorrentes do código, entre eles, podemos citar o Singleton, padrão que busca mediante um construtor private, variável da própria classe static e um getter publico forçar o programa a possuir apenas uma instância única daquela classe e dando um acesso global e facilitado àquela instancia.

2.7. Factory

Esse padrão consiste em fornecer uma interface para a criação de objetos de uma superclasse, permitindo que seja escolhido o tipo da subclasse, sendo muito útil para promover flexibilidade ao código e desacoplamento, sendo de fácil modificação caso necessite implementar novas subclasses ou remover alguma.

2.8. Facade

Facade é com certeza uma dos padrões de projetos mais populares, servido de fachada, ela fornece uma interface simplificada para uma aplicação, promove desacoplamento e esconde a complexidade do código, normalmente está no nível mais alto do código e acabam sendo instanciadas no front-end.

2.9. Prototype

Prototype permite a criação de uma cópia de uma classe porem em uma instância diferente, garantindo que a alteração de uma instância não resulta na alteração do seu clone e vice-versa, sendo útil assim, quando desejamos que uma classe não sofra alteração direta por outra, ou que uma classe não conheça o endereço de memória de uma instância mais comprometedora.

3. Metodologia

Antes de descrever melhor as estratégias utilizadas para resolver o problema, é de suma importância citar o Ambiente de Desenvolvimento Integrado(IDE) utilizado que foi o IntelliJ da empresa [JetBrains 2024], no qual, utilizei como principal plataforma para desenvolver o programa que rodará esse código e principalmente debugar o código de modo a encontrar seu real funcionamento.

A resolução do problema começou com as discussões nas sessões PBLs e estudos dos assuntos previamente citados na Seção 2, após isso, começou-se o processo de escrita do código, partindo das classes de mais baixo nível (Produto) até as classes de mais alto nível no programa (LojaFacade), após desenvolver todos os problemas, começou-se o período de testes unitários, foram feitos no total 40 testes até ter certeza de que estava seguindo o comportamento desejado, com testes e classes feitos, partiu-se para a interface gráfica onde eu fiz esboços a mão de como gostaria que funcionasse para então codificara em Swing.

4. Resultados e discussões

Todo o desenvolvimento do PBL passou pelo requisito de tentar seguir os princípios SOLID ao máximo e a implementação recomendada de 7 padrões de projeto.

4.1. Principais classes

A classe de mais baixo nível nesse programa é a abstrata Produto, que gera seus 3 tipos como subclasses.

Após o produto foi criado a classe estoque, classe essa que é capaz de armazenar e fazer as devidas manipulações de todos os produtos da aplicação num Map de <Intenger, TuplaEstoque>, sendo Intenger um ID gerado por meio de uma classe geradora de ID, e TuplaEstoque uma tupla entre produto e sua quantidade de estoque.

Logo após isso, foi criado também a classe Pedido e Carrinho, sendo Carrinho basicamente um estoque separado, que estará associada a cada cliente. Já pedidos, ele armazena uma instância de Carrinho e guardar outras informações do cliente, mas, basicamente, sua função é apenas armazenar informações e ser capaz de atualizar a situação do pedido.

Chegamos finalmente na classe Usuário, Usuário é uma classe bem simples que apenas armazena informações comuns a qualquer Usuário desse sistema, como seu login, nome e senha. Quando o Cliente herda dela, é adicionado mais funções como armazenamento e exibição de pedidos, armazenamento e manipulação de um carrinho, armazenamento e criação de PagamentoStrategy, é por via de Cliente que é gerado um pedido.

Temos também a classe Dono herdando de Usuário, sendo Dono a classe capaz de manipular e gerenciar o Estoque, além de gerenciar os pedidos de qualquer cliente.

Por fim, chegamos a classe ControllerLoja, que acopla as funções de Dono e Cliente numa classe só, fazendo a checagem do que deveria ser capaz de fazer conforme a pessoa que está logada.

4.2. Prototype e ID Generator

Foi criado uma classe capaz de gerar IDs únicos para Cliente, Produto e Pedidos, implementações a parte, essa logica foi essencial para podermos utilizar Prototype em Pedido e Produto, como dito na Seção 2.9, era necessário criar uma cópia da instância dessas 2 classes para a finalidade de impedir que por algum bug futuro fosse permitido que algum cliente mal-intencionado conseguisse mudar os preços de produtos ao modificar os produtos que estavam salvos na sua instância, ou, ao remover um produto do estoque, por compartilharem a mesma instancia, remover o produto do pedido.

4.3. Factory e Produto

Para tornar o código mais expansível e simples de entender, foi criado uma Factory para os tipos de Produto, sendo eles, Eletronico, Roupa e Alimento, essa mesma factory também implementa o Prototype de cada Produto, deixando o código simples de entender e de se expandir.Segue na figura 2, o diagrama de onde foi usado prototype e factory ao mesmo tempo

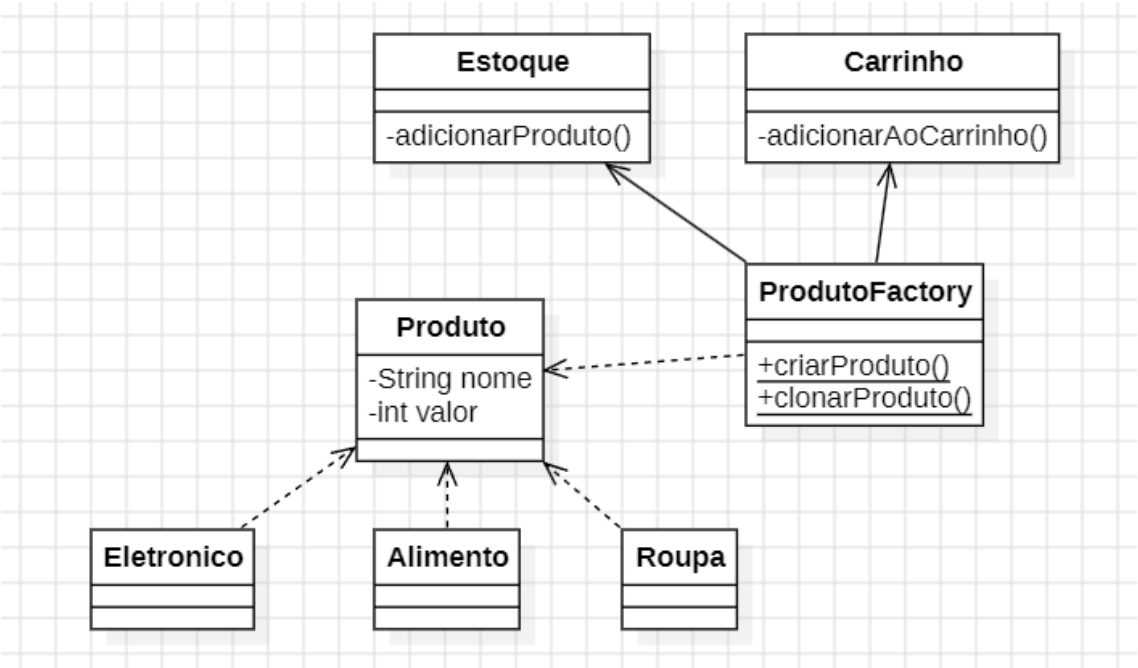

Figura 1. Diagrama de Prototype e Factory

4.4. Singleton e seus usos

A capacidade de Singleton criar um acesso global e instancia única para a classe Dono foi essencial, principalmente para o uso de Prototype de Pedido ao efetuarCompra em Cliente, tornando de fácil acesso armazenar o pedido original em Dono, e a cópia em Cliente. Singleton também foi utilizado em IDGenerator e em ControllerLoja, apenas para forçar o sistema a ter uma instância única dessas classes, porém, foi preciso também implementar formas de salvar essa instancia e garantir que ao carregar o programa, a instância seria carregada também.

4.5. Pagamento e Strategy

Pagamento é uma classe criada apenas para simbolizar o ato de pagar, como, não foi pedido implementar ainda questões de saldo do Cliente, foi criado as classes para permitir autenticar um pagamento em PayPal, cartão de crédito e transferência bancaria. Após isso, foi utilizado do padrão Strategy, padrão esse que sinaliza a atividade efetuarPagamento em comum para todos que implementem ela, mas, deixa a cargo de cada classe Strategy, a forma de como será implementada, ou seja, a estratégia que será utilizada.

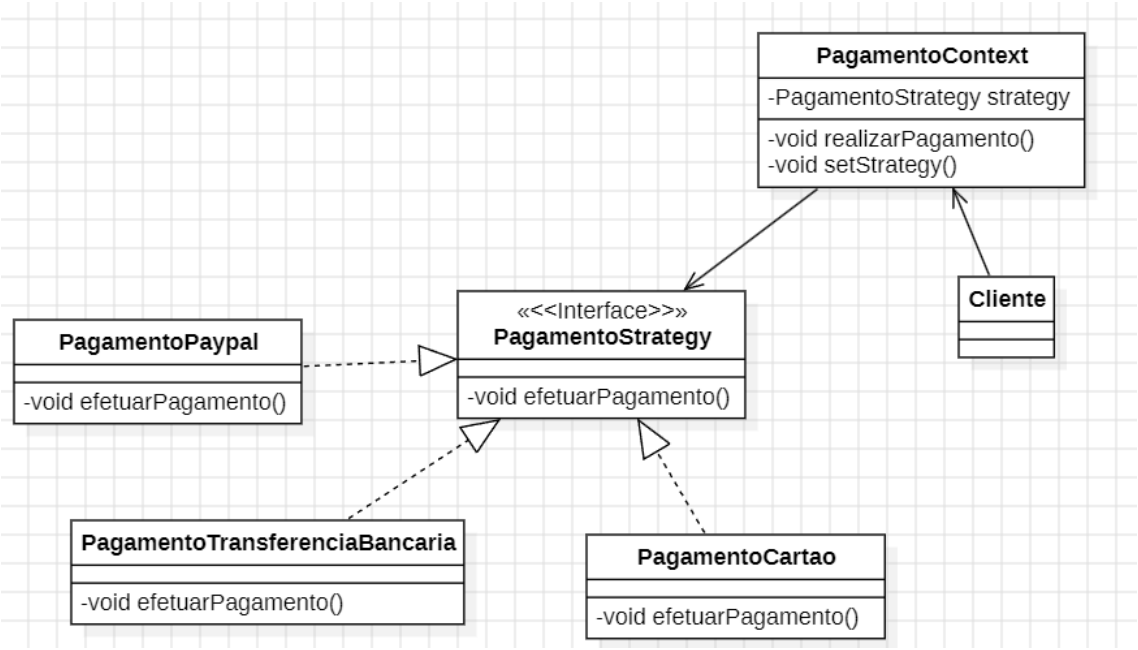

Figura 2. Diagrama de Strategy

4.6. Facade e Loja

Para adquirir todas as vantagens que Facade traz ao projeto e organizar o código antes de chegarmos ao front-End, pegou-se o controllerLoja e instanciou ele na LojaFacade, encapsulando todas as implementações em uma classe que apenas chama as funções desejadas.

4.7. Iterator e Observer

Iterator é uma classe que já vem implementada no Java, foi feito uso dos seus benefícios que incluem encapsular objetos em uma só interface comum, uniformidade e consistência para iterar sobre coleções. Iterator foi amplamente utilizado sempre que era necessário passar pelos elementos de uma coleção.

Observer, igualmente a Iterator, já vem implementado por padrão no Java, quando chamamos um Listener de Swing, estamos utilizando Observer, seu uso foi necessário para identificar sempre que o estado de algum objeto mudava, no caso, os botões eram clicados, para assim, executar uma ação que normalmente envolvia exibir alguma informação ou utilizar de alguma função implementada na Facade.

4.8.

5. Conclusão

Portanto, apesar de acreditar que os princípios SOLID não foram seguidos com esmero total e com tudo documentado acima, é possível considerar que os principais requisitos do trabalho foram concluídos e proveitosos. Quanto a bugs não foram encontrados durantes os testes manuais e automatizados, porem, acredito que possa existir algum que passou batido. Para possíveis pontos de melhoria, podemos citar a inserção de imagens dos produtos, a capacidade de buscar produtos através de seu nome e não só seus IDs, um sistema

de notificação sobre o estado do seu pedido, formas de alterar preço de produtos no estoque para o dono, realizar promoções e ser capaz de alterar informações de segurança da classe Cliente (login e senha).

Referências

JetBrains (2024). Intellij idea: The capable & ergonomic java ide. https://www.jetbrains.com/idea/. Acessado em: 3 de maio de 2024.

Oracle. Java SE Overview. https://www.oracle.com/br/java/technologies/java-se-glance.html. Acessado em: 6 de maio de 2024.

Refactoring Guru (2024). Catálogo de padrões de projeto. Acessado em: 25 de julho de 2024.