E-commerce

Grupo: Projeto RA2 1

Isabelle de Melo Vizzu, Julia Helena de Almeida e Maria Fernanda Moreira Perez.

**Breve explicação do contexto do projeto:**

O projeto consiste em uma típica loja virtual, contendo classes como: produto, funcionário, cliente, pedido e entre outros. Ações clássicas dentro de um e-commerce são realizadas pelo sistema, como um cliente poder fazer ou cancelar um pedido, um cliente pode adicionar um produto ao seu carrinho ou aos seus favoritos, enquanto um funcionário pode, por exemplo, editar o estoque de um produto. Assim, esse software faz atividades essenciais de uma loja on-line.

**O diagrama de Classes:**

**Diagrama, Esquemático

Descrição gerada automaticamente**

Figure 1-Diagrama de Classe do E-commerce

Diagrama de Classe também anexado na entrega como ‘Diagrama do projeto de Design 2.0’.

**Justificativa da aplicação de cada padrão de projeto usado:**

1. *Singleton*:

O padrão *Singleton* foi preciso nessa aplicação em duas ocasiões específicas, tanto para a implementação do carrinho quanto no favorito. Essas classes usaram o *Singleton* pelo fato de que não havia necessidade de que houvesse mais do que um para cada cliente, aliás não faria sentido se o cliente tivesse mais do que um carrinho ou mais do que um favorito. Por isso, foi optado o uso do padrão *Singleton*,que resolveu essa problemática, garantindo que as classes citadas tivessem apenas uma instância.

Em relação a implementação, as classes que usufruíram do padrão *Singleton* foram o ‘Carrinho’ e o ‘Favorito’. Ambos adicionaram o atributo ‘instancia’, do tipo da classe, isto é, ‘Carrinho’ ou ‘Favorito’, e implantaram o método *getInstance*() que, basicamente, criava uma instância do ‘Carrinho’ ou do ‘Favorito’ caso ainda não tivesse uma, contudo, se já houvesse uma instância, então o método apenas retornava essa instância.

1. *Observer*:

O padrão *Observer* foi implementado para notificar o cliente quando um produto, anteriormente fora de estoque e adicionado aos seus favoritos, estiver disponível novamente. Esse padrão foi escolhido para garantir que o cliente seja informado automaticamente sobre o retorno do item de seu interesse, eliminando a necessidade de acessar a loja repetidamente para verificar a disponibilidade manualmente.

Na implementação, a classe ‘Cliente’ atuou como o observador, implementando a interface *Observer*, cuja função era receber as notificações e enviá-las. A classe ‘Produto’, por sua vez, foi o observado e implementou métodos de adicionar e remover observadores e de notificá-los. Tendo em vista seu objetivo de gerenciar a lista de observadores e enviar notificações quando necessário, como no caso de um produto voltar ao estoque.

1. *Facade:*

O padrão de projeto *Facade* foi implementado com duas classes principais: FacadeCliente e FacadeFuncionario. A adoção desse padrão foi motivada pela necessidade de simplificar a execução de operações aparentemente simples, mas que, na realidade, podem envolver múltiplos métodos e classes. Um exemplo clássico é o processo de cancelamento de uma compra em nosso sistema.

Por mais que cancelar uma compra possa parecer uma tarefa trivial, ela envolve algumas etapas e duas classes, nesse caso ‘Pedido’ e ‘Cliente’. Sem o uso do *Facade*, essas interações seriam espalhadas dentro das classes, o que tornaria o sistema mais suscetível a erros quando uma alteração fosse feita.

Assim, com o uso do padrão *Facade* esse problema foi mitigado. O *Facade* atuou como uma interface simplificada para o subsistema, ocultando sua complexidade interna, tanto para o ‘Cliente’ quanto para o ‘Funcionario’. Consequentemente, um método não precisou conhecer os detalhes do funcionamento de outros métodos ou o passo a passo para realizar uma ação. Logo, delegamos a complexidade dessa operação e de outras operações para o *Facade*, facilitando uma possível manutenção do sistema.

1. *Strategy*:

O padrão *Strategy* foi implementado por meio de uma interface, utilizada pelas classes do pacote financeiro. Esse padrão foi adotado para permitir a troca independente do comportamento (método de pagamento) escolhido.

Na implementação, a classe Pagamento possui um atributo privado que armazena a interface *Strategy,* facilitando a mudança da estratégia de pagamento, caso necessário. A classe Boleto implementa essa interface diretamente, enquanto a classe Cartao, sendo abstrata, delega a implementação do método da interface para suas classes filhas.

**Justificativa da não aplicação dos padrões *Adapter* e *Bridge:***

1. *Adapter:*

O motivo para que o padrão *Adapter* não ter sido implementado em nosso projeto foi pôr o código não conter nenhuma API e nenhum objeto com uma interface incompatível que não colaborasse com os outros objetos, assim esse padrão não teria utilidade nesse projeto.

1. *Bridge:*

Este padrão não foi utilizado pois o padrão *Bridge* é utilizado para dividir uma classe grande ou um conjunto de classes fortemente ligadas em duas (ou mais) hierarquias separadas, sendo essas implementação e abstração, isso para impedir a exagerada quantidade de classes que precisariam ser criadas ao usar duas hierarquias. Mas esse problema em si não é encontrado em nosso projeto, já que não há duas hierarquias de dimensões diferentes e além de nem ser um projeto extenso a ponto de ser necessário o uso do padrão *Bridge*.

**Link do Código-fonte do projeto:** [**Isavizzu/Design-Ecomerce (github.com)**](https://github.com/Isavizzu/Design-Ecomerce)

**Reflexão sobre as maiores dificuldades neste projeto:**

Algumas das maiores dificuldades enfrentadas durante o projeto estariam em torno de: o que iria ou não ser uma classe, a definição das associações delas umas com as outras, além da dificuldade em estabelecer cada pacote e definir quais classes ficariam nele, já que em alguns momentos tivemos certas dúvidas se uma classe se encaixaria melhor no pacote ‘Venda’ ou em ‘Financeiro’, por exemplo.

Outra dificuldade encontrada está relacionada diretamente aos padrões, visto que foi complexo entender o ‘como’ implementar certos tipos de padrões e compreender quais ajustes seriam necessários.

Por fim, uma última dificuldade foi reconhecer os benefícios da aplicabilidade dos padrões de projeto, para equivaler o ‘trabalho das mudanças’ com as vantagens fornecidas pelos padrões. Ou seja, literalmente entender o porquê do uso dos padrões e o porquê é melhor usá-los do que não os usar.