UNIVERSIDADE FEDERAL DE MINAS GERAIS

Ágatha Soares

Eduardo Fiuza

Lucas Costa

Vitor Mafra

Trabalho final para a disciplina de Programação e Desenvolvimento de *Software* II: e-commerce

Professores: Flavio Figueiredo

Júlio César

BELO HORIZONTE

2019

INTRODUÇÃO

Esse trabalho é a consolidação de todo o aprendizado na disciplina de Programação e Desenvolvimento de Software II, ao longo do primeiro semestre de 2019. Conceitos inéditos como modularização, orientação a objetos, testes de unidade, makefile, desenvolvimento em grupo e Github foram implementados ao longo do trabalho de modo a aplicar tudo em um *software* de médio porte que junta diversas funções em um único ambiente. O resultado disso é um programa escrito em C++, funcional, robusto e que engloba muito do que aprendemos ao longo da disciplina. Todo o código do trabalho pode ser encontrado e baixado em https://www.github.com/pds2/20191-team-21

MOTIVAÇÃO

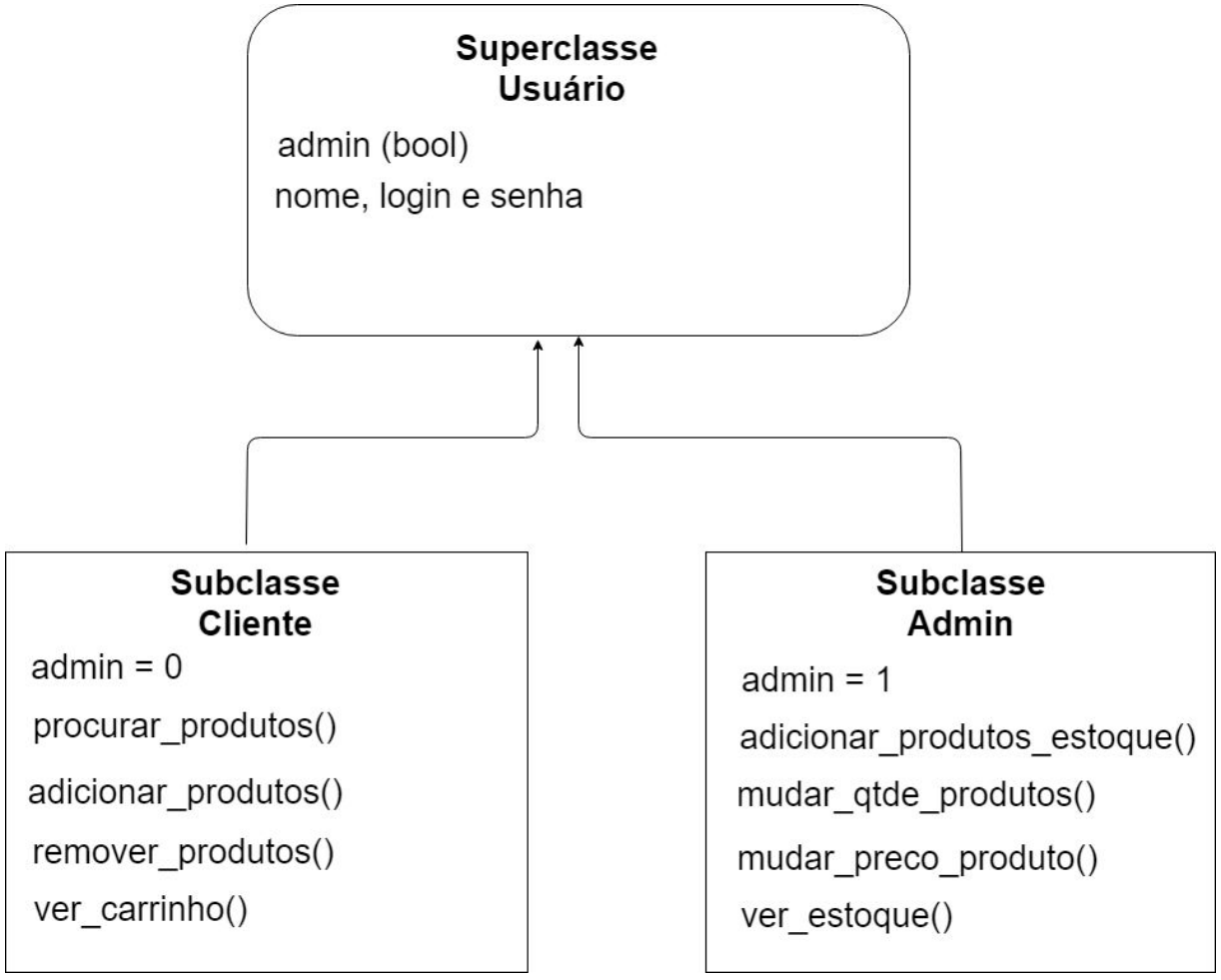
É evidente o aumento expressivo do número de comércios eletrônicos nos últimos anos. O avanço tecnológico possibilitou a simulação dessa dinâmica no mundo digital e conectou as pessoas a esse ambiente. Foi justamente a importância do comércio eletrônico na vida das pessoas e na economia moderna que nos motivou a buscar entender melhor o que acontece por trás de todo esse processo. Dessa forma, tivemos que pensar como administradores de lojas eletrônicas e como clientes para que pudéssemos criar um programa, ainda que simples, que pudesse englobar toda essa experiência dos usuários (comprador e vendedor).

DECISÕES DE IMPLEMENTAÇÃO

Para que a dinâmica de compra/venda pudesse ocorrer precisaríamos de dois tipos de agentes nesse processo, isto é, quem pudesse vender e quem pudesse comprar. Vale observar aqui que, apesar desses dois atores utilizarem do mesmo instrumento de manuseio do programa, as intenções de ambos são completamente diferentes. Portanto, ainda que os dois sejam usuários do programa, cada um deles teria comportamentos diferentes e, portanto, níveis de acesso diferentes, por exemplo. Assim, usando os conceitos de orientação a objetos, modelamos e implementamos o seguinte esquema de usuários para que o papel de cada um no processo compra/venda fosse atendido:

- Classe abstrata Usuário, que tem os atributos de “nome”, “login” e “senha”, comuns a todos os tipos de usuários, que não é instanciada durante o programa, já que isso representaria uma quebra de contrato.
- Classe derivada Administrador, que herda os atributos e métodos de “Usuário”, mas que tem seu comportamento específico sendo ali definido (o qual será melhor explorado abaixo)
- Classe derivada Cliente, quem também herda as informações e ações determinadas em usuário mas que, assim como o “Administrador”, também tem as suas peculiaridades respeitadas em outros atributos e métodos específicos (os quais também serão detalhados abaixo)

Segue um diagrama que ilustra a herança descrita acima:

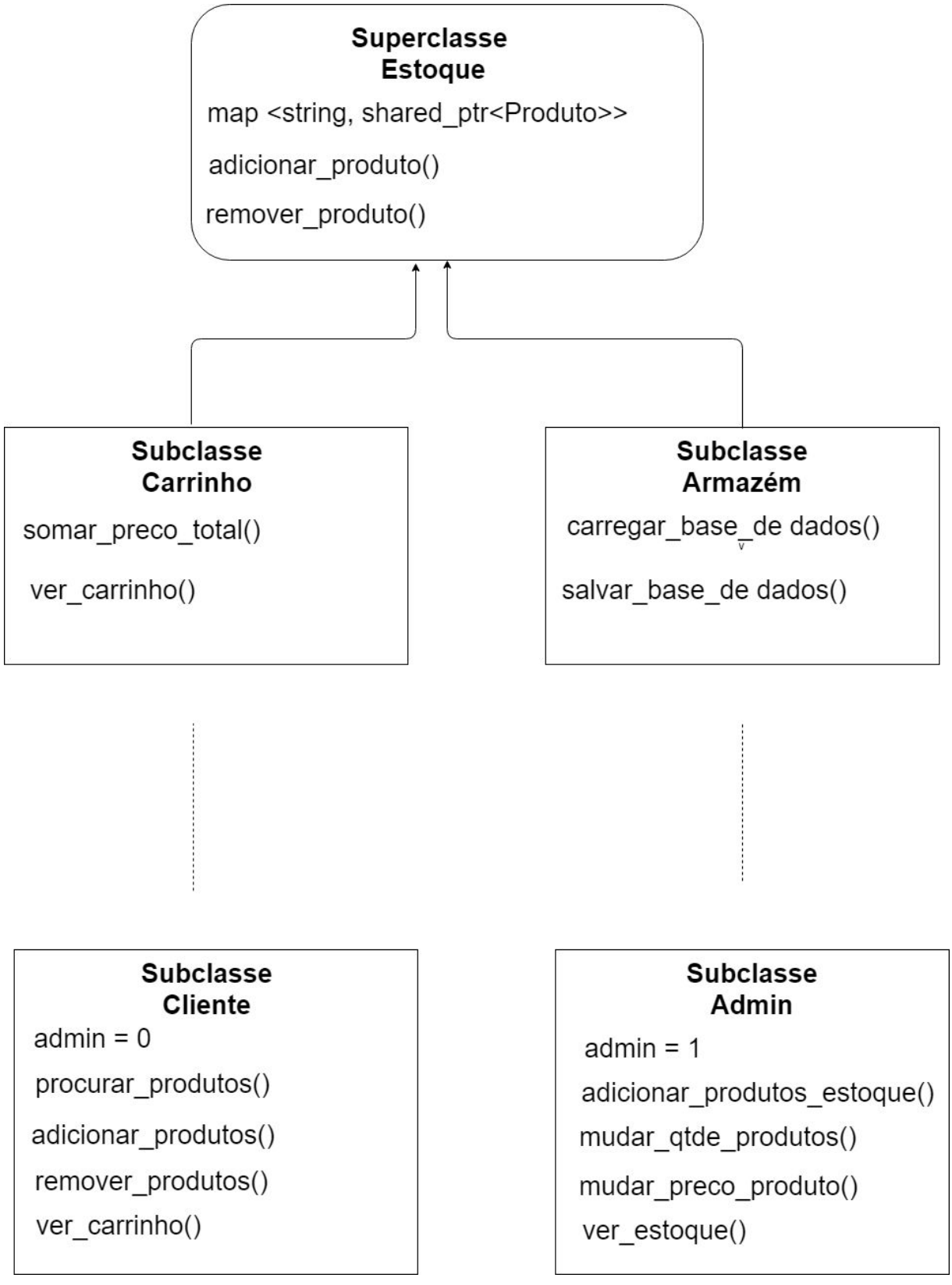


Outro ponto que liga, ainda que indiretamente, os clientes e os administradores da loja são os produtos. Nossa loja pretendia simular a venda de produtos de um comércio no ramo da moda e, por isso, inicialmente criamos uma superclasse produtos que englobava todos eles. Sendo assim, criamos (também anteriormente) subclasses que herdavam características e comportamentos da classe-mãe “produtos”, mas que possuíam suas características específicas de cada tipo de produto. Essa foi uma ação que descobrimos, na entrevista, ser “overkill” e que forçava uma herança em uma relação em que idealmente não possuía nenhuma. Dessa forma, removemos essa herança na nova versão do nosso programa e a classe “Produtos” segue sendo uma das mais importantes e centrais de toda a dinâmica, mas sem nenhuma herança ligada à ela.

Ainda sobre heranças e a nova versão do software, implementamos uma superclasse Estoque que tem subclasses Armazém e Carrinho. Essa foi uma herança que não havíamos pensado anteriormente, mas que depois da implementação do primeiro versionamento do código passou a fazer todo o sentido, afinal, o carrinho do cliente era um “estoque” próprio e interessante ao usuário que comprava os produtos, enquanto o Armazém passou a representar todos os produtos presentes no estoque da loja. Além, é claro dessa visão mais teórica dessa modelagem, ela se aplica perfeitamente nos conceitos de orientação a objetos.

O funcionamento do Estoque é um map que tem uma chave em formato string e que recebe um ponteiro compartilhado da classe Produtos e mapeia isso nessa nova classe - afinal, os produtos não têm a responsabilidade de se organizarem por si só. Assim, as subclasses seguem esse contrato e herdam esse comportamento de mapeamento da classe Produtos, mas com suas peculiaridades. Somente o Administrador tem a capacidade de modificar o estoque, por exemplo, portanto somente ele pode alterar a quantidade de produtos dentro da classe Armazém. E assim segue essa dinâmica no Estoque, tanto para a modificação do Armazém, através do administrador, quanto para a personalização do Carrinho por meio das ações do cliente.

Novamente, decidimos demonstrar essa relação através de outro diagrama de herança, que segue abaixo (que mostra também, parte da relação apontada anteriormente):


Superclasse
Estoque
map <string, shared_ptr<Produto>>
adicionar_produto()
remover_produto()
Subclasse
Carrinho
somar_preco_total()
ver_carrinho()
Subclasse
Armazém
carregar_base_de dados()
salvar_base_de dados()
Subclasse
Cliente
admin = 0
procurar_produtos()
adicionar_produtos()
remover_produtos()
ver_carrinho()
Subclasse
Admin
admin = 1
adicionar_produtos_estoque()
mudar_qtde_produtos()
mudar_preco_produto()
ver_estoque()

Por fim, para um programa funcional e robusto, todos esses dados deveriam ser salvos em uma base de dados que se mantivesse mesmo depois do encerramento do programa. Sendo assim, decidimos implementar uma simulação de banco de dados usando arquivos do tipo txt. Esses arquivos se organizam pelo tipo de dados que eles armazenam, isto é, os dados de login e senha dos usuários são armazenados em arquivos diferentes do arquivo em que são salvas as informações do produto. Além de uma divisão conceitual, essa decisão nos ajudou também na modificação e leitura de cada um desses arquivos, fazendo com que o programa saiba exatamente quais catálogos ele deve acessar, o que traz, ainda, mais robustez ao código.

Classes, atributos e métodos

Superclasse: **user**

- Atributos
    - admin (variável do tipo boolean que assume o valor 1 na subclasse user_admin e 0 na subclasse user_client)
    - name (variável do tipo string que armazena o nome do usuário)
    - login (variável do tipo string que armazena o login do usuário)
    - password (variável do tipo string que armazena a senha do usuário)
- Métodos
    - get_name
    - get_login
    - get_password

Subclasses:

user_admin

- Métodos
    - add_product_warehouse (método sem retorno que adiciona um produto ao estoque)

- change_product_quantity (método sem retorno que muda a quantidade de um produto no estoque)
- change_product_price (método sem retorno que muda o preço de um produto presente no estoque)
- remove_product (método sem retorno que remove um produto do estoque)
- show_storage (método sem retorno que mostra os produtos cadastrados presentes no estoque)

user_client

- Atributos
  - client_cart (objeto do tipo carrinho)
- Métodos
  - search_product (método sem retorno que procura um produto no estoque)
  - see_cart (método sem retorno que exibe os produtos presentes no carrinho do cliente)
  - add_product (método sem retorno que adiciona produtos ao carrinho)
  - remove_product (método sem retorno que remove produtos do carrinho)

Classe: **product**

- Atributos
  - price (variável do tipo double que armazena o preço do produto)
  - quantity (variável do tipo unsigned int que armazena valores positivos referentes à quantidade do produto)
  - features (variável do tipo map<string> que armazena características do produto)
- Métodos
  - set_feature
  - update_quantity

- set_quantity
- set_price
- get_feature
- get_price
- get_quantity
- get_id

Superclasse: **storage**

- Atributos
  - storage (variável do tipo map<string, shared_ptr<Product>> que recebe um ponteiro compartilhado que aponta para um produto e mapeia isso como um estoque)
- Métodos
  - add_product
  - remove_product
  - get_product

Subclasses:

storage_warehouse

- Métodos
  - load_db
  - save_db

storage_basket

- Métodos
  - get_total_price
  - view_basket