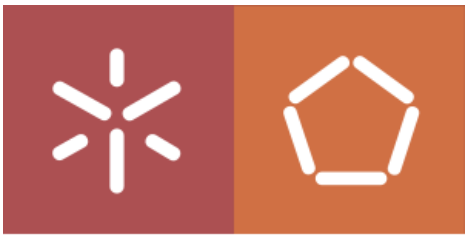

Universidade do Minho
Escola de Engenharia

Programação Orientada aos Objetos

Trabalho Prático

Grupo 96

2022/2023

A91671 – João Manuel Novais da Silva

Conteúdo

1.Introdução

Este projeto consistiu no desenvolvimento de uma aplicação que conseguisse simular um sistema de *Marketplace* Vintage utilizando a linguagem de programação Java de forma a pôr em prática as estratégias e conhecimentos adquiridos ao longo do semestre na UC de Programação Orientada aos Objetos.

Este sistema deve permitir a compra e venda de artigos novos ou usados de vários tipos. Os utilizadores da aplicação podem escolher criar uma conta como Comprador, Vendedor ou Transportadora. Os vendedores podem adicionar novos artigos para venda. Os compradores podem optar pela compra dos artigos disponíveis. E as transportadoras são associadas a cada artigo postado pelo vendedor e podem ter fórmulas customizadas para o custo de expedição.

Os artigos podem ser sapatilhas, malas e t-shirts que são caracterizadas por um conjunto de propriedade que vão ser discutidas posteriormente a esta introdução. As compras destes artigos são organizadas em encomendas que podem ser posteriormente finalizadas, expedidas e entregues ao comprador correspondente. Estas também podem ser devolvidas num prazo de tempo pré-definido.

O sistema toma controlo do stock dos artigos disponíveis, assim como as encomendas efetuadas pelos compradores, vendas feitas pelo vendedores e dados de todos os utilizadores, nomeadamente os logins.

O sistema também foi implementado de maneira a ser possível realizar a simulação necessária para fazer saltos no tempo para o futuro. Esta implementação será abordada com mais detalhe seguidamente.

2.Classes

2.1Main

Classe responsável com o arranque do programa, apenas tem um método *main* que declara uma nova instância de *View* e chama o método *run* desta.

2.2 View

A classe *View* está responsável por permitir a interação entre o utilizador e o sistema. Neste caso esta tem um atributo:

- Private Controller controller

O seu método principal (public void run()) é usado chama os métodos da classe Menu e os inputs do utilizador com os métodos do *Controller*.

Com o intuito de manter uma melhor legibilidade desta classe várias excertos do método run() foram substituídos por chamadas a métodos, disto é o caso do método public void criarComprador(), por exemplo.

2.3 Controller

A classe Controller teve a tarefa de gerir o fluxo da aplicação, servindo de porta de conexão entre os inputs do utilizador e a arquitetura do sistema. A partir deste é possível de interagir com todas as classes que formam a estrutura dos objetos necessário do sistema. Por isso esta classe necessita de toda a informação disponível ao sistema num dado momento, daí os seus atributos serem:

- Private Dados dados;
- Private LoginParser login.

Com isto o controller tem acesso à lista de logins do sistema assim como o registo de todos os objetos fundamentais para o funcionamento do sistema. Como é de esperar esta classe possuí todas os métodos que são chamados pela *View*, por exemplo,

public void adicionaComprador(String email, String nome, String morada, int nif, String password).

2.4 Dados

A classe Dados existe para ser mais fácil a manutenção dos dados necessários para o sistema, esta tem vários atributos:

- Private HashMap<String, Transportadora> transportadoras → HashMap que guarda todas as transportadoras (premium e normais);
- Private HashMap<String, Utilizadores> utilizadores →HashMap que guarda os tipos de utilizadores (compradores e vendedores);
- Private HashMap<String, Artigos> artigos → HashMap que guarda todos os artigos do sistema (sapatilhas, malas e t-shirst);
- Private HashMap<String, Encomendas> encomendas → HashMap que guarda todas as encomendas geradas pelos compradores;
- Private LocalDateTime dateTimeDoSistema → Variável de tempo utilizada pelo sistema para fazer saltos no tempo;
- Private HashMap<Integer, String> novasFormulas → HashMap que guarda as novas fórmulas introduzidas pelas transportadoras antes de haver um salto no tempo;
- Private HashMap<Integer, Float> novasMargens → HashMap que guarda as novas margens de lucro introduzidas pelas transportadoras antes de haver um salto no tempo;
- Private List<Artigos> vendidos → Lista de artigos vendidos antes de haver um salto no tempo;
- Private List<Encomenda> encomendasFinalizadas → Lista de encomendas pagas antes de haver um salto no tempo;
- HashMap<Comprador, String> faturas → Lista de faturas criadas após a compra duma encomenda que esperam até a própria encomenda seja entregue.

2.5 Menus

Nesta classe são guardados os métodos que são chamados pela *View* para escrever no terminal os menus de utilização do sistema assim como devolver a escolha dos utilizadores.

2.6 Parser

A classe *Parser* ficou responsável por escrever e ler os dados do sistema num ficheiro. Para tal é implementado o *Serializable* nas classes cujos objetos vão ser guardados. Os métodos que executam este processo são:

- Public static void escreveFicheiro(String filename, Dados dados);
- Public satic Dados lerFicheiro(String filename).

```java
public static void escreveFicheiro(String filename, Dados dados) throws IOException {
    try {
        FileOutputStream fileOut = new FileOutputStream(filename);
        ObjectOutputStream out = new ObjectOutputStream(fileOut);
        out.writeObject(dados);
        out.close();
        fileOut.close();
        System.out.println("Os objetos foram gravados para: " + filename);

    } catch (IOException e){
        e.printStackTrace();
    }

}

3 usages  JoaoManuelNovaisdaSIlva
public static Dados lerFicheiro(String filename) throws IOException, ClassNotFoundException {
    Dados dados = null;


    try {
        FileInputStream fineIn = new FileInputStream(filename);
        ObjectInputStream in = new ObjectInputStream(fineIn);
        dados = (Dados) in.readObject();
        in.close();
        fineIn.close();

    } catch (IOException | ClassNotFoundException e){
        e.printStackTrace();
    }
    return dados;
}
```

Figura 1 - Parser feito para ler e escrever os objetos num ficheiro .ser

2.7 LoginParser

Esta classe funciona de maneira parecida com a classe *Parser*, mas está encarregue de ler e escrever os dados de login dos utilizadores. Para poder ser feito as verificações necessárias para o login as informações dos ficheiros são guardadas nos seguintes atributos:

- Private HashMap<String, List<String>> contasCompradores → Lista de compradores onde a key é o email do comprador e os values uma lista com a password e o id;
- Private HashMap<String, List<String>> contasVendedores → Lista de vendedores onde a key é o email do comprador e os values uma lista com a password e o id;
- Private HashMap<String, List<String>> contasTransportadoras → Lista de transportadoras onde a key é o email do comprador e os values uma lista com a password e o id;

Além dos métodos esperados de ler e escrever as informações dos utilizadores num ficheiro de texto, também existe um método de validação de logins (public int validaLogin(TipoConta tipo, String email, String password) que devolve o id da conta em questão ou -1 caso as credenciais estejam erradas). Para além disto também é permitido, na aplicação, fazer alterações á conta dos utilizadores daí ser necessário mudar os email no ficheiro de texto e a existência dos respetivos métodos: public void alterarEmail(TipoConta tipo, String currentEmail, String newEmail, int id)

2.8 Utilizador

A classe Utilizador serve de base para as contas de vendedor e comprador, por isso como foi indicado no enunciado do projeto os utilizadores têm os seguintes atributos:

- Private int id → Id único do utilizador;

- Private String email → Email também usado para login;
- Private String nome → Nome do utilizador;
- Private String morada → Morada introduzida pelo utilizador;
- Private int nif → Número de identificação fiscal;

2.8.1 Comprador

A classe Comprador estendesse de Utilizador e apensa adiciona um novo atributo:

- Private ArrayList<Artigos> comprados → Lista de artigos comprados pelo comprador;

Como tal também é necessário de métodos que adicionem e removam artigos deste atributo daí a existência dos métodos public void adicionarCompradores(ArrayList<Artigos> compras) e public void removerComprados(ArrayList<Artigos> devolvidos).

2.8.2 Vendedor

Com a classe Vendedor são adicionados os atributos seguintes:

- Private ArrayList<Artigos> emVenda → Lista de produtos que o vendedor tem em venda;
- Private ArrayList<Artigos> vendidos → Lista de produtos vendidos pelo vendedor;
- Private float valorFaturado → Valor total faturado ao longo do uso do sistema, é usado para fazer estatística do sistema.

Além dos métodos para adicionar artigos ás lista também é definido o método public int compareTo(Vendedor other) para ser possível ordenar uma lista de

Compradores baseando se no valor faturado para podermos saber que foi o vendedor que mais faturou desde sempre.

2.9 Artigos

Esta classe é descrita pelos seguintes atributos:

- Private boolean isUsed → Se um artigo é usado ou não;
- Private String desciption → Descrição do produto;
- Private String barCode → Código de barras do produto, único para cada artigo;
- Private float price → Preço base do produto;
- Private float discountPrice → Preço do desconto a aplicar aos artigos;
- Private Estado usedState → Use do enum Estado que pode ser mau, médio ou bom;
- Private Transportadora transportadoraAssociada → Transportadora encarregue a entregar o produto;
- Pirvate int stock → Qunatidade disponível do artigo.

De maneira a certificar que cada artigo tem um código de barras diferente este é conseguido a partir do HashCode do objeto.

2.10 Malas

Estendida da classe Artigo esta adiciona a informação em relação à:

- Private Dimenssao dimension → Uso do enum Dimenssao que pode ser pequeno, medio ou grande;
- Private TexturaMala texture → Uso do enum TexturaMala que pode ser pele, tecido ou sintético;

- Private int collectionYear → Ano de coleção da mala.

Como é de esperar dum tipo de artigo temos um método que atualiza o valor do desconto dependendo do ano em que o sistema se encontra:

- Public void atualizaPrecoDesconto(int currentYear).

```
3 usages  new *
public void atualizaPrecoDesconto(int currentYear){
    if(this.getIsUsed()){
        float newDiscount = (this.getDicountPrice() + (this.getDicountPrice()*0.05f*(currentYear-this.collectionYear)));
        if(newDiscount > this.getPrice()) System.out.println("As sapatilhas não podem ter mais disconto!");
        else this.setDicountPrice(newDiscount);
    }else System.out.println("O artigo não é usado logo não aumenta o desconto");
}
```

Figura 2 - Definição da atualizaPrecoDesconto das malas

2.10.2 MalasPremium

Também foram implementadas as versões premium dos artigos, neste caso esta classe estende Malas e adiciona a informação relacionada com:

- Private String author → Autor da mala.

A atualização do preço é diferente para produtos premium, estes valorizam ao longo do tempo daí a definição do método:

- Public void atualizaPrecoMalaPremium(int currentYear)

```java
public void atualizaPrecoMalasPremium(int currentYear){
    float newPrice = this.getPrice();
    if(this.getDimensions() == Dimenssao.Pequeno){
        newPrice = (this.getPrice() + (this.getPrice()*0.05f*(currentYear- this.getCollectionYear())));
    }else if (this.getDimensions() == Dimenssao.Medio){
        newPrice = (this.getPrice() + (this.getPrice()*0.10f*(currentYear- this.getCollectionYear())));
    }else if (this.getDimensions() == Dimenssao.Grande){
        newPrice = (this.getPrice() + (this.getPrice()*0.20f*(currentYear- this.getCollectionYear())));
    }
    this.setPrice(newPrice);
}
```

Figura 3 - Definição da atualizaPrecoMalasPremium

2.11 Sapatilhas

Como mencionado no enunciado as sapatilhas possuem:

- Private int shoeSize → Tamanho da sapatilha;
- Private boolean hasLaces → Existência de atacadores;
- Private String color → Cor das sapatilhas;
- Private int collectionYear → Ano da coleção da sapatilha;

A atualização do preço do desconto é idêntica ao das mala é adicionado 5% do valor a cada ano que passa, fazendo com que o preço geral do produto desça 5% todos os anos.

```java
public void atualizaPrecoDesconto(int currentYear){
    if(this.getIsUsed()){
        float newDiscount = (this.getDicountPrice() + (this.getDicountPrice()*0.05f*(currentYear-this.collectionYear)));
        if(newDiscount > this.getPrice()) System.out.println("As sapatilhas não podem ter mais disconto!");
        else this.setDicountPrice(newDiscount);
    }else System.out.println("O artigo não é usado logo não aumenta o desconto");
}
```

Figura 4 - Definição da atualizaPrecoDesconto das sapatilhas

2.11.1 SapatilhasPremium

Como mencionado anteriormente foram implementadas as versões premium das sapatilhas e malas, e como a anterior a SapatilhasPremium têm:

- Private String authors → Autor da sapatilha premium.

Como a Mala Premium também é considerado que o valor da sapatilha premium valoriza ao longo do tempo, neste caso o preço valoriza 10% todos os anos.

```
public void atualizarPrecoSapatilhasPremium(int currentYear){
    float newPrice = (this.getPrice() + (this.getPrice()*0.10f*(currentYear- this.getCollectionYear())));
    this.setPrice(newPrice);
}
```

2.12 TShirt

O último tipo de artigo é uma t-shirt e os seus atributoes são:

- Private TamanhoTShirt size → Uso do enum TamanhoTShirt que pode ser S, M, L ou XL;
- Private PadraoTShirt pattern →Uso do enum PadraoTShrit que pode ser liso, riscas ou palmeiras.

2.13 Encomenda

A encomenda é criada quando um comprador compra um novo produto e este ainda não tem encomendas pendentes, os atributos duma encomenda são:

- Private int id → Id da encomenda;
- Private Comprador comprador → Comprador a que pertence a encomenda;
- Private ArrayList<Artigos> artigos → Lista dos artigos associados a esta encomenda;
- Private Dimenssao dimensão → Uso do enum Dimenssao que pode ser pequeno, medio ou grande;
- Private float precoProdutos → Preço final dos produtos;
- Private float custoExpedicao → Preço de expedição calculado a partir das fórmulas das transportadoras;
- Private EstadoEncomenda estado → Uso do enum EstadoEncomenda que pode ser pendente, finalizado, expedido ou entregue;
- Private LocalDateTime dataCriação → Data e hora da criação da encomenda;

- Private LocalDataTime dataEntrega → Data e hora da entrega da encomenda;
- Private LocalDateTime dataAlteração → Data e hora da última alteração ao estado da encomenda.

Além disto a classe também possuí métodos para adicionar artigos à lista de artigos, métodos para pagar e devolver a encomenda e, finalmente, métodos para calcular o custo de expedição da encomenda.

```java
public void pagarEncomenda(LocalDateTime tempo){
    if(this.estado == EstadoEncomenda.Pendente) {
        this.estado = EstadoEncomenda.Finalizada;
        this.ultimaAlteracao = tempo;
        this.comprador.adicionarComprados(this.artigos);
        System.out.println("Valor a pagar: " + this.custosExpedicao);
        System.out.println("A sua encomenda foi paga com sucesso!");
    }else System.out.println("A sua encomenda já se encontra no estado: " + this.estado);
}

1 usage  JoaoManuelNovaisdaSilva *
public void devolucao(LocalDateTime tempo){
    if(this.estado == EstadoEncomenda.Entregue) {
        Duration duration = Duration.between(this.dataEntrega, tempo);
        long horasPassadas = duration.toHours();
        if (horasPassadas >= 48) {
            System.out.println("Desculpe mas já passaram 48h desde a sua entrega, não pode devolver a encomenda!");
            return;
        } else {
            System.out.println("A sua encomenda será devolvida e o valor da venda será devolvido!");
            System.out.println("Valor devolvido: " + this.custosExpedicao);
            this.comprador.removerComprados(this.artigos);
        }
    } else System.out.println("Não é possível devolver a encomenda porque esta encontra-se no estado: " + this.estado);
}
```

Figura 5 - Definição de pagarEncomenda e devolucao

```java
public float calculaPrecoExpedicao(ArrayList<Artigos> a, Dimenssao d){
    HashMap<Transportadora, ArrayList<Artigos>> dividida = divideEmHashMap(a);
    HashMap<Transportadora, Float> preco = calculaPrecoArtigosDeTransportadoraEncomenda(dividida);
    float precoExp = calculaCustosDeExpedicaoTransportadoraEncomenda(preco);

    if(d == Dimenssao.Pequeno) precoExp += Transportadora.valorExpedicaoPequena;
    else if(d == Dimenssao.Medio) precoExp += Transportadora.valorExpedicaoMedia;
    else if(d == Dimenssao.Grande) precoExp += Transportadora.getValorExpedicaoGrande;

    return precoExp;
}
```

Figura 6 - Definição de calculaPrecoExpedicao

2.14 Transportadora

A transportadora é responsável por transportar a encomenda e por isso deverá ter fórmulas para cálculo do custo de expedição:

- Static float valorExpedicaoPequena = 2.00f → valor de expedição de encomendas pequenas;
- Static float valorExpedicaoMedio = 4.20f → valor de expedição de encomendas de tamanho médio;
- Static float valorExpedicaoGrande = 6.00f → valor de expedição de encomendas de tamanho grande;
- Static float imposto = 0.23f → Valor do imposto universal para todas as transportadoras;
- Private int id → Id único da transportadora;
- Private String name → Nome da transportadora;
- Private int nif → Número de identificação fiscal da transportadora;
- Private String email → Email usando também para login da transportadora;
- Private ArrayLisr<Artigos> artigosAssociados → Artigos que podem ser entregues por esta transportadora;
- Private float margemLucro → Margem de lucro introduzida pela transportadora valor entre 0 e 1 normalemente;
- Private TipoFormula tipoFormula → Uso do enum TipoFormula que pode ser normal ou customized;
- Private String formula → Fórmula de calculo do preço de expedição;
- Private float valorFaturado → Valor total faturado pela transportadora.

Além dos métodos habituais foram definidos métodos que reconhecem uma fórmula introduzida pela transportadora e, utilizando a biblioteca MVEL, calcula o valor dessa fórmula.

```java
public float calculaFormula(float preco){
    if(this.tipoFormula == TipoFormula.Default){
        return calculaFormulaDefualt(preco);
    } else{
        return  calculaFormulaCusomized(this.formula, preco);
    }
}

1 usage  JoaoManuelNovaisdaSIlva
public float calculaFormulaDefualt(float preco){
    return (preco*(margemLucro)*(1+imposto));
}

1 usage  JoaoManuelNovaisdaSIlva
public static float evaluateFormula(String formula, Map<String, Float> variables) {
    Object result = MVEL.eval(formula, variables);
    return Float.parseFloat(result.toString());
}

1 usage  JoaoManuelNovaisdaSIlva *
public float calculaFormulaCusomized(String line, float preco){
    Map<String, Float> variables = new HashMap<>();


    variables.put("preco", preco);
    variables.put("imposto", imposto);
    variables.put("margem", margemLucro);
    return evaluateFormula(line, variables);
```

Figura 7 - Cálculo da fórmula para o preço de expedição

2.14.1 TransportadoraPremium

Foram implementadas transportadora premium que apenas elas podem ser usadas para distribuir os produtos premium, e além disso têm fórmulas de cálculo do preço de expedição diferentes

- Static float impostoPremium = 5.00f → Imposto base de transportadoras premium;
- Private String formulaPremium → Fórmula de calculo de transportadoras premium.

Como a classe anterior, esta também tem um mecanismo para calcular o custo de expedição.

```java
public float calculaFormulaPremium(float preco){
    if(super.getTipoFormula() == TipoFormula.Default){
        return calculaFormulaDefualtPremium(preco);
    } else{
        return  calculaFormulaCusomizedPremium(this.formulaPremium, preco);
    }
}

1 usage  JoaoManuelNovaisdaSIlva
public float calculaFormulaDefualtPremium(float preco){
    return (preco*(super.getMargemLucro())*(1+imposto))+impostoPremium;
}

1 usage  JoaoManuelNovaisdaSIlva
public static float evaluateFormulaPremium(String formula, Map<String, Float> variables) {
    Object result = MVEL.eval(formula, variables);
    return Float.parseFloat(result.toString());
}

1 usage  JoaoManuelNovaisdaSIlva *
public float calculaFormulaCusomizedPremium(String line, float preco){
    Map<String, Float> variables = new HashMap<>();

    variables.put("preco", preco);
    variables.put("imposto", imposto);
    variables.put("margem", getMargemLucro());
    variables.put("impostoPremium", impostoPremium);
    return evaluateFormulaPremium(line, variables);

}
```

Figura 8 - Cálculo da fórmula para o preço de expedição

3. Estrutura do Projeto

O projeto segue a estrutura *Model View Controller* (MVC), estando por isso organizado em três camadas:

- A camada dos dados (o modelo) é composta pelas classes Artigo, Sapatilha, SapatilhaPremium, Mala, MalaPremium, TShirts, Encomenda, Transportadora, TransportadoraPremium, Comprador e Vendedor.
- A camada de interação com o utilizador é composta pela classe View e Menus.
- A camada de controlo do fluxo do programa é composta pela classe Controller.

O projeto tem em mente a topologia de encapsulamento.

4. Exception

4.1 InputInvalidoException

Esta exception foi criada para devolver uma exceção customizada no caso do utilizador selecionar uma opção inválida nos menus.

5. Diagrama de Classes

6. Conclusão

No geral, apesar da realização deste projeto ter sido feita por apena um elemento este está convencido que consegui realizar os objetivos base pedidos pela equipa docente respeitando as regras e recomendações feitas pelos professores. A realização desta aplicação apresentou-se como um meio robusto de aprendizagem dos conceitos de Programação Orientada aos Objetos e também ao desenvolvimento de aplicações úteis.