Universidade do Minho

Departamento de Informática

Relatório do Trabalho Prático

Programação Orientada aos Objetos

2022/2023

Grupo 98

Ricardo Miguel Queirós de Jesus (a100066)

Rui Pedro Fernandes Madeira Pinto (a100659)

Hugo Arantes Dias (a100758)

Conteúdo

1 Introdução

No âmbito da cadeira de Programação Orientada aos Objetos, o grupo desenvolveu, em Java, um sistema de *marketplace* denominado de **Vintage**, com o objetivo de permitir a compra e venda de artigos novos ou usados de diversos tipos.

De forma a ter uma interação o mais intuitiva possível, o nosso grupo implementou uma interface minimalista via terminal onde é possível criar e efetuar o login de utilizadores, efetuar encomendas, criar transportadoras e também fazer login como transportadora e alterar os valores das mesmas.

O presente relatório está estruturado com a descrição das classes, técnicas utilizadas para resolução de problemas propostos, modo de construção dos menus e, no final, encontra-se em anexo, encontra-se o diagrama de classes, com todas as classes, varáveis de instância e métodos das mesmas, tal como as relações entre as diferentes classes presentes no nosso projeto.

2 Classe Artigo e respetivas subclasses

No nosso projeto, existem 3 tipos de artigos, as **Malas**, as **Sapatilhas** e as **T-Shirt's**. O grupo optou por criar o superclasse **Artigo** que contém o seguinte:

Variavél	Tipo
codBarras	String
stock	int
transportadora	String
precoBase	double
marca	String
descricao	String
estado	String
nome	String
dataLancamento	LocalDate
numDonos	int
avalestado	int

A classe artigo é abstrata, contendo como método abstrato o clone(), contém também os construtores da classe, métodos de clone, equals e toString, contém também um método que dado um desconto passado como argumento, altera o valor do preço atual do artigo. Nesta classe optamos, por colocar uma **String** com o nome da transportadora associado ao invés de colocar uma **Transportadora** diretamente.

2.1 Subclasses Mala, Sapatilha e TShirt

As subclasses **Mala**, **Sapatilha** e **TShirt** contém apenas variáveis adicionais mais especificas de cada um, como se pode ver abaixo:

Sapatilha

Variável	**Tipo**
tamanho	int
temAtacadores	boolean
cor	String

Mala

Variável	**Tipo**
tamanho	int
material	String
anoColecao	int

TShirt

Variável	**Tipo**
tam	enum Tamanho
padrao	enum Padrao

Todas estas subclasses contém variáveis de instância específicas de cada, contudo, as váriaveis da classe **TShirt** são representadas por dois *enum*, a enum **Tamanho**, que contém os tamanhos disponíveis para as T-Shirts e a enum **Padrao** que contém os 3 padrões disponíveis, que no nosso caso são apenas os padrões liso, riscas e palmeiras.

3 Classes Encomenda e Transportadora

No nosso projeto, implementamos, tal como nos foi pedido no enunciado as classes **Encomenda** e **Transportadora**, que tratam das encomendas feitas e das transportadoras presentes no sistema de *marketplace* implementado.

3.1 Encomenda

A classe **Encomenda** contém o seguinte:

Variável	Tipo
codSistema	String
codSistemaUtilizador	String
artigos	Map<String, Artigo>
tamanho	String
precoFinal	double
status	StatusEncomenda
data	LocalDate
vintageProfit	double

A classe **Encomenda** contém as variáveis apresentadas na tabela ao lado. Nesta classe são guardados os artigos num *Map*, que utiliza com *key* o código de barras associado a cada artigo da encomenda. Esta classe contém os contrutores de classe, métodos gerais como o *clone()*, *equals* e *toString()* e também os respetivos *getters* e *setters*.

Quanto aos outros métodos presentes nesta classe, temos:

- **insereUmArtigo(Artigo artigo)**: método que recebe um artigo e adiciona à lista de artigos da encomenda, usando o *deep clone*.
- **custoProdutos()**: método que em numa encomenda, percorre a lista de artigos e soma o preço base dos mesmos e retorna-o.
- **custoTotalEncomenda()**: método que numa encomenda, percorre a lista de artigos e soma o preço total dos artigos e retorna esse valor.
- **vintageProfit()** : método que percorre os artigos e calcula o valor que o sistema **Vintage** ganhou com a encomenda, tendo em conta os valores de taxa de satisfação de serviço associado a cada artigo(0,25€ para um artigo usado e 0,50€ para um artigo novo) e retorna esse *profit*.

3.2 Transportadora

A classe **Transportadora** contém o seguinte:

Variável	Tipo
nome	String
valorBase	double
margemLucro	double

A classe **Transportadora** contém os contém as variáveis apresentadas na tabela ao lado. Esta classe contém também os métodos *clone()*, *equals(Object o)* e *toString()* e os *getters* e *setters* das variáveis de instância presentes na classe.

4 Classe Utilizador

A classe **Utilizador** contém o seguinte:

Variável	Tipo
codigoSistema	String
email	String
nome	String
morada	String
nif	String
profit	double
produtosAVendaCodBarras	List<String>
encomendasFeitas	List<Encomenda>
artigosCarrinho	List<Artigo>
produtosVendidos	<String, Artigo>

A classe **Utilizador** contém as variáveis apresentadas na tabela ao lado. O utilizador tem guardado em si, uma lista com o código de barras dos produtos à venda.
O utilizador conta também com uma lista de produtos vendido guardados num *Map*, uma lista de encomendas feitas e também uma lista chamada de *artigosCarrinho* que guarda artigos num carrinho, no qual poderá ser feito o respetivo *checkout* esvaziando o mesmo e adicionando esses artigos que estavam no carrinho para a lista de encomendas feitas.

5 Estatísticas sobre o estado do programa

Para realizar as estatísticas propostas no enunciado do trabalho prático, o grupo decidiu colocar os métodos que tratam das mesmas, numa classe **ModelQueries** e **ControllerQueries**, usando uma abordagem **MVC**, que iremos explicar mais abaixo de forma mais detalhada.
Segue abaixo, uma explicação detalhada dos métodos utilizados para fazer o tratamento estatístico diretamente:

5.1 Vendedor que mais faturou num período ou desde sempre

Para resolver esta **query** utilizamos o seguinte método para obter o vendedor que mais faturou desde sempre:

```
public Utilizador getVendedorMaisFaturouSempre(Map<String, Utilizador> lstUsers) {
    Utilizador usrComMaisProfit = null;
    double maiorProfit = Double.NEGATIVE_INFINITY;

    for(Utilizador utilizador : lstUsers.values()) {
        if(utilizador.getProfit() > maiorProfit) {
            maiorProfit = utilizador.getProfit();
            usrComMaisProfit = utilizador;
        }
    }

    return usrComMaisProfit;
}
```

Neste método ele percorre a lista de utilizadores passado como argumento, e vai sempre aumentando a variável inicializada a **Double.NEGATIVE_INFINITY** à medida que encontra um profit maior, e depois retorna o utilizador com a maior faturação desde sempre, a partir da variável profit, obtida através do método *getProfit()* de cada utilizador.

Quanto ao método usado para ver o vendedor que mais faturou num dado período de tempo:

```
public Utilizador getVendedorQueMaisFaturouPeriodo(Map<String, Utilizador> lstUsers, LocalDate di, LocalDate df) {
    Utilizador ret = null;
    double maiorProfit = Double.NEGATIVE_INFINITY;

    for(Utilizador uti : lstUsers.values()) {
        double profitAtual = 0.0;
        for(Artigo art : uti.getProdutosVendidos().values()) {
            if(art.getDataComprado().isAfter(di) && art.getDataComprado().isBefore(df)) {
                profitAtual += art.getPrecoBase();
            }
        }
        if(profitAtual>maiorProfit) {
            maiorProfit = profitAtual;
            ret = uti.clone();
        }
    }
    return ret;
}
```

Neste método a lista de utilizadores, passada como argumento é percorrida, e dentro de cada utilizador é percorrida a lista de produtos vendidos e caso esses produtos vendidos estejam entre as datas passadas como argumento, ele faz o *profit* dos artigos que foram vendidos nesse período de tempo, e compara com a variável *maiorProfit* e devolve o utilizador que faturou mais no período de tempo indicado.

5.2 Transportadora com maior volume de faturação

Quanto ao método que trata desta estatística, usamos o seguinte:

```java
public String transportadoraComMaisFaturacao(Map<String, Transportadora> lstTransportadoras, Map<String, Utilizador> listaUtilizadores) {
    String ret = null;
    double val = Double.NEGATIVE_INFINITY;

    for(Transportadora trans : lstTransportadoras.values()) {
        double currentVal = 0.0;
        for(Utilizador user : listaUtilizadores.values()) {
            for(Artigo art : user.getProdutosVendidos().values()) {
                if(art.getNomeTransportadora().equals(trans.getNome())) {
                    currentVal = currentVal + (trans.valorExpedicao() * (1 - trans.getMargemLucro())/100);
                }
            }
        }
        if(currentVal > val) val = currentVal;
        ret = trans.getNome();
    }

    return ret;
}
```

Neste método é usado a lista de transportadoras, e a lista de utilizadores. Este métdodo percorre a lista de transportadora e para cada transportadora, ele percorre os utilizadores e os artigos vendidos pelos mesmos e para cada artigo verifica se está associada a transportadora. Terminando estes ciclos, é retornado o nome da transportadora com mais faturação.

5.3 Listar encomendas emitidas por um vendedor

O método usado para tratar desta estatística é o seguinte:

```java
public Map<String, Encomenda> encsVendedor(Map<String, Encomenda> listaEncomendas, Utilizador user) {
    Map<String, Encomenda> encsUser = new HashMap<String, Encomenda>();

    for(Encomenda enc : listaEncomendas.values()) {
        for(Artigo art : enc.getArtigos().values()) {
            if (user.getProdutosVendidos().containsValue(art)) {
                encsUser.put(enc.getCodSistema(), enc);
            }
        }
    }

    return encsUser;
}
```

Neste método é retornado um *Map¡String, Encomenda¿* que basicamente, é uma lista com as encomendas emitidas por um vendedor.

5.4 Ordenação dos maiores compradores/vendedores num período de tempo

No caso dos maiores compradores num período de tempo, utilizamos o seguinte método que trata diretamente desta estatística:

```java
public String topNCompradores(int num, Map<String, Utilizador> listaUtilizadores, LocalDate di, LocalDate df) {
    StringBuilder sb = new StringBuilder();
    List<Utilizador> utilizadoresOrdenadosPorDinheiroGasto = new ArrayList<>(listaUtilizadores.values());

    Collections.sort(utilizadoresOrdenadosPorDinheiroGasto, new Comparator<Utilizador>() {
        @Override
        public int compare(Utilizador u1, Utilizador u2) {
            double dinheiroGastoU1 = u1.dinheiroGasto(di, df);
            double dinheiroGastoU2 = u2.dinheiroGasto(di, df);
            if(dinheiroGastoU1 < dinheiroGastoU2) {
                return 1;
            } else if(dinheiroGastoU1 > dinheiroGastoU2) {
                return -1;
            } else {
                return 0;
            }
        }
    });

    utilizadoresOrdenadosPorDinheiroGasto = utilizadoresOrdenadosPorDinheiroGasto.subList(0, Math.min(num, utilizadoresOrdenadosPorDinheiroGasto.size()));
    for(Utilizador uti : utilizadoresOrdenadosPorDinheiroGasto) {
        sb.append(uti.toString()).append("\n");
    }
    return sb.toString();
}
```

Neste método, é possível escolhar quantos maiores compradores se pretende visualizar. Este método ordena a lista de utilizadores, através do seu dinheiro gasto em compras de artigos, através de um comparador criado dentro deste método. E depois cria uma *"sublist"* com o número de compradores pretendido e depois dá-se *return* dessa sublista numa *(*string).

No caso dos maiores vendedores num período de tempo, utilizamos o seguinte método:

```java
public String topNVendedores(int num, Map<String, Utilizador> listaUtilizadores, LocalDate di, LocalDate df) {
    StringBuilder sb = new StringBuilder();

    List<Utilizador> utilizadoresList = new ArrayList<>(listaUtilizadores.values());
    utilizadoresList.sort(Comparator.comparingDouble(u -> -u.dinheiroGanho(di, df)));
    utilizadoresList = utilizadoresList.subList(0, Math.min(num, utilizadoresList.size()));

    for(Utilizador uti : utilizadoresList) {
        sb.append(uti.toString()).append("\n_____\n");
    }

    return sb.toString();
}
```

Neste método, tal como o anterior, há a possibilidade de escolher o número de maiores vendedores que pretendemos visualizar. No caso deste método, utilizando um *Comparator*, ordena-mos a lista de utilizadores fornecida como argumento e depois "cortamos"essa lista para o número de vendedores que pretendemos ver. No final retornamos essa list em formato *String*.

5.5 Determinar quando dinheiro ganhou o Vintage

Para resolver esta *query*, utilizamos o seguinte método que a resolve diretamente:

```java
public double vintageProfit(Map<String, Encomenda> listaEncomendas) {
    double lucro = 0.0;

    for(Encomenda enc : listaEncomendas.values()){
        lucro += enc.vintageProfit();
    }

    return lucro;
}
```

Neste método, o *profit* do **Vintage** é feito através da lista de encomendas realizadas, através de um método presente na classe encomenda que devolve o valor total do *profit* do **Vintage** na encomenda. No ciclo *foreach* é somado o lucro total e depois esse lucro é retornado.

6 Ficheiros

O nosso método de leitura de ficheiros começa por no início dar Load de todos os ficheiros .txt que temos (utilizadores.txt, artigos.txt, encomendas.txt e transportadoras.txt). Cada um destes ficheiros é carregado, povoando assim todas as Maps necessárias com as informações provenientes destes ficheiros. Além disto, as strings correspondentes a cada ficheiro são também carregadas numa variável da classe Versao. Esta variavel, é usada por todo o projeto, e visa manter sempre a versão mais atualizada do projeto até ao momento. Por exemplo, se for criado um artigo ou uma transportadora, este é adicionado à variável da classe Versao responsável por carregar o conteúdo do ficheiro respetivo a este tipo de variável previamente carregado. Ou seja se tivermos a tratar de um artigo, quando é registado um novo artigo na plataforma, este é também carregado para a variavel versaoArtigosTxt da classe Versão, assim esta variável carrega sempre todas as versões mais atualizadas do programa.

Isto tudo é feito para que os ficheiros se mantenham intactos até que se queira realmente efetuar o save. Quando o utilizador quer efetuar o save, é chamada uma função que passa as strings que estavam previamente no objeto da classe Versao, para os ficheiros a elas destinadas. A variável versaoArtigosTxt para o ficheiro artigos.txt, versaoTransportadorasTxt para o ficheiro transportadoras.txt etc. Além disto quando é efetuado o Save, a versao é carregada para uma Map que carrega todas as versões usadas nesta sessão, identificadas pela hora de criação da mesma.

Quanto ao load de ficheiros, este é efetuado acessando a Map, e carregando a versão desejada. O utilizador pode escolher a versão que deseja carregar introduzindo a hora de criação da mesma no ecrã.

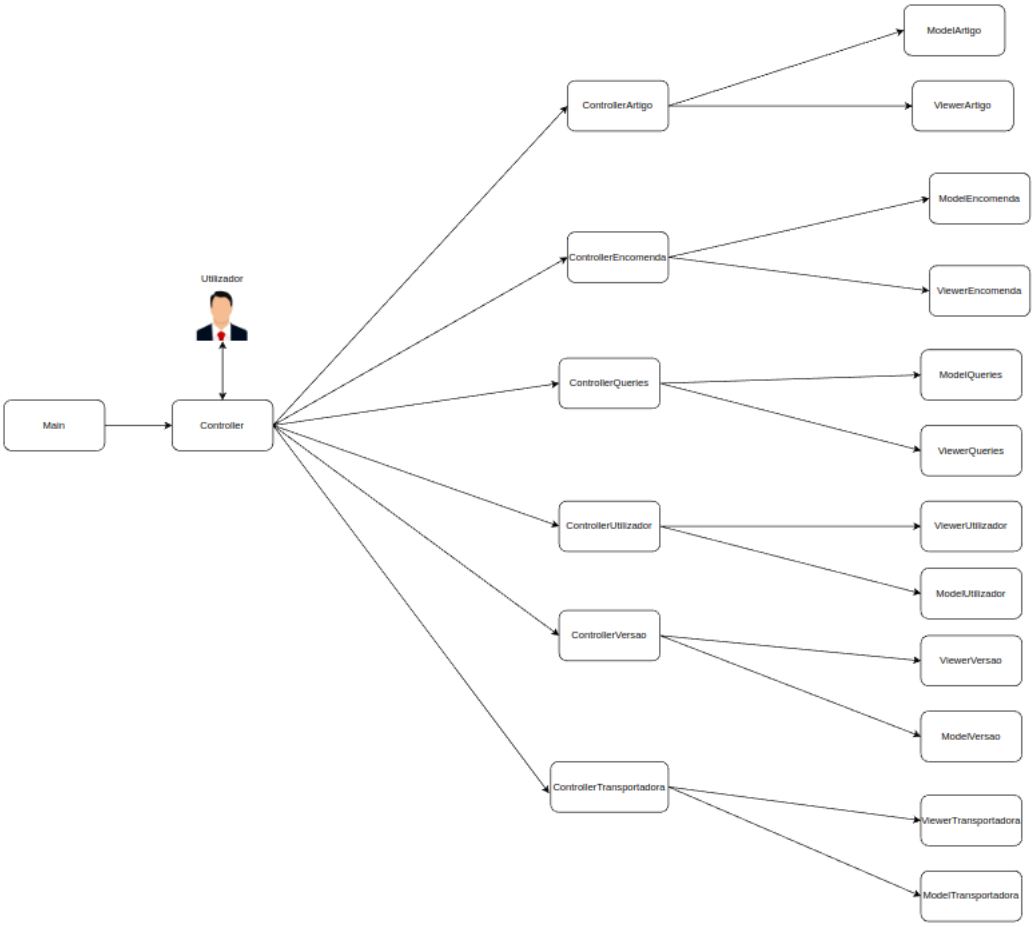
É importante salientar que apesar de termos a ideia correta, e alguns dos métodos já feitos, não conseguimos aplicar a atualização do ficheiro referente aos Utilizadores, nem a atualização do ficheiro referente aos Artigos, por erros que não fomos capazes de encontrar a solução.

7 Estrutura e abordagem MVC

No início do projeto do projeto, optamos por ter um módulo geral no qual teria em si todas as informações do programa(excluindo as classes).No entanto, com o avançar do projeto, reparamos que a nossa estratégia não era a ideal para o projeto, por isso optamos adotar uma arquitetura *"MVC"* (Model-View-Controller) que é uma arquitetura que promove uma maior modularidade e encapsulamento do código.

Com a arquitetura *"MVC"*, para cada classe do nosso projeto, nós temos um *"Model"*, que vai ser responsável por tratar dos dados relativos aquela classe, um *"View"*, que vai ser responsável por tratar dos dados para apresentar ao utlizador, e um *"Controller"*, que vai ser responsável por servir de ponte entre o *"Model"* e *"View"* da classe e um *"Controller"* geral, que serve para comunicar com o utlizador.

7.1 Estrutura



8 Diagrama de Classes

9 Conclusão

O grupo considera que na globalidade do projeto, poderia ter feito mais, dado que não conseguimos completar o avanço de datas e a parte estatística não estar 100% funcional.

Sabemos que poderiamos ter uma melhor modularização e encapsulamento, pois o *Viewer* não é utilizado como é suposto numa abordagem MVC e também devido à classe Utilizador conter dentro de si vários Artigos numa lista, o que quebra também o encapsulamento.

Quanto aos aspetos positivos, o nosso grupo tem a apontar a forma como é possível gerir grandes projetos e assegurar a segurança dos seus dados. Os elementos do grupo desenvolveram também capacidades de resolução de problemas que tivemos de enfrentar e também um desenvolvimento ao nível da capacidade do trabalho de grupo.