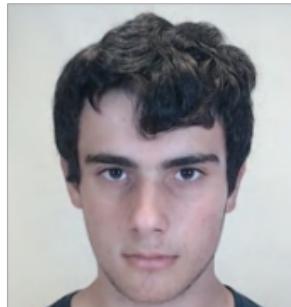


UNIVERSIDADE DO MINHO  
LICENCIATURA EM ENGENHARIA INFORMÁTICA

POO - Projeto Prático  
Grupo 54

David Teixeira (A100554)      João Pastore (A100543)  
Jorge Nuno Rodrigues (A101758)

Ano Letivo 2022/2023



# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>5</b>
<b>2</b>	<b>Classes</b>	<b>6</b>
2.1	Main . . . . .	6
2.2	Controller . . . . .	6
2.3	Model . . . . .	6
2.4	View . . . . .	7
2.5	CarrierManager . . . . .	7
2.6	ItemManager . . . . .	8
2.7	Order Manager . . . . .	8
2.8	User Manager . . . . .	8
2.9	Querier . . . . .	9
2.10	BiggestEarnerAllTime . . . . .	9
2.11	BiggestEarnerAllTimeTimeFrame . . . . .	9
2.12	BiggestCarrier . . . . .	10
2.13	EmmitedOrderList . . . . .	10
2.14	PodiumSeller . . . . .	11
2.15	PodiumSpenders . . . . .	11
2.16	VintageProfit . . . . .	11
2.17	Item . . . . .	12
2.18	Bag . . . . .	12
2.19	Sneaker . . . . .	12
2.20	Tshirt . . . . .	13

2.21	Premium . . . . .	13
2.22	PremiumBag . . . . .	13
2.23	PremiumSneaker . . . . .	13
2.24	PremiumCarrier . . . . .	14
2.25	SystemDate . . . . .	14
2.26	Bill . . . . .	14
2.27	Carrier . . . . .	14
2.28	Order . . . . .	16
2.29	User . . . . .	16
2.30	Util . . . . .	17
2.31	CarrierAlreadyExistsException . . . . .	17
2.32	InvalidCommand . . . . .	17
2.33	InvalidID e MissedIdException . . . . .	17
2.34	OrderNotReturnable . . . . .	17
2.35	UserAlreadyExistsException . . . . .	17
2.36	UserIsAdminException . . . . .	18
<b>3</b>	<b>Testes Unitários</b>	<b>19</b>
3.1	BillTest . . . . .	19
3.2	CarrierTest . . . . .	20
3.3	IDTest . . . . .	20
3.4	ItemTest . . . . .	20
3.5	OrderTest . . . . .	21
3.6	Pointer Test . . . . .	21
<b>4</b>	<b>Funcionalidades do Programa</b>	<b>22</b>
<b>5</b>	<b>Especificações Técnicas das Funcionalidades</b>	<b>24</b>
<b>6</b>	<b>Arquitetura do Projeto</b>	<b>27</b>
<b>7</b>	<b>Diagrama de Classes</b>	<b>28</b>



# Capítulo 1

## Introdução

Este projeto consistiu no desenvolvimento de uma aplicação de Marketplace chamada Vintage. Esta aplicação permite a compra e venda de vários tipos de artigos com funcionalidades distintas como controlo de utilizadores, gestão de encomendas e uma simulação de time skip (avançar no tempo).

Algumas *features* da aplicação:

- Autenticação e criação de perfis de utilizadores.
- Criação de artigos e informações relevantes.
- Funcionalidade para criação, atualização e gestão de encomendas.
- Funcionalidade para compra e venda de artigos.
- Atualização da informação do sistema e geração de faturas (*bills*).
- Geração de estatísticas e afins (*queries*).
- Suporte para transportadoras/itens premium e fórmulas de cálculo (noção de *Premium*).
- Automatização da simulação através da leitura de arquivos.

# Capítulo 2

## Classes

### 2.1 Main

Responsável pela inicialização do programa. O método *main* inicia *Controller*, que atua como ligação entre *Model* e *View*;

### 2.2 Controller

```
private Model m;
```

Classe, que juntamente com *Model* e *View*, é responsável pela gestão geral da aplicação. O *Controller* limita-se a chamar os métodos disponíveis da classe *model* que *View* necessita, mas sem aceder de forma direita aos dados da aplicação. Deste modo, o *Controller* atua como intermediário (uma ligação) entre *Model* e *View*.

### 2.3 Model

```
private ItemManager itemManager;
private UserManager userManager;
private OrderManager orderManager;
private CarrierManager carrierManager;
private double vintageProfit;
private User currentUser;
```

*Model* é a classe que assegura a parte das regras e camada computacional (a parte lógica/matemática do programa). Um objeto *Model* é construído sobretudo a partir de quatro managers (*Item*, *User*, *Order* e *Carrier*), que possuem *maps* e vários métodos de gestão. Através destes managers, *Model* manipula informação relacionada com cada gestor.

Algumas das features incluem :

- Registo Artigos, Utilizadores e Encomendas.
- Manipulação de carriers.
- Adição de Artigos a Utilizadores.
- Modificação do estado de Artigos (*sold*, *listed*).
- *Time skip.*

## 2.4 View

```
private Controller _cont = null;
```

Classe que interage com o utilizador. Relaciona-se de forma direta e única com Controller. A View obtém informações (via Scanner) provenientes do utilizador que, por sua vez, serão úteis ao Model. Estes inputs são passados para o Controller, que encaminha para Model. A View captura a informação do utilizador e também mostra (sob a forma de texto no terminal) informações, como por exemplo estatísticas que advêm de Querier e o carrinho de cada Utilizador.

## 2.5 CarrierManager

```
private TreeMap<String , Carrier> carrierMap ;
```

Classe do tipo *Manager*. Gere o *Map* de transportadoras;

Esta classe apresenta métodos de manipulação do *carrierMap*, tais como :

- Pesquisa de Transportadoras, via *String* (nome) da Transportadora.
- Adição e remoção de Transportadoras ao *Map* de transportadoras.
- Criação de uma cópia do *carrierMap*.

Nesta classe, os objetos são gravados por **composição**.

## 2.6 ItemManager

```
private HashMap<Integer , Item> soldItemsMap ;  
private HashMap<Integer , Item> listedItemsMap ;
```

Classe *Manager*. Gere os *Maps* de itens.

Esta classe apresenta métodos de manipulação dos *soldItemsMap* e *listedItemsMap*, tais como:

- Pesquisa de Artigos, via *id* do Artigo.
- Adição, remoção e atualização (no caso do Artigo passar de listado a vendido) de Artigos aos *Maps* de itens vendidos ou a vender.
- Atribuição de uma lista de itens vendidos e listados.

Nesta classe, os objetos são gravados por **composição**.

## 2.7 Order Manager

```
private HashMap<Integer , Order> orderMap ;
```

Classe *Manager*. Gere o *Map* de encomendas.

Esta classe apresenta métodos de manipulação do *orderMap*, tais como :

- Pesquisa, remoção e adição de uma Encomenda, via *id* da encomenda .
- Atribuição de uma lista de encomendas feitas por um Utilizador, via *id* do Utilizador.

Nesta classe, os objetos são gravados por **composição**.

## 2.8 User Manager

```
private Map<Integer , User> userMap ;
```

Classe *Manager*. Gere o *Map* de utilizadores.

Esta classe apresenta métodos de manipulação do *userMap*, tais como :

- Pesquisa (também possível via *email* do Utilizador), remoção e adição de um Utilizador, via o seu respetivo *id*.
- Atribuição de uma lista com todos os Utilizadores.
- Criação de uma cópia do *userMap*.
- Remoção de faturas de uma encomenda.

Nesta classe, os objetos são gravados por **composição**.

## 2.9 Querier

```
public interface Querier  
{  
    public Object execute() throws NullPointerException;
```

*Querier* é uma interface implementada em todo o tipo de classe que seja responsável pelo cálculo de estatísticas, ou seja, *queries*. Assim, *querier* é responsável pelo agrupamento de todas as *queries*.

## 2.10 BiggestEarnerAllTime

```
private Map<Integer ,User> hm;
```

Esta classe implementa *Querier*, tratando-se de uma *Query* (operação de consulta). É responsável por consultar *qual é o vendedor que mais facturou desde sempre*.

*BiggestEarnerAllTime*, utiliza uma cópia de *usersMap* (mapa de utilizadores). Com o auxílio de mecanismos de controlo de erros, verifica primeiramente a eventualidade de existir um mapa vazio, declarando um erro através da mensagem: *"No user is in the Model"*.

Não havendo erro, inicializa *User biggestEarner* a *null*. De seguida, limita-se a atravessar a cópia de *usersMap*, guardando sempre em *biggestEarner* o vendedor que apresenta o maior volume de faturação até ao momento.

Para obter o maior volume de faturação de cada vendedor, utiliza-se o método **soldItemsValue()**, definido em *User.java*.

No final da travessia, retorna *biggestEarner*.

## 2.11 BiggestEarnerAllTimeTimeFrame

```
private Map<Integer ,User> hm;  
private LocalDate date1;  
private LocalDate date2;
```

Esta classe implementa *Querier*, tratando-se de uma *Query* (operação de consulta). É responsável por consultar *qual é o vendedor que mais facturou num período*.

*BiggestEarnerAllTimeTimeFrame*, utiliza uma cópia de *usersMap* (mapa de utilizadores), e coloca em *date1* a data mais antiga e em *date2* a mais recente . Com o auxílio de mecanismos de controlo de erros, verifica primeiramente a eventualidade de existir um mapa vazio, declarando um erro através da mensagem: *"No user is in the Model"*.

Não havendo erro, inicializa *User biggestEarner* a *null*. De seguida, limita-se a atravessar a cópia de *usersMap*, guardando sempre em *biggestEarner* o vendedor que apresenta o maior volume de faturação até ao momento, **que se encontre dentro das datas anteriormente colocadas**.

Para obter o maior volume de faturação de cada vendedor, utiliza-se o método **soldItemsValue()**, definido em *User.java*.

No final da travessia, retorna *biggestEarner*.

## 2.12 BiggestCarrier

```
private Map<String , Carrier> hm;
```

Esta classe implementa *Querier*, tratando-se de uma *Query* (operação de consulta). É responsável por consultar *qual o transportador com maior volume de facturação*.

*BiggestCarrier*, utiliza uma cópia de *carriersMap* (mapa de transportadoras). Com o auxílio de mecanismos de controlo de erros, verifica primeiramente a eventualidade de existir um mapa vazio, declarando um erro através da mensagem: *"No Carrier is in the Model"*.

Não havendo erro, inicializa *Carrier biggestEarner* a *null*. De seguida, limita-se a atravessar a cópia de *carriersMap*, guardando sempre em *biggestEarner* a transportadora que apresenta o maior volume de faturação até ao momento.

Para obter o maior volume de faturação de cada transportadora, utiliza-se o método **getTotalEarning()**, definido em *Carrier.java*.

No final da travessia, retorna *biggestEarner*.

## 2.13 EmmitedOrderList

```
private Map<Integer , User> hm;
private int id;
```

Esta classe implementa *Querier*, tratando-se de uma *Query* (operação de consulta). É responsável por *listar as encomendas emitidas por um vendedor*.

*BiggestCarrier*, utiliza uma cópia de *usersMap* (mapa de utilizadores). Com o auxílio de mecanismos de controlo de erros, verifica primeiramente a eventualidade de existir um mapa vazio, declarando um erro através da mensagem: *"No User is in the Model"*.

Não havendo erro, inicializa *User u*, que irá corresponder ao utilizador dado através do método **get()**. O argumento *id* em **get()** será introduzido via *Scanner* em *View*.

Caso **get()** retorne *null*, o sistema declara *"User not in the system"*.

Se existir de facto um *User* com o *id* introduzido, é retornado o resultado do método **getEmmitedOrder()**, definido em *User.java*.

## 2.14 PodiumSeller

```
private Map<Integer ,User> hm;
private LocalDate date1 ;
private LocalDate date2 ;
```

Esta classe implementa *Querier*, tratando-se de uma *Query* (operação de consulta). É responsável por *fornecer uma ordenação dos maiores vendedores do sistema durante um período a determinar*.

*BigestEarnerAllTimeTimeFrame*, utiliza uma cópia de *usersMap* (mapa de utilizadores), e coloca em *date1* a data mais antiga e em *date2* a mais recente . Com o auxílio de mecanismos de controlo de erros, verifica primeiramente a eventualidade de existir um mapa vazio, declarando um erro através da mensagem: *"No user is in the Model"*.

Não havendo erro, inicializa *List<User> topSellers*, que corresponde a um *ArrayList* de *Users* cujos valores são obtidos através do método **values()**, que tem como argumento o mapa de utilizadores.

Seguidamente, ordena *topSellers* por ordem decrescente com base no valor total dos produtos vendidos por um utilizador vendedor num período de tempo específico. São realizados dois loops para comparar cada par de vendedores *user1* e *user2* e trocar as suas respetivas posições caso o valor de vendas de *user2* seja maior do que de *user1*.

É utilizado **soldItemsValueFrame()** definido em *User.java*, como método de comparação.

No final da travessia, é retornada a lista *topSellers*.

## 2.15 PodiumSpenders

```
private Map<Integer ,User> hm;
private LocalDate date1 ;
private LocalDate date2 ;
```

Esta classe implementa *Querier*, tratando-se de uma *Query* (operação de consulta). É responsável por *fornecer uma ordenação dos maiores compradores do sistema durante um período a determinar*.

Este método é idêntico no seu todo ao método referido anteriormente; a única diferença significativa corresponde ao método a ser utilizado como factor de comparação, que no caso é **boughtValueFrame()** definido em *User.java*.

## 2.16 VintageProfit

```
private double vp;
```

Esta classe implementa *Querier*, tratando-se de uma *Query* (operação de consulta). É responsável por *determinar quanto dinheiro ganhou o Vintage no seu funcionamento*. Limita-se a retornar um *double*, que é obtido via **getVintageProfit()** definido em *Model.java*.

## 2.17 Item

```
private String description;
private String brand;
private double basePrice;
private Carrier carrier;
private double conditionScore;
Stack<Integer> previousOwners;
private int id;
private int userId;
```

Para implementar os artigos, começamos com a classe abstrata *Item*. Essa classe serve como um ponto de partida para algo mais específico, como os próprios artigos (mala, sapatinha e tshirt).

A classe *Item* possui atributos que todos os tipos de artigos compartilham. A partir dessa classe, as subclasses são criadas para gerar os artigos reais.

A decisão de usar uma classe abstrata *Item* foi feita para permitir a reutilização de código e simplificar a implementação dos artigos.

Esta classe é relativamente simples em termos de código, possuindo apenas construtores de classe, *getters*, *setters* e implementações dos métodos usuais **toString()**, **equals()** e **clone()**.

## 2.18 Bag

```
private double dimension;
private String material;
private LocalDate releaseDate;
```

Bag trata-se de uma subclassse de *Item*. Corresponde à implementação do tipo de artigo Mala. Aos atributos que todos os itens apresentam, acrescentamos nesta classe os atributos dimensão (em cm<sup>3</sup>), material e data de lançamento. É também criado um método **getPrice()**, que se baseia na fórmula apresentada no enunciado do projeto prático.

Possui construtores de classe, *getters*, *setters* e implementações dos métodos usuais **toString()**, **equals()** e **clone()**.

## 2.19 Sneaker

```
private double size;
private SneakerType type;
private String color;
private LocalDate releaseDate;
```

Sneaker trata-se de uma subclassse de *Item*. Corresponde à implementação do tipo de artigo Sapatinha. Aos atributos que todos os itens apresentam, acrescentamos nesta classe os atributos tamanho, tipo (se possui atacadore/tilhos), cor e data de lançamento. É também criado um método **getPrice()**, que se baseia na fórmula apresentada no enunciado do projeto prático.

Possui construtores de classe, *getters*, *setters* e implementações dos métodos usuais **toString()**, **equals()** e **clone()**.

## 2.20 Tshirt

```
private TshirtSize size;  
private TshirtPattern pattern;
```

Tshirt trata-se de uma subclassse de *Item*.

Aos atributos que todos os itens apresentam, acrescentamos nesta classe os atributos tamanho (S,M,L,XL) e padrão (liso, riscas, palmeiras). É também criado um método **getPrice()**, que se baseia no enunciado do projeto prático.

Esta possui construtores de classe, *getters*, *setters* e implementações dos métodos usuais **toString()**, **equals()** e **clone()**.

## 2.21 Premium

```
public interface Premium
```

Premium trata-se de uma interface que serve meramente para etiquetar uma entidade que seja do tipo Premium. Seguidamente serão descritas tais entidades:

## 2.22 PremiumBag

PremiumBag trata-se de uma subclassse de *Bag*, que implementa a interface *Premium*. Corresponde à implementação do tipo de artigo Mala Premium. Não é acrescentado qualquer tipo de novo atributo a esta classe, em comparação com *Bag*. O único factor diferenciador trata-se do método **getPrice()**, que segue uma fórmula de cálculo de preço dedicada a itens do tipo *Premium*.

Possui construtores de classe, *getters*, *setters* e implementações dos métodos usuais **toString()** e **clone()**.

## 2.23 PremiumSneaker

PremiumSneaker trata-se de uma subclassse de *Sneaker*, que implementa a interface *Premium*. Corresponde à implementação do tipo de artigo Sapatilha Premium. Não é acrescentado qualquer tipo de novo atributo a esta classe, em comparação com *Sneaker*. O único factor diferenciador trata-se do método **getPrice()**, que segue uma fórmula de cálculo de preço dedicada a itens do tipo *Premium*.

Possui construtores de classe, *getters*, *setters* e implementação do método usual **clone()**.

## 2.24 PremiumCarrier

PremiumCarrier trata-se de uma subclassse de *Carrier*, que implementa a interface *Premium*. Corresponde à implementação do tipo de artigo Transportadora Premium. Não é acrescentado qualquer tipo de novo atributo a esta classe, em comparação com **Carrier**.

Possui construtores de classe, *getters*, *setters* e implementação do método usual **clone()**.

## 2.25 SystemDate

```
private static LocalDate date;
```

SystemDate tem como objetivo armazenar o valor correspondente a uma data obtida a partir de uma simulação para o cálculo de preços de um item premium. Atua como uma classe intermediária entre os itens premium e o input do utilizador.

Possui *getters* e *setters*.

## 2.26 Bill

```
private final int billNumber;
private TypeBill type;
private Map<Integer ,Item> items;
private double totalCost;
private double portsTax;
private Order order;
private static int bill_count = 1;
```

Bill trata-se de uma classe responsável pela integração de faturas no sistema. Possui construtores de classe, *getters*, *setters* e implementações dos métodos usuais **toString()**, **equals()** e **clone()**.

Para além disso, possui métodos de adição/remoção de itens, cálculo de custo total dos itens, cálculo da quantidade dos itens e afins.

## 2.27 Carrier

```
private String name;
private double taxSmall;
private double taxMedium;
private double taxBig;
private double totalEarning;
```

Carrier trata-se de uma classe responsável pela integração de transportadoras no sistema. Possui construtores de classe, *getters*, *setters* e implementações dos métodos usuais **toString()**, **equals()** e **clone()**.

Para além disso, possui métodos de atualização do ganho de transportadoras (adição ou remoção do ganho). Por implementar a interface Comparable possui, também, um método **compareTo()**, que compara *Carriers* com base no ganho total de cada uma.

## 2.28 Order

```
private List<Item> collection;
private HashMap<String, Integer> carrierHelper;
private List<User> sellers;
private User buyer;
private TypeOfSize dimension;
private double itemPrice;
private double satisfactionPrice;
private OrderState state;
private LocalDate date;
private int id;
private double endPrice;

private static int currentID = 1;
```

Order trata-se de uma classe responsável pela integração de encomendas no sistema. Possui construtores de classe, *getters*, *setters* e implementações dos métodos usuais **toString()**, **equals()** e **clone()**.

Para além disso, possui métodos de remoção e adição de itens assim como cálculo de preços.

Por implementar a interface Comparable possui, também, um método **compareTo()**, que compara *Orders* com base na data de cada uma.

## 2.29 User

```
private int id;
private String email;
private String name;
private String address;
private int nif;
private String password;
private Map<Integer, Bill> bills;
private List<Item> systemItems;
private List<Item> sellingItems;

private static int currentID = 1;
```

User trata-se de uma classe responsável pela integração de utilizadores no sistema. Possui construtores de classe, *getters*, *setters* e implementações dos métodos usuais **toString()**, **equals()** e **clone()**.

Para além disso, possui métodos de remoção, adição e manipulação de itens assim como cálculo de preços.

Por implementar a interface Comparable possui, também, um método **compareTo()**, que compara *Users* com base no valor total dos items que venderam, obtido através do método **soldItemsValue()**.

## **2.30 Util**

Util trata-se de uma classe responsável unicamente por manipular e transformar *Strings* que advêm do input do utilizador. Em util, existem métodos de conversão de *Strings* para formatos como *Date*, *TshirtSize*, *TshirtPattern*, *SneakerType*, entre outros.

## **2.31 CarrierAlreadyExistsException**

CarrierAlreadyExistsException corresponde a uma classe que é responsável por representar, sob a forma de uma *exception*, os casos onde os utilizadores tentam adicionar ao sistema uma transportadora que já se encontra inserida.

## **2.32 InvalidCommand**

InvalidCommand corresponde a uma classe que é responsável por representar e identificar, sob a forma de uma *exception*, os casos onde os utilizadores inserem comandos inválidos no ficheiro utilizado para a automatização do sistema.

Fornece informações tais como o comando inválido e a linha onde foi escrito.

## **2.33 InvalidID e MissedIdException**

InvalidID e MissedIdException correspondem a classes que são responsáveis por representar e identificar, sob a forma de uma *exception*, os casos onde os utilizadores inserem ID's que são inválidos ou que não existem.

## **2.34 OrderNotReturnable**

OrderNotReturnable corresponde a uma classe que é responsável por representar e identificar, sob a forma de uma *exception*, os casos onde uma encomenda não possa ser devolvida.

## **2.35 UserAlreadyExistsException**

UserAlreadyExistsException corresponde a uma classe que é responsável por representar e identificar, sob a forma de uma *exception*, os casos onde o utilizador pretende inserir um *User* já existente no sistema.

## **2.36 UserIsAdminException**

UserIsAdminException corresponde a uma classe que é responsável por representar e identificar, sob a forma de uma *exception*, os casos onde o utilizador pretende efetuar uma operação que não seja possível quando está no modo de administrador do sistema.

# Capítulo 3

## Testes Unitários

De forma a testar a qualidade e verificar possíveis erros nas classes, foram desenvolvidos vários testes em JUnit que testam a veracidade e autenticidade do código. O principal objetivo dos testes é verificar as fórmulas numéricas das classes, como por exemplo as expressões usadas para calcular preços dos itens. No total, existem 26 testes que estão presentes em 6 ficheiros principais: “BillTest”, “CarrierTest”, “IDTest”, “ItemTest”, “OrderTest” e ”Pointer Test”.

### 3.1 BillTest

```
<public void smallBillBought()>
<public void mediumBillSameCarriersBought()>
<public void mediumBillDiffCarriersBought()>
<public void BigBillBought()>
<public void getAmountSmallSold()>
<public void getAmountMediumSold()>
<getAmountBigSold()>
```

“BillTest”, tem como objetivo testar os cálculos usados para determinar o preço de uma fatura. Nos testes, são adicionados itens a uma “Bill” e é testado se os preços totais (“totalCost”) e as taxas das transportadoras (“portsTax”) são calculadas corretamente. Consequentemente é verificado se o resultado final (“Amount”) está correto, tendo em conta que ele depende diretamente dos outros dois.

Além disto, como o “Amount” é calculado de maneira diferente se a “Bill” estiver no estado “Bought” ou “Sold”, são feitos também testes para ambos os dados.

Algumas coisas importantes de referir são, que dentro dos testes “MediumBill” e “BigBill” as funções que removem itens da “Bill” também são testadas, removendo alguns itens da Bill e verificando se o resultado também muda e se o tamanho da “Bill” é alterado (por exemplo, passar de Big para Medium; o que por sua vez afeta as variáveis anteriores também). Outra coisa, é que são feitos testes com transportadoras diferentes de modo a verificar se as taxas de transporte são aplicadas corretamente no caso de haver itens associados a transportadoras diferentes.

## 3.2 CarrierTest

```
<public void smallCarrier()>
<public void mediumCarrier()>
<public void BigCarrier()>
```

O “CarrierTest” testa o ganho total de uma transportadora (“TotalEarning”) com a adição de itens. Assim como em “BillTest”, são realizados 3 testes para os possíveis tamanhos das transportadoras.

Nos testes “mediumCarrier” e “BigCarrier” é também testada a mudança no tipo de tamanho e no ganho total, aquando da remoção de itens.

## 3.3 IDTest

```
<public void userID()>
<public void itemID()>
```

O conteúdo do “IDTest” é relativamente direto. Estes testes verificam se os identificadores dos itens e utilizadores (ID) são incrementados corretamente quando são criados novos objetos.

## 3.4 ItemTest

```
<public void bagPrice()>
<public void tshirtPriceSmooth()>
<public void tshirtPriceNotSmooth()>
<public void sneakerPrice()>
<public void premiumBag()>
<public void premiumSneakerNoOwners()>
<public void premiumSneakerWithOwners()>
```

O “ItemTest” testa o método de cálculo utilizado para saber o preço de todos os tipos de itens possíveis de comprar e registar (malas, sapatilhas e t-shirts).

No caso das t-shirts, é testado também o preço dependendo do tipo de padrão do produto (já que isso afeta o preço).

Nas sapatilhas, por outro lado, é testado o preço caso estas já tenham tido (ou não) utilizadores passados, visto que isto também afeta o preço das mesmas.

Por fim, é também testado o preço para a criação de itens premium levando em conta que o cálculo usado para saber o preço dos itens premium (ou seja, somente sapatilhas e malas) é bastante diferente dos itens padrões.

### 3.5 OrderTest

```
<smallOrder()>
<mediumOrder()>
<bigOrder()>
```

O ”Order Test”, verifica se tanto o preço dos objetos que vão ser adicionados às próprias (“Item.Price”) como a taxa da transportadora (mais IVA) (por exemplo, “carrier.TaxSmallwithIVA”) estão corretos e por fim, o cálculo do preço final da encomenda, que depende dos dois anteriores.

Assim, como no caso dos testes das transportadoras e faturas, existem testes para cada um dos possíveis tipos de tamanhos de encomendas, e nos de tamanho “medium” e “big” também é testada a adaptação das encomendas quando são removidos itens.

### 3.6 Pointer Test

```
<userManager()>
<itemManager()>
<orderManager()>
<carrierManager()>
```

Finalmente, os últimos testes são os aos apontadores. Estes testes são feitos de modo a verificar que é possível aceder aos principais dados dos objetos através dos “Managers”.

No total, foram feitos 4 testes, um para cada tipo de “Manager” existente, *i.e.*, “user”, “item”, “order” e “carrier”. Em cada teste, tenta-se aceder às informações principais e relevantes de cada Manager (ou seja, informações sobre os itens no “ItemManager”, do utilizador no “UserManager”...). Contudo, também se tenta a aceder a informações mais específicas e que derivam das principais, como por exemplo, aceder a informações sobre a transportadora através dos itens ou aceder a informações sobre os itens através dos utilizadores.

## Capítulo 4

# Funcionalidades do Programa

Em sucessão aos tópicos referidos anteriormente, é indispensável explicitar de que forma as diversas Classes interagem entre si, de modo a cumprir com as funcionalidades esperadas do projeto como a criação de utilizadores, inserção de artigos e criação de encomendas.

Primeiramente, de modo a manter uma experiência autêntica aos utilizadores, o grupo achou relevante incluir um utilizador com um estatuto especial de administrador. O “admin” tem acesso a diversas funcionalidades particulares que, normalmente, não estão disponíveis a utilizadores comuns, como por exemplo, realizar cálculos estatísticos sobre o programa. No entanto, o administrador perde também funcionalidades normais do utilizador, visto que o papel dele será de gerir o programa, mas sem interagir com as entidades que o englobam à exceção das “Carriers”.

Assim, começaremos por especificar as tais funcionalidades exclusivas ao administrador. Em primeiro lugar, vale a pena salientar que o uso de exceções no projeto é robusto, tendo assim existido a necessidade de criar uma exceção que diferencia o “admin” do resto dos outros utilizadores. Passando às funcionalidades em concreto, o admin dispõe da seguinte lista de opções:

- Criar “Carriers”. Um ponto essencial do sistema são as “Carriers”. Estas devem estar disponíveis antes dos utilizadores começarem a utilizar o programa, pois é necessário para adicionar artigos ao programa. Neste instante é esperado que o administrador forneça o nome da “Carrier” assim como as taxas requisitadas e o seu estatuto.
- Alterar “Carriers”. Após a incorporação de uma transportadora no sistema, é esperado que possam ser realizadas alterações às taxas da mesma. Em coincidência com o enunciado, apenas as encomendas feitas em instantes depois da alteração ficam sujeitas às novas taxas.
- Executar “Queries” estatísticas. Uma vez que a estatística pressupõe um acesso privilegiado ao conteúdo da Vintage, definimos que apenas o administrador pode requisitar esse tipo de informação. Estas estatísticas correspondem a todas que estão no enunciado.

Em segundo lugar temos o utilizador comum da Vintage. Este tipo de utilizador não tem acesso às funcionalidades anteriores dispondo, em contrapartida, de funcionalidades que permitem o verdadeiro funcionamento do programa. As funcionalidades definidas são as seguintes:

- Registar um artigo. De forma a que possa existir comércio entre os utilizadores é uma condição necessária que haja “Itens”. Estes podem ser “Tshirts”, “Bags” ou “Sneakers” e, nos últimos dois tipos, ainda dispõe do estatuto “Premium”.

- Fazer uma encomenda. Um requisito para o funcionamento do programa é a realização de “Orders”. As “Orders” são registadas através de uma lista de códigos dos artigos fornecida pelo utilizador. No momento do registo são listados os “itens” do sistema que não pertencem ao utilizador.
- Ver e retornar encomendas. Uma funcionalidade disponível para os utilizadores é ainda a devolução de encomendas. Para esse efeito, são disponibilizadas as encomendas feitas pelo utilizador numa lista.
- Consultar faturas. Após a conclusão de uma encomenda, surgem então as faturas resultantes da operação. Estas faturas aparecem tanto para o comprador como para os vendedores.
- Meus artigos. A qualquer momento o utilizador tem a possibilidade de ver os seus artigos listados e ver os itens que comprou na Vintage no caso de querer voltar a vender o mesmo.

Há funcionalidades que são comuns a todos os utilizadores, e até mesmo em casos de nenhum utilizador estiver na sua conta. Essas funcionalidades são as seguintes:

- Registar um utilizador. A qualquer instante, a Vintage permite o registo de utilizadores novos no sistema. O registo requer informações relativas à conta como o email, password, nome, morada e NIF. Como fatores de autenticação da conta, apenas serão necessários email e password.
- Salto no tempo. A função de “Time Skip” é essencial para o contexto do programa, pois permite que se possa observar o desenrolar das operações feitas anteriormente. Esta função recebe uma data no futuro e avança a o programa até à data pretendida.
- Login. O procedimento de “login” é necessário para aplicação, porque dá a conhecer ao sistema que utilizador está a fazer uma operação, e também permite ao sistema saber se a operação é valida para o utilizador em questão.
- Logout. O programa dá também a possibilidade ao utilizador de terminar a sua sessão.
- Simulação via Texto. Uma funcionalidade pedida no enunciado é a criação e utilização de um ficheiro com os comandos em texto previamente escritos pelo utilizador. Esta funcionalidade recebe o “path” para o ficheiro em questão e executa as funcionalidades lá descritas.
- Salvaguarda do estado da aplicação. O programa possui a capacidade de guardar (sob o formato .ser) as informações relevantes das entidades que estão em memória. Além disso, também é possível recuperar o estado gravado a qualquer momento, permitindo que se possa reiniciar o programa a partir desse ponto.

## Capítulo 5

# Especificações Técnicas das Funcionalidades

Seguidamente à apresentação sumária das funcionalidades, é ainda relevante expor o que realmente acontece no programa, com intuito de mostrar e justificar as escolhas de Design do grupo. Neste tópico iremos focar nos no que acontece dentro do *Model*, incluindo as relações entre os objetos e os respetivos gestores.

Primeiramente, nós optamos por utilizar classes de gestão em vez de ter as estruturas de dados que compõe os objetos diretamente no *Model*. A motivação por de trás desta decisão está relacionada com conceitos como a modificabilidade e escalonamento do programa, pois acabamos por ter um único ponto onde podemos fazer alterações sem corromper o *Model*, e também por questões de abstração, onde a implementação dos managers fica escondida ao *Model*.

Começando pelas *Carriers*, como referido previamente no documento, estas não possuem nenhum apontador para outro objeto do sistema. Após receber os *inputs* necessários para criar um objeto da classe, o objeto é criado no *Model* e é feita a composição no *CarrierManager*. Para guardar as diferentes *Carriers*, optamos por utilizar um *Map* com o nome da *Carrier* como chave, pois permite um acesso rápido ao objeto de modo a aplicar cálculos como o lucro da transportadora.

Cada transportadora define valores para diferentes tipos de taxas (*small,medium,big*), que variam de acordo com as dimensões da encomenda. Além disso, cada encomenda está sujeita a um imposto sobre o valor acrescentado (IVA) de 13%. O lucro de cada transportadora é calculado multiplicando o preço do item pela respetiva taxa com isenção de IVA.

Os artigos, no contexto do problema, são uma entidade que visa poder ser expandida com facilidade. Com isso em mente, o grupo decidiu criar uma hierarquia com a classe abstrata “Item” e respetivas subclasses “Tshirt”, “Sneaker” e “Bag”. O uso de uma classe abstrata provou ser bastante útil, na medida em que minimizou o volume de código a ser escrito, visto que as subclasses são vistas na mesma como “Itens”, não tendo assim de especificar para todas as operações a ser realizadas sobre os artigos.

Semelhante às transportadoras, os “Itens” estão compostos na classe “ItemManager” em dois mapas: Um para artigos à venda, e outro para artigos do sistema. A classe ainda possui um apontador para a respetiva “Carrier”, de modo a facilitar os cálculos das taxas em operações futuras. Ainda na medida de eficiência, os “Itens” possuem referências dos utilizadores antigos e do utilizador atual sob a forma de um Inteiro (ID do “User”). Esta pequena referencia ajuda particularmente na operação de devolução de uma encomenda, e também faz com que seja mais

rápido chegar ao dono do “Item” em questão.

Os “Users” são a classe com mais interações. Estes possuem referências para “Items”, e também possuem as “Bills”. À semelhança das outras classes, os “Users” também possuem um gestor chamado “UserManager”. Apesar de não possuírem uma referência direta às encomendas onde participam, os utilizadores são associados às várias encomendas onde participam através das faturas.

As estatísticas dos utilizadores são calculadas através das suas “Bills”, de forma a reduzir o número de variáveis da classe “User”. O Model possui um apontador para o utilizador que possui sessão iniciada no sistema, e é desta forma que a aplicação sabe quem é o autor da encomenda e, naturalmente, procede ao tratamento correto da mesma.

Como já foi referido anteriormente, existe um “User” administrador que tem acesso privilegiado às funções do sistema. A intenção principal do grupo com a distinção entre utilizadores e “Admin” atuaria como forma de proteção de dados.

*Order* corresponde à classe responsável pela implementação de uma encomenda no sistema. Possui referências para as classes *Item* e para *User*. *Order*, como também se confirma em *User*, possui um *Manager* dedicado, o **Order Manager**. Integra, também, uma estrutura auxiliar *CarrierHelper*, que corresponde a um mapa que contém informações acerca de transportadoras.

Esta classe é responsável por guardar informações acerca de uma encomenda, como por exemplo o seu conteúdo/ a sua coleção (em *Items*), os vendedores e o comprador associados, assim como o seu estado (*Pending*, *Finished*, *Dispatched*), tamanho (*Little*, *Medium*, *Big*), data da encomenda, o preço total (em *Items* mais taxas), entre outros.

Em termos de estado da encomenda, no momento em que se passam três dias após o pedido e o seu estado seja *finished*, é gerada uma fatura tanto para o utilizador como para o comprador. O vendedor recebe o dinheiro resultante da transação, e a transportadora a respetiva comissão. São, também, removidos os itens associados à encomenda do vendedor, e adicionados ao ”inventário” do utilizador.

Em relação a *features* relacionadas com encomendas, é possível devolver uma encomenda (caso não se tenham passado mais de 16 dias após a data da criação da encomenda). Se for possível, os itens são devolvidos ao vendedor, e a fatura resultante da encomenda é apagada.

Em suma, *Order* tem como objetivo principal permitir que sejam transferidos itens de utilizador para utilizador, assim como também saber quanto é que cada vendedor ganha em cada encomenda, o que será útil para o cálculo de estatísticas.

A nível do cálculo de estatísticas sobre o estado do programa, introduzimos a interface **Querier**, a partir da qual introduzimos classes (todas estas queries) responsáveis por efetuar tais cálculos. A maioria destas classes utiliza uma cópia dos mapas dos *managers* , (embora as queries apenas consultem dados) para mantermos este projeto fiel ao paradigma de Programação Orientada aos Objetos. Estas cópias advêm de um método definido em *Model* . As queries são executadas via método *querierExecution*, definido em *Controller* . Embora cada *query* de certa forma corresponda a um método (o que a executa propriamente), optamos por criar uma classe para cada uma das *queries*. No geral, cada *query* segue o seguinte formato:

- Recebe uma cópia de um mapa do *manager* desejado e manipula datas (caso a *query* dependa de um time frame);
- Executa a *query*, utilizando travessias e comparações sob as cópias dos *maps*;

A escolha da interface serve unicamente para etiquetar a classe que a implementa, de modo que esta seja facilmente identificada como uma calculadora de estatísticas (neste caso).

A escolha desta arquitetura para a definição de *queries*, permite que sejam adicionadas novas queries de maneira bastante rápida e intuitiva, pois não é exigida grande alteração/criação de código.

**Bill** trata-se de uma entidade responsável pela criação de faturas, que exteriormente acede a outras duas entidades, sendo estas *Item* (que acede via o seu respetivo mapa em Item Manager) e *Order*. Esta classe tem a sua implementação simplificada, onde apenas se realizam operações de cálculo de taxas e manipulação de itens. *Order* tem uma relação de agregação com *Bill*, i.e., uma *Order* existe sem *Bill*, mas uma fatura nunca pode ser gerada sem uma encomenda associada. Da mesma forma, *Bill* também é dependente de *Item*, pois uma fatura tem que objetivamente ter produtos associados a esta. Esta escolha de relações entre entidades foi implementada no âmbito de “obrigar” esta classe a não ter valores vazios/inválidos, i.e., cada entidade que depende de outra necessariamente apenas será criada caso tenha de facto valores associados à mesma.

Em relação à implementação da noção de *Premium* (existência de produtos e transportadoras Premium), optamos pela utilização de uma interface designada *Premium*, responsável por agrupar todo o tipo de Classes que siga esse padrão/característica.

Para além disso, como a implementação de uma interface só por si não seria suficiente, foram criadas subclasses de outras classes não-*Premium*, que no seu todo são iguais às classes de onde surgem, mas que possuem fórmulas de cálculo de preços distintas.

O principal objetivo da criação de uma interface e de subclasses *Premium* é **minimizar** a produção de código e promover a sua reutilização.

Finalmente, o uso de *exceptions* permitiu-nos identificar e tratar facilmente eventuais erros no sistema. A implementação de classes que seguiam o padrão de exceções facilitou o *debug* do nosso código, tornando-o mais fácil de ser interpretado.

## Capítulo 6

# Arquitetura do Projeto

A nível arquitetural, como referido ao longo da descrição das classes, o nosso projeto orienta-se sob o padrão de arquitetura de software MVC (*Model-View-Controller*).

O *Model* representa a lógica de e os dados da *app*. A *View* é responsável pela apresentação dos dados do modelo aos utilizadores, enquanto que *Controller* atua como um intermediário entre os dois.

O objetivo geral do MVC é separar os dados, a lógica e a interface em componentes distintos para facilitar o desenvolvimento, a manutenção e a escalabilidade do programa. Para além disso, este padrão é importante para os factores encapsulamento e modularidade, que são muito valorizados nesta Unidade Curricular.

# Capítulo 7

## Diagrama de Classes

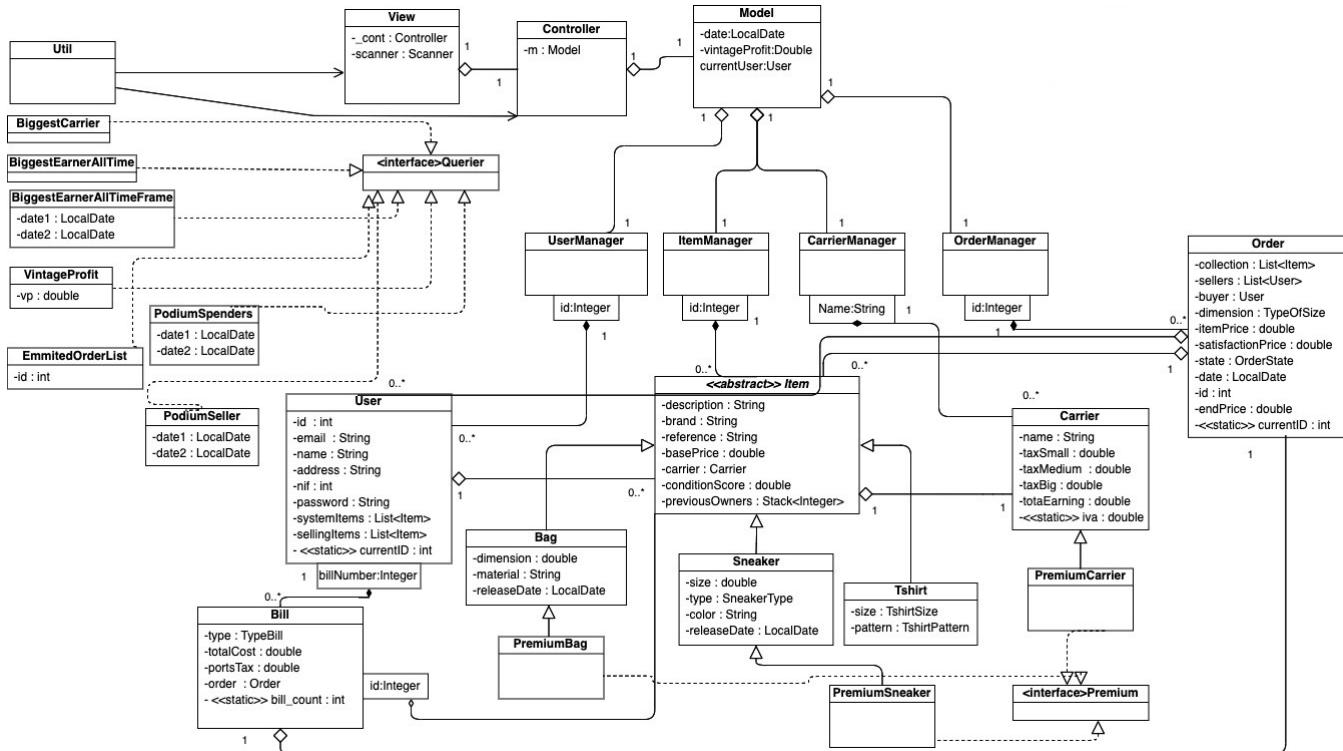


Figura 7.1: Diagrama de Classes.

**Nota:** A classe `SystemDate` foi omitida do Diagrama.

# **Capítulo 8**

## **Conclusão**

Ao finalizarmos este projeto prático da Unidade Curricular Programação Orientada aos Objetos (POO), podemos afirmar que foram utilizadas metodologias e abordagens adequadas para resolver o problema proposto. Com o desenvolvimento completo das funcionalidades solicitadas pelos Docentes, consideramos que o trabalho foi bem executado e atingiu todos os objetivos.

Durante o processo de desenvolvimento do sistema/aplicação, foi necessário aplicar conceitos teóricos de POO na prática, o que possibilitou um melhor entendimento da linguagem utilizada (Java) e da forma como as classes e objetos podem ser implementados de maneira eficiente. Além disso, a adoção de um modelo de boa prática de desenvolvimento de sistemas de software (MVC) permitiu a entrega de um produto final que nos parece adequado face aos requisitos desta UC.