Relatório Trabalho Prático POO - Grupo 47

Francisco Macedo Ferreira (A100660)
Júlio José Medeiros Pereira Pinto (A100742)
Rui Lopes Martins (A100643)

14 de maio de 2023



Conteúdo

1 Introdução

Este relatório tem como intuito abordar o projeto da UC Programação Orientada a Objetos do ano letivo 2022/2023.

Falaremos das decisões tomadas e funcionalidades implementadas pelo grupo e o método de raciocínio para desenvolvimento do projeto final. O objetivo deste trabalho foi desenvolver um sistema de *marketplace*, ***Vintage***, que permite a compra e venda de produtos novos ou usados.

2 Interface

A nossa interface segue mais ou menos o estilo de um terminal emulado. Ações são executadas a partir de comandos tal como: ***'item create'*** ou ***'carrier create DHL'***.

```
Welcome to the Vintage Marketplace!

Choose which view you want to use.
Available views: User (user), Admin (admin) or Exit (exit)
> admin
Admin Control View - Type 'help' to see available commands
> help
Available commands in admin view:
 - carrier <subcommand> - Parcel carrier related commands
 - help - Show this help
 - logout - Logs out of the current view
 - time <subcommand> - Time related commands
 - user <subcommand> - User related commands
> ...
```

(Inputs do utilizador são marcados pelo símbolo >)

Escolhemos esta solução já que a achamos mais intuitiva e fácil de navegar, em oposição a menus dentro de menus, como tínhamos explorado nas aulas da UC.

2.1 Conceito de *View*

De forma a separar utilizadores de administradores (e, possivelmente, outros *roles*), decidimos criar e implementar o conceito de *View*. Uma *View*, genericamente, é apenas uma função *run()*.

As *views user* e *admin* implementadas são necessárias para separar ações relativas ao programa numa globalidade (mudança de tempo, controlo de transportadoras) e ações relativas ao programa para um determinado utilizador (controlar artigos à venda, comprar artigos, etc). Estas duas são estendidas de uma *view* especializada em execução de comandos que moldam a interação com a aplicação. Cada uma tem uma lista de comandos únicos que podem ser executados com contexto da *view* atual.

Nem todas as *views* são especializadas em execução de comandos. Por exemplo, a funcionalidade de login e registo das contas de utilizadores é uma *view* que pede ao utilizador informações necessárias para execução da *view* de *users*:

```
Available views: User (user), Admin (admin) or Exit (exit)
> user
[USER] Enter the email to login (or 'cancel'): francisco@gmail.com
[USER] User with email francisco@gmail.com does not exist.
[USER] Do you want to register that email (y/n)? y
[USER] Creating user with email francisco@gmail.com...
[USER] Enter your username: francisco
[USER] Enter your name: Francisco
[USER] Enter your address: Francisco Street
[USER] Enter your tax number: 199356100
[Francisco] Logged in as Francisco (francisco@gmail.com)
[Francisco] > ...
```

2.2 Comandos

A interação com o programa tem como pilar a execução de comandos, logo, é necessário um sistema robusto e flexível para cumprir tal requisito.

A cada chamada de execução, a implementação do comando recebe um *array* de *Strings* populado com os argumentos escritos pelo utilizador:

```
> comando arg1 arg2 arg3
```

Resultaria em:

```
void Command.execute(..., [arg1, arg2, arg3]);
```

Esta separação é feita pelo *CommandManager* que recebe o comando cru (*"comando arg1 arg2 arg3"*), encontra o comando *"comando"*, *tokeniza* os parâmetros do comando e chama a função em questão para execução.

Este método *execute()* também recebe um *Logger*, para escrita no terminal, e um *InputPrompter* para pedir input ao utilizador (exemplo: pedir informações da conta ao registar um novo utilizador). Isto será útil na secção da automatização.

Comandos podem ser registados em qualquer *view*, exceto em casos que seja necessário fornecer, por exemplo, o utilizador *logado*. Nesse caso, os comandos recebem no construtor uma *UserView* (*view* do utilizador) que tem métodos para se saber qual o utilizador *logado* atualmente.

2.2.1 Subcomandos

Comandos podem também ter subcomandos:

```
> comando subcomando arg1 arg2 arg3
```

Estes subcomandos são uma extensão de um comando, que, à semelhança do *CommandManager*, encontra o subcomando pretendido numa lista de subcomandos registados no comando e *tokeniza* os parâmetros posteriores.

Subcomandos podem ser também registados em subcomandos, sendo, por exemplo, possível o seguinte comando: *item stock add ...* (*add* é um subcomando de *stock*, que é um subcomando de *item*).

2.2.2 Help (Comando de Ajuda)

Para clarificação de quais comandos o utilizador pode executar numa *view*, existe o comando *help*, que enumera todos os comandos disponíveis. Este comando recebe uma interface *CommandRepository* que tem o propósito de fornecer os tais comandos disponíveis, tal que o *CommandManager* e a classe de subcomandos (*ParentCommand)* a implementa.

```
> help
Available commands in user view:
 - cart <subcommand> - Shopping Cart commands
 - help - Show this help
 - item <subcommand> - Item management commands
 - logout - Logs out of the current view
 - order <subcommand> - Order commands
 - time - Displays the current time
 - user <subcommand> - User related commands
```

Este comando pode ser registado em qualquer *CommandRepository*. Isto dá-nos a flexibilidade de existir um subcomando de ajuda em todos os subcomandos com muita facilidade:

```
> cart help
Available commands in cart:
 - cart add <itemId> - Adds the item given to the user's shopping cart
 - cart help - Show this help
 - cart list - Lists the items in shopping cart
 - cart order (customId) - Makes an order out of the current shopping cart
 - cart remove <item> - Remove the given item from the shopping cart
```

3 Arquitetura

A arquitetura do programa foi um quesito bem investido no trabalho. Não seguimos explicitamente as nomenclaturas do modelo de arquitetura MVC ensinado nas aulas devido também à escolha de existência de comandos. Ainda assim, o conceito arquitetural foi todo implementado seguindo as regras de MVC.

3.1 Nomenclaturas usadas

Cada modelo segue as seguintes nomenclaturas e camadas de abstração:

- **Modelo:** Classe onde se guarda informação do modelo
- **ModeloManager:** Repositório que guarda e faz gestão de todos os modelos em memória
- **ModeloFactory:** Utilitário para criação de modelos com muita informação (ex: artigos, encomendas)
- **ModeloController:** Lógica para modificação de informação de modelos

De modo geral na aplicação, o ***Controller*** é um *facade* do ***Factory*** e do ***Manager*** que contem todos os métodos necessários para controlo completo do modelo correspondente.

Desta forma, se quisermos modificar alguma informação de um modelo:

- Requisitamos o modelo atual ao *Controller*.
- O *Controller* acede ao *Manager* e encontra o modelo pelo identificador.
- O *Manager* retorna uma cópia do modelo na memória.
- Usamos o *Controller* para alterar informações.
- O *Controller* altera o valor no modelo e atualiza o modelo anterior presente no *Manager*.

Exemplo do fluxo de código no ***ItemController*** ao atualizar de stock de um artigo:

```
Item item = this.itemManager.get(itemId);
...
item.setCurrentStock(newStock);
this.itemManager.updateItem(itemId, item);
```

Nota: Algumas entidades/componentes mais simples podem não ter todas estas camadas.

Todos estes ***Controllers*** são incluídos num outro *facade* geral, chamado ***Vintage***. Este *facade* geral é a classe que é entregue às *views* (e subsequentemente para os comandos) para modificação do programa num nível mais alto.

Apresentamos aqui um diagrama de classes da arquitetura com omissões:

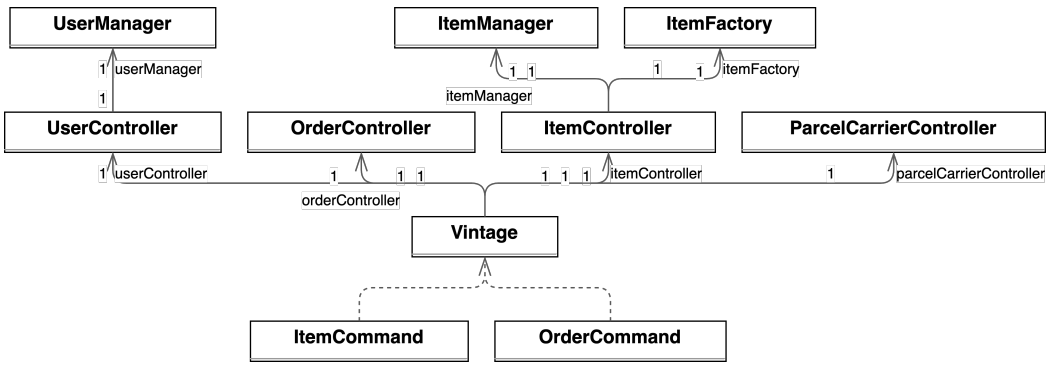

Figura 1: Arquitetura geral do programa

4 Modelos

O programa tem presente vários modelos com as suas abstrações únicas. Todos os modelos foram implementados de forma a se manter uma expansão de funcionalidades sustentável do programa.

4.1 Item (Artigos)

Propuseram os seguintes tipos de artigos: *Mala*, *Sapatilha*, *Tshirt* e artigos *premium* de *Mala* e *Sapatilha*. Todos esses tipos de artigos estendem uma classe *Item* que contém informações comuns de todos os artigos (condição, descrição, marca, preço base, transportador).

As especializações de artigos, alteram um método abstrato da classe *Item*: *getPriceCorrection*. Este método recebe o ano atual da simulação e é usado para calcular o preço final do artigo, com, por exemplo, os ajustes de desgaste dependendo do tipo de artigo. Também incluem na classe informações adicionais de acordo com o tipo de artigo (ex: Mala tem dimensão).

Os artigos *premium* são especializações do tipo de artigo que são premium de, e modificam o método de correção de preço. Adicionam também mais informações se necessário.

Com isto, temos o seguinte diagrama de classes:

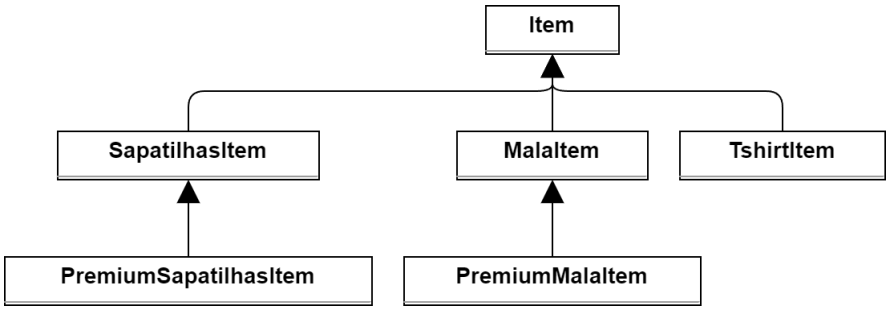

Figura 2: Diagrama de classes dos artigos

Para referência de, por exemplo, o vendedor ou o transportador do artigo, é guardado um id único dessa entidade (*UUID* para vendedor e *String* para nome único do transportador) e não a referência do objeto na memória em si. Com isto, evitamos problemas de referências penduradas, possível dessincronização de objetos e necessidade de outras entidades precisarem de estar presentes em memória. Este pensamento foi aplicado no resto dos modelos do programa.

4.1.1 Propriedades de Artigos

Para fácil reutilização de código, na criação e inspeção de artigos, tipos de artigos têm uma lista de propriedades necessárias para a sua criação e inspeção. Por exemplo, uma sapatilha tem as propriedades: tamanho, tem cordão, cor, coleção e as propriedades comuns em todos os tipos (vendedor, id, condição, stock atual, etc).

Desta forma, para criação de artigos, basta termos a lista de propriedades necessárias do tipo do artigo, e questioná-las ao utilizador:

```
> item create
Choose the type of item you want to create:
'Mala', 'Sapatilhas', 'TShirt', 'Mala Premium', 'Sapatilhas Premium'
Insert the item type: Mala
Insert initial item stock (1-20): 2
Insert the item condition (new, used): new
Insert the item description: Mala Top
Insert the item brand: Marca Fixe
Insert the item base price: 9.90
Insert the parcel carrier to use: DHL
Insert the bag dimension area: 15
Insert the item material: Pele
Insert the item collection year: 2023
Insert the item depreciation rate over years in percentage (0-100): 0
Registered item Mala (5NL-IX1) successfully.
```

Na criação, um dicionário de propriedades é passado para o ***ItemFactory***. Essa classe é responsável por passar as propriedades ao construtor de classe do artigo correspondente e retornar o item criado.

Ao inspecionar o artigo, todas as propriedades obrigatórias do tipo de artigo são informadas:

```
> item inspect 5NL-IX1
Item 5NL-IX1 details:
 - STOCK: 2
 - ITEM CONDITION: New
 - DESCRIPTION: Mala Top
 - BRAND: Marca Fixe
 - BASE PRICE: 9.90
 - PARCEL CARRIER NAME: DHL
 - DIMENSION AREA: 15
 - MATERIAL: Pele
 - COLLECTION YEAR: 2023
 - DEPRECIATION RATE OVER YEARS: 0
```

Isto permite-nos reutilizar propriedades de vários itens ao fazer as perguntas e ao inspecionar.

Adição de novos tipos de artigos é facilitada, já que, não será necessário modificação da implementação dos comandos, apenas será necessário adicionar o texto de pergunta das novas propriedades e a forma de conversão de texto para o tipo de objeto necessário (ex: stock inicial em texto ser convertido para inteiro).

4.2 Parcel Carrier (Transportadores)

Existem dois tipos de transportadores: normal e premium.

- Transportadores normais só conseguem enviar artigos não-premium.
- Transportadores premium só conseguem enviar artigos premium.

Nota: Na criação de artigos, é necessário que haja transportadores compatíveis com o tipo de artigo criado.

No código, esta ideia foi implementada com o seguinte diagrama de classes:

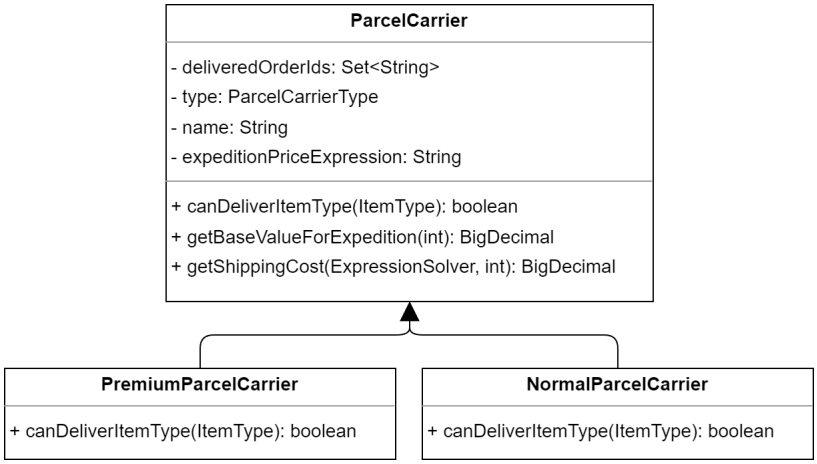

Figura 3: Diagrama de classes dos transportadores

A classe abstrata *ParcelCarrier* tem informações do nome da transportadora, da expressão customizável do preço da transportadora, do tipo, e uma lista de ids de encomendas entregues (para cálculo de estatísticas) que é populado na criação de uma encomenda.

NormalParcelCarrier e *PremiumParcelCarrier* dão *override* a um método abstrato que determina se um tal tipo de artigo pode ser enviado por este transportador ou não.

4.2.1 Fórmula customizável de preço

Na criação do transportador, é perguntado ao utilizador se pretende escolher uma fórmula para o preço diferente da padrão.

```
> carrier create DHL
Do you want to set a custom expedition price expression? (y/n)
The default one is: basePrice * (1 + tax) * 0.9
Boolean > y
Please enter the expression using the following variables:
'basePrice', 'tax'
Expression > basePrice * (1 + tax) * 0.5
Parcel carrier DHL created successfully.
```

Esta fórmula é validada e calculada quando necessária a partir de uma biblioteca *Exp4j* (abstraída pela interface *ExpressionSolver*). A fórmula pode conter qualquer expressão matemática com recurso às variáveis pré-definidas *basePrice* (preço tendo em conta o número de artigos a transportar) e *tax* (taxa global de transporte do Vintage).

4.3 User (Contas de utilizador)

De modo a guardar informação de utilizadores existe uma classe *User* que armazena nome, endereço, email e número de contribuinte do utilizador. Esta também guarda uma lista de artigos que o utilizador tem à venda, uma lista de artigos que estão no seu carrinho (para futura finalização de encomenda), uma lista de encomendas finalizadas e uma lista de encomendas em que outro cliente comprou um artigo deste. As listas, como referido acima, guardam identificadores e não referências de objetos.

Estas listas são modificadas, como explicado acima, a partir do ***UserController***.

4.4 Order (Encomendas)

Encomendas são criadas no momento da finalização. Depois de artigos serem adicionados ao carrinho (*cart add <item>*), estes podem ser encomendados (caso haja stock) a partir do comando *cart order*.

```
> cart order
Order summary:
 - 9,99: Item AAA-AAA (Mala) Mala Topzona
 - 0,50: Item AAA-AAA Condition Satisfaction Service Tax
 - 4,99: Item BBB-BBB (Sapatilha) Nike Air Usada
 - 0,25: Item BBB-BBB Condition Satisfaction Service Tax
 - 0,48: DHL Shipping Cost (2 items)
Total: 16,21

Confirm the purchase (y/n): y

Order (ORD-BSRS08) created successfully.
Your order will be delivered soon.
```

Uma encomenda guarda informação dos artigos encomendados numa classe (por composição), do comprador, de linhas da fatura, da data de encomenda, do preço total e do estado atual da encomenda.

Ao fazer a encomenda, artigos que têm transportadores comuns são separados para cálculo do custo de entrega de tal transportador. Após a finalização, o stock dos itens é diminuído em um, modelos que tenham referências a encomendas (ex: lista de encomendas entregues pela transportadora) são atualizados e o carrinho do cliente é limpo.

Nota: Por simplicidade, não é possível encomendar mais que um artigo igual numa só encomenda. Isso é possível fazendo várias encomendas separadas.

Passado um determinado número de dias da encomenda (definido 3 dias no nosso programa), encomendas que estão no estado *ORDERED* passam para *DELIVERED*. Aí, o cliente tem um

determinado número de dias para proceder à devolução da encomenda na totalidade (por simplicidade, não é possível devolver apenas alguns artigos da encomenda). Caso o utilizador a devolva, o stock dos artigos encomendados é aumentado em um e o estado da encomenda passa para *RETURNED*.

4.4.1 OrderedItem (Artigos encomendados)

Para guardar informação dos artigos encomendados, a encomenda guarda, por composição, *OrderedItem*s que contem informação do id do item, do preço do item no momento da encomenda e o transportador que foi usado.

Este é usado para manter informação de artigos encomendados, sem se preocupar com possíveis modificações no modelo do artigo.

4.4.2 InvoiceLine (Linha de fatura)

De forma a comunicar ao utilizador como o preço da encomenda está a ser calculado, a encomenda guarda, por composição também, uma lista de *InvoiceLine*s.

Cada uma destas guarda um preço e um texto e representa as linhas apresentadas na confirmação da encomenda mostrada acima.

Para cálculos estatísticos, esta classe foi estendida em vários tipos de linha de fatura, para, por exemplo, ser possível identificar quanto é que certa transportadora ganhou em taxas numa encomenda (*ParcelShipmentCostInvoiceLine*).

Segue aqui o diagrama de classes das linhas de fatura:

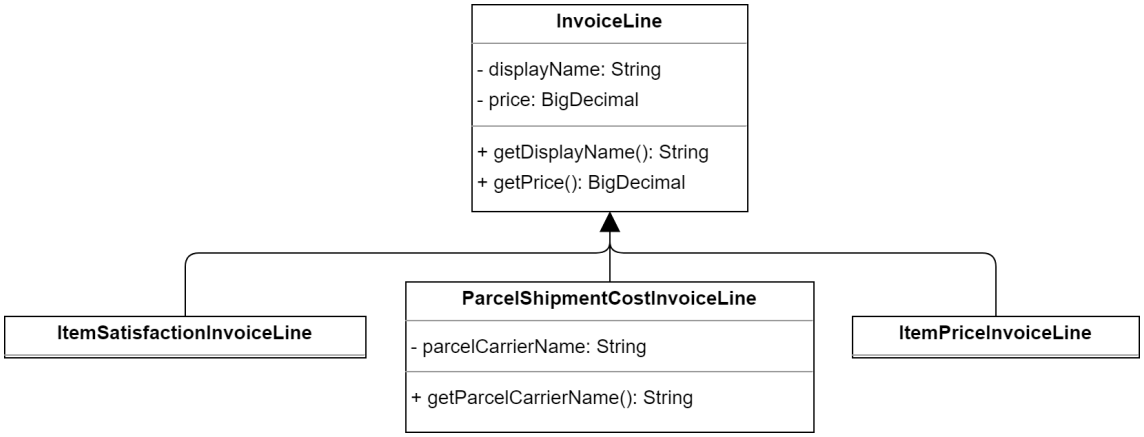

Figura 4: Diagrama de classes das linhas de fatura

5 Funcionalidades

5.1 Scripting (Automatização da simulação)

No início do programa são carregados *scripts* a partir de um ficheiro *scripts.txt*. Cada linha do ficheiro, isto é, cada *script*, tem o formato: '[data], [ação]'.

A forma como automatizamos a simulação foi fazer com que sejam executados comandos (da mesma forma que um utilizador/administrador faria) em determinadas *views*, numa data.

Isto deu-nos a flexibilidade de todas as ações que um utilizador pode fazer também estarem disponíveis no ambiente da automatização sem quaisquer modificações no sistema de *scripts*.

Para já só temos um 'tipo' de *script*, que é o que executa comandos, mas outros podem ser implementados futuramente para, por exemplo, executar ações mais críticas que não estão disponíveis por comandos.

Exemplo de um *scripts.txt*:

```
02/01/2023,admin,carrier create DHL < n
02/01/2023,admin,user register chico@gmail.com < chico < Chico < ...
```

```
02/01/2023,admin,user register buyer@gmail.com < buyer < Buyer < ...
02/01/2023,user:chico@gmail.com,item create AAA-AAA < Mala < 5 < ...
02/01/2023,user:buyer@gmail.com,cart add AAA-BBB < y
```

Pegando na segunda linha, a lógica de execução de um *script* de comando é a seguinte:

- O *script* recebe 'admin,user register chico@gmail.com < chico < Chico < ...'.
- Deteta que tem que executar o comando 'user register chico@gmail.com' na *view 'admin'*, fornecendo as linhas de *input* 'chico', 'Chico', etc, sequencialmente.
- Este fornecimento de linhas é feito por uma nova implementação de um *InputPrompter* (*BufferedInputPrompter)* que o comando recebe. Em vez de usar o *STDIN* como forma de saber *input*, pega na informação com recurso a um *Iterator*, que avança a cada requisição de input.
- De forma a não poluir a interface, uma nova implementação de *Logger* é passado. Esta nova implementação (*OnlyWarnLogger*) é um *decorator* do *Logger* tradicional, que apenas imprime *warnings*.

Nota: *View* de administração é identificada a partir de 'admin'. Comandos podem ser executados em *views* de utilizadores a partir da sintaxe 'user:<email>'.

A execução do *script* acima, tem, então, o seguinte *output*:

```
[cart add AAA-BBB < y - WARN] Item with the id AAA-BBB does not exist.
[02/01/2023] Ran sucessfully 5 scripts.
```

A não existência artigo AAA-BBB é propositaada, para demonstração das funcionalidades mencionadas dos *scripts*.

5.2 Avançar do tempo

Para avançar o tempo, na *view* de *admin* existe um comando *time*. Ao chamar '*time jump <dias>*', o tempo do programa é avançado em tantos dias.

Ao avançar o tempo, estados de encomendas são alterados e *scripts* dos dias avançados são executados. Estas ações são chamadas pelo *facade Vintage* geral desta forma:

```
this.orderController.notifyTimeChange(..., newDate);
this.scriptController.notifyTimeChange(..., previousDate, newDate);
```

O avançar do tempo é feito dia a dia, para ser possível, por exemplo, por *scripts*, fazer uma encomenda e devolvê-la passado 3 dias.

```
> time jump 10
[02/01/2023] Ran sucessfully 7 scripts.
[08/01/2023] Ran sucessfully 5 scripts.
[09/01/2023] Delivered 2 orders.
[09/01/2023] Ran sucessfully 3 scripts.
[10/01/2023] Ran sucessfully 2 scripts.
Time jumped from 01/01/2023 to 11/01/2023.
```

5.3 Persistência em disco do estado do programa

Para guardar o estado do programa em disco, recorremos à funcionalidade de serialização de classes do Java. Um ***PersistentManager*** é responsável por criar o ficheiro binário e guardar, num dicionário, as classes que mantém informação do programa.

Não é salva a classe da aplicação completa de uma só vez para minimização do tamanho e simplificação do ficheiro. Isto também torna a persistência mais robusta a alterações na funcionalidade do programa.

Caso uma classe de um modelo seja alterada (desserialização não será possível com sucesso), aparece uma mensagem de binário corrompido e o programa começa no estado inicial.

Deste modo, os ***Managers*** (classes que guardam as entidades) são guardados em ficheiro desta forma:

```
persistentManager.addReferenceToSave("userManager", this.userManager);
persistentManager.addReferenceToSave("parcelCarrierManager", ...);
persistentManager.addReferenceToSave("itemManager", this.itemManager);
persistentManager.addReferenceToSave("orderManager", this.orderManager);
persistentManager.addReferenceToSave("timeManager", this.timeManager);
persistentManager.save();
```

E carregados desta forma:

```
Map<String, Object> data = persistentManager.loadPersistentData();
this.userManager = data.getOrDefault("userManager", ...);
this.parcelCarrierManager = data.getOrDefault(...);
this.itemManager = data.getOrDefault("itemManager", new ItemManager());
this.orderManager = data.getOrDefault("orderManager", ...);
this.timeManager = data.getOrDefault("timeManager", ...);
```

O carregamento é feito no início do programa, e o *salvamento* é feito na saída do programa. Por conveniência, o *salvamento* não é feito se o programa fechar inesperadamente.

5.4 Estatísticas

StatsManager é a classe responsável na implementação de resolução de estatísticas, tais como: quem foi o maior vendedor num certo espaço de tempo ou quem foi a transportadora que lucrou mais. Esta classe acessa informação de outros entidades utilizando o *facade* geral *Vintage* e calcula todas as estatísticas referidas previamente.

Todas as estatísticas pedidas foram implementadas:

1. Qual é o vendedor que mais faturou num período ou desde sempre
2. Qual o transportador com maior volume de faturação
3. Listar as encomendas emitidas por um vendedor
4. Fornecer uma ordenação dos maiores compradores/vendedores do sistema durante um período a determinar
5. Determinar quanto dinheiro ganhou o Vintage no seu funcionamento

5.4.1 Qual é o vendedor que mais faturou num período ou desde sempre

O ***StatsManager*** acessa todos os utilizadores e dentro de cada utilizador as encomendas entregues e soma o preço dos artigos contido nas mesmas.

Execução do comando:

```
> stats topseller
Top seller in specified date range is Joao with 142,00 made in sales.
```

Também é possível especificar um período de tempo com '*stats topseller <from> <to>*'.

Não é uma solução muito eficiente, mas não esperamos que a aplicação seja utilizada em grandes escalas. Não optámos por guardar o total de dinheiro ganho dentro do modelo do utilizador para evitar possíveis discrepâncias de dados e simplicidade de implementação, devido à necessidade de cálculo de faturações em períodos de tempo.

5.4.2 Qual o transportador com maior volume de faturação

Da mesma forma que a estatística anterior, para o cálculo do transportador com maior volume de faturação, acessamos todos os transportadores, dentro dos transportadores todas as encomendas relacionadas e são somadas as taxas de envio encontradas.

Essas taxas de envio são encontradas internamente pelo modelo *Order* a partir das *InvoiceLine*s.

Execução do comando:

```
> stats topparcelcarrier
Top parcel carrier is DHL with 12,06 received.
```

5.4.3 Listar as encomendas emitidas por um vendedor

Para esta, bastou-nos acessar a lista de encomendas enviadas por tal vendedor.

Execução do comando:

```
> order listseller chico
Orders containing seller Chico:
 - ORD-BSRS08
> order inspect ORD-BSRS08
ORD-BSRS08 summary:
 - 9,99: Item AAA-AAA (Mala) Mala Topzona
  ...
```

5.4.4 Fornecer uma ordenação dos maiores compradores/vendedores do sistema durante um período a determinar

Usando a mesma lógica da primeira estatística, conseguimos calcular a lista de maiores compradores e vendedores num determinado período.

Execução do comando sem período:

```
> stats toplist 3
Top 3 buyers in specified date range:
  #1 Maria - 98,75
  #2 Julio - 66,91
  #3 Rui - 52,00

Top 3 sellers in specified date range:
  #1 Joao - 142,00
  #2 Chico - 57,92
  #3 Rui - 15,00
```

Execução do comando com período entre 01/01/2023 e 10/01/2023:

```
> stats toplist 3 01/01/2023 10/01/2023
Top 3 buyers in specified date range:
  #1 Maria - 98,75
  #2 Julio - 60,91
  #3 Rui - 0,00

Top 3 sellers in specified date range:
  #1 Joao - 97,00
  #2 Chico - 42,92
  #3 Rui - 15,00
```

5.4.5 Determinar quanto dinheiro ganhou o Vintage no seu funcionamento

Para o cálculo desta estatística, é percorrida a lista de todas as encomendas feitas e somados todos os preços de satisfação de artigos. Encomendas que foram devolvidas são filtradas neste processo.

A informação do preço de satisfação de artigos está presente nas linhas da fatura (do tipo *ItemSatisfactionInvoiceLine*) e é calculada no modelo da encomenda, internamente.

Execução do comando:

```
> stats vintage
Total gained in Vintage from taxes: 4,50
```

6 Outros

6.1 Gradle Package Manager

De forma a facilitar o controlo de dependências e compilação do programa, optamos por usar Gradle devido à sua simplicidade e flexibilidade. Com isto, conseguimos adicionar várias dependências no projeto, tal como *Exp4j* (referida acima nos Transportadores), *Mockito*, e *JUnit* com muita facilidade. O Gradle também foi responsável por rodar os testes unitários ao longo do desenvolvimento do projeto.

6.2 Testes unitários

Ao longo do projeto fomos fazendo alguns testes unitários, que foram especialmente muito úteis enquanto que a interface gráfica não estava completamente desenvolvida. Os testes foram desenvolvidos com recurso ao *JUnit* para asserções e execução dos testes e *Mockito* para simplificação de testes de componentes que dependem de outros.

Devido a questões de cumprimento do prazo de entrega, não conseguimos atingir o nosso objetivo de testar todos os componentes da aplicação mas os fundamentais estão a ser cobertos pelos testes desenvolvidos. De qualquer forma, os testes unitários serviram bem o desenvolvimento da aplicação, já que, mudanças nos componentes cobertos pelos testes, eram rapidamente validadas.

6.3 Documentação

Ao longo do projeto não conseguimos dedicar tanto tempo a esta vertente como pretendido. Apenas algumas decisões e funções no código estão propriamente documentadas. Mesmo assim consideramos que o código está bastante legível com métodos nomeados apropriadamente.

6.4 Diagrama de classes

No relatório foram apresentados diagramas de classes simplificados. Um diagrama de classes mais completo está presente na mesma pasta deste relatório.

7 Conclusão

Com o desenvolvimento deste trabalho, conseguimos consolidar o nosso conhecimento de POO e Java. Conceitos como hierarquia, encapsulamento e modularidade. Este permitiu-nos ficar mais familiarizados com o paradigma e com tecnologias orientadas a objetos, tecnologias estas que se demonstraram muito úteis no desenvolvimento de programas como o *Vintage*.

Perante os desafios apresentados, acreditamos que o nosso trabalho teve um desempenho muito satisfatório. Aplicando conhecimentos teóricos e práticos adquiridos durante a unidade curricular. A nossa dedicação e empenho resultaram, a nosso ver, num bom trabalho prático que nos irá, com certeza, ajudar em futuros desafios académicos e profissionais.