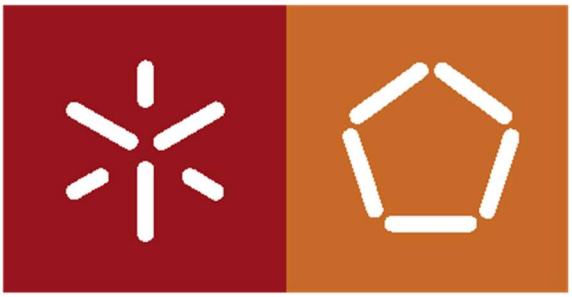


Laboratórios de Informática III

Mestrado Integrado em Engenharia Informática
2ºano - 2ºSemestre
Universidade do Minho

PROJECTO DE JAVA:
Desenvolvimento de uma aplicação Desktop usando o modelo MVC para a gestão das vendas de uma cadeia de distribuição

Grupo 37

a85367
Luís Francisco Mendes Lopes
a85367@alunos.uminho.pt

a83732
Gonçalo Rodrigues Pinto
a83732@alunos.uminho.pt

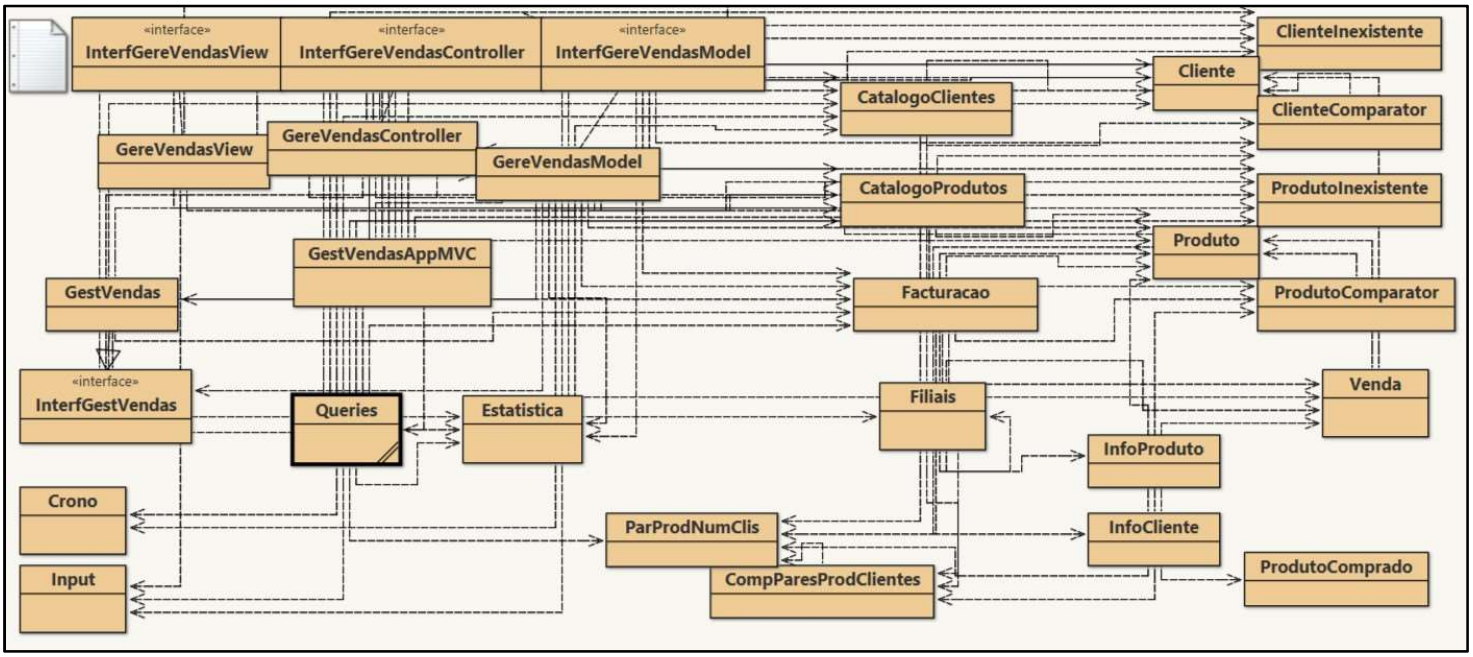
a83693
Rui Filipe Chaves
a83693@alunos.uminho.pt

Índice

Introdução

No 2º semestre do 2º ano do Curso de Engenharia Informática da Universidade do Minho, existe uma Unidade Curricular denominada por "Laboratórios de Informática III", que tem como objetivo ajudar os estudantes a consolidar experimentalmente os conhecimentos teóricos e práticos adquiridos em Unidades Curriculares anteriores e a apresentar aos alunos os desafios que se colocam a quem concebe e programa aplicações com grandes volumes de dados e com elevada complexidade algorítmica e estrutural. O presente projeto pretende implementar uma aplicação Desktop em Java, baseada na utilização de interfaces e das coleções de *Java Collections Framework* cujo objetivo é a realização de consultas interativas de informações relativas à gestão básica de uma cadeia de distribuição. Trata-se de um projeto em larga escala, onde a quantidade de dados a tratar é elevada. Este projeto foi desenvolvido tendo em conta os conceitos de modularidade e encapsulamento, criação de código reutilizável, uma escolha otimizada das estruturas de dados e testes de performance. O projeto pretende seguir o modelo *Model-View-Controller – MVC* para isso foram implementadas classes com esses mesmos nomes e as suas respetivas interfaces. O *model* consiste nos dados da aplicação, lógica e funções. A *view* encarangue-se da representação dos dados. O *controller* descodifica o que recebeu da *view* e converte em comandos para o *model*.

Classes implementadas



Catálogo Clientes

```
public class Cliente implements Serializable{
    private String codigo;
```

```
public class CatalogoClientes implements Serializable{
    private Map<Character,TreeSet<Cliente>> cc; /*Letra, Todos os clientes começados por essa letra sem repeticoes*/
```

A classe Cliente é uma classe simples, tem apenas uma variável privada que é o código do cliente a ele associado. Esta classe possui a declaração dos construtores por omissão, parametrizado e de cópia, dos métodos de acesso e alteração do seu estado interno e do método *clone*, *toString*, *equals*, *compareTo* e *hashcode*.

O nosso catálogo de clientes como a imagem acima o representa é uma associação entre carácter(chave) e um conjunto de clientes (valor). Decidimos implementar desta forma pois como os clientes começam sempre por uma letra, assim temos todos os mesmos clientes começados pela mesma letra mapeados pela mesma chave. Deste modo, fazemos com que todas as funções que envolvem procurar um código de cliente sejam mais eficientes, em vez de pesquisar numa grande associação, pesquisam em conjuntos mais pequenos. Esta classe além da declaração dos construtores por omissão, parametrizado e de cópia, dos métodos de acesso e alteração do seu estado interno e do método *clone*, *toString*, *equals* e *hashcode* possui outros métodos como adicionar um cliente a esta associação, verificar a existência de um dado cliente e retornar quantos clientes estão registados.

Associado aos clientes temos também mais duas classes o ClienteComparator e o ClienteInexistente que servem para ordenar os clientes nos conjuntos começados por uma dada letra e para alertar o utilizador que um dado cliente não está registado, respetivamente.

Catálogo Produtos

```
public class Produto implements Serializable{
    private String codigo;
```

```
public class CatalogoProdutos implements Serializable{
    private Map<Character,TreeSet<Produto>> cp; /*Letra, Todos os produtos começados por essa letra sem repeticoes*/
```

A classe Produto também é uma classe simples, apenas tem uma variável privada que é o código do produto a ele associado. Esta classe possui a declaração dos construtores por omissão, parametrizado e de cópia, dos métodos de acesso e alteração do seu estado interno e do método *clone*, *toString*, *equals*, *compareTo* e *hashcode*.

O nosso catálogo de produtos como a imagem acima o representa é uma associação entre carácter(chave) e um conjunto de produtos (valor). Decidimos implementar desta forma pois como os produtos começam sempre por uma letra e assim temos todos os mesmos produtos começados pela mesma letra mapeados pela mesma chave. Deste modo, fazemos com que todas as funções que envolvem procurar um código de produto sejam mais eficientes pois em vez de pesquisar num grande conjunto pesquisam num mais pequeno. Esta classe além da declaração dos construtores por omissão, parametrizado e de cópia, dos métodos de acesso e alteração do seu estado interno e do método *clone*, *toString*, *equals* e *hashcode* possui outros métodos como adicionar um produto a esta associação, verificar a existência de um dado produto e retornar quantos produtos estão registados.

Associado aos produtos temos também mais duas classes o ProdutoComparator e o ProdutoInexistente que servem para ordenar os produtos nos conjuntos começados por uma dada letra e para alertar o utilizador que um dado produto não existe, respetivamente.

Venda

Esta classe representa uma venda para isso tem como variáveis de instância um Cliente, um Produto, o tipo de venda (Normal ou promoção), a filial associada, a quantidade vendida, o mês que foi registada e o preço. Esta classe possui a declaração dos construtores por omissão, parametrizado e de cópia, dos métodos de acesso e alteração do seu estado interno e do método *clone*, *toString*, *equals.*

ParProdNumClis

Um dos objetivos da aplicação é permitir consultas de dados, consultas essas que envolvem por vezes, coleções de "pares de coisas" ou mesmo triplos, para isso, esta classe é utilizada para responder a tais consultas que envolvem pares. Esta classe é composta por uma *String* o código e um inteiro que representa o total ou quantidade associada. Esta classe possui a declaração dos construtores por omissão, parametrizado e de cópia, dos métodos de acesso e alteração do seu estado interno e do método *clone*, *toString*, *equals.* Esta classe tem associado a classe CompParesProdClientes que implementa a interface *Comparator<T>* onde ordena pelo inteiro e caso o inteiro seja igual pelo nome.

Faturação

```
public class Facturacao implements Serializable{
    private Map<Produto,Integer> fact; /*Codigo, unidades vendidas*/
    private int[][] tVendas; /*Total de unidades vendidas mensais, por tipo N ou P*/
    private double[][] tFF1; /*Total facturado mensal na Filial 1, por tipo N ou P*/
    private double[][] tFF2; /*Total facturado mensal na Filial 2, por tipo N ou P*/
    private double[][] tFF3; /*Total facturado mensal na Filial 3, por tipo N ou P*/
    private double tFacturado; /*Total Facturado global*/
```

A nossa faturação é composta por uma associação onde a chave é um Produto e o valor que mapeia é um inteiro que representa o número de unidades vendidas por esse produto; é composta também por quatro matrizes de 12x2, em que o número de linhas representa os meses do ano e as colunas revelam o modo, 0 se for normal e 1 se for promoção. Uma destas matrizes representa o total de unidades vendidas e as outras três referem o total faturado nas diferentes filiais, por fim existe um valor que indica quanto foi faturado globalmente. Esta classe além da declaração dos construtores por omissão, parametrizado e de cópia, dos métodos de acesso e alteração do seu estado interno e do método *clone*, *toString*, *equals* e *hashcode* possui outros métodos tais como: adicionar um produto ao conjunto acima referenciado; adicionar uma faturação, isto é, recebe uma venda retira os dados essenciais, preenche as suas variáveis com esses valores; indicar quantos produtos não foram comprados, para isso, percorre o conjunto e verifica quais os produtos estão a mapear o valor zero; dado um determinado mês existem métodos que retornam o valor presente nas matrizes respetivas, e por fim, um método que recebe um determinado valor e retorna uma lista ordenada de ParProdNumClis de tamanho do valor recebido dos produtos que mais unidades venderam.

Filiais

```
public class InfoProduto implements Serializable{
    private int unidadesVendidas; /*Unidades vendidas globais*/
    private int[] comprasM; /*Unidades vendidas separadas por mes*/
    private double[] totalFM; /*Total facturado separado por mes*/
```

```
public class ProdutoComprado implements Serializable{
    private int[] comprasN; /*Total de unidades vendidas num determinado mes em modo normal*/
    private int[] comprasP; /*Total de unidades vendidas num determinado mes em modo promo*/
    private int unidadesVendidas; /*Total de unidades vendidas*/
    private double totalPago; /*Total de facturado global*/
```

```
public class InfoCliente implements Serializable{
    private Map<Produto,ProdutoComprado> compras; /*Codigo do produto que o cliente comprou, informacao sobre as vendas produto */
    private int[] totalgasto; /*Total gasto mensal por um cliente*/
    private int[] comprasMesN; /*Total de compras em modo normal por mes*/
    private int[] comprasMesP; /*Total de compras em modo promo por mes*/
```

```
public class Filiais implements Serializable{
    private Map<Cliente,InfoCliente> comprasCliente; /*Codigo do cliente, Informacao das suas compras*/
    private Map<Produto,InfoProduto> comprasProduto; /*Codigo do produto, Informacao das suas vendas*/
    private Map<Cliente,Double> compradoresF1; /*Codigo do cliente, quanto gastou na filial1 */
    private Map<Cliente,Double> compradoresF2; /*Codigo do cliente, quanto gastou na filial2 */
    private Map<Cliente,Double> compradoresF3; /*Codigo do cliente, quanto gastou na filial3 */
```

ProdutoComprado é uma classe que representa a informação de um dado produto ter sido vendido, para isso possui como variáveis de instância dois *array's*, cada posição do *array* corresponde a um mês e a informação nela contida representa o total de unidades vendidas mensal, em modo normal; o outro array corresponde ao mesmo tipo de informação mas em modo promoção; detém também dois inteiros que representam o total de unidades vendidas e o total faturado global. Esta classe além de possuir a declaração dos construtores por omissão, parametrizado e de cópia, dos métodos de acesso e alteração do seu estado interno e do método *clone*, *toString*, *equals, hashcode,* possui igualmente um método para adicionar informação às suas variáveis e um método que retorna o número de unidades vendidas num determinado mês.

InfoCliente representa as compras de um cliente desta forma, possui uma associação que representa as suas compras onde o valor chave é o produto comprado que mapeia a informação da classe ProdutoComprado; possui três array's de 12 posições que correspondem aos meses do ano, um representa o total gasto por mês, outro o total de compras em modo normal e o último o total de compras em modo promoção. Esta classe além de possuir a declaração dos construtores por omissão, parametrizado e de cópia, dos métodos de acesso e alteração do seu estado interno e do método *clone*, *toString*, *equals, hashcode,* possui outros métodos além dos mencionados anteriormente, tais como: adicionar informação às suas variáveis recebendo uma venda; devolver quantas unidades comprou num determinado mês, tanto em modo normal como em modo promoção; retornar quantos produtos comprou num determinado mês; devolver quanto gastou num determinado mês; dado um produto e um mês devolver quantas unidades comprou nesse mês; retornar, de forma decrescente, pela quantidade comprada, um conjunto de ParProdNumClis dos produtos que um dado cliente comprou; permite verificar se comprou um dado produto, bem como, calcular quantos produtos comprou e quanto gastou num dado produto.

A classe InfoProduto como anteriormente foi referido representa as vendas de um produto. Possui como variáveis de instância o número de unidades vendidas, um array que representa o número de unidades vendidas mensal e outro array o total faturado num mês. Esta classe além de possuir a declaração dos construtores por omissão, parametrizado e de cópia, dos métodos de acesso e alteração do seu estado interno e do método *clone*, *toString*, *equals, hashcode*; tem, ainda outros métodos que permitem atualizar a informação das suas variáveis, retornar o número de unidades vendidas num determinado mês e o total faturado num determinado mês.

A classe Filiais é uma classe complexa que apresenta cinco associações: a primeira representa as compras dos clientes, isto é, cada código mapeia a informação das suas compras representada pela classe InfoCliente; a segunda representa as vendas de produtos, o valor chave é o produto e o valor que mapeia é da classe InfoProduto; as restantes associações representam as compras pelas diferentes filiais, desta forma o Cliente(chave) mapeia o valor de quanto gastou numa determinada filial(valor). A classe Filiais além

de possuir a declaração dos construtores por omissão, parametrizado e de cópia, dos métodos de acesso e alteração do seu estado interno e do método *clone*, *toString*, *equals, hashcode;* possui outros métodos que permitem adicionar um cliente e adicionar um produto aos conjuntos correspondentes criando as suas listas de compras e vendas, respetivamente, independentemente de comprarem ou venderem; adicionar uma compra, isto é, recebe uma venda que mediante o produto e cliente em causa atualiza as compras dos clientes, produtos e da filial envolvida; retornar quantos clientes tem a sua variável comprasCliente; devolver quantos produtos foram vendidos e dado um mês retorna quantos clientes efetuaram compras. Existem mais métodos implementados além dos anteriormente abordados, métodos esses que serão descritos abaixo e que permitem atingir os objetivos propostos.

Estatística

Esta classe foi desenvolvida com o intuito de cumprir a consulta estatística definida como objetivo. Para isso tem como diferentes variáveis: o nome do último ficheiro lido, o total de linhas lido, o número de registos errados, o total de produtos, o total de produtos comprados, o total de produtos que não foram comprados, o total de clientes, o total de clientes que compraram, o total de clientes que nada compraram, o total de quantas vendas com o preço igual a 0 foram registadas, a faturação global, um *array* de 12 posições que representa o total de compras mensal, outros três *array's* de 12 posições que representam o total faturado por cada filial (cada filial com o seu *array*) , um outro *array* de 12 posições que demonstra o total faturado em cada mês e por fim um *array* de 12 posições de quantos clientes efetuaram compras por mês. Esta classe possui a declaração dos construtores por omissão, parametrizado e de cópia, dos métodos de acesso e alteração do seu estado interno e do método *clone*, *toString*, *equals, hashcode.*

InterfGestVendas e GestVendas

A classe GestVendas foi desenvolvida de forma a agregar toda a informação anteriormente referida para isso possui como variáveis de instância um catálogo de produtos, um catálogo de clientes, uma faturação, uma estatística e informação das filiais através da classe Filiais. Em todas as classes construídas a classe em causa possui a declaração dos construtores por omissão, parametrizado e de cópia, dos métodos de acesso e alteração do seu estado interno e do método *clone* e *equals.* Como um dos objetivos deste trabalho é a implementação de interfaces foi implementada também a interface InterfGestVendas que detém os métodos de acesso às variáveis.

Queries

Esta classe não possui qualquer variável de instância, foi desenvolvida simplesmente para responder às diversas consultas interativas definidas como objetivos. Esta classe recebe sempre o model nas diferentes queries e mais alguma informação se assim for preciso. Demonstra em todos os métodos quanto tempo demora a executar cada consulta. No fim apresenta os resultados obtidos.

A queryEstatistica vai ao *model* buscar a estatística e utiliza o método toString desta para apresentar os valores.

A query1 pede ao *model* a filial e utiliza um dos métodos presente na classe Filiais que percorre a associação de compra dos produtos anteriormente descrita, verificando quais os produtos que não tiveram nenhuma unidade vendida , de modo a obter uma lista de produtos que não foram vendidos.

A query2 recebe um mês e utiliza os métodos presentes na faturação e na filial para obter as informações de quantos vendas foram registadas nesse determinado mês e quantos clientes o fizeram, respetivamente.

A query3 recebe um código de cliente, verifica se o código existe utilizando o método presente no catálogo de clientes, se existir obtém através de *array's* quantas compras efetuou em cada mês, quantos produtos comprou e quanto gastou.

A query4 faz o mesmo que a query3 mas agora com um código de Produto.

A query5 recebe um cliente, verifica se existe, caso exista, invoca o método presente na classe Filiais que permite obter um conjunto de ParProdNumClis, ordenado pelo número de unidades que o cliente em causa comprou e apresenta o resultado sob a forma de uma lista.

A query6 utiliza um dos métodos presentes da classe faturação permitindo obter uma lista dos produtos mais vendidos, do tipo anteriormente referido para definir pares de informação, cujo tamanho desta lista é um valor que o utilizador inseriu, obtém ainda uma lista com o número de clientes que compraram esses produtos.

A query7 devolve através de conjuntos os três clientes que mais gastaram em cada filial.

A query8 devolve uma lista ordenada, de forma decrescente, cujo tamanho é indicado pelo utilizador, de clientes que mais compraram produtos diferentes. Não se encontra a funcionar.

Por fim, a query9 recebe um código de Produto e um valor, apresentando uma lista de ParProdNumClis ordenada com os clientes que mais unidades compraram desse produto.

Funcionamento da aplicação

De uma forma resumida passamos a explicar o funcionamento da aplicação em causa.

A aplicação inicia-se em GestVendasAppMVC onde está definida a *main* implementada como os docentes assim o explicaram e forneceram.

Inicialmente, cria-se o model através da interface, isto é, as estruturas de dados são inicializadas primeiro e só posteriormente são carregados os ficheiros iniciais, através do método createData, onde o utilizador insere quais os ficheiros iniciais que pretende ler, posteriormente é efetuado o processo de leitura e o correspondente preenchimento nas estruturas criadas.

De seguida, é criado a view e o controller, este vai ser preenchido pela view e o model anteriormente criados. O controller utiliza o método startController onde é criado uma instância de Queries, objeto esse que é passado para a função *engine*, função esta que está encarregue de: invocar os menus presentes na view, de interpretar os valores recebidos quer sejam inteiros quer sejam Listas de objetos e posteriormente invocar as consultas respetivas através dos métodos correspondentes que se encontram na classe Queries. Esta função é responsável de ler novos dados, guardar e carregar em ficheiro de objetos a informação que armazenou anteriormente.

Medidas de Performance

De seguida, apresentam-se os tempos que a aplicação demorou a carregar e a ler os ficheiros fornecidos pelos docentes. Os tempos apresentados abaixo têm em consideração o preenchimento de todas as estruturas anteriormente abordadas, armazenamento esse que implicou a validação de cada venda e o seu "parsing".

```
################## GestVendasAppMVC ######################
Ficheiro para clientes? (não se esqueça da terminação .txt)
Clientes.txt
Ficheiro para produtos? (não se esqueça da terminação .txt)
Produtos.txt
Ficheiro para vendas? (não se esqueça da terminação .txt)
Vendas_1M.txt
Tempo: 7.5232989.
Leitura dos ficheiros efectuada com sucesso.
```

```
################## GestVendasAppMVC ######################
Ficheiro para clientes? (não se esqueça da terminação .txt)
Clientes.txt
Ficheiro para produtos? (não se esqueça da terminação .txt)
Produtos.txt
Ficheiro para vendas? (não se esqueça da terminação .txt)
Vendas_3M.txt
Tempo: 28.521083.
Leitura dos ficheiros efectuada com sucesso.
```

```
################## GestVendasAppMVC ######################
Ficheiro para clientes? (não se esqueça da terminação .txt)
Clientes.txt
Ficheiro para produtos? (não se esqueça da terminação .txt)
Produtos.txt
Ficheiro para vendas? (não se esqueça da terminação .txt)
Vendas_5M.txt
Tempo: 52.451721.
Leitura dos ficheiros efectuada com sucesso.
```

Apresenta-se de seguida os tempos que o programa demorou a executar, por exemplo, a query5, query7 e query9, respetivamente com o ficheiro de Vendas_5M.txt:

```
>>>5
Insira um código de cliente (ex: G1999)
X4536
Tempo: 0.0054462
```

```
>>>7
Tempo: 0.0273125
```

```
>>>9
Insira um código de um produto existente (ex: GR1999)
FD1089
Insira um valor
200
Tempo: 0.0279136
```

Seguidamente, decidimos alterar as estruturas de dados do seguinte modo, onde tínhamos anteriormente implementado coleções do tipo TreeMap<> substituiu-se por coleções do tipo HashMap<>, e vice-versa. Aplicou-se a mesma estratégia nas coleções TreeSet<> por HashSet<>, e vice-versa, e finalmente substituiu-se a implementação de ArrayList<> por coleções do tipo Vector<>.

```
################## GestVendasAppMVC ######################
Ficheiro para clientes? (não se esqueça da terminação .txt)
Clientes.txt
Ficheiro para produtos? (não se esqueça da terminação .txt)
Produtos.txt
Ficheiro para vendas? (não se esqueça da terminação .txt)
Vendas_1M.txt
Tempo: 7.9792942.
Leitura dos ficheiros efectuada com sucesso.
```

```
################## GestVendasAppMVC ######################
Ficheiro para clientes? (não se esqueça da terminação .txt)
Clientes.txt
Ficheiro para produtos? (não se esqueça da terminação .txt)
Produtos.txt
Ficheiro para vendas? (não se esqueça da terminação .txt)
Vendas_3M.txt
Tempo: 27.2047564.
Leitura dos ficheiros efectuada com sucesso.
```

```
################## GestVendasAppMVC ######################
Ficheiro para clientes? (não se esqueça da terminação .txt)
Clientes.txt
Ficheiro para produtos? (não se esqueça da terminação .txt)
Produtos.txt
Ficheiro para vendas? (não se esqueça da terminação .txt)
Vendas_5M.txt
Tempo: 43.7138558.
Leitura dos ficheiros efectuada com sucesso.
```

```
>>>5
Insira um código de cliente (ex: G1999)
X4536
Tempo: 0.0073097
```

```
>>>7
Tempo: 0.0859847
```

```
>>>9
Insira um código de um produto existente (ex: GR1999)
FD1089
Insira um valor
200
Tempo: 0.0281443
```

Conclusão

O presente relatório descreveu, de forma sucinta, as principais componentes da aplicação desenvolvida para a avaliação do projeto Java da unidade curricular de LI3.

Consideramos que os principais objetivos foram cumpridos com a exceção da consulta número 8 a não funcionar, a consulta número 10 não foi implementa pois não sabíamos como concretizá-la, na consulta estatística não conseguimos mostrar quantos vendas tiveram o preço a zero. Por fim, ficou em falta criar uma exceção de forma a garantir que o programa não pare de executar quando os ficheiros introduzidos pelo utilizador não existam.

Ao nível da interface, alguns aspetos poderiam ter sido melhorados, bem como, a otimização de algumas queries de modo a serem mais eficientes tal como nas medidas de Performance.

Sentimos que a realização deste projeto consolidou os nossos conhecimentos da linguagem Java, nomeadamente na modularidade e encapsulamento de dados.