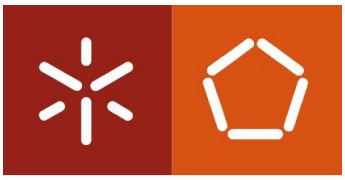

Universidade do Minho

Departamento de Informática

GestVendas

Grupo Nº 56

João Teixeira (A85504)

Emanuel Rodrigues (A84776)

José Ferreira (A83683)

14 de Junho de 2019

Conteúdo

Capítulo 1

Introdução

O objetivo deste projeto é construir um sistema de gestão de vendas modular, de forma a ser capaz de armazenar informações de vendas e relacionar produtos, clientes e vendas de forma eficiente, aplicando uma arquitetura *Model, View, Controller* e conhecimentos sobre algoritmos e estruturas de dados. Como objetivo, é também necessário garantir o encapsulamento dos dados armazenados de forma a que não sejam possíveis alterações indevidas por agentes externos.
Ao longo deste relatório vamos descrever as nossas abordagens a estes problemas e apresentar alguns testes de performance do nosso projeto final.

Capítulo 2

Problema

Três ficheiros são fornecidos contendo informação sobre as transações de uma distribuidora:

- O primeiro contém ids de produtos.
- Outro, ids de clientes.
- O último integra informações sobre cada venda efetuada ao longo de um ano.

Com base nesses ficheiros é preciso responder a 12 queries fornecidas:

1. Estatísticas sobre os ficheiros lidos.
2. Números gerais sobre os dados carregados.
3. Lista dos códigos de produtos não comprados.
4. Número de vendas realizadas e clientes distintos, num dado mês e numa dada filial.
5. Número de produtos distintos e gastos totais mês a mês para um dado cliente.
6. Determinar mês a mês quantas vezes foi comprado, por quantos clientes e o total faturado de um dado produto.
7. Produtos mais comprados por um dado cliente e a sua quantidade.
8. X produtos mais vendidos todo o ano e o número de clientes que o compraram.
9. Determinar os 3 maiores compradores em cada filial.
10. X clientes que compraram mais produtos diferentes e quantos.
11. X clientes que compraram um dado produto e qual o valor gasto.
12. Calculara a faturação total de um dado produto, mês a mês e filial a filial.

Capítulo 3

Módulos e API

Tendo em conta as características do projeto, concluímos que a arquitetura que melhor abrangia os critérios pedidos era a Arquitetura do tipo *MVC* (Modelo, Apresentação e Controlador). Um dos critérios que mais fez o grupo gravitar em direção a esta escolha foi a modularidade inerente a esta arquitetura.

3.1 Model

É na *Package Model* onde está toda a parte de algoritmos e dados, nunca esta conhecendo ou dada a conhecer à camada da Apresentação.

3.1.1 Cliente

O módulo de Cliente tem uma API capaz de lidar com a informação de um cliente individual, e sua respetiva validação.

3.1.2 Clientes

O módulo de Clientes tem uma API capaz de lidar com a informação de todos os clientes, respetivo armazenamento e pesquisa.
Como estrutura principal de armazenamento dos Clientes individuais, após testes testar várias estruturas, optamos por utilizar os HashMaps presente na biblioteca *java.util.Map*.

3.1.3 Produto

À semelhança do módulo de Cliente, o módulo de Produto tem uma API desenhada para tratar da informação referente a um produto individual, bem como a sua validação.

3.1.4 Produtos

No módulo de Produtos, a API está construída de maneira a que seja capaz de lidar, de forma eficiente com o armazenamento e pesquisa de todos os produtos.
Para este módulo, como no módulo de Clientes, decidimos utilizar os mesmos HashMaps da biblioteca *java.util.Map*.

3.1.5 Venda

No módulo de Venda, temos uma API capaz de dar parse de uma string com o formato previamente definido, colocar numa estrutura com os campos necessários de maneira a preservar a informação, para posteriormente ser tratada pelo módulo de Filiais e Faturação.

3.1.6 Factura

Neste módulo, a API está definida de forma a conseguir, a partir de uma Venda, criar/atualizar uma fatura, contendo esta, a faturação relativa a um produto.

3.1.7 Faturas

O módulo de Faturas é capaz de comportar informação sobre toda a faturação, organizada por produtos e guardada numa Hashtable. Uma estrutura do tipo Faturas para além de guardar a faturação individual de cada produto, para um mais rápido acesso, guarda também valores totais de faturação e número de vendas.

3.1.8 Filial

Para o módulo de filial, está definida uma API capaz de fazer a ligação entre clientes e os respetivos produtos por eles comprados, e vice-versa.
Uma estrutura do tipo Filial é composta por duas Hashtables, uma que contém informação relativa a todos os clientes que fizeram compras na dada filial, que produtos compraram e respetiva faturação, e uma segunda que contém informação relativa a todos os produtos vendidos na dada filial, mais concretamente que clientes adquiriram o produto.

3.1.9 Constantes

No módulo constantes, são guardadas todas as informações referentes ao número de filiais, ficheiros a ler, entre outros, que são lidas de um ficheiro de configs.

3.1.10 GestVendasModel

O módulo GestVendasModel é o módulo que junta todos os acima descritos, contendo este uma estrutura que contém um Catálogo de Produtos, um Catálogo de Clientes, uma estrutura Faturação e um array de Filiais. Este módulo faz a ponte entre todos os módulos internos e o exterior, sendo este o módulo que ao qual o controlador faz pedidos, e tem métodos capazes de responder a todas as queries pedidas.

3.2 View

Esta *package* contém as classes de apresentação dos resultados ao utilizador.

3.2.1 GestVendasView

Esta classe representa os vários Menus e as relações entre eles. Para permitir conhecer o caminho percorrido até ao menu que se está a observar esta classe contém uma stack com os menus percorridos.

3.2.2 Navigator

A fim de facilitar a apresentação de determinadas queries foi criada uma classe que apresenta uma lista de valores sobre a forma de páginas.
De forma a que esta representação se ajuste ao terminal que está a ser utilizado, o número de linhas e de colunas que vai ser representado em cada página navegável é calculado com base no tamanho do terminal presente. Para tal, a classe Terminal é utilizada.

3.2.3 Table

Alguns dos dados obtidos nas queries variam em duas variáveis discretas finitas, como por exemplo ao mês a mês e filial a filial em simultâneo.
Assim, naturalmente, a melhor forma de representar tais dados é através de uma tabela. Para tal foi desenvolvida uma classe para representar tabelas com um *generic type parameter*.
Consequentemente, esta classe apenas precisa de duas listas de etiquetas (uma para as linhas e outra para as colunas) e uma Lista de Listas com os dados para representar uma tabela visualmente apelativa em que cada coluna automaticamente adapta o seu tamanho ao seu conteúdo.

3.3 Controller

Cria a ponte entre o View e o Model. Assim, o Controller é o único que conhece a view e o Model, sendo que tanto a View como o Model apenas conhecem o Controller.

3.3.1 GestVendasController

Esta classe atua como o controlador do nosso projeto. Para além de funcionar como a ponte entre a lógica e a apresentação também cronometra o tempo demorado pela lógica a responder às queries pedidas e passa esse valor à *View* para ser apresentado ao utilizador.

3.4 Exceptions

Esta Package contém todas as classes de exceções utilizadas ao longo do projeto. Estas são usadas para assinalar, por exemplo, uma filial ou um cliente inválido passado como argumento.

3.5 Utils

A *package* Utils contém classes que são utilizadas em várias partes do projeto e que não pertencem a uma parte específica da arquitetura *MVC*.

3.5.1 Crono

A Classe Crono está desenhada para medir o tempo decorrido ao longo da execução do projeto. Para tal possui dois métodos, o *start* e o *stop* que devem ser chamados no inicio e no fim da porção de código que se quer cronometrar, respetivamente. Por fim, para apresentar o resultado ao utilizador, a função *toString* foi implementada de forma a apresentar os tempos calculados de forma legível.

3.5.2 Terminal

A classe Terminal calcula o tamanho do terminal onde o programa está a correr. Para tal realiza duas *System Calls* com dois *Fork exec* (uma para obter a altura e outra para obter a largura do terminal). Para recalcular o tamanho basta chamar o método *update*.

3.5.3 StringBetter

A Classe StringBetter contém métodos que permitem formatar texto quando este é impresso na *Shell*. Estes métodos incluem mudar a cor, negrito e itálico e repetir o texto N vezes.
Afim de agilizar a utilização da classe, todos os métodos devolvem uma instância da classe de forma a ser possível o encadeamento de métodos.
Assim, se um utilizador quiser escrever *Hello World* a negrito, sublinhado e a vermelho basta escrever.

```
out.println(new StringBetter("Hello World").bold().under().red()
```

Capítulo 4

Testes e Benchmarks

Durante a execução do nosso projeto, foram efetuados diversos testes, quer ao nível de formas de leitura do ficheiro, em particular, *BufferedReader* e *Files*, bem como testes ao nível das diversas implementações das Interfaces *Map*, *Set* e *List*.

4.1 Tempos de execução

Com recurso à classe fornecida *Crono*, efetuamos alguns benchmarks, obtendo a tabela C.1, com os tempos médios de execução, com os diversos ficheiros de vendas fornecidos.
Nestes vários benchmarks, testamos diferentes implementações das Interfaces *Map*, nomeadamente, *HashMap* e *TreeMap*, concluindo assim que os tempos de que os *HashMap* são muito mais rápidos no carregamento da informação para memória, mantendo os tempos das queries praticamente inalterados, como é possível comprovar na tabela C.3.
Foram também testados tempos de execução das Implementações da Interface *List*, nomeadamente *ArrayList* e *Vector*, visível na tabela C.2 não havendo diferenças notáveis, acabando assim por optar pela implementação mais comum *ArrayList*.

4.2 Tempos de Leitura

Ao nível de leitura, foram efetuados benchmarks para comparar a performance de *Files* e de *BufferedReader* vistos em no apêndice B. Sendo o *Files* mais lento e já que o *Files* utiliza internamente um *BufferedReader*, acabamos por utilizar o *BufferedReader* para a leitura dos ficheiros com a informação. Testamos também a leitura e validação dos ficheiros tirando partido do paralelismo intrínseco da JVM8, utilizando uma Stream¡Venda¿ verificando que os tempos de carregamento baixavam para perto de metade, no caso do ficheiro com 5 milhões de vendas.

Capítulo 5

Conclusão

Para concluir, conseguimos cumprir todos os requisitos propostos, conseguindo implementar todos os módulos e estrutura-los como pedido, sendo assim capaz de responder a todas as queries da forma que nos pareceu mais eficiente.
Como trabalho futuro, gostaríamos de melhorar a maneira da organização da informação referente às filiais, de forma a baixar tempo de resposta a queries que utilizam informação nele presente.

Apêndice A

Diagramas de Classes

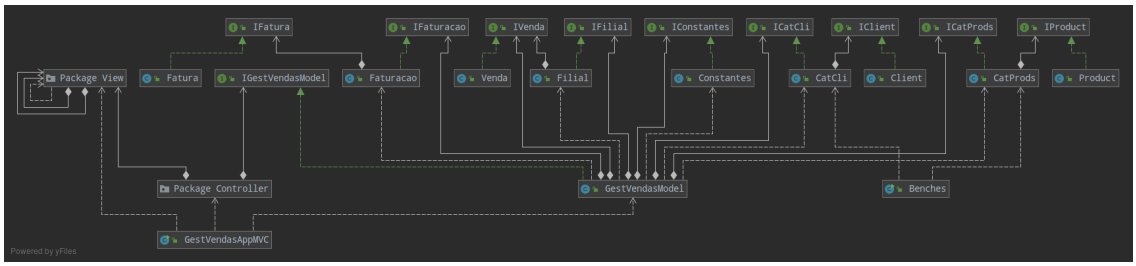

Figura A.1: Diagrama de Classes do Model

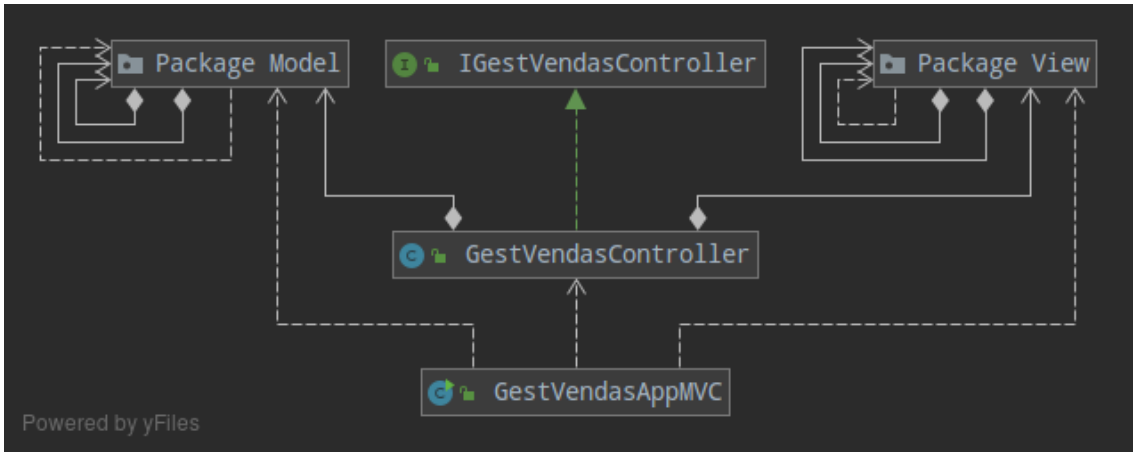

Figura A.2: Diagrama de Classes do Controller

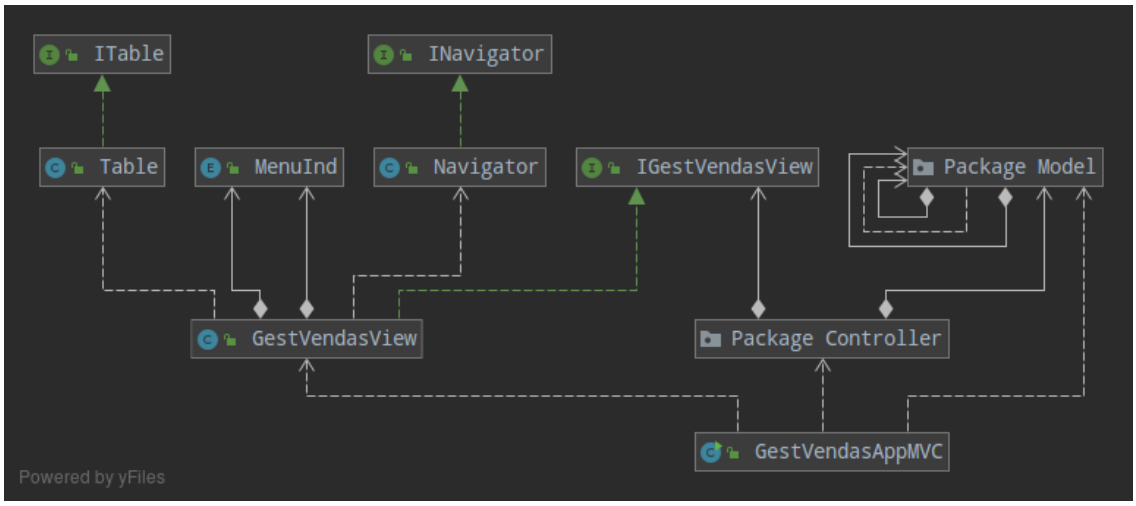

Figura A.3: Diagrama de Classes do View

Apêndice B

Benchmarks BufferedReader vs Files

	1 Milhão	3 Milhões	5 Milhões
Leitura	162	434	750
Parse	871	1884	3013
Validação	1316	3120	4380
Validação Paralela	795	1535	2203

Tabela B.1: Tempo (em ms) de tempos de BufferedReader

	1 Milhão	3 Milhões	5 Milhões
Leitura	173	480	773
Parse	861	1948	3017
Validação	1139	2755	4380
Validação Paralela	828	1572	2222

Tabela B.2: Tempo (em ms) de tempos Files

Apêndice C

Tabela de Tempos de Execução

	1 Milhão	3 Milhões	5 Milhões
Load Time	2002	4522	7359
Query 1	68	32	31
Query 2	39	80	119
Query 3	55	20	14
Query 4	1	1	1
Query 5	13	22	17
Query 6	950	2306	3730
Query 7	71	111	72
Query 8	240	1115	1526
Query 9	9	14	10
Query 10	45	100	230

Tabela C.1: Tempo (em ms) das queries para um dado número de vendas

	1 Milhão	3 Milhões	5 Milhões
Load Time	2117	4824	7270
Query 1	21	32	34
Query 2	78	86	117
Query 3	5	20	18
Query 4	1	1	1
Query 5	11	26	18
Query 6	858	2405	3942
Query 7	67	121	57
Query 8	267	900	1418
Query 9	7	5	10
Query 10	33	104	176

Tabela C.2: Tempo (em ms) das queries para um dado número de vendas, utilizando *Vector* em vez de *ArrayList*

	1 Milhão	3 Milhões	5 Milhões
Load Time	5060	16100	25369
Query 1	74	32	27
Query 2	69	88	94
Query 3	45	10	18
Query 4	1	2	1
Query 5	48	15	14
Query 6	929	2528	3690
Query 7	62	68	80
Query 8	269	816	1430
Query 9	7	5	10
Query 10	50	111	155

Tabela C.3: Tempo (em ms) das queries para um dado número de vendas, utilizando *TreeMap* em vez de *HashMap*