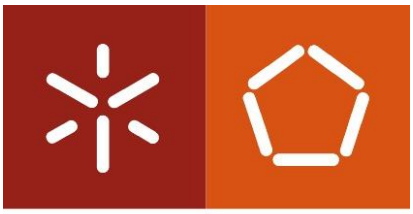

Universidade do Minho

Mestrado Integrado em Engenharia Informática

LI3 - Sistema de Gestão de Vendas - Java

Grupo 12

Sofia Santos (A89615) Ana Filipa Pereira (A89589)
Carolina Santejo (A89500)

Ano Letivo 2019/2020



Conteúdo

Capítulo 1

Introdução e principais desafios

Este projeto consistiu no desenvolvimento de um Sistema de Gestão de Vendas (SGV) na linguagem de programação Java, sendo a continuação de um projeto anterior realizado em C. Tal como o projeto desenvolvido previamente, este é um projeto de programação em larga escala, e como foi necessário recorrer a princípios mais avançados de programação, como o uso de estruturas de dados eficientes para armazenar e consultar grandes quantidades de informação, garantindo sempre o encapsulamento dos dados. Para este projeto usámos apenas estruturas de bibliotecas standard do Java, como TreeSet ou HashMap.

Neste projeto, em comparação ao anterior, foi mais fácil desenvolver formas eficazes de armazenar os dados, visto que esta linguagem de programação já nos disponibiliza bastantes estruturas de dados com vantagens diferentes, e não temos que nos preocupar com a alocação e libertação manual de memória. Para mostrar grandes quantidades de informação ao utilizador usámos o método que usámos no projeto em C, percorremos uma lista por páginas, sendo que o tamanho de cada página depende do tamanho de cada elemento, para termos algo mais consistente.

Capítulo 2

Classes, interfaces e estruturas de dados

2.1 Catálogo (Catalog)

Como os catálogos de clientes e de produtos são muito parecidos, apenas muda o tipo de dados que armazenam, criámos uma interface Catalog que as classes dos catálogos de clientes e produtos implementam. Desta forma podemos dizer às classes que implementam esta interface que métodos devem implementar.

2.2 Clients

Para armazenar a informação de todos os clientes criámos a classe Clients que utiliza um TreeSet para armazenar instâncias da classe Client. A API desta classe permite realizar operações de inserção e procura.

2.3 Products

Tal como a classe dos clientes, a classe Products usa um TreeSet para armazenar instâncias da classe Product. Esta classe permite realizar operações de inserção e procura.

2.4 CatalogItem

Como usámos a interface Catalog, precisamos também de uma interface CatalogItem que as classes Client e Product implementam.

2.5 Client

Esta classe armazena informação relativa a um cliente, que no caso do nosso projeto se resume ao seu código, mas que num projeto de mais larga escala podia ser usada para armazenar outro tipo de dados, como o nome do cliente, idade, etc. A API desta classe permite validar um código de cliente e obter o código de cliente de uma instância.

2.6 Product

Tal como a classe Client, esta classe, para além de ter uma API semelhante, também armazena informação relativa a um produto, que mais uma vez inclui apenas o código de produto, mas também poderia conter mais informação, caso tal fosse relevante ou necessário.

2.7 Faturas (Bills) e Fatura (Bill)

Para armazenar as faturas usámos um HashMap, que associa um código de produto a uma classe à qual chamámos Bill (Fatura). Uma fatura permite armazenar, para um dado produto, o seu código, o lucro total obtido com a venda desse produto, o número total de vezes que esse produto foi vendido (registos de venda) e a quantidade total vendida. O lucro total, a quantidade total de vendas e a quantidade total vendida de um produto estão separadas por filial, mês e modo (isto é, se o produto foi comprado em promoção ou não). Temos assim a nossa definição de uma fatura:

```
public class Bill {
    private String productID;
    private double[][][] totalProfit;
    private int[][][] totalSales;
    private int[][][] totalQuantitySold;
}
```

A API da classe Bill permite obter informação relativa a essa fatura, e da classe Bills permite inserir faturas e obter dados relativos a todas as faturas.

2.8 Filial (Branch)

Para armazenar as relações entre clientes e produtos por filial usámos uma estrutura de dados constituída por dois HashMaps, um que faz corresponder cada cliente aos produtos que este comprou, e outro que faz corresponder cada produto aos clientes que o compraram. Na classe GestVendas é usada uma lista de três elementos, onde cada elemento é uma instância desta classe, para podermos ter uma instância por cada filial.

O primeiro Map (clientsWhoBought) associa cada produto (chave) aos clientes que o compraram (valor). Esta lista de clientes é outro HashMap, que a cada cliente (chave) faz corresponder a informação relativa à compra do produto inicial por esse cliente (valor), nomeadamente a quantidade desse produto comprada, o número de registos de venda e o dinheiro gasto nesse produto, todos divididos por mês.

O segundo Map (productsBoughtBy) associa cada cliente (chave) aos produtos que este comprou (valor). Tal como no outro Map, neste a lista de produtos é também um HashMap, que a

cada produto (chave) faz corresponder a informação relativa à compra desse produto pelo cliente inicial (valor). Esta informação inclui a quantidade desse produto comprada, o dinheiro gasto nesse produto e o número de registos de venda associados ao cliente dado, todos divididos por mês.

A informação dos clientes e dos produtos mencionada acima é armazenada em instâncias da classe CliProdInfo. Estes dois HashMaps acabam por conter na sua maioria a mesma informação, mas esta duplicação de informação resulta numa enorme eficiência na execução das queries. Se apenas tivéssemos o Map que associa cada produto aos clientes que o compraram, por exemplo, e quiséssemos uma lista de todos os produtos comprados por um dado cliente, teríamos que percorrer o Map todo, visto que o cliente pode ter comprado qualquer produto, enquanto que usando dois Maps podemos simplesmente consultar uma vez o HashMap que associa cada cliente aos produtos que comprou e teríamos imediatamente a nossa resposta. Passamos assim de milhares de consultas a poucas dezenas (a complexidade passa de O(N) para O(log N), e como temos milhares de clientes e de produtos, a diferença é significativa).

Esta classe permite realizar operações de inserção, pesquisa e obtenção de dados.

2.9 Informação dos ficheiros (FileInfo)

Esta classe armazena informação relativa aos ficheiros lidos aquando do carregamento do GestVendas, nomeadamente os caminhos dos ficheiros, o número total de linhas e o número de linhas válidas. Em relação às vendas, são guardados também a faturação total, o número de vendas de custo 0 e o número de clientes que fizeram compras, para responder à primeira consulta estatística. Esta classe permite realizar operações de consulta e alteração de dados.

2.10 StructuresInfo

Esta classe armazena a informação necessária para responder à segunda consulta estatística, mais nomeadamente o número total de compras por mês, a faturação total por mês e filial, tal como o valor global, e o número de distintos clientes por mês e por filial.

2.11 GestVendas

O nosso Sistema de Gestão de Vendas é composto pelas seis estruturas principais mencionadas acima. A classe GestVendas permite preencher as suas variáveis de instância a partir de ficheiros e responde a todas as queries. Podemos então definir a nossa classe GestVendas da seguinte forma:

```
public class GestVendas {
    private Clients clients;
    private Products products;
    private Bills bills;
    private ArrayList<Branch> branches;
    private FileInfo fileInfo;
    private StructuresInfo structuresInfo;
}
```

Esta classe também permite que armazenemos o seu estado atual através de ObjectStreams, e permite carregar um estado armazenado em disco.

2.12 Venda (Sale)

Definimos uma classe auxiliar que nos permite armazenar uma venda, tal como consta no ficheiro das vendas. Esta classe permite-nos aceder à informação de cada venda de forma mais eficiente, aquando do preenchimento dos módulos das faturas e das filiais. Esta classe apenas inclui operações de criação, consulta de dados e validação de uma venda, visto que é uma classe temporária, que apenas é usada para auxiliar o carregamento de dados dos ficheiros na GestVendas.

2.13 UI/QueriesUI

As classe UI e QueriesUI, tal como os nomes indicam, tratam da parte visual do programa, como mostrar ao utilizador os resultados das queries, os menus do programa, e os *input prompts*. Através da classe Formatting, foi possivel imprimir texto colorido no terminal. A classe UI trata da User Interface geral do programa, enquanto que a classe QueriesUI é específica das queries, tornando assim o código mais organizado.

2.14 Queries

A classe Queries é a responsável pelo fluxo das queries, ou seja, é a que trata de receber *input* do utilizador, chamar a função da query correspondente, fornecendo-lhe o *input* necessário, e enviar o *output* para a classe QueriesUI, para poder ser apresentado ao utilizador.

2.15 Controlador (Controller)

Por último, a classe Controlador é a classe principal, ou seja, é esta que controla o fluxo de todo o programa. Esta classe controla que query será executada, ou que ficheiros serão carregados no programa. A classe GestVendasAppMVC, a que contém o método main, é apenas responsável por correr o método run da classe Controller, a partir daí o Controller trata de correr o programa.

Capítulo 3

Estrutura do projeto

O nosso projeto segue a estrutura *Model View Controller* (MVC), estando por isso organizado em três camadas:

- A camada de dados (o modelo) é composta pela classe GestVendas, que por sua vez é constituído pelas classes referidas no capítulo anterior, Clients, Products, Bills, Branch, FileInfo e StructuresInfo. Para além destas, temos ainda a classe auxiliar Venda, e as subclasses Client, Product e Bill.
- A camada de interação com o utilizador (a vista, ou apresentação) é composta pelas classes UI e QueriesUI.
- A camada de controlo do fluxo do programa (o controlador) é composta pelas classes Controller e Queries. É esta camada que gere o funcionamento do programa, interagindo com as outras duas camadas para que este possa funcionar da forma pretendida. As outras duas camadas nunca interagem diretamente uma com a outra.

Nesta estrutura, o controlador é responsável por enviar pedidos ao modelo, como por exemplo responder a uma query. O modelo calcula essa resposta e envia-a ao controlador, que irá pedir à vista para a apresentar ao utilizador. Temos assim três camadas distintas que funcionam juntas para formar o nosso programa.

Capítulo 4

Complexidade das estruturas

Decidimos usar TreeSets nos catálogos de clientes e de produtos para podermos aceder aos dados dos catálogos de forma ordenada. Podíamos ter usado HashSets, por exemplo, mas nesse caso teríamos que ordenar os dados sempre que fôssemos aceder a instâncias dessas classes, o que iria ter um grande impacto na performance, e implicaria mudar a primeira query para passar a ordenar a lista que devolve.

Para os dados das faturas, usámos HashMaps em vez de TreeMaps puramente por escolha nossa. Comparando os tempos médios de execução das queries 5 a 9, em segundos, usando ambas as estruturas vemos que a diferença é tão pequena que não terá um impacto significante no desempenho do nosso programa, e podemos assim usar qualquer uma das duas estruturas.

Query	HashMap	TreeMap
5	0.0022	0.0020
6	0.6347	0.6733
7	0.4939	0.4849
8	0.3838	0.3511
9	0.0003	0.0004

Para os dados das filiais usámos HashMaps porque a performance com HashMaps é melhor, em geral, em comparação com TreeMaps, apesar da diferença não ser muito significativa. A única query que sofre bastante com esta troca é a query 10, cujo tempo de execução quase que duplica. Nesta tabela é possível ver o tempo médio de execução das queries 5 a 9 usando HashMaps e TreeMaps, em segundos (o ficheiro de vendas usado é o de 1 milhão de vendas):

Query	HashMap	TreeMap
5	0.0022	0.0015
6	0.6347	0.6509
7	0.4939	0.6376
8	0.3838	0.4450
9	0.0003	0.0003

No caso das estruturas relativas aos resultados das queries, foi utilizada a mais adequada para cada situação, como Maps ou Lists.

Capítulo 5

Testes de desempenho

Para medir o desempenho do programa usámos duas classes, a classe Benchmark para medir o tempo médio de execução das queries, e a classe PerformanceTests, para medir o tempo de leitura dos ficheiros de vendas, com ou sem parsing e validação. Estas classes podem ser corridas em separado do resto do programa, permitindo-nos realizar estes testes sem interferir com o programa principal. Estas classes usam a classe Crono para medir os tempos de execução.

A seguinte tabela permite-nos visualizar os tempos médios de execução de cada query, em segundos, para cada um dos três ficheiros de vendas (o tempo médio é a média do tempo de execução de 5 medições):

Query	1M Vendas	3M Vendas	5M Vendas
1	0.0437	0.0559	0
2	0.4937	1.0281	0
3	0.0023	0.0006	0
4	0.0010	0.0003	0
5	0.0022	0.0010	0
6	0.6347	0.9005	0
7	0.4939	1.5213	0
8	0.3838	1.9895	0
9	0.0003	0.0001	0
10	2.6814	12.523	0

Pela análise da tabela, podemos concluir o seguinte:

- para a maioria das queries, o tempo de execução não sofre uma alteração muito grande com o aumento do número de vendas.
- as queries cujo tempo de execução é afetado pela quantidade de vendas são as queries 2, 6, 7, 8 e 10, com a query 10 sendo a mais afetada.

Para além dos testes de tempo de execução, realizámos também testes de performance da leitura dos ficheiros de vendas, cujos resultados podem ser consultados na seguinte tabela:

Tipo de teste	1M Vendas	3M Vendas	5M Vendas
Leitura	0.5403	1.2304	2.0249
Leitura + parsing	2.2272	6.1081	10.028
Leitura + parsing + validação	3.1269	9.1028	16.245

Esta tabela mostra-nos aquilo que já seria de esperar. Quantas mais vendas contiver o ficheiro que estamos a ler, maiores vão ser os tempos de leitura, parsing e validação. Vemos também que o que mais afeta o tempo de execução é o parsing de cada linha, mais do que a validação e mais do que o número de vendas.

Capítulo 6

Conclusão

Tal como o projeto em C, sentimos que o nosso projeto está bem conseguido. O nosso programa é capaz de responder a todas as queries e mostrar os resultados de forma intuitiva, e fá-lo com tempos aceitáveis.

Tentámos melhorar a estrutura do projeto quando passámos de C para Java, o que tínhamos feito em C estava um pouco confuso e complexo, e acreditamos que neste projeto conseguimos simplificar algumas coisas, tornando o código mais simples de ser entendido por alguém que o veja.

Apêndice A

Diagrama de Classes

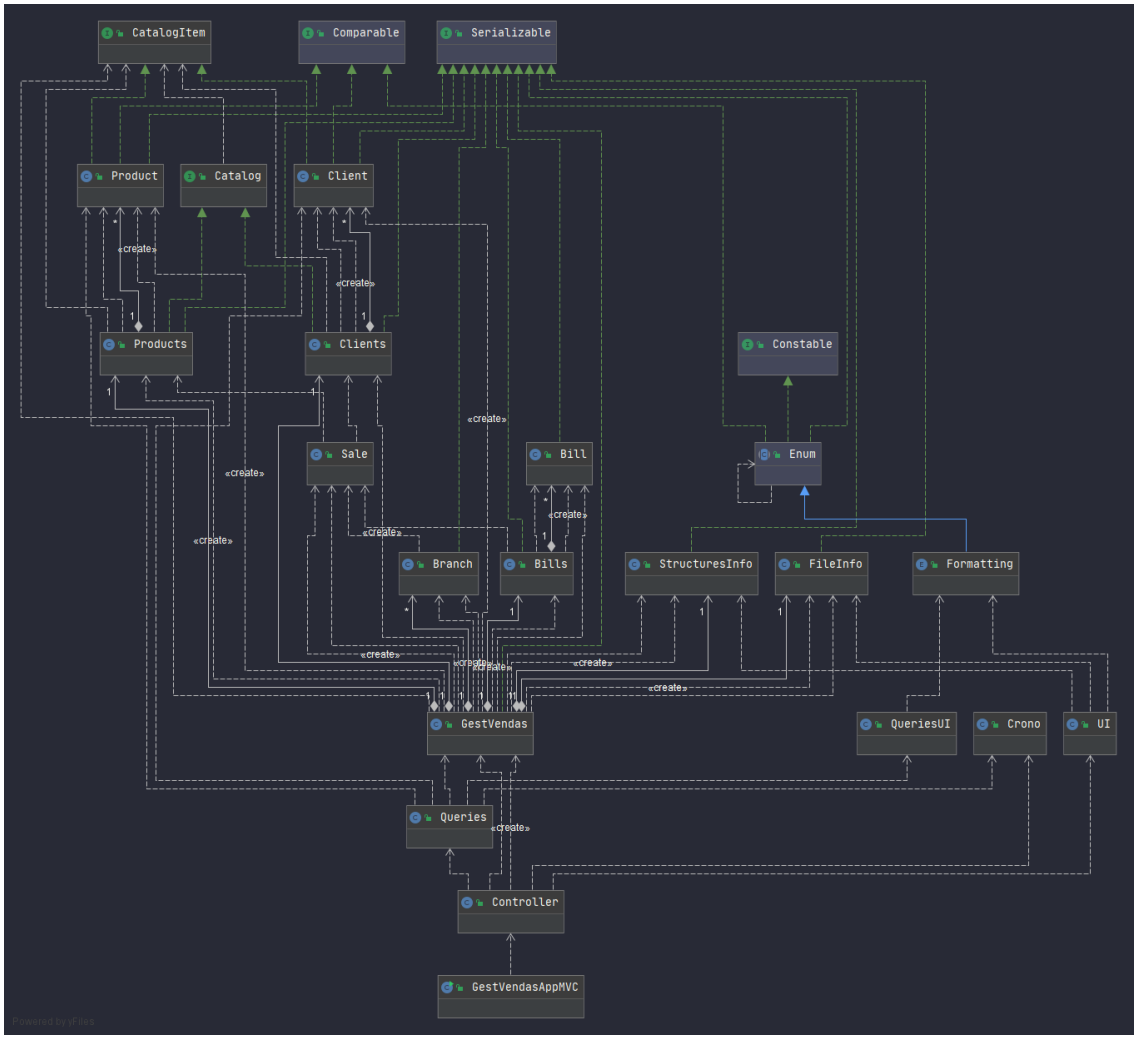

Figura A.1: Diagrama de classes do programa, gerado pelo *IntelliJ*

Apêndice B

Desenho da Estrutura de Dados

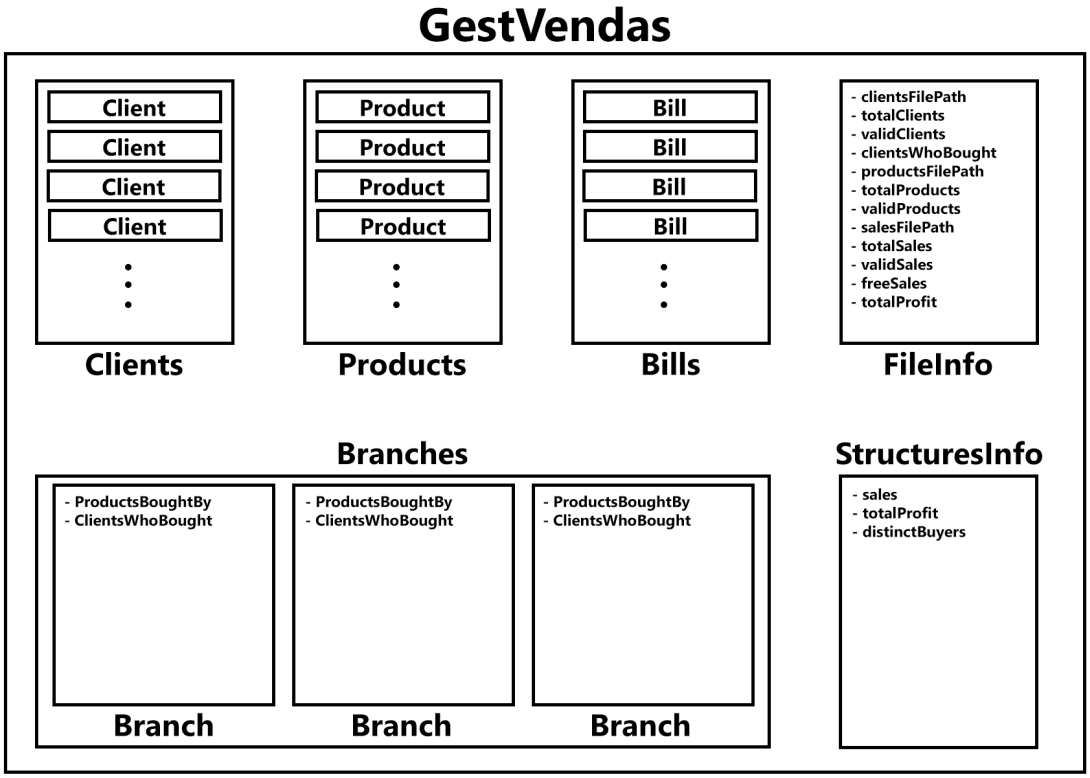

Figura B.1: Representação visual da classe GestVendas