Escola de Engenharia
Universidade do Minho

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA
Mestrado Integrado em Engenharia Informática
Laboratórios de Informática III

VENDUM

Sistema de Gestão de Vendas

Grupo 8

Renato Cruzinha
A75310

Ricardo Pereira
A73577

Ana Rita Guimarães
A79987

Braga, 7 de Junho de 2019

Conteúdo

1. Introdução

No âmbito da unidade curricular Laboratórios de Informática III do 2º Ano do Mestrado Integrado em Engenharia Informática foi proposto o desenvolvimento de um projeto em linguagem *C* e uma posterior adaptação para a linguagem *JAVA* sobre uma gestão de vendas com três filiais que tem como objetivo ajudar à consolidação dos conteúdos teóricos e práticos e enriquecer os conhecimentos adquiridos nas u.c.'s de *Programação Imperativa*, *Algoritmos e Complexidade* e *Programação Orientada aos Objetos*.

Considerou-se este projeto um grande desafio pela elevada complexidade que apresenta, uma vez que será uma aplicação com grandes volumes de dados, algo que a torna altamente complexa a nível algorítmico e estrutural. Nesta segunda etapa, foi incentivada a execução de testes de escalabilidade do programa desenvolvido, havendo para o efeito dois novos ficheiros de vendas, um com três milhões de entradas e outro com cinco. Nesse sentido, o desenvolvimento do mesmo será realizado a partir dos princípios da modularidade (divisão do código fonte em unidades separadas coerentes), do encapsulamento (garantia de proteção e acessos controlados aos dados), da conceção de código reutilizável e da escolha otimizada das estruturas de dados.

2. Desenvolvimento

2.1 Descrição das classes e estruturas de dados

Como seria de esperar, este projeto foi desenvolvido através da construção de classes, cada uma delas com uma determinada função imprescindível. É extremamente importante fazê-lo, tendo em conta a abstração do "mundo real", que nos ajuda a perceber e concretizar o nosso problema de uma forma mais objetiva. Esta forma de construir programas ajuda ainda à deteção de erros, tornando-os mais evidentes quando acontecem.

A análise dos dados que seriam importados para as estruturas tinha já sido feita na primeira fase deste projeto, e por esse motivo não foi necessário fazer testes exaustivos que nos mostrassem os dados com que estávamos a trabalhar.

Nesta etapa, foi novamente feita uma meticulosa análise das *queries* que nos foram propostas de forma a que as estruturas que desenvolvemos respondessem da forma mais rápida e eficiente possível. Como era esperado, a ideia geral não variou muito, variando sim, os tipos de dados e estruturas auxiliares que a representam.

2.1.1 Módulo venda (*venda*)

Ao contrário do cliente e do produto, uma venda tem mais do que um atributo, isto é, cada linha disponibilizada pelo ficheiro *Vendas_<XM>.txt* engloba sete campos. Por esse motivo foi necessário criar uma estrutura onde pudessemos guardar cada campo associado a um registo de uma venda. Destes, existem dois campos que devem ser devidamente analisados, os do cliente e do produto, que devem existir no catálogo de clientes e de produtos, respetivamente. São estas operações que fazem com que a leitura dos ficheiros de dados, abordada mais à frente, seja muito demorada.

Neste módulo estão implementadas todas as funções que verificam cada campo de um registo, e ainda mais algumas, nomeadamente, a função que cria uma estrutura do tipo *Venda* e inicializa os seus campos e a função que preenche cada campo de uma estrutura já criada.

2.1.2 Módulo catálogo de clientes (*Cclientes*)

Um catálogo de clientes não é nada mais que um *list* com vinte e seis posições em que cada posição está contido um *set*. Este número deve-se ao facto de a cada *set* estar associada uma letra do alfabeto, isto é, criados todos os *sets* correspondentes a cada posição, um cliente que seja válido terá como destino a árvore cuja letra associada é a única existente no seu código. Isto permite-nos fazer procuras (aproximadamente) vinte e seis vezes mais rápidas, assumindo que a atribuição de letras aos códigos é completamente aleatória..

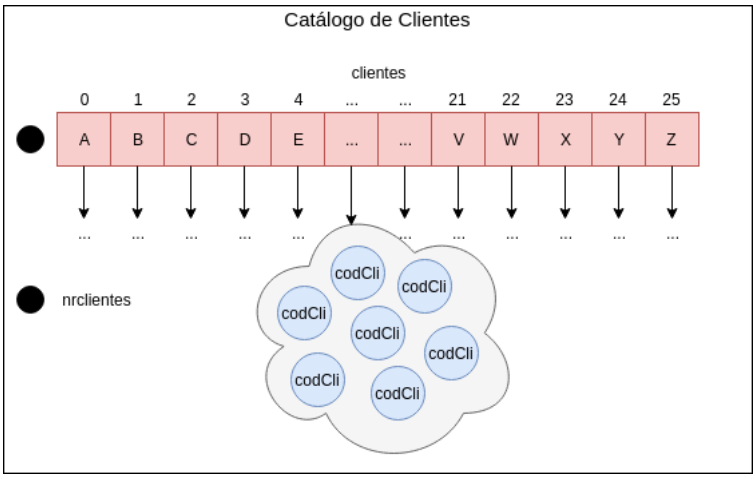

Figura 2.1: Catálogo de clientes

Como não poderia deixar de ser, este módulo alberga todos os métodos que alteram, criam ou manipulam estruturas deste género, como o registo de clientes, a verificação da existência de um determinado cliente, a colonagem de subestruturas, etc.

2.1.3 Módulo catálogo de produtos (*Cprodutos*)

Da mesma forma que dividimos os clientes por um *list* unidimensional, os produtos foram divididos por um *list* "bidimensional"(sendo na verdade um *list* de *list*s), uma vez que o seu código é sempre constituído por duas letras. Desta vez, teremos um list de vinte e seis por vinte e seis posições em que cada uma aponta para um *set*. Cada linha está associada à primeira letra do código do produto e cada coluna, à segunda, assumindo que os primeiros índices são as linhas e os segundos as colunas. As funções que identificamos no ponto anterior estão também implementadas neste módulo.

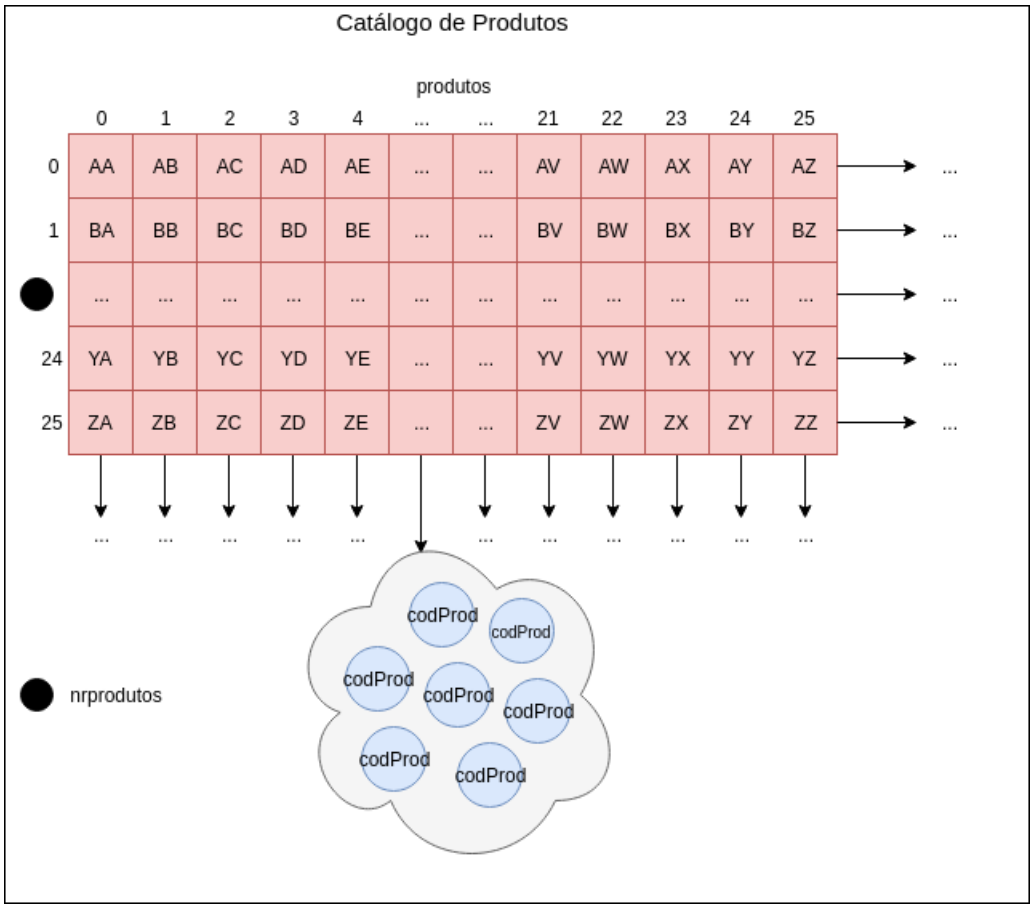

Figura 2.2: Catálogo de produtos.

2.1.4 Módulo catálogo de faturação (*Cfaturacao*)

Foi proposto que se organizasse num catálogo toda a faturação do sistema de gestão de vendas e após uma leitura cuidada e intensiva de todas as *queries* mencionadas, decidimos, sem fazer referência aos clientes, criar um módulo denominado *cfaturacao*, tal como na etapa anterior. Este módulo exporta as estruturas de dados que vemos nas seguintes imagens:

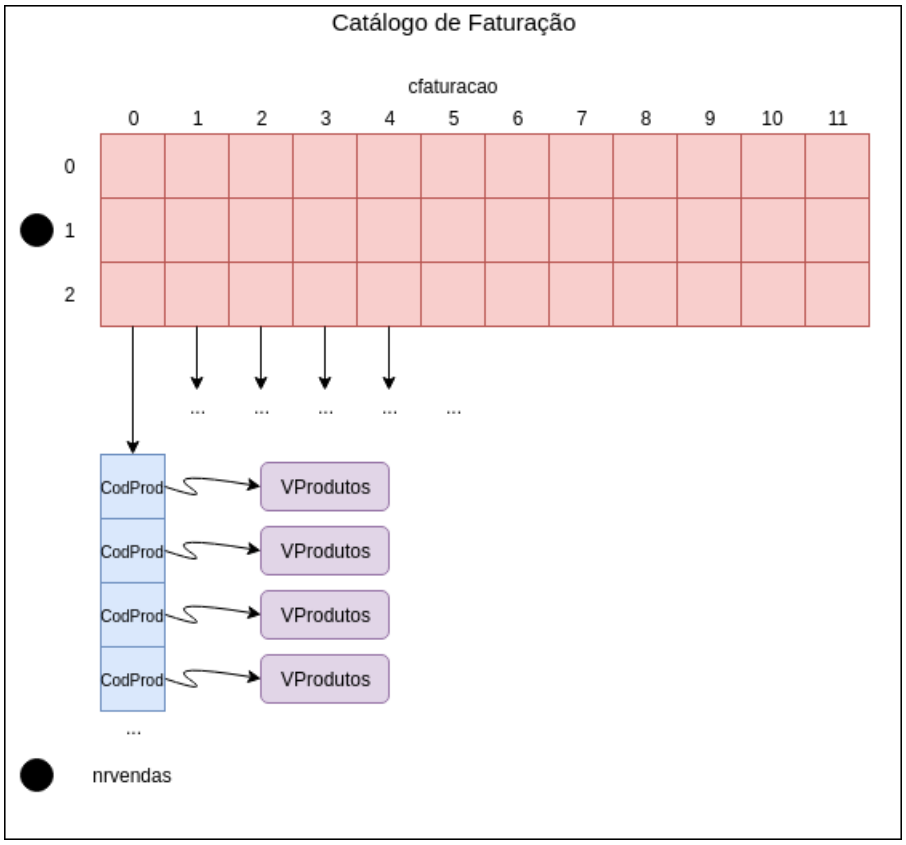

Figura 2.3: Catálogo de faturação.

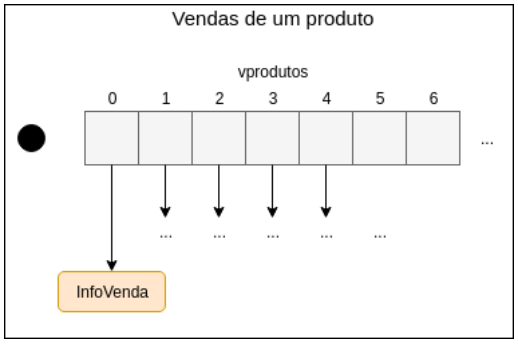

Figura 2.4: Vendas de um produto.

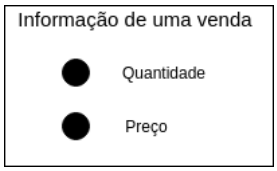

Figura 2.5: Informação de uma venda.

Esta estrutura é, talvez, a mais complicada de compreender, visto que existem muitos detalhes que devem ser estudados. Para começar, deliberou-se que as principais características de um registo, atendendo àqueles dez requisitos, seriam o mês e a filial, pelo que se construiu um *list* de *list*s com doze por três posições (número de meses e número de filiais, respetivamente), onde cada uma tem um *map* associado. Neste, as chaves são códigos de produtos e os respetivos valores são todas as vendas associadas a esse produto naquele mês e naquela filial.

Anteriormente tinha-se decidido fazer a divisão de forma diferente, mas desta vez verificou-se que esta seria a melhor solução, tendo ainda em conta que o modo de compra não é considerado em qualquer requisito e por isso, não fazia sentido guardá-lo nas estruturas. Até agora falou-se de *list*s cujo tamanho é sabido, ou era equivalente ao número de meses ou ao número de filiais e foi por esse motivo que foram organizados dessa forma, no entanto, o número de registos de venda por produto num mês e numa filial varia bastante e cada um deve ser guardado independentemente de outros. Isto levou a que fosse criada uma nova classe, a *VProdutos*, cujas instâncias corresponderão aos valores do *map* falado anteriormente. Dentro desta, existe um *list* em que cada posição tem uma instância da classe *InfoVenda*, criada para representar a informação referente a uma venda, nomeadamente a quantidade e o preço unitário.

Como é evidente, qualquer método/construtor que atue sobre este módulo está lá implementado, como é o caso dos que inicializam/inserem valores nas "subestruturas"ou então das funções que ajudam na resposta a determinadas *queries*.

2.1.5 Módulo catálogo de filiais (*Cfiliais*)

Tal como aconteceu com a classe *Cfaturacao*, foi pedido que se criásse um módulo *Cfiliais* que relacionasse os clientes, os produtos e as quantidades, para além dos parâmetros mês e filial, cuja existência é indispensável em qualquer um dos dois módulos agora mencionados. Mais uma vez, o modo de compra não aparece nem é inserido nesta estrutura, no entanto, caso fosse necessário, adicionar-se-ia esse atributo nas respetivas estruturas.

No *Cfiliais*, decidiu-se que, depois de um estudo exaustivo das interrogações impostas, devia-se implementar um *list* de *list*s de doze por três posições (meses e filias, respetivamente), tal como no *Cfaturacao*. Também à semelhança deste último, decidiu-se associar a cada posição um *map* que ligasse cada cliente aos respetivos produtos comprados, criando-se para o efeito uma nova classe chamada *ProdsCliente*.

A classe *ProdsCliente* representa todos os produtos comprados por um cliente num determinado mês, numa determinada filial e como não poderia deixar de ser, na sua composição inclui um *map* cujas chaves são os códigos de produtos e os respetivos valores, instâncias da classe *VProdutos* estudada anteriomente.

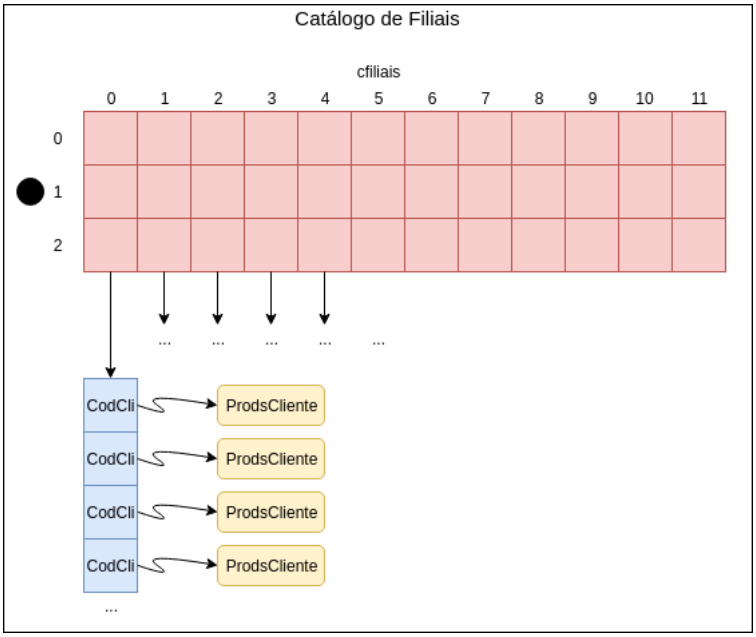

Figura 2.6: Gestor de filiais.

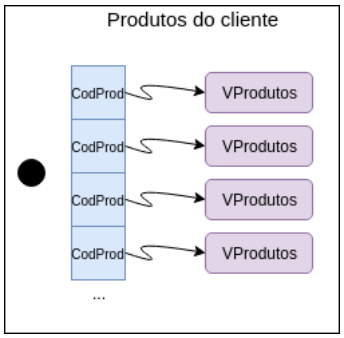

Figura 2.7: Produtos de um cliente.

Tal como todos os outros descritos anteriormente, este módulo tem métodos que retornam, por exemplo, os produtos mais vendidos, os clientes que mais compraram um produto, o número de clientes de uma filial, etc. Implementa ainda os construtores que criam a estrutura principal e as correspondentes "subestruturas".

2.1.6 Módulo lista de *strings* (*LStrings*)

A função deste módulo é possibilitar ao utilizador a visualização dos resultados de uma forma prática. Sem este módulo, teríamos que imprimir os dados todos seguidos no terminal onde executamos a aplicação, ficando alguns indisponíveis, se a quantidade impressa fosse mesmo elevada.

Por esse motivo, importamos para um *list strings* todos os resultados que queremos imprimir. Esse *list* está englobado na classe *LStrings* juntamente com dois campos, que nos indicam o número de elementos na lista e o número de linhas por página apresentada.

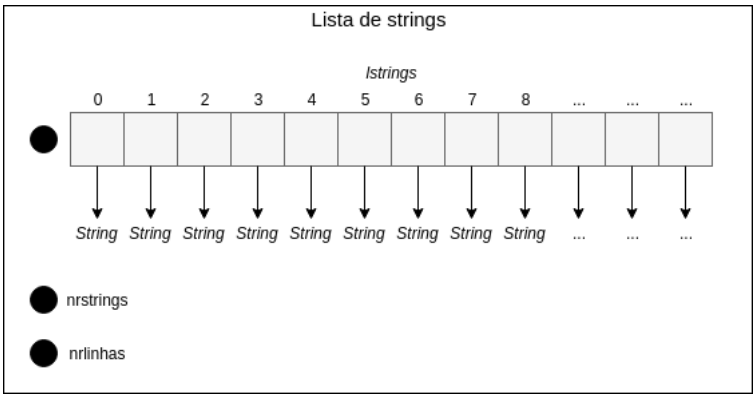

Figura 2.8: Listas de *strings*

Quando as *queries* assim o exigirem, o utilizador irá vizualizar a primeira página e conforme a instrução que submeter, poderá avançar ou retroceder nas várias "páginas"de resultados que existem.

2.1.7 Módulo leitura dos dados (*Leitor*)

A leitura dos dados é crucial para o bom funcionamento do programa. É nesta classe que estão presentes os métodos que leem os ficheiros *.txt* que contêm os dados. Temos um método que utiliza *BufferedReader* e outro que usa função *readAllLines*; é este último que efetivamente usamos, servindo o outro para realizar testes comparativos.

2.1.8 Módulo controlador (*GereVendasController*)

A execução da aplicação é, no fundo, um conjunto de funções que são chamadas sempre que escolhemos a *querie* a ser executada. Ora, para podermos estabelecer esta comunicação com o sistema de gestão

de vendas, é impreterível que tenhamos um módulo que trate deste ponto e que informe e peça todos os dados necessários ao utilizador para que a *query* escolhida apresente a informação correta. Para que tal seja possível, o mesmo apresenta como variáveis de instância um atributo do tipo *GereVendasView* e outro do tipo *GereVendasModel* executando métodos de uma ou de outra, conforme o que for necessário para responder às *queries*.

É necessário referir que na etapa anterior as queries eram tratadas num módulo próprio, não obstante, decidimos nesta fase que o controlador era o sítio indicado para as declarar.

2.1.9 Módulo *model* (*GereVendasModel*)

Esta classe inclui todas as estruturas que contêm os dados carregados e por esse motivo, irá ser usada apenas pelo controlador.

2.1.10 Módulo *view* (*GereVendasView*)

Toda a parte gráfica/estética que irá ser apresentada ao utilizador da aplicação está construida sob esta classe que terá métodos que irão ser chamados pelos controlador de forma a fomentar uma melhor interação com o utilizador.

2.1.11 Módulo globais (*Globais*)

Em qualquer programa são necessários determinados parâmetros que, por se repetirem e por serem comuns a todos os outros módulos, não faz sentido que sejam constantemente declarados. No nosso caso, variáveis como o número de meses e o número de filiais, são números exatos que não variam. Por esse motivo, decidimos criar um módulo que englobasse todas estas variáveis de tal forma que pudessem ser utilizadas por qualquer outro módulo.

2.1.12 Módulo heap (*MaxHeapDouble* e *MaxHeapInt*)

Ambas as classes têm o mesmo funcionamento, a diferença é que uma opera *integers* e outra opera *doubles*. O seu propósito é manter um *map*, que guarda como uma das suas variáveis de instância, organizado por ordem descrescente das chaves (algo que é bastante requisitado nos encargos propostos). O tipo da chave é um inteiro/double e o valor é um conjunto de *String*s, todas elas associadas a uma determinada chave. Estruturas como estas são necessárias às *queries* que pedem os "N clientes", os "N produtos", etc.

2.1.13 Módulos auxiliares (*ClisProd*, *Crono*, *Input*, *InfoCliente*)

Durante o projeto foram necessárias algumas classes que não foram referidas em cima, dada a pouca importância que teem na aplicação, não obstante, em determinados aspetos são bastantes úteis e por essa razão explicamos as vantagens de cada uma a seguir:

- **ClisProd** - Reune numa das suas variáveis de instância todos os clientes que compraram um determinado produto;
- **Crono** - Útil no cálculo do tempo de uma determinada ação da aplicação;
- **Input** - Classe que alberga os vários métodos de leitura de tipos de dados (ex.: *String*, *int*, *double*, etc.);
- **InfoCliente** - Necessária para o cálculo e reunião da informação dos vários produtos de um cliente.

2.2 Arquitetura da aplicação

Como em qualquer programa em linguagem *C* ou linguagem *JAVA*, existe um módulo que contém a função/método *main*, a primeira a ser invocada pelo sistema operativo e neste etapa é a classe *GestVendasAPP* que a tem definida.

Posteriormente é criado um *Controller* que irá conter uma instância de *Model* e outra de *View*. É aqui que vemos a arquitetura *MVC* implementada, onde um controlador opera e conjuga os módulos visão e modelo colocando-os em interação. Esta arquitetura permite a separação da camada de dados da camada gráfica, podendo, caso seja pretendido, aplicar outros modelos gráficos à mesma camada de dados.

É no *Controller* que são utilizadas várias instâncias de classes que ajudam na resposta às várias *queries*, como por exemplo, instâncias de *Input* e/ou *LStrings*.

Finalmente, a classe *Model* terá uma instância de *CClientes*, outra de *CProdutos*, outra de *CFaturacao* e outra de *CFiliais*. As duas útlimas irão ter *Maps* cujos valores serão instâncias das classes *VProdutos* e *ProdsCliente*, respetivamente.

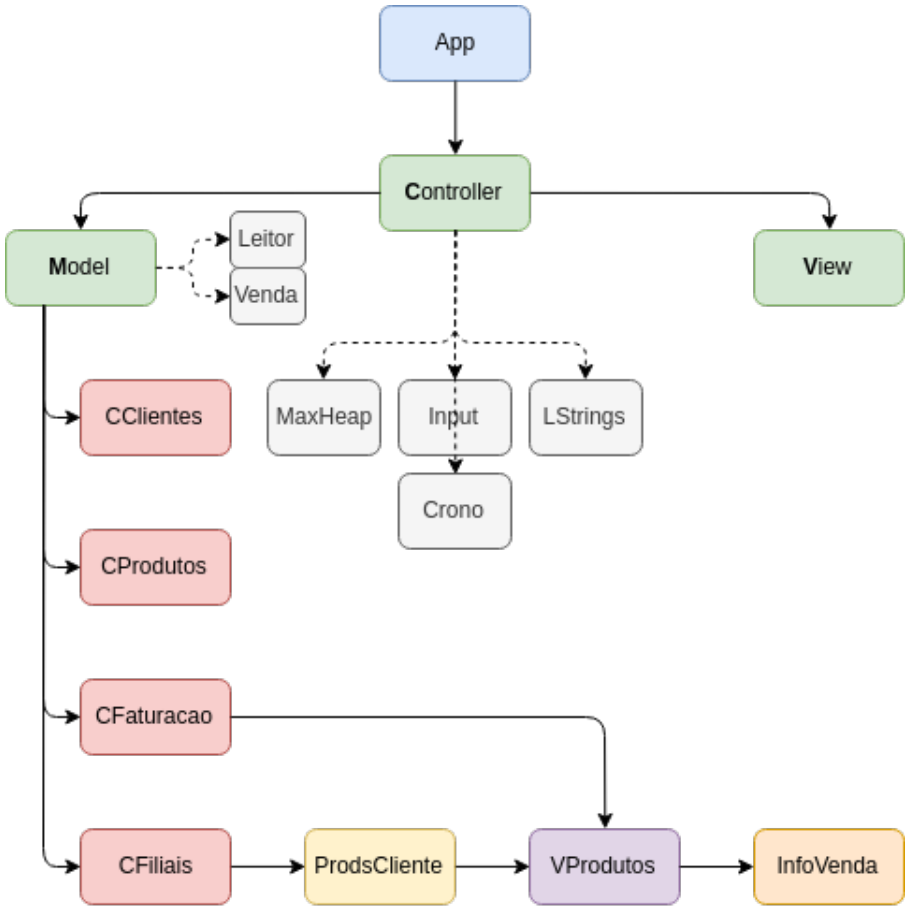

Figura 2.9: Arquitetura do *SGV*

A arquitetura escolhida contém ainda outras classes que simplesmente ajudam na agregação dos dados que se querem obter para a resposta a determinadas queries, não sendo apresentadas no esquema acima por essa razão. São exemplo dessas classes a *ClisProd* e a *InfoCliente*.

2.3 Complexidade das estruturas

O objetivo, desde o início, foi minimizar o tempo de obtenção das respostas referentes às diversas *queries*. Como já foi explicado, uma forma instantânea de tornar o programa mais rápido foi dividir os clientes e os produtos por *set*s, cada um associada a uma letra.

De resto, a complexidade de cada estrutura do sistema de vendas é totalmente controlada pelos métodos oferecidos pelas interfaces *Collection* e *Map*; saliente-se que qualquer uma destas estruturas eram/-foram e continuam a ser/estar extremamente otimizadas, garantindo, por exemplo, tempos de procura incrivelmente rápidos. Para termos uma breve noção, nos métodos *get*/*put*, no melhor caso a complexidade é de O(1), isto acontece quando a função de *hash* é bastante boa a ponto de evitar colisões, que é seguramente o que acontece com mais frequência. Contudo, quando acontece o infortúnio de haver várias entradas com o mesmo código *hash* a complexidade passar a ser O(N), o que, apesar de ser bastante raro, caracteriza o pior caso. É neste sentido que todo o o projeto foi construído.

2.3.1 Resultados obtidos

Recorde-se que na primeira fase do projeto, teve-se a preocupação inicial de comparar a diferença entre *arrays* dinâmicos e estruturas organizadas com *avl*s. As diferenças registadas foram absurdas e neste projeto, essa possibilidade não foi sequer equacionada. Aquilo que de facto se fez, foram vários testes de forma a comparar o uso de várias estruturas da própria linguagem *JAVA*, sendo que todas elas estão já extremamente otimizadas.

Posto isto, foi pedido que se analisassem e permutassem estruturas que se tinham implementado por outras pertencentes às mesmas coleções, isto é, onde se usassem *HashMap*s passava-se a usar *TreeMap*s, onde se usassem *HashSet*s passava-se a usar *TreeSet*s e onde se usassem *List*s passava-se a usar *Vector*s e vice-versa para cada um dos casos. A ideia era que se anotassem e verificassem as descrepâncias temporais usando umas ou outras estruturas, algo que está registado nas tabelas que se seguem.

Leituras	Clientes	Produtos	Vendas_1M	Vendas_3M	Vendas_5M
Files	0.024	0.038	0.851	2.104	3.252
BufferedReader	0.020	0.037	0.628	1.799	2.981

Tabela 2.1: Tempos de leitura sem *parsing*, validação e/ou inserções nas estruturas (segundos).

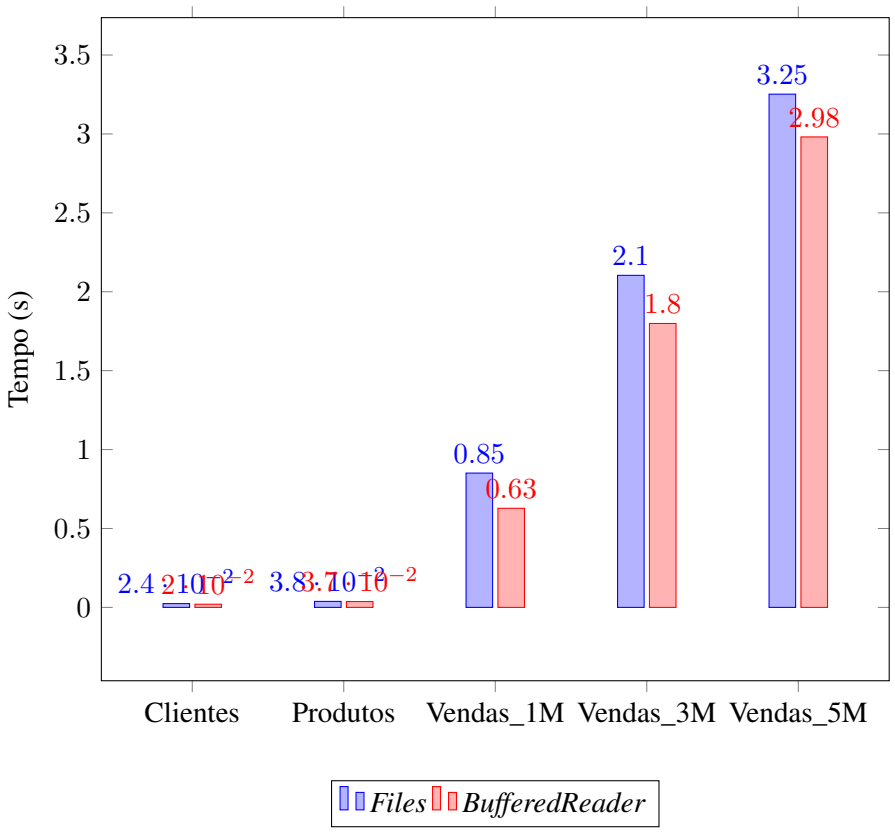

Figura 2.10: Gráfico relativo à tabela 2.1.

A tabela 2.1 e o respetivo gráfico mostram os tempos obtidos depois da leitura de todos os ficheiros sem qualquer tipo de *parsing*, *validação* ou *inserção*. Note-se que com *Files*, a leitura parece ser mais demorada, mas dada a insignificante diferença, decidiu-se que se fariam as leituras deste modo porque em teoria, esta é uma classe bastante mais otimizada que a classe *BufferedReader*. A explicação para não se conseguir notar a diferença deve-se ao facto de estarmos a utilizar ficheiros cujo tamanho ainda não nos permite ver essa descrepância.

Inserções s/ validação	Clientes	Produtos	Vendas_1M	Vendas_3M	Vendas_5M
Files	0.053	0.110	5.552	22.117	40.617
BufferedReader	0.046	0.111	5.341	19.382	48.454

Tabela 2.2: Tempos de inserção nas estruturas com *parsing* e sem validação (segundos).

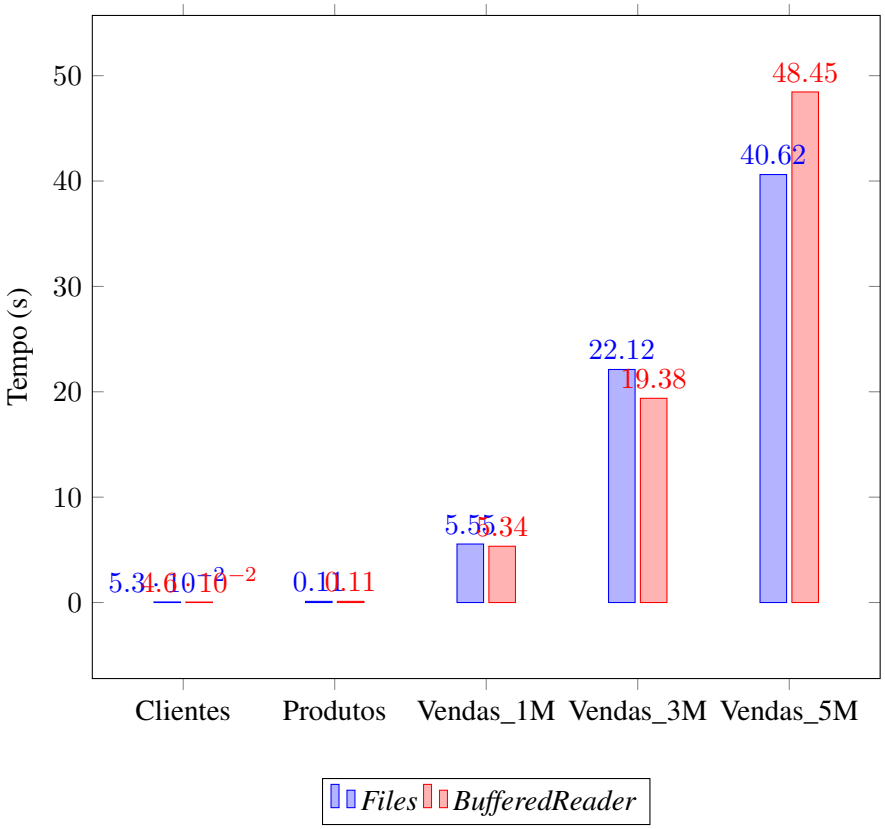

Figura 2.11: Gráfico relativo à tabela 2.2.

A tabela 2.2, apresenta os tempos de inserção nas estruturas com *parsing* e sem validação. Consegue-se constatar que os tempos de leitura e inserção dos dados presentes nos vários ficheiros de vendas vão aumentando à medida que aumenta o tamanho dos ficheiros, algo expectável. No gráfico, conseguimos ainda detetar uma descrepância significativa entre o uso da classe *Files* e da classe *BufferedReader*, apresentando a primeira o melhor tempo.

Seria previsível que nas inserções com validações de dados os tempos aumentassem um pouco, algo que se verifica na tabela a seguir.

Inserções c/ validação	Clientes	Produtos	Vendas_1M	Vendas_3M	Vendas_5M
Files	0.029	0.100	5.936	25.804	60.179
BufferedReader	0.040	0.106	6.532	25.174	63.936

Tabela 2.3: Tempos de inserção nas estruturas com *parsing* e validação (segundos).

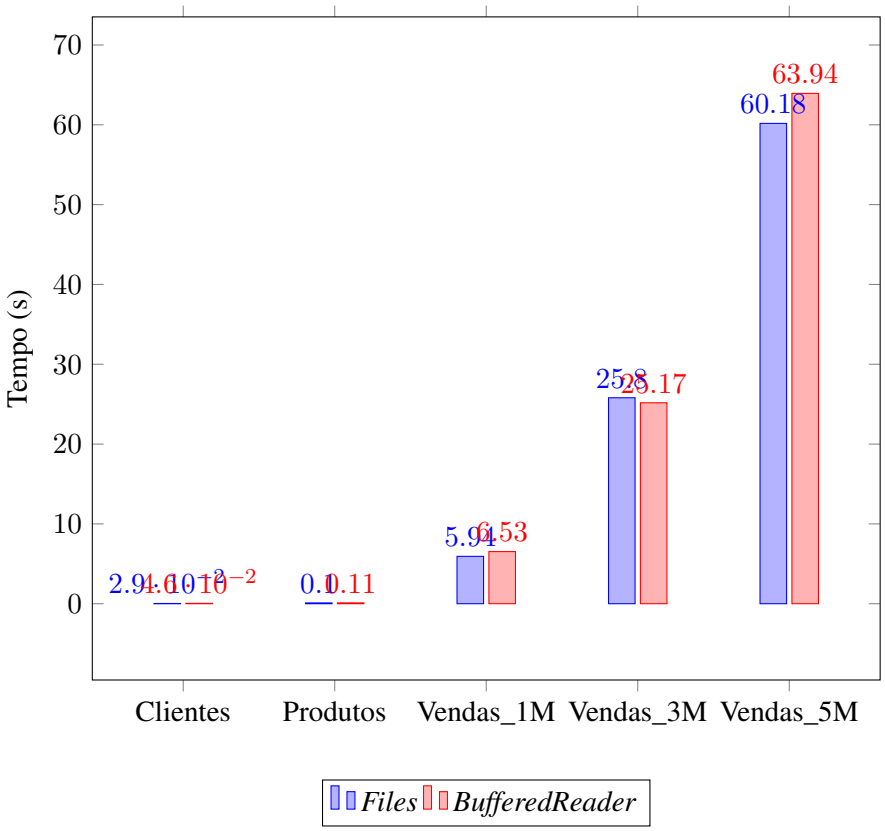

Figura 2.12: Gráfico relativo à tabela 2.3.

Para se ver o impacto que a validação dos dados tem nos tempos da aplicação, decidiu-se medi-los e compara-los com os tempos anteriores. A tabela 2.3 mostra um resultado previsível, aumentando sensivelmente em vinte segundos o tempo de inserção dos dados nas estruturas para o ficheiro com cinco milhões de vendas. A leitura com a classe *Files* teve vantagem, mais uma vez, em relação à leitura com a classe *BufferedReader*.

Queries	*Query 5*	*Query 6*	*Query 7*	*Query 8*	*Query 9*
HashMap, HashSet, ArrayList	0.006	1.424	0.131	0.794	0.075
TreeMap, TreeSet, Vector	0.002	1.977	0.210	0.746	0.107

Tabela 2.4: Tempos das *queries* com diferentes estruturas.

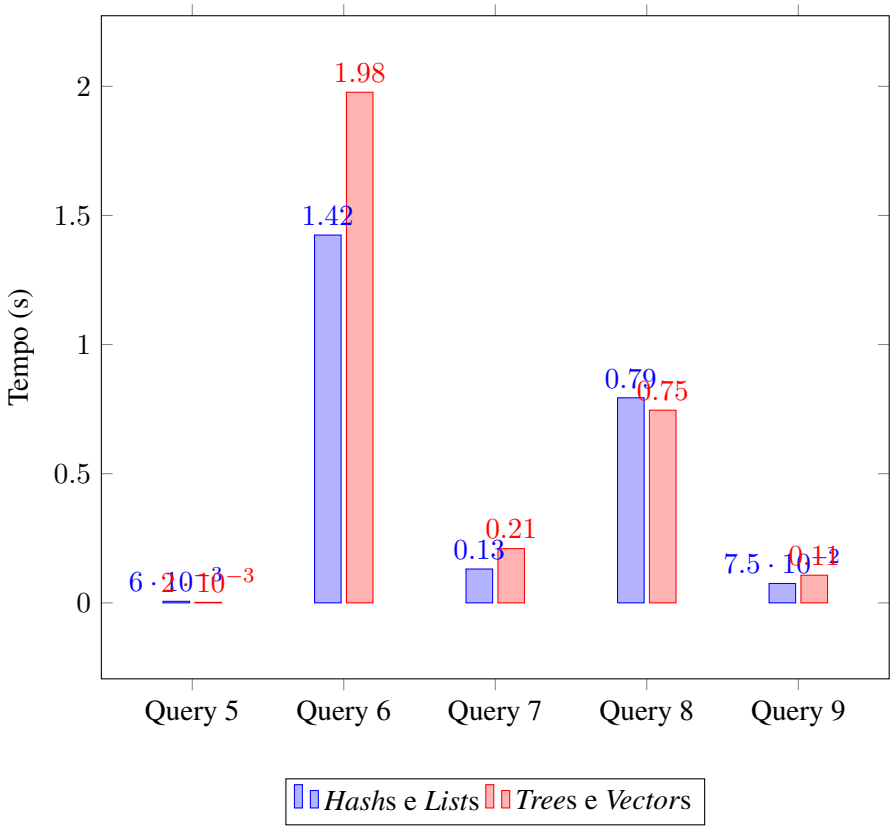

Figura 2.13: Gráfico relativo à tabela 2.4.

Resta-nos analisar o tempo de resposta às *queries* do utilizador. A *query* onde a diferença é mais significativa é a sexta, onde a leitura com estruturas de *Hash* e *List* é 0.553 segundos mais rápida que com as outras estruturas. De resto, na tabela 2.4 não se verifica qualquer descrepância extraordinária que prove a superioridade das estruturas de *Hash* e *List* em relação às suas concorrentes.

Apesar da tabela apresentar dados relativamente iguais, efetuou-se ainda um teste de inserção com *parsing* e validação nas estruturas de dados em que as interfaces e coleções utilizadas foram os *TreeMap*s, *TreeSet*s e os *Vector*s. Com estas estruturas o tempo de inserção que foi registado para o ficheiro de um milhão de vendas foi de 10.520 segundos, o dobro do tempo medido com as estruturas concorrentes (5.936 segundos). O mesmo foi verificado para os ficheiros com três e cinco milhões de entradas, o que reitera a **necessidade do uso de estruturas do tipo *HashMap*, *HashSet* e *List*** caso a ordenação não seja necessária.

3. Conclusão

A unidade curricular de Laboratórios de Informática III revelou-se uma ótima oportunidade para aprimorarmos os nossos conhecimentos de linguagem *C* e *JAVA*. Foi extremamente útil na medida em que ficamos com uma melhor preceção daquilo que é um projeto que lida com grandes volumes de dados, algo que ficou ainda mais evidente nesta segunda fase, visto que verificamos que triplicar ou quintiplicar o número de entradas do ficheiros vendas, torna a nossa aplicação extremamente mais lenta.

Na fase anterior, a construção dos diferente módulos permitiu-nos começar a perceber como funcionam linguagens de alto nível, nomeadamente, a linguagem *JAVA*, e nesta fase, podemos verificar isso mesmo. Dividir o código escrito pelos diversos módulos não só é uma muito boa prática, como nos ajuda a abstrair a aplicação do mundo real para o mundo da programação e a linguagem em questão foi construída nesse sentido, isto é, tudo aquilo que existe e que é passível de ser programado, pode ser visto como um módulo com vários atributos e vários métodos, pode ser visto como um objeto abstrato.

Por explorar fica o conceito de *interface*, que apesar de já ter sido compreendido não entrou na conceção deste projeto, algo que mais tarde, certamente, será tido em conta.