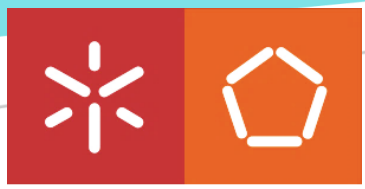


Universidade do Minho
Escola de Engenharia

LABORATÓRIOS DE INFORMÁTICA III

RELATÓRIO PROJETO JAVA

SISTEMA DE GESTÃO DE VENDAS - SGV

Grupo 67

Ana Luísa Carneiro A89533
Ana Rita Peixoto A89612
Luís Miguel Pinto A89506

ÍNDICE

INTRODUÇÃO

O projeto “Sistema de Gestão de Vendas” propõe a implementação de um programa que seja capaz de processar grandes volumes de dados de maneira eficiente, utilizando os princípios de encapsulamento, modularidade, asbtração e programação com interfaces.

No decorrer do desenvolvimento deste projeto, foram surgindo alguns desafios, os quais passamos a enumerar:

- O primeiro desafio consistiu em perceber como funciona a leitura de ficheiros em Java, quais as opções que tinhamos ao dispor e qual seria a mais benéfica.
- Surgiu outro desafio, em particular para algumas queries, que foi descobrir uma estratégia que conseguisse ordenar valores por ordem decrescente numa coleção, e ao mesmo tempo preservar essa ordem.
- Também consideramos que foi desafiante construir este projeto, dado que foi a primeira vez que tivemos contacto com um paradigma de programação orientado a objetos, em particular, destacamos a ambientação ao uso de interfaces.

ESTRUTURA DE DADOS

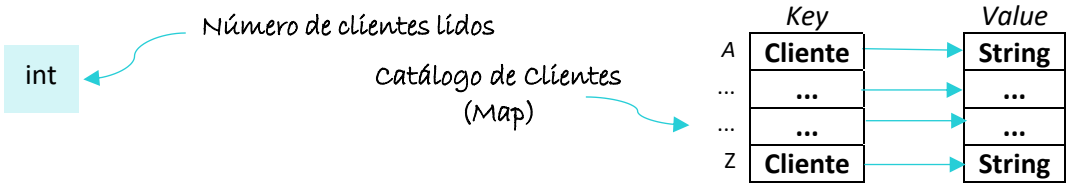
Catálogo de Clientes

Na estrutura para o **Catálogo de Clientes** optamos por uma implementação que permitisse associar dois objetos. Neste caso, associar um objeto da classe *Cliente* (classe que trata de efetuar os métodos relativos a um código de cliente), com um objeto da classe *String* (de modo a guardar a concatenação em *String*, das filiais onde o *Cliente* efetuou compras), da seguinte forma:

Para tal, optamos por utilizar a implementação *Map* (enquanto variável de instância) ,de forma a aumentar a abstração, para efetuar a correspondência entre os objetos referidos anteriormente. Especificamente, optamos pela coleção *TreeMap*, para que fosse possível populacionar esta estrutura com os Clientes por ordem alfabética. Deste modo, foi necessário redefenir o método compareTo da classe *Cliente*.

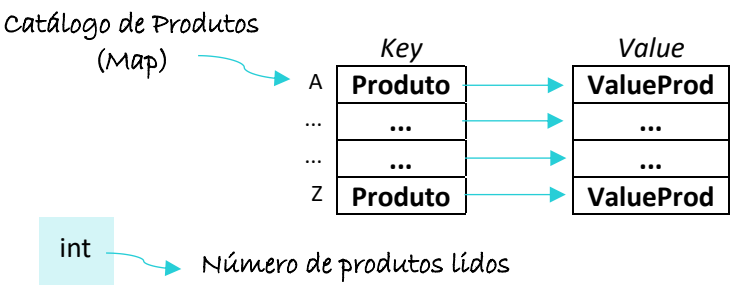
Além disso, nesta classe para o Catálogo de Clientes, está também presente uma variável do tipo *int*, que permita armazenar o número de clientes que foram lidos do ficheiro "Clientes.txt".

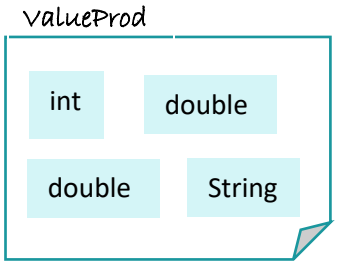
Em suma, de acordo com o referido anteriormente, podemos esquematizar as variáveis de instância da classe desenvolvida do seguinte modo:



Catálogo de Produtos

No caso da estrutura para o **Catálogo de Produtos**, a solução foi análoga. Para manter registo do número de produtos lidos optamos por usar um '*int lidas*' (visto que é pratico e eficaz), e um '*Map<Produto, ValueProd > catalogo*' para armazenar o dito catalogo de produtos, esta decisão baseou-se principalmente na necessidade de manter uma correspondência unívoca entre produto e informação do produto. Deste modo, implementamos as classes Produto e ValueProd, a primeira contendo uma 'String produto' com o nome/código do produto e a segunda com informação sobre o numero de vendas('int vendas'), quantidade ('double qnt'), faturacao a ('double preco') e filiais onde foi comprado ('String filial'). Assim tem-se :



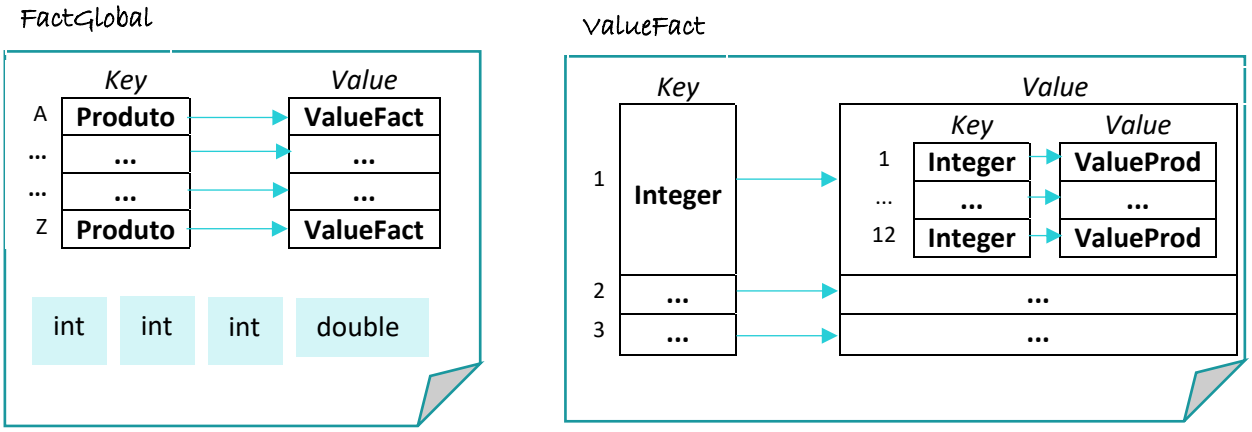


Faturação Global

Para a **Faturação Global**, consideramos conveniente encadear diversas coleções, de modo a repartir cada objeto da classe *Produto* nas filiais e meses onde foi vendido. Em adição, implementamos também 4 variáveis onde fosse possível guardar informação acerca do número de vendas lidas, número de vendas válidas, número de vendas cuja faturação é zero e, por fim, uma variável com a faturação total de todas as vendas ocorridas.

Deste modo, começamos por criar uma variável de instância da coleção *Map*, que permitisse estabelecer uma correspondência entre um objeto da classe *Produto(key)*, e uma outra classe *ValueFact(value).* Dentro desta última, está implementado um *Map* que associa a cada filial (*key)* um *Map(value)*, cuja *key* trata os meses, e o *value* é um objeto da classe *ValueProd* onde são armazenadas diversas informações úteis sobre cada produto, tais como o número de unidades vendidas, a faturação e o número de vendas.

De forma a manter os objetos da classe *Produto* organizados por ordem alfabetica, foi implementado na classe *Produto*, um método compareTo. Além disso, no momento de inserção dos objetos *Produto* no *Map*, especificamos a estrutura para *TreeMap*, para que respeitasse a ordem pretendida.

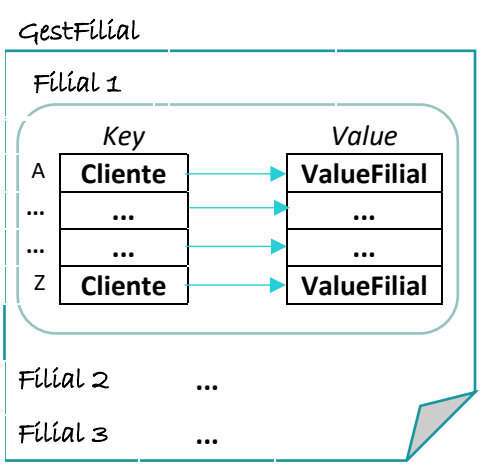


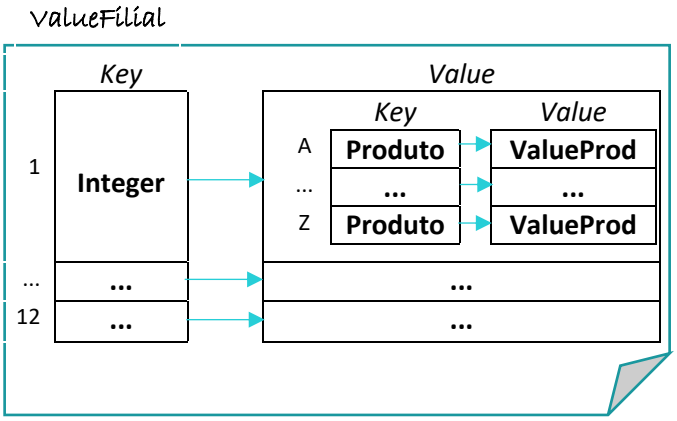
Gestão de Filial

Em relação à **Gestão de Filial**, optamos por criar 3 mapeamentos do tipo *'Map<Cliente, ValueFilial>'*, um para cada filial. Esta decisão colmata dois aspetos importantes das consultas : (1) Obter resultados independentes por filial; (2) Associar a um cliente a respetiva informação sobre as suas compras.

Assim sendo, usamos como chave de mapeamento um Cliente, estrutura já mencionada em cima, e para valor de mapeamento criamos uma classe 'ValueFilial'. Ora, visto que algumas consultas necessitam de resultados mensais, achamos sensato dividir esta ultima classe por meses e criar, mais uma vez, um mapeamento entre mês e informação mensal *'Map<Integer, Map<Produto, ValueProd>',* a "informação mensal" traduz-se portanto num mapeamento entre Produto e ValueProd, ambas classes já mencionadas anteriormente.

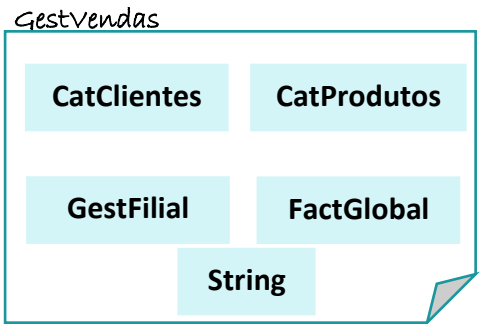
Em resumo, teremos em cada filial uma informação mensal de cada cliente, e dentro dessa informação mensal consta, para cada produto comprado pelo cliente, as propriedades associadas a compra do produto. Assim, a estrutura implementada baseia-se na seguinte esquematização:





GestVendas

Assim, em consequência do referido anteriormente, surge o módulo **GestVendas** que agrega todas as estruturas anteriormente expostas, de modo a completar a estrutura principal para o Sistema de Gestão de Vendas proposto a desenvolver. De acordo com a representação esquemática, a classe GestVendas contém as diversas estruturas onde estão armazenados os dados lidos dos ficheiros, e também 1 valor da classe *String* onde está guardado o caminho do último ficheiro de vendas lido.



ARQUITETURA DA APLICAÇÃO

Estrutura do projeto

O projeto encontra-se inserido num *package* inicial "SGV" que foi dividido em três *packages* ("Model", "View", "Controller") de modo a respeitar o padrão MVC.

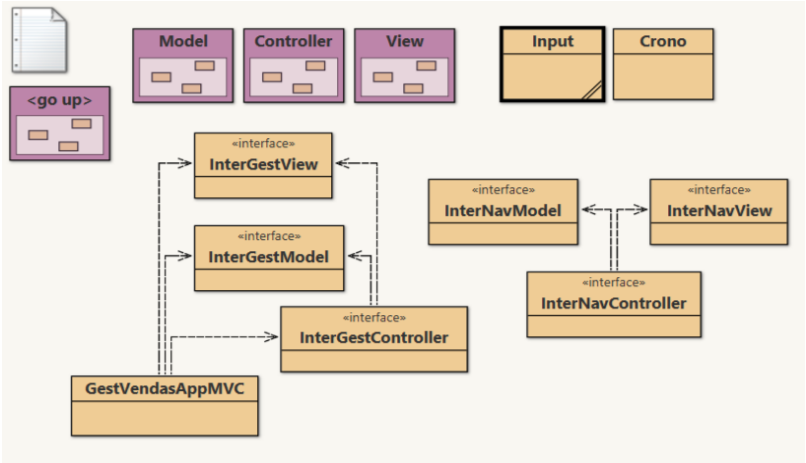


Figura 1: Package SGV

Assim, o *package* model, que contém as camadas de dados e algoritmos, está organizado de acordo com o seguinte diagrama de classes[1]:

[1] Obtidos através do BlueJ.

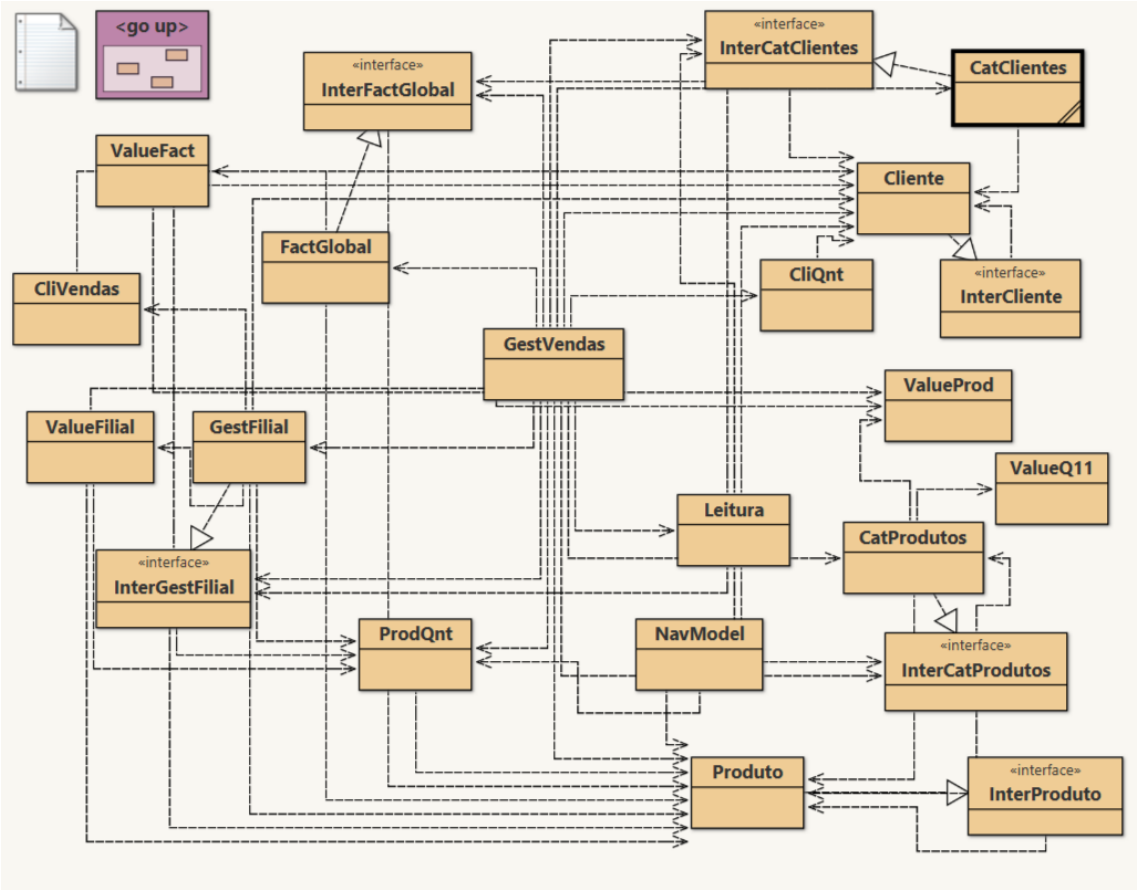


Figura 2: Package Model

E, por fim, ilustramos os *packages* View (que se encarrega da apresentação ao utilizador) e Controller (controlo de fluxo) nos seguintes diagramas[2]:

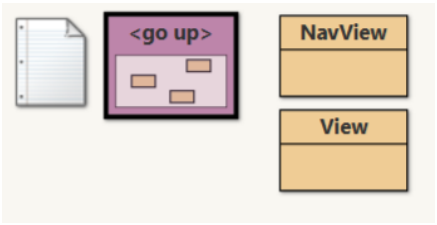


Figura 3: Package View

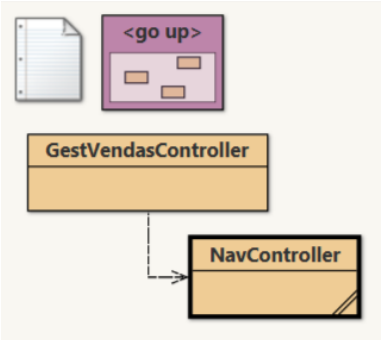


Figura 4: Package Controller

Funcionalidades das classes

Antes de mais, com efeito de oferecer uma melhor experiência de navegabilidade ao utilizador foi desenvolvida uma ferramenta de apresentação de resultados (por páginas), que devido à sua complexidade e de forma a cumprir os requisitos do projeto, foi dividida em 3 classes:

NavController: "mini-controlador-de-fluxo", após ser invocado pelo controlador *GestVendasController* tem temporariamente o controlo (enquanto o utilizador navega pelos resultados), ou seja, simplesmente avança ou recua páginas da apresentação de resultados.

[2] Obtidos através do BlueJ.

NavModel: modelo de dados específico para o conjunto navegador.
NavView: "vista secundária" do programa, apresenta no ecrã páginas e páginas com resultados, podendo estes ser *ranks*, produtos ou clientes.

Em adição às classes mencionadas anteriormente, desenvolvemos também:

GestVendasAppMVC: responsável pela inicialização do programa.

GestVendasController: trata da sincronização do programa com o utilizador, i.e, controlador principal do fluxo do programa.

View: responsável pela apresentação no ecrã da interação atual com o utilizador.

GestVendas: contém o conjunto de todas as funcionalidades suportadas pela aplicação.

Leitura: responsável pela leitura dos dados (ficheiros clientes, produtos e vendas) e redireccionamento de forma a serem devidamente tratados.

CliQnt: classe auxiliar à query 9, contendo para isso um cliente e uma quantidade.

Cliente: contém os métodos referentes a cada cliente.

Produto: contém os métodos referentes a cada produto.

ProdQnt: classe auxiliar à query 5, contendo para isso um produto e uma quantidade.

GestFilial: organiza os dados conforme a filial, apesar de vocacionado para os clientes faz também a ligação do cliente com os respetivos produtos.

CatClientes: contem a estrutura com o catálogo dos clientes e respetivas funções sobre clientes.

FactGlobal: classe onde estão armazenados os produtos divididos pelos meses e filiais onde foram comprados.

CatProdutos: contem a estrutura com o catálogo de produtos e respetivas funções sobre produtos.

ValueProd: classe auxiliar ao *Produto* que contém informações do mesmo.

ValueFilial: complementa a classe GestFilial.
ValueFact: complementa a classe FactGlobal.
ValueQ11: classe auxiliar à query 6.

Em suma, as classes referidas estão inseridas nos packages respetivos:

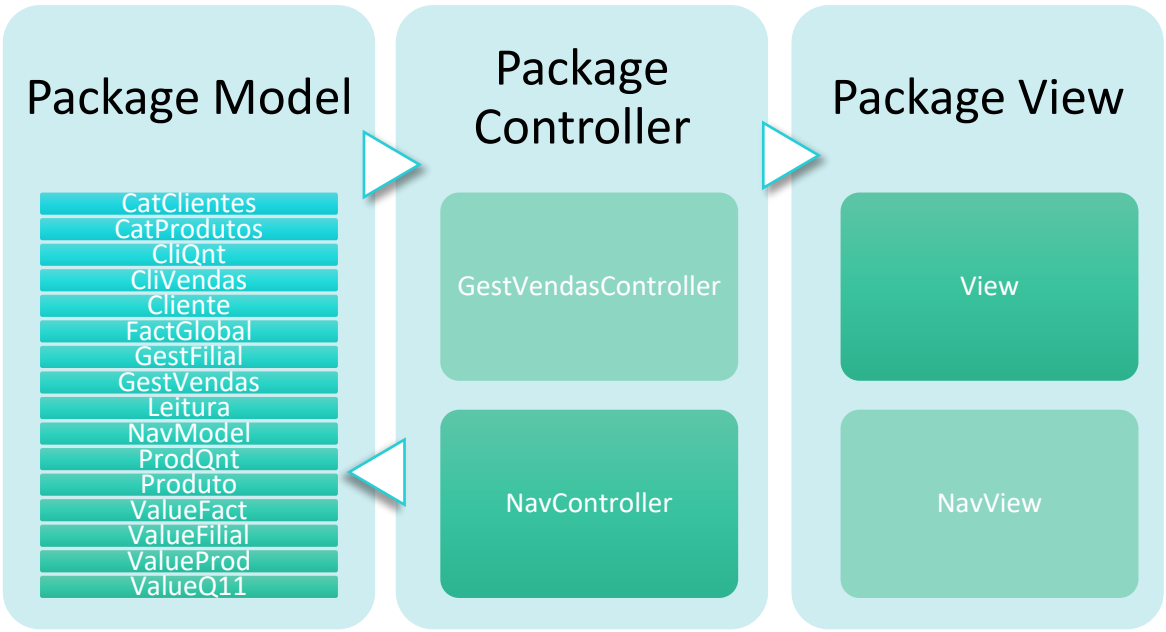


Figura 5: Modelo MVC

TESTES DE PERFORMANCE

Os testes de performance foram realizados na mesma máquina através da média de 3 medições. Para determinar os tempos de execução usou-se a **classe Crono** fornecida pelos docentes. Todos as tabelas com os tempos de execução encontram-se em Anexo.

PERFORMANCE DE LEITURA

Para os testes de performance na leitura testou-se o armazenamento dos dados em diferentes estruturas – **ArrayList**, **HashSet** e **TreetSet** – para os ficheiros de 1M, 3M e 5M de vendas, sendo que a leitura destes ficheiros para o sistema foi feita através da classe **BufferedReader**. Não foram testadas estruturas como o *Map* pois essas são semelhantes aos *Sets* com a particularidade de terem o campo *value* que para efeito destes testes não foi usado.

Leitura Sem Parsing (Ver Anexos)

Testes de performance realizados sem divisão dos parâmetros das vendas, com o armazenamento nas estruturas a ser feito em **String**, tal como está no ficheiro de vendas. Assim para as estruturas de TreeSet e HashSet, o *compareTo* tal como o *hashCode* utilizados são os definidos na classe String.

Leitura Com Parsing (Ver Anexos)

Testes de performance realizados com divisão dos campos das vendas. Para o efeito foi criado uma **classe Venda** cujas variáveis de instância são os parâmetros divididos da String venda. Para essa divisão foi usado o método *split* da classe String.

Para o armazenamento na estrutura TreeSet foi usado o *compareTo* criado na classe Venda que compara objetos da classe Venda primeiro pelo parâmetro produto, pelo parâmetro quantidade e em seguida pelo parâmetro preço.

Leitura Com Parsing e Validação (Ver Anexos)

Testes de performance realizados com divisão dos campos das vendas assim como validar se estes campos estão corretos, isto é determinar se a venda é valida. A inserção nas estruturas, tal como na leitura com parsing, foi feita através da classe Venda.

Para validar campos como o produto e cliente é necessário a leitura e armazenamento do catálogo de produtos e clientes no sistema. Assim esses campos são válidos se o cliente e o produto também estiverem nos catálogos. Para isso armazenaram-se esses catálogos numa TreeSet, uma vez que a pesquisa por um produto ou cliente é muito mais eficiente e rápida quando há uma ordem entre os elementos. Como teste usou-se o armazenamento dos catálogos em ArrayList mas o processo foi bastante demorado (>4minutos).

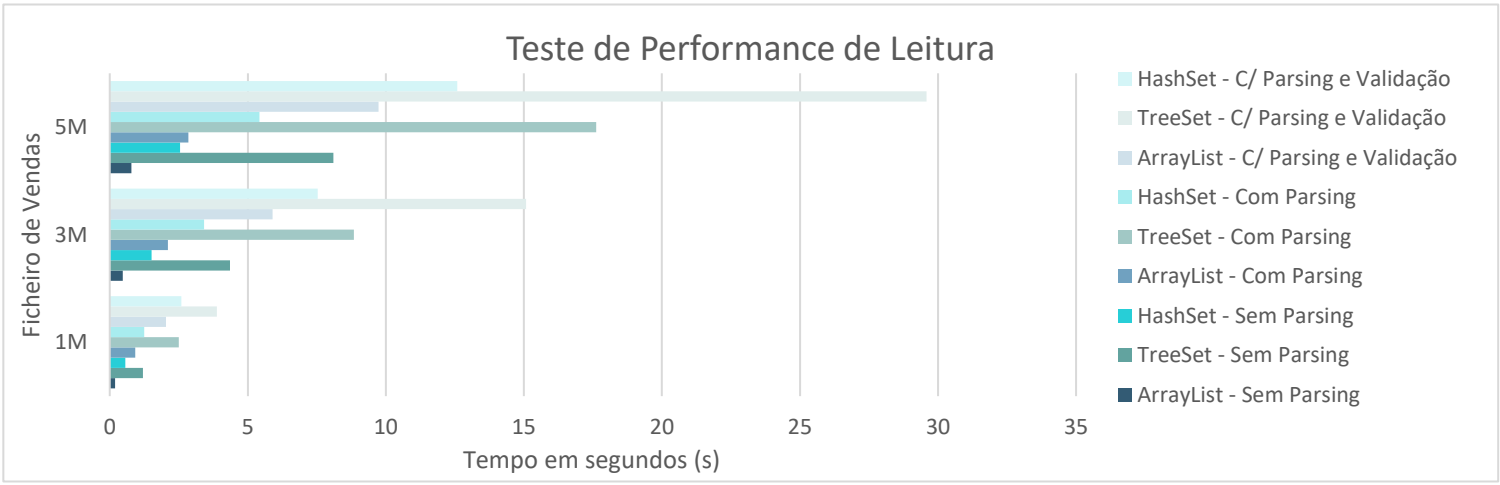
Para o armazenamento das vendas na estrutura de TreeSet manteve-se o *compareTo* criado na classe Venda.

Interpretação de Resultados

Pelo gráfico conseguimos ver que o armazenamento numa estrutura como o TreeSet, ou seja, em que os seus elementos possuem uma certa ordem, faz se de forma menos eficiente uma vez que tem de haver comparação entre o elemento que se vai inserir e o que já lá esta.

O mesmo podemos dizer para o HashSet já que temos de definir um *hashCode* para cada elemento que vai ser introduzido. Para além disso, tal como era previsto, o tempo de execução vai aumentando com o nº de vendas presentes no ficheiro.

No gráfico abaixo está representado as médias dos testes de leitura[3]:



PERFORMANCE DAS QUERIES

Os testes de performance das queries 5 a 9 foram executados para os ficheiros de 1M, 3M e 5M de vendas com a introdução dos mesmos dados.

Query5: Cliente: Z5000
Query6: Top: 10
Query7: -----
Query8: Top: 10
Query9: Produto: WD1962 | Top: 10

Execução em TreeMap

Numa primeira instância executou-se o código criado aquando da realização do programa *GestVendas*, ou seja, com a implementação inicial em **TreeMaps** da estrutura da classe *CatProdutos* (usada para obtenção dos resultados da query6) e da estrutura da classe *GestFilial* e das suas classes auxiliares (usada para obtenção dos resultados da query5,7,8 e 9).[4]

Ficheiro		Query5	Query6	Query7	Query8	Query9
Vendas_1M	Trial 1	0,006	2,487	0,234	14,2398	0,15046
	Trial 2	0,001	2,4161	0,30499	13,779	0,1136
	Trial 3	0,00189	2,457	0,1428	13,28496	0,10455
	Media	**0,002963**	**2,453367**	**0,227263**	**13,76792**	**0,12287**
Vendas_3M	Trial 1	0,01319	3,4558	0,5196	11,40197	0,2322
	Trial 2	0,004818	3,8867	0,4019	10,9122	0,1837
	Trial 3	0,001519	3,3156	0,4073	11,1726	0,1867
	Media	**0,006509**	**3,5527**	**0,442933**	**11,16226**	**0,200867**
Vendas_5M	Trial 1	0,01295	4,0899	1,4428	12,5245	0,2837
	Trial 2	0,0093	4,1316	0,8238	11,738	0,2452
	Trial 3	0,0026	4,3714	0,8358	11,9857	0,2345
	Media	**0,008283**	**4,197633**	**1,034133**	**12,08273**	**0,254467**

[3] Gráfico gerado com apoio ao Microsoft Excel
[4] Os tempos de execução da tabela encontram-se em segundos

Pela tabela vemos que os tempos de execução para as queries 5,6,7 e 9 são os esperados para aquilo que era pedido, no entanto conseguimos ver que há uma grande discrepância entre os tempos de execução da query8 para as outras queries. Neste query era pedido para determinar o top N dos clientes que mais compraram produtos diferentes, assim era necessário percorrer todos os clientes que compraram em cada filial. Contudo como os clientes estão divididos por filiais é necessário verificar se o cliente já foi verificado numa outra filial, isto é, se o cliente comprou na filial 1 e na filial 2 (por exemplo), deste modo existe uma perda de eficiência. Para além disso, como será sempre necessário percorrer todos os clientes para gerar o top N, a performance vai ser semelhante independentemente do N que o utilizador peça.

Execução em HashMap (Ver Anexos)

Apos a primeira execução em todos os ficheiros de vendas foram feitas mudanças nas classes de *CatProdutos* e *GestFilial* (e classes auxiliares desta última) para que estas, em vez de implementarem TreeMaps nas suas estruturas, implementassem **HashMaps**.

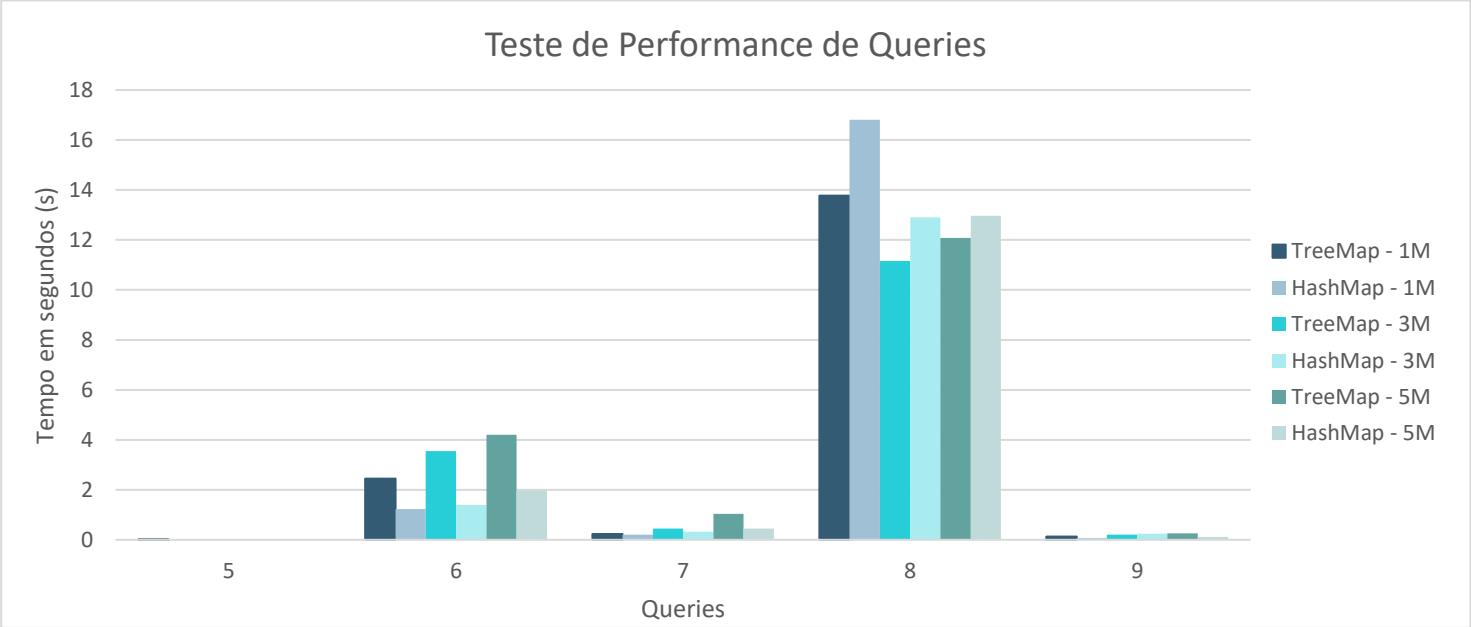
Decidiu-se manter as estruturas para as queries inalterada e mudaram-se só as estruturas de armazenamento de dados (package Model), visto que o tempo de execução é exclusivo da obtenção de dados e não na passagem de dados para o utilizador.

Como foram implementadas nas classes abstração das estruturas através da implementação de interfaces como a Map, a alteração do código fez se de forma eficiente. Para além das alterações já mencionadas, para que o código executasse de forma correta foi necessário a criação, quer na classe Produto quer na classe Cliente, dos métodos *hashCode* e *equals*, sendo que o primeiro retorna o *hashCode* da String produto/cliente.

Interpretação de resultados

Pelo gráfico podemos ver que para as queries 5,7 e 9 os tempos de execução mantiveram-se semelhantes. Contudo para a query6 o HashMap foi mais eficiente e para a query8 o TreeMap teve uma execução mais rápida.

No gráfico abaixo está representado as médias dos testes das queries para cada ficheiro de vendas e cada estrutura implementada[5]:



[5] Gráfico gerado com apoio ao Microsoft Excel

CONCLUSÃO

Dado por concluído o nosso projeto, apresentamos uma reflexão sobre possíveis melhorias, assim como aspetos a valorizar.

No desenvolvimento do projeto de Java, tivemos em consideração as críticas construtivas dos docentes, e as melhorias que consideramos que podíamos ter implementado no projeto de C:

- Implementamos funcionalidades extra no navegador:
  - Permitir ao utilizador avançar para uma dada página à sua escolha;
  - Habilitamos a pesquisa por um qualquer elemento no navegador.
- Tentamos manter as classes encapsuladas, opacas e abstratas. Assim, qualquer informação sobre uma dada classe, é consultada e gerida na própria classe.

Por outro lado, consideramos que existem pontos a melhorar no nosso projeto:

- Nas queries 6 e 8, os tempos de execução verificam-se mais elevados do que o pretendido.
- Na eficiência do trabalho, pois não verificamos o fator consumo de memória na tomada de decisões.
- Em relação à estrutura para a Gestão de Filial, que em C consideramos demasiado complexa, criamos em Java uma implementação mais simples, no nosso parecer. No entanto, segue a mesma linha de raciocínio, pelo que, consideramos que ainda poderia ser melhorada.

Por fim, consideramos que existem também aspetos positivos:

- Os tempos de execução das restantes queries (exceto das queries 6 e 8) corresponderam às expectativas.
- Apresentamos uma interface simples e completa ao utilizador.
- O nosso programa está funcional.

De um modo geral, o trabalho cumpriu com os requisitos propostos e apesar de haver melhorias, achamos que o balanço é positivo.

ANEXOS

Tabelas com as medições feitas assim como a media dos tempos de execução dos testes de performance. Os tempos de execução em todos as tabelas estão em segundos.

LEITURA SEM PARSING

ESTRUTURAS		Vendas_1M.txt	Vendas_3M.txt	Vendas_5M.txt
ArrayList	Trial 1	0,2013	0,4289	0,8386
	Trial 2	0,1983	0,4975	0,6968
	Trial 3	0,1802	0,4789	0,8131
	Media	**0,193267**	**0,468433**	**0,782833**
TreeSet	Trial 1	1,2175	4,302	8,0951
	Trial 2	1,218	4,3667	8,2228
	Trial 3	1,1834	4,3809	7,9713
	Media	**1,2063**	**4,349867**	**8,0964**
HashSet	Trial 1	0,5034	1,5379	2,6907
	Trial 2	0,561	1,5517	2,4765
	Trial 3	0,6147	1,443	2,47048
	Media	**0,5597**	**1,510867**	**2,545893**

LEITURA COM PARSING

ESTRUTURAS		Vendas_1M.txt	Vendas_3M.txt	Vendas_5M.txt
ArrayList	Trial 1	0,8556	2,2129	3
	Trial 2	0,9663	2,122	3,2616
	Trial 3	0,9538	1,99918	2,288
	Media	**0,925233**	**2,11136**	**2,849867**
TreeSet	Trial 1	2,4687	8,83659	17,724
	Trial 2	2,5497	8,8602	17,4895
	Trial 3	2,49698	8,81898	17,632
	Media	**2,505127**	**8,83859**	**17,61517**
HashSet	Trial 1	1,2268	3,5	5,2827
	Trial 2	1,2086	3,4428	5,4842
	Trial 3	1,3089	3,3106	5,5135
	Media	**1,2481**	**3,4178**	**5,4268**

LEITURA COM PARSING E VALIDAÇÃO

ESTRUTURAS		Vendas_1M.txt	Vendas_3M.txt	Vendas_5M.txt
ArrayList	Trial 1	1,9748	5,7202	9,6133
	Trial 2	2,07857	6,06532	9,8953
	Trial 3	2,054	5,92158	9,68439
	Media	**2,03579**	**5,902367**	**9,730997**
TreeSet	Trial 1	3,8738	15,005	29,783
	Trial 2	3,8749	15,396	29,2247
	Trial 3	3,8886	14,853	29,755
	Media	**3,8791**	**15,08467**	**29,58757**
HashSet	Trial 1	2,5849	7,8307	12,2038
	Trial 2	2,6037	7,41078	13,1966
	Trial 3	2,6048	7,3602	12,3668
	Media	**2,5978**	**7,533893**	**12,58907**

QUERIES – EXECUÇÃO EM HashMap

Ficheiro		Query5	Query6	Query7	Query8	Query9
Vendas_1M	Trial 1	0,005	1,2602	0,2915	17,7705	0,08897
	Trial 2	0,00037	1,189	0,156	16,1551	0,0586
	Trial 3	0,00119	1,227	0,15489	16,489	0,06621
	Media	**0,002187**	**1,2254**	**0,200797**	**16,80487**	**0,07126**
Vendas_3M	Trial 1	0,0152	1,49341	0,38029	14,0199	0,119
	Trial 2	0,00149	1,3486	0,2926	12,46102	0,56082
	Trial 3	0,003613	1,3559	0,2834	12,22	0,0678
	Media	**0,006768**	**1,399303**	**0,318763**	**12,90031**	**0,249207**
Vendas_5M	Trial 1	0,00586	1,8143	0,4911	13,014	0,1524
	Trial 2	0,00216	2,3123	0,4178	13,054	0,1103
	Trial 3	0,00779	1,78086	0,4249	12,818	0,0923
	Media	**0,00527**	**1,969153**	**0,4446**	**12,962**	**0,118333**