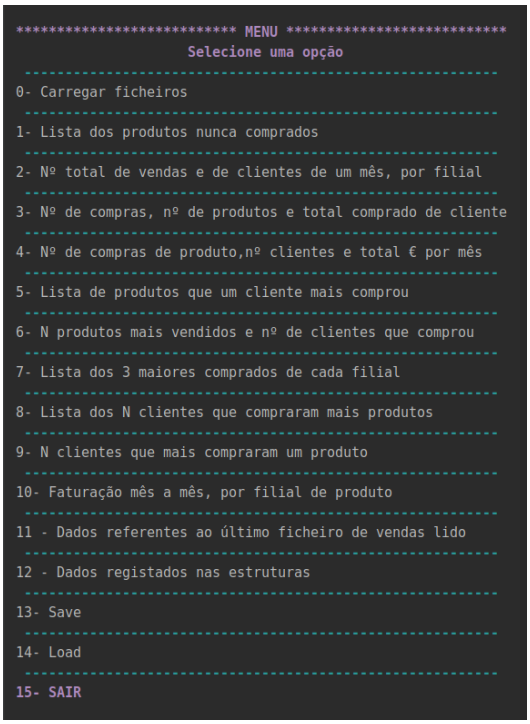
Relatório do Projeto de C

Desenvolvimento de uma Aplicação Desktop usando o modelo MVC para a Gestão das Vendas de uma Cadeia de Distribuição



Grupo 26:

Ema Dias (A89518)

Raquel Costa (A89464)

1 ÍNDICE

2 INTRODUÇÃO

No âmbito da disciplina LI3, após desenvolvermos o projeto de C, realizamos o de JAVA, que consiste numa aplicação de Desktop, usando o modelo MVC para a gestão de vendas de uma cadeia de distribuição.

Para tal, pretendia-se que a mesma fosse capaz de ler e armazenar em estruturas de dados, neste caso, coleções de JAVA , os ficheiros fornecidos, para uma posterior consulta através de queries e para serem submetidos a alguns testes de performance de modo a conseguirmos avaliar e concluir sobre as nossas opções ao longo do projeto.

Essencialmente, temos uma classe "Main" que é a agregadora de todo o projeto e uma classe "GestVendas" responsável por englobar todas as estruturas utilizadas no Model tais como : Filial, Faturaçao, Catálogos,... que nos permitem guardar e organizar todas as informações disponibilizadas seja por um ficheiro .txt ou .dat.

De modo a tornar o programa mais genérico e mais flexível tivemos em conta conceitos tais como, os princípios da programação com interfaces, o MVC e ainda o encapsulamento. Deste modo, conseguimos com que o acesso aos dados fosse feito de uma forma controlada e organizada.

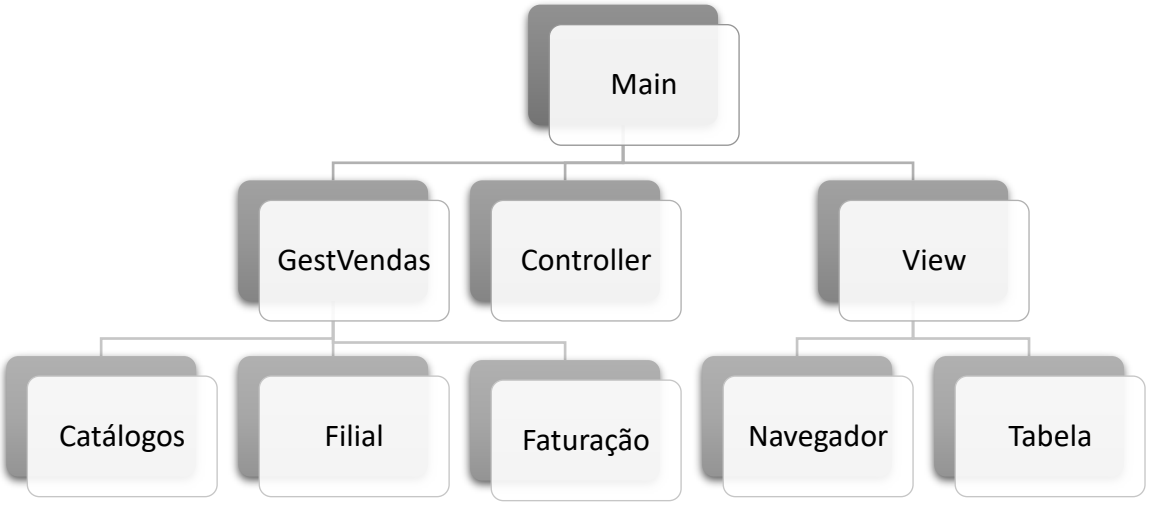


Figura 1- Esquema geral do projeto

3 ARQUITETURA DO PROJETO

De modo a organizarmos o nosso projeto dividimos as classes e as suas interfaces em 4 packages (Model, Controller, View, TestesPerformance). Este último package será abordado mais à frente.

3.1 MODEL-VIEW-CONTROLLER

Os 3 packages referentes ao modelo MVC e as classes de cada um deles:

- Model: Contém todas as classes e interfaces referentes ao model tal como podemos ver no esquema da figura 2.
- Controller: Tem a classe "Controller" e a sua respetiva interface.
- View: Contém a classe "View", "Navegador" e "Tabela", onde estas últimas duas dependem ambas da primeira. (figura 3).

Figura 2- Classes do package Model e as suas dependências

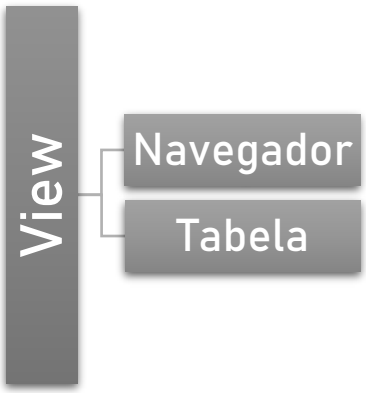


Figura 3-Classes do package View e as suas dependências

3.2 INTERFACES

Ao programar usando interfaces, tornamos o nosso programa muito mais genérico, versátil e moldável de acordo com as necessidades do utilizador, isto é, se quisermos trocar as estruturas que estão no model podemos fazê-lo sem ter de mudar seja o que for nas classes referentes à View e ao Controller.

Além disso, ao programarmos os nossos métodos usando interfaces e sem definir os tipos dos parâmetros de entrada ou dos resultados poucas são as alterações que temos de fazer se quisermos alterar qualquer coisa nas estruturas. Isto, porque, a substituição de uma coleção por outra do mesmo tipo preserva a linguagem uma vez que por terem a mesma API os métodos são os mesmos.

4 CLASSES E INTERFACES

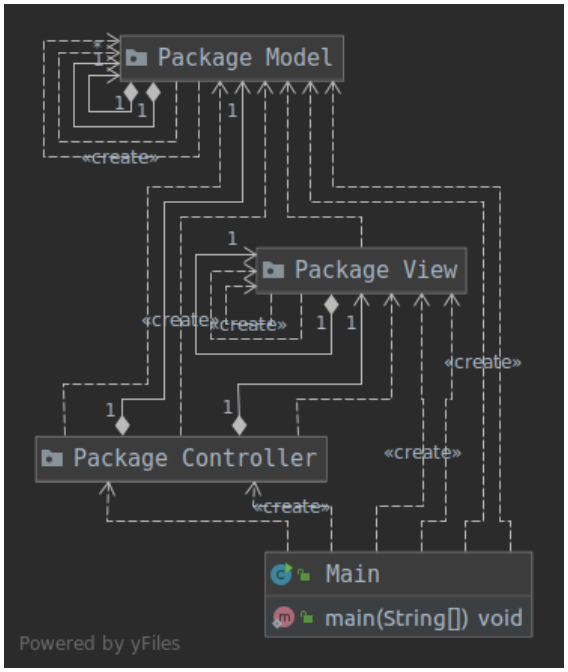


Figura 3- Diagrama de Classes

Nesta figura, apresentamos o diagrama entre as várias classes dentro de cada package. Tal como referido anteriormente, o nosso trabalho encontra-se divido em 3 packages (excluindo o 4º que se destina aos testes de desempenho), onde nos baseamos nos princípios da estrutura MVC. Sendo assim, a Main inicializa o Controller, a View e o Model. Desta forma, permite-nos mudar/trocar, por exemplo o Model, tal como nós fizemos com a experiência do package TestesPerformance.

- **Interação classes MVC + Main**

A main, portanto, está encarregada de englobar todo o projeto.

Tal como é observável no diagrama, o Controller responsabiliza-se por "recolher o pedido" do utilizador onde as várias opções que este poderia tomar foram anteriormente apresentadas pela View, e após isto o Controller responsabiliza-se por consultar ou atualizar os dados nas classes pertencentes ao Model, e de seguida este envia os dados atualizados de novo para o Controller onde esta atualiza de View de modo que os dados obtidos possam ser apresentados ao utilizador.

4.1 READFILE

O objetivo desta classe é efetuar a leitura dos ficheiros escolhidos pelo utilizador, sejam eles os "default" (Produtos.txt, Clientes.txt, Vendas_1M.txt) ou outros que o mesmo deseje.

Ao percorrer o ficheiro dos produtos e dos clientes as strings contidas nos mesmos são adicionadas à estrutura "CatalogoProdutos" e "CatalogoClientes", respetivamente. Como as mesmas são HashSets, não serão inseridos produtos/clientes repetidos.

Ao percorrer o ficheiro Vendas_1M.txt avalia-se antes de alocar nas estruturas, se cada venda em análise é válida, e só depois é que se aloca esta na Filial f1,f2 e f3 e Faturação.

Resumindo, através de um ciclo, lê-se todas as linhas de cada um dos ficheiros fornecidos, de maneira a que as estruturas sejam preenchidas.

O método de *readFile* é invocado aquando da realização da query 0. Ou seja, para as posteriores queries serem realizadas com base nas estruturas é necessário as mesmas estejam preenchidas, o que implica que haja inicialmente esta leitura.

4.2 RWESTADO

Esta classe é responsável por guardar o estado do programa num ficheiro.dat e por ler um ficheiro desse mesmo tipo e preencher posteriormente as estruturas principais do programa usando o objeto da classe "GestaoVenda", para tal fizemos uso de "ObjectStreams".

Além disso, quando o utilizador prime a opção de salvar o estado do programa ele tanto pode dar um nome ao ficheiro ou não, caso não dê, gera-se automaticamente um ficheiro com o nome "GestVendas.dat".

4.2.1 Limitações / Pontos Positivos

Apesar da opção de fazer "save" ou "load" de um estado, de modo a evitar ter de esperar o tempo todo que se demora a ler um ficheiro de grandes dimensões, concluímos que realmente isto não foi algo que conseguimos atingir, uma vez que os tempos de ler um ficheiro do tipo ".dat" acabaram por ser muito maiores do que aqueles ao ler um ficheiro do tipo ".txt".

4.3 CATÁLOGOS

Como já foi referido, os catálogos implementados nas classes "CatalogoClientes" e "CatalogoProdutos" são hashSets, o que permite uma enorme facilidade no momento de validação das vendas, já que estes têm associado o método contains. Assim, além de não alocarmos Strings de códigos já existentes nos Sets, conseguimos validar as vendas de forma mais eficiente.

É importante também mencionar que são dependentes das classes "Produto" e "Cliente" uma vez quando lemos o código de um produto ou cliente convertemos os mesmos para um objeto de cada uma destas duas classes. Sendo assim, os catálogos são formados por (Hash)Set<IProduto> e (Hash)Set<ICliente>.

4.3.1 Limitações / Pontos Positivos

Através dos testes de performance e desempenho que fizemos pudemos concluir que, provavelmente, em vez de usarmos HashSets deveríamos ter usado TreeSets de modo a melhorar a eficiência do programa. Pois, deste modo, se quisermos procurar um determinado Produto ou Cliente, ao invés de percorrer os Catálogos todos até encontrarmos o que procuramos, só iriamos percorrer no máximo metade do Catálogo, uma vez que a procura de um elemento numa árvore se baseia no conceito de procura binária.

4.4 FILIAL

Para a filial, nós decidimos criar uma classe que se denomina como InfoVenda, onde estão definidas as variáveis: cliente, produto, preço, quantidade, tipo (N ou P), mês e filial, que são as informações que se obtém numa linha de venda. O que permitiu na classe Filial termos um (Hash)Map <String, (Tree)Set<IInfoVenda>>.

Consideramos que um Hashmap seria a escolha ideal, já que na estrutura do trabalho a grande maioria das vezes é nos pedida uma relação cliente-produtos, cliente-quantidade comprada ou, até mesmo, cliente-dinheiro gasto. Assim, a key deste Hashmap é ICliente e o value um TreeSet de InfoVenda's.

Desta forma, é possível associar a um cliente a sua árvore de compras. Para ter conhecimento de algum campo da InfoVenda basta fazer "get" da InfoVenda em questão, e para inserir os campos da mesma, aquando da leitura do ficheiro, basta efetuar um "set" do campo respetivo na sua InfoVenda.

A filial é uma estrutura global, que depois, na classe GestaoVendas é diferenciada por 3, já que o ficheiro contém 3 tipos de filial, a 1, 2 e 3 (*private IFilial f1,f2,f3*). Assim, aquando da leitura do ficheiro, antes de inserir na estrutura, verifica-se o seu campo filial, e só aí se insere na sua respetiva filial, através do método *addVenda*.

4.4.1 Limitações / Pontos Positivos

Organizamos a (Tree)Set de cada key por meses de modo a ser mais fácil encontrar, por exemplo, as compras que um cliente fez num determinado mês. Deste modo evitamos percorrer o Set integralmente quando apenas se necessita de um mês em específico.

Resumindo, decidimos optar por esta estrutura, pois inicialmente estávamos a utilizar um HashMap <String, List<IInfoVenda>> e percebemos que não era eficiente, uma vez que a List não estaria organizada de forma alguma, e, desta forma, não obteríamos proveito de uma procura binária, nem do algoritmo Red- Black tree, cuja complexidade é O(log n).

4.5 FATURAÇÃO

Para organizar a faturação global de cada produto e informações relacionadas, optamos por um método muito similar ao da classe responsável pela Filial.

Primeiramente, optamos por criar uma classe "InfoFatur" que se responsabiliza por guardar as seguintes informações:

- Código do produto;
- Mês da faturação
- Filial da faturação
- Número de clientes diferentes que compraram este produto em N
- Número de clientes diferentes que compraram este produto em P
- Total da faturação em modo P
- Total da faturação em modo N
- Número total de vendas deste produto nesta filial e mês

Na classe da Faturação, usamos um (Hash) Map<IProduto, (Tree) Set<IInfoFatur>>, escolhemos usar um HashMap pelos mesmos motivos que o fizemos na Filial.

Inicialmente, à medida que vamos lendo o ficheiro responsável pelos códigos de todos os produtos, adicionamos cada um deles à classe Faturação e consequentemente ao HashMap, como as TreeSets não permitem valores "null" pois geram "NullPointerException", optamos por criar uma InfoFatur a zeros. Portanto, nós sabemos que um produto não foi comprado quando, por exemplo, o campo do número total de vendas do produto é 0.

4.5.1 Limitações / Pontos Positivos

Quando nos deparamos com uma InfoFatur que tenha o mesmo mês e filal para o mesmo Produto de uma outra InfoFatur, somamos estas de modo a reduzir a altura da Tree.

O método de comparação da TreeSet organiza as InfoFatur's por filial e aquelas que têm a mesma filial são organizadas consoante o seu mês. Desta forma é muito mais eficiente para respondermos às queries, pois conseguimos obter faturações de um produto num determinado mês ou filial de uma forma mais rápida.

4.6 CONTROLLER

Esta classe é aquela que contém a interação do utilizador, ou seja recebe o que o utilizador introduz, através de scanner, seja a opção da query que quer efetuar ou os inputs para a mesma, e, para isso, realizamos um "do while" com todos os casos das queries disponíveis.

Assim, esta classe é responsável por ler os pedidos do utilizador, distribuí-los chamando as queries da GestaoVendas e enviar os resultados para a View os apresentar.

4.7 DADOSFICHEIRO E DADOSESTADO

Nestas duas classes é onde guardamos os resultados das queries estatísticas. Para resolver algumas delas conseguimos reutilizar uma grande parte do código que já tínhamos usado para as queries ditas "normais" e para outras conseguimos as informações necessárias enquanto fazíamos a leitura dos ficheiros.

4.8 GESTAOVENDAS

A classe GestaoVendas é em tudo semelhante ao módulo SGV do projeto C, pois é nesta classe que se situa a inicializa as estruturas do projeto, ou seja o "CatalogoClientes", o "CatalogoProdutos", as 3 "Filial", a "Faturação", o "DadosFicheiro" e, por fim, o "DadosEstado".

Além disso, é nesta classe que se definem as queries, ou seja é esta classe que dá resposta às perguntas escolhidas pelo utilizador, retorna a resposta ao "controller" que por sua vez chama a "view" para apresentar os resultados, respeitando a estrutura MVC.

Estas queries invocam métodos das outras classes para auxiliar o cálculo e reúnem todas as respostas ora num array, ora set, ou map.

Em suma, é esta classe que agrega todas as classes, tendo um acesso global ao projeto e permitindo, deste modo, efetuar as queries de forma estruturada. É ela que nos permite inicializar a parte do Model do projeto, independentemente das escolhas das coleções em cada estrutura, desde que correspondam às variáveis de instância desta mesma classe.

4.9 VIEW

A View é a classe que apresenta os resultados ao utilizador, ou seja, é esta que contém os métodos que recebem como input o resultado à query selecionada pelo utilizador e apresenta através de *System.out.print** as respostas.

Esta classe não calcula nem efetua interações que não sejam de apresentação de resultados, são assim definidos métodos genéricos que exibem resultados. Estes métodos são invocados no Controller quando este já tem acesso à resposta que a classe GestaoVendas lhe retornou, ou então mostra ao utilizador quais as opções que poderá tomar (tal como por exemplo o menu) e após isso, no controller faz-se scan da opção que o utilizador tomou.

É nesta classe que chamamos o Navegador e a Tabela de forma a apresentar os dados de uma forma mais organizada.

4.9.1 Navegador

O navegador toma um papel importante no nosso projeto porque é ele o responsável por apresentar o resultado de um grande conjunto de dados, mostrando apenas ao utilizador um subconjunto desses mesmos de cada vez. O utilizador pode tomar várias funções tais como:

- "P" - próxima página de resultados;
- "A" - página anterior de resultados;
- "T" - número total dos resultados;
- "N"- escolher a página à qual o utilizador quer aceder;
- "M"- voltar ao Menu inicial.

4.9.2 Tabela

A tabela organiza os resultados que envolvam, por exemplo, os meses do ano, e não só, de uma forma mais clara. Além disso, para a query 10, esta classe tem também uma função que é basicamente uma junção de um navegador e de uma tabela, pois cada página deste "Navegador-Tabela" corresponde a uma tabela da faturação feita por um produto do Catálogo dos produtos em cada uma das 3 filiais ao longo dos meses do ano.

5 TESTES DE DESEMPENHO

No package "TestesPerformance" temos as seguintes classes:

- Crono – cronometra o tempo
- TesteQ – avalia o tempo de cada query

As próximas classes são "estruturas de teste" onde decidimos substituir onde usamos HashMap<> por TreeMap<> ou vice-versa, usar HashSet<> onde se usou TreeSet<> e vice-versa, e usar Vector<> onde se usou ArrayList<> :

- CatalogoClientesTeste
- CatalogoProdutosTeste
- FilialTeste
- FaturacaoTeste

5.1 GLOBAL

Especificações	PC 1	PC 2
Processador	Intel Core i5-8265U, 1.60 GHz, 1.80 GHz	Intel Core i7-8750 Hexa-Core, 2.20 GHz c/Turbo até 4.10 GHz, 9 MB cache
Sistema Operativo	Windows 10 64 bits	Windows 10 64 bits
Memória RAM	8 GB	16 GB DDR4
Placa Gráfica	GeForce MX250, NVIDIA compatível	Intel HD Graphics 630 +Nvidia GeForce GTX 1050 Ti 4GB
Armazenamento	SSD 512 GB	SSD 256 GB + HDD 1TB
Uso de Virtual Box?	Sim.	Sim.
VB – Base Memory	4096 MB	6251 MB
VB – Processor(s)	1 CPU	2 CPU

Figura 4- Especificações dos PC's usados

	Ficheiros\Tempos	PC 1	PC 2
Estruturas	Vendas_1M	9.3793 s	8.8764 s
	Vendas_3M	67.0068 s	56.2921 s
	Vendas_5M	*	152.5457 s
Estruturas Teste	Vendas_1M	10.4874 s	11.9189 s
	Vendas_3M	*	61.2848 s
	Vendas_5M	*	167.1414 s

Figura 5- Tempos de leitura e preenchimento das estrutruras nos 3 ficheiros

* java.lang.OutOfMemoryError

5.2 POR QUERY

5.2.1 Vendas_1M

- Tempo de cada query com as estruturas e estruturas de teste:

```
Estatística-> Leitura de Ficheiros:  Demorou: 0.616673816 s
Estatística-> Load de Estados:  Demorou: 5.624576102 s

QUERY 1: Demorou: 0.097641017 s
QUERY 2: Demorou: 0.156784163 s
QUERY 3: Demorou: 3.56247E-4 s
QUERY 4: Demorou: 1.505021892 s
QUERY 5: Demorou: 0.009384768 s
QUERY 6: Demorou: 2.518138423 s
QUERY 7: Demorou: 0.554435526 s
QUERY 8: Demorou: 0.479277336 s
QUERY 9: Demorou: 0.170563962 s
QUERY 10: Demorou: 1.291427401 s
```

Figura 6- Queries PC 1, Estruturas

```
Estatística-> Leitura de Ficheiros:  Demorou: 0.322930446 s
Estatística-> Load de Estados:  Demorou: 6.122691855 s

QUERY 1: Demorou: 0.085240745 s
QUERY 2: Demorou: 0.063111574 s
QUERY 3: Demorou: 2.52804E-4 s
QUERY 4: Demorou: 0.796026643 s
QUERY 5: Demorou: 0.00466984 s
QUERY 6: Demorou: 1.259021149 s
QUERY 7: Demorou: 0.25882808 s
QUERY 8: Demorou: 0.161171326 s
QUERY 9: Demorou: 0.113392645 s
QUERY 10: Demorou: 0.803793066 s
```

Figura 7- Queries PC 2, Estruturas

```
Estatística-> Leitura de Ficheiros:  Demorou: 2.245810395 s
Estatística-> Load de Estados:  Demorou: 6.498072091 s

QUERY 1: Demorou: 0.619600199 s
QUERY 2: Demorou: 0.132419169 s
QUERY 3: Demorou: 0.004337142 s
QUERY 4: Demorou: 1.867135011 s
QUERY 5: Demorou: 9.77397E-4 s
QUERY 6: Demorou: 6.073137762 s
QUERY 7: Demorou: 0.349516877 s
QUERY 8: Demorou: 0.48106833 s
QUERY 9: Demorou: 0.099534455 s
QUERY 10: Demorou: 2.506569872 s
```

Figura 8- Queries PC 1 ,Estruturas Teste

```
Estatística-> Leitura de Ficheiros:  Demorou: 0.965758732 s
Estatística-> Load de Estados:  Demorou: 5.803930595 s

QUERY 1: Demorou: 0.289287871 s
QUERY 2: Demorou: 0.035766539 s
QUERY 3: Demorou: 4.11591E-4 s
QUERY 4: Demorou: 0.361700102 s
QUERY 5: Demorou: 0.003623983 s
QUERY 6: Demorou: 2.085643766 s
QUERY 7: Demorou: 0.260045426 s
QUERY 8: Demorou: 0.25411279 s
QUERY 9: Demorou: 0.063022325 s
QUERY 10: Demorou: 1.716785526 s
```

Figura 9- Queries PC 2, Estruturas Teste

5.2.2 Vendas_3M e 5M

PC 1		
	Ficheiros	
Queries	Vendas_3M	Vendas_5M
1	4.6098 s	*
2	0.6321 s	*
3	0.0011 s	*
4	6.4613 s	*
5	0.0266 s	*
6	29.1898 s	*
7	2.4912 s	*
8	14.4751 s	*
9	1.1433 s	*
10	*	*
Estatística Leitura .txt	*	*
Estatística Leitura .dat	*	*

PC 2		
	Ficheiros	
Queries	Vendas_3M	Vendas_5M
1	0.8484 s	0.8376 s
2	0.3945 s	1.0690 s
3	0.0017 s	0.0014 s
4	4.6658 s	9.4240 s
5	0.0205 s	0.0253 s
6	7.7508 s	13.8431 s
7	1.6624 s	2.9243 s
8	0.8856 s	2.3806 s
9	0.5253 s	1.7130 s
10	2.0069 s	*
Estatística Leitura .txt	2.0537 s	*
Estatística Leitura .dat	26.6250 s	*

Figura 5- Tabelas com os tempos de cada query para cada PC

* java.lang.OutOfMemoryError

5.3 Reflexão sobre os Resultados

- Leitura dos ficheiros:

Primeiramente, é importante referir que o uso de Virtual Box tem uma grande influência na performance do nosso programa, tal como foi possível ver o PC 1 não conseguiu sequer ler o ficheiro de Vendas_5M.txt (figura 5), outros dados que também desempenham um papel importante são as especificações tanto do PC como da VB (figura 4).

Com os testes de performance conseguimos constatar que as estruturas pelas quais nós optamos mostraram ser as mais eficientes, tal como é possível observar através dos tempos de execução durante a leitura e preenchimento das estruturas.

- Tempos das Queries:

Mais uma vez, é possível verificar que as estruturas de teste não trazem qualquer beneficio em tempo de execução, o que comprova que, muito

provavelmente, a nossa escolha foi a mais acertada. Podemos tomar como exemplo, os resultados da query 6 no PC 1, onde o tempo de execução aumenta basicamente o triplo de uma estrutura para a outra. Em relação, às queries de estatística um dos pontos negativos que podemos apontar é que muito provavelmente poderíamos ter melhorado os seus tempos, principalmente naquelas que estão relacionadas com a leitura do estado.

- Memory Usage

Após os testes de performance, vimos que ambos os PC's obtiveram uma mensagem de erro relacionada com a memória da heap. Portanto, decidimos recorrer à ferramenta "VisualVM", de modo a poder visualizar o consumo de memória ao longo da execução do programa.

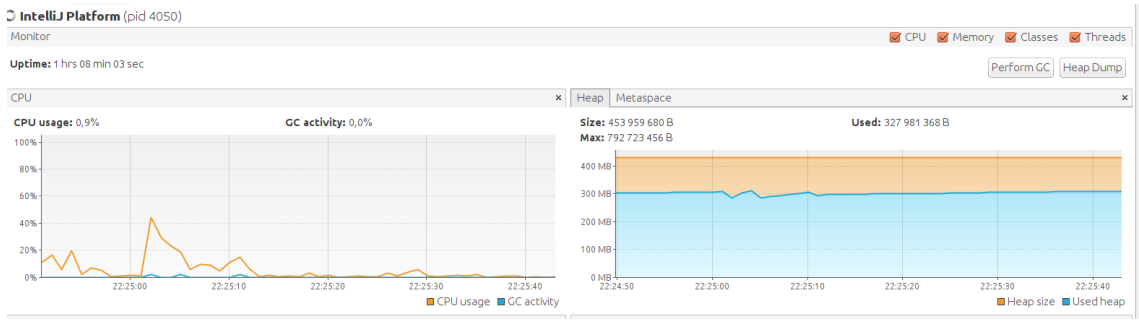


Figura 6- Used Heap com ficheiro Vendas_1M (PC2)

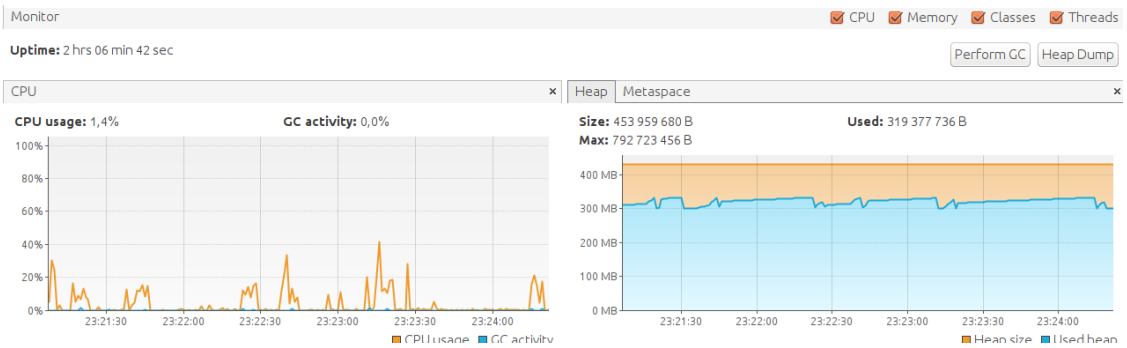


Figura 7- Used Heap com ficheiro de Vendas_5M (PC2)

Tal como é possível observar na figura 8, o tamanho da heap no PC 1 é apenas de 50 MB (ao contrário do PC 2 que é de ~453MB) por isso, com base nestes dados não é de estranhar a extrema diferença dos resultados nas queries, e a incapacidade de ler o ficheiro Vendas_5M.

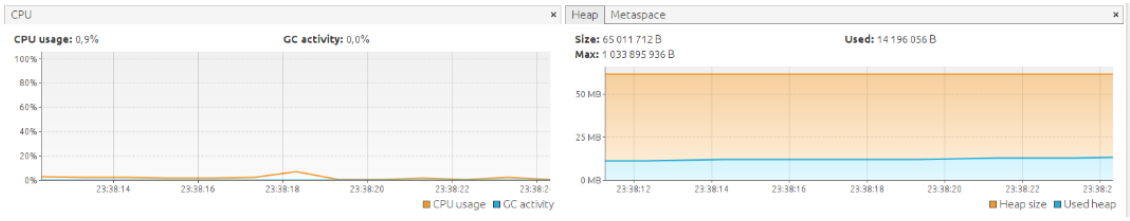


Figura 8- Heap Size (PC1)

6 CONCLUSÃO

6.1 REUTILIZAÇÃO DO CÓDIGO

Ao longo do trabalho, um dos nossos principais objetivos foi tentar reutilizar o código ao máximo, de modo a termos, em geral, um programa genérico, flexível e mesmo assim eficiente. Como exemplo, podemos observar as classes Navegador e Tabela onde tentamos manter o mais geral possível. Temos a noção de que provavelmente poderíamos ter reutilizado mais o código principalmente em relação às queries, mas por vezes, revelavam-se ser desafios bastante difíceis.

6.2 HASH VS TREE

No final do projeto, constatámos que efetivamente o uso de TreeSet ao invés de HashSet teria vindo a beneficiar os tempo de execução no que toca aos catálogos que desenvolvemos, já que os TreeSet proporcionariam uma procura binária.

Relativamente à ordem, os HashSets não estão ordenados e os TreeSet têm a possibilidade de o serem, com um Comparator ou Comparable, tendo também implementados métodos tais como, first(), last(), etc, o que também teria sido vantajoso.

No entanto, é de realçar que operações de inserção ou de remoção são mais rápidas a executar em HashSets, uma vez que a complexidade destas é O(1), enquanto no TreeSet é O(log n).

Já relativamente a HashMap em comparação com TreeMap, o primeiro não garante qualquer tipo de ordem nos elementos, ou seja não podemos prever qualquer ordem quando iteramos keys/values, já o TreeMap contém os elementos pela sua ordem natural.

Relativamente à rapidez, concluímos que os HashMap têm um tempo constante na sua execução, O(1), o que nos permite beneficiar bastante em operações como add( ), remove( ) e contains ( ). No entanto, sabemos que existem algumas desvantagens, tais como Memory Overhead, e uma eficiência mais baixa quando existem colisões.

6.3 ARRAYLIST VS VECTOR

Uma das grandes diferenças entre o vector e o arraylist é que este ultimo não é sincronizado, ou seja vários threads podem aceder e executar simultaneamente, ao contrário do vector onde apenas um thread pode executar. No que toca ao aumento do tamanho, o ArrayList só aumenta metade do seu tamanho, ao invés do vector que aumenta para o dobro, quando é necessário haver resize.

No fundo, o uso do ArrayList é mais eficiente, pois ,uma vez, que é não é sincronizado, opera de forma mais rápida. Além disso, a qualquer momento é possível convertê-lo a sincronizado usando java collections framework utility class.

6.4 PROGRAMAÇÃO COM INTERFACES

Após termos realizado os testes de performance apercebemos-mos da importância da programação dos métodos com interfaces em vez de classes concretas para definir os tipos dos parâmetros de entrada e dos resultados, uma vez que permitem com que o nosso programa seja moldável. E, sendo assim, a substituição de uma coleção por outra do mesmo tipo preserva o código usado, uma vez que, por usarem a mesma API, os métodos são os mesmos.

6.5 REFLEXÃO FINAL

Em suma, conseguimos responder a todas as queries pedidas trabalhando com as estruturas que nos pareceram mais eficientes, como trabalho futuro, voltaremos a tentar reutilizar o código ao máximo, de modo a tornar o nosso programa o mais genérico e moldável possível. Temos a apontar também que algo que gostaríamos de ter melhorado é o consumo de espaço na heap e um menor tempo de execução nas queries de estatística, já que algumas lidam com grandes conjuntos de dados.