



Escola de Engenharia
Departamento de Informática
Universidade do Minho

Gesthiper

Laboratórios de Informática III

Abril 2015

Grupo nº 50:

André Santos (a61778)



Bruno Pereira (a72628)



Índice

1.	Arquitectura da aplicação e Estruturação Main.....	2
2.	Módulos de Dados	4
2.1	Catálogo Produtos.....	4
2.2	Catálogo Clientes.....	5
2.3	Contabilidade	6
2.4	Módulo Compras.....	7
2.5	Classes Matriz_Int_12x2 / Matriz_Double_12x2.....	9
2.6	Classes Hipermercado e Compra	10
3.	Interface com Utilizador.....	11
4.	Navegação nos resultados	15
5.	Performance Leitura e Queries	17
5.1	Performance leitura ficheiros.....	17
5.2	Performance Queries	18
6.	Medidas Performance.....	20
6.1	leitura	20
6.2	Estruturas	21
7.	Conclusão	24
8.	Bibliografia	25

1. ARQUITECTURA DA APLICAÇÃO E ESTRUTURAÇÃO MAIN

Esta secção tem por objectivo apresentar a arquitectura geral da aplicação, dando especial destaque à forma como o main está organizado. Será explicada a ligação que o main tem com os diferentes módulos de dados e como o utilizador interage com esses módulos. Detalhes da implementação e considerações sobre cada um desses módulos serão deixados para secções posteriores.

O pilar de funcionamento da aplicação são 4 módulos de dados, a saber: catálogo de produtos, catálogo de clientes, contabilidade e compras. Cada um destes módulos tem a sua respectiva classe e corresponde portanto a um tipo de dados que pode ser instanciado. Por sugestão do enunciado do projecto, foi criada uma classe Hipermercado que agrega, por composição, estes 4 módulos. Assim, todas as classes que necessitam de aceder aos módulos, fazem-no sempre a partir da classe Hipermercado. Por sua vez, o acesso à classe Hipermercado faz-se a partir do main, que no caso do nosso programa se encontra na classe Gesthiper. O main declara por isso uma variável *static* do tipo Hipermercado bem como um método get e set que permitem o acesso a essa variável.

```
public static Hipermercado hipermercado;

(...)

public static Hipermercado getHipermercado() {
    return hipermercado;
}

public static void setHipermercado(Hipermercado novo_hiper) {
    hipermercado = novo_hiper;
}
```

A função main propriamente dita começa por isso por inicializar esta variável. Depois chama a função *MenuLeitura.menuLeitura()*, que permite ao utilizador escolher os ficheiros que quer ler. Por último, chama a função *MenuQueries.menuPrincipal()*, que apresenta ao utilizador as queries disponíveis e as realiza quando seleccionadas. Quando o utilizador decide sair do menu das queries, o programa termina. Tem-se por isso a seguinte função main:

```
public static void main(String[] args) {
    hipermercado = new Hipermercado();
    MenuLeitura.menuLeitura();
    MenuQueries.menuPrincipal();
}
```

De uma forma geral a arquitectura da aplicação pode ser representada pelo seguinte esquema:

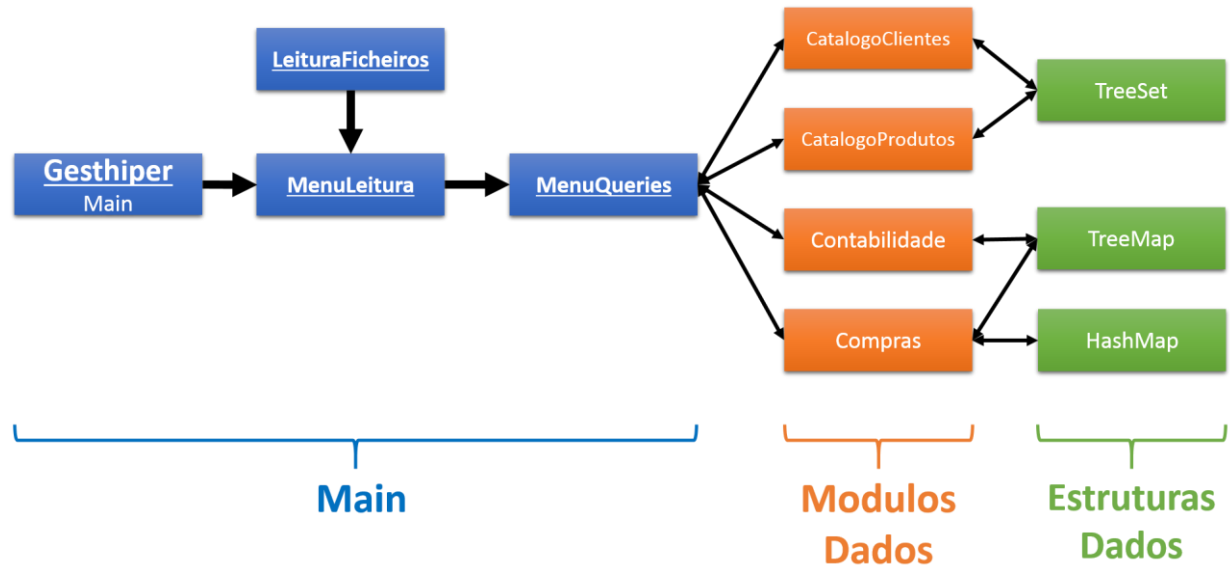


Figura 1 - Arquitectura geral da aplicação

Na figura 1 cada rectângulo corresponde ao nome de uma classe do programa. Visto o esquema pretender representar apenas a arquitectura geral, omitem-se algumas classes secundárias.

Como se pode ver o “main” na verdade é constituído por 4 ficheiros/classes:

- **Gesthiper** – Onde se localiza a função main, apresentada anteriormente.
- **LeituraFicheiros** - Funções que tratam da leitura de ficheiros e de povoar as estruturas de dados. Estas funções representam a parte computacional da leitura de ficheiros, não tendo associado nenhum I/O ou interacção com o utilizador.
- **MenuLeitura** – Parte da interface onde o utilizador escolhe de que forma, e que ficheiros pretende ler para inicializar o programa. Relativamente à leitura, esta parte trata apenas da interface e tratamento do input do utilizador, chamando as funções de *LeituraFicheiros* para ler os ficheiros e povoar as estruturas de dados.
- **MenuQueries** – Parte da interface responsável por mostrar ao utilizador as queries disponíveis e as executar quando seleccionadas, fazendo pedidos aos módulos.

A decisão de dividir o main nestes 4 ficheiros teve dois objectivos:

- Primeiro, evitar ter um número de ficheiros com elevado número de linhas de código com as consequentes dificuldades de navegação e procura de partes específicas de código que isso traria.
- Segundo, dividir o código de forma lógica, de acordo com a funcionalidade, o que por sua vez também facilita a navegação pelo código.

A divisão da parte computacional da parte de I/O na leitura dos ficheiros veio-se a revelar bastante útil numa fase posterior do projecto, em que para mudar o ficheiro de compras a meio da execução do programa foi necessário chamar uma das funções de *LeituraFicheiros* sem ter associada a interface do *MenuLeitura*.

Relativamente aos módulos de dados, a decisão de ter os 4 apresentados teve a ver directamente com os objectivos do projecto, em que especificamente foram pedidos estes 4 módulos.

2. MÓDULOS DE DADOS

2.1 CATÁLOGO PRODUTOS

Para o catálogo de produtos optou-se por usar um `ArrayList` com 27 posições, em que cada posição guarda um apontador para um `TreeSet<Produto>`. Cada um destes `TreeSet` contém os códigos dos produtos começados por uma determinada letra. Para o índice 0 está o `TreeSet` correspondente à árvore com os códigos começados A, para o índice 1, a letra B..etc. Tem-se assim 26 posições no array, uma para cada letra. Além disso, num espírito de programação defensiva, foi reservado também o último índice para casos especiais, em que o código não começa por uma letra. Cada `TreeSet` é um `TreeSet<Produto>`, inicializado com um `Comparator` que ordena os Produtos de acordo com o seu código.

A decisão de se usar `TreeSets` para guardar os produtos teve a ver com a natureza da própria classe `Produto`, em que cada `Produto` é apenas constituído por um código, ficando assim fora de questão o uso de classes `Map<K,V>`. Uma vez que em nenhuma querie é necessário acessos a este módulo, a decisão de usar um `TreeSet` em detrimento de um `HashSet` ou `LinkedHashSet` foi mais ou menos arbitrária.

O motivo para se ter um `ArrayList` de `TreeSets` e não apenas um único `TreeSet` visa facilitar procuras no catálogo. Embora o programa deste projecto não necessite de consultar os módulos dos catálogos, assume-se que o objectivo principal deste módulo, além de conter uma lista de todos os produtos existentes, seja facilitar a obtenção dos produtos por letra, em que é útil ver o catálogo como vários conjuntos em vez de um único.

```
private ArrayList<TreeSet<Produto>> catalogo;
```

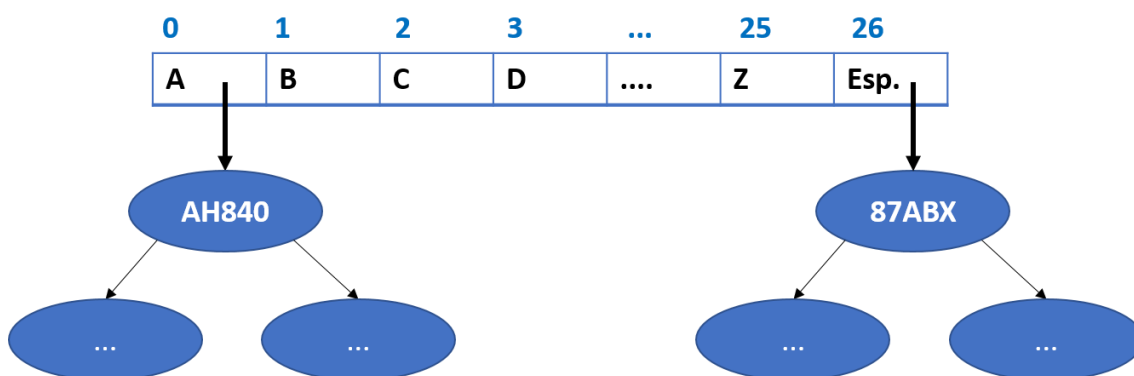


Figura 2- Estrutura do catálogo de produtos

2.2 CATÁLOGO CLIENTES

A estrutura do catálogo de clientes é em tudo semelhante ao catálogo de produtos apresentado anteriormente. Todas as considerações feitas para o catálogo de produtos são válidas para o catálogo de clientes. Recomenda-se por isso a leitura da secção anterior. Deixa-se apenas para referência o esquema da estrutura:

```
private ArrayList<TreeSet<Cliente>> catalogo;
```

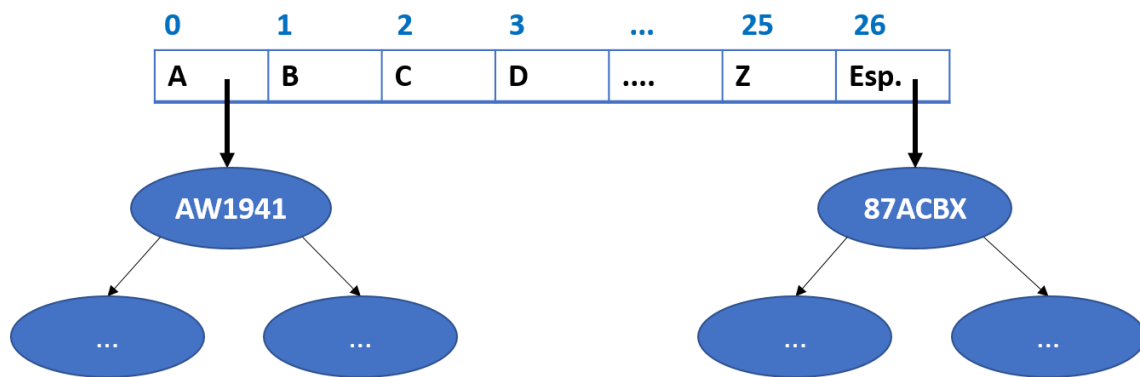


Figura 3- Estrutura do catálogo de clientes

2.3 CONTABILIDADE

O módulo da contabilidade é constituído fundamentalmente por um TreeMap, em que o Produto é a chave e tem associado a si uma ficha de produto (FichaProdutoContabilidade). Nessa ficha de produto é guardada informação relativa a vendas, número de compras e facturação de cada Produto, para cada mês e tipo de compra. De forma a evitar travessias pelo TreeMap, para procurar informação, existe também informação global sobre o nº de compras, vendas e facturação de todos os produtos, também por mês e tipo de compra.

Assim, a classe **Contabilidade** é constituída pelas seguintes variáveis de instância:

```
private Matriz_Int_12x2 totalComprasPorMes;  
private Matriz_Int_12x2 totalUnidadesVendidasPorMes;  
private Matriz_Double_12x2 totalFacturadoPorMes;  
TreeMap<Produto, FichaProdutoContabilidade> arvoreProdutos;
```

Em que a **FichaProdutoContabilidade** consiste em:

```
private Matriz_Int_12x2 numUnidadesProdutoVendidasPorMes;  
private Matriz_Int_12x2 numComprasProdutoPorMes;  
private Matriz_Double_12x2 facturacaoProdutoPorMes;
```

A decisão de usar um TreeMap teve a ver em primeiro lugar, com a necessidade de se poder procurar rapidamente uma ficha de produto, dado um produto. Isto motivou a escolha de um Map. Em segundo lugar, tendo em vista a forma como os resultados deveriam ser apresentados, era necessário que o Map estivesse ordenado por produto, o que motivou a que o Map escolhido fosse um TreeMap. Um HashMap permitiria inserções mais rápidas, e tempo de pesquisa quase igual (ver secção de testes de performance), no entanto os resultados de uma travessia a um HashMap teriam que ser posteriormente ordenados, o que deixa de justificar a vantagem na inserção e ligeira maior rapidez de acesso.

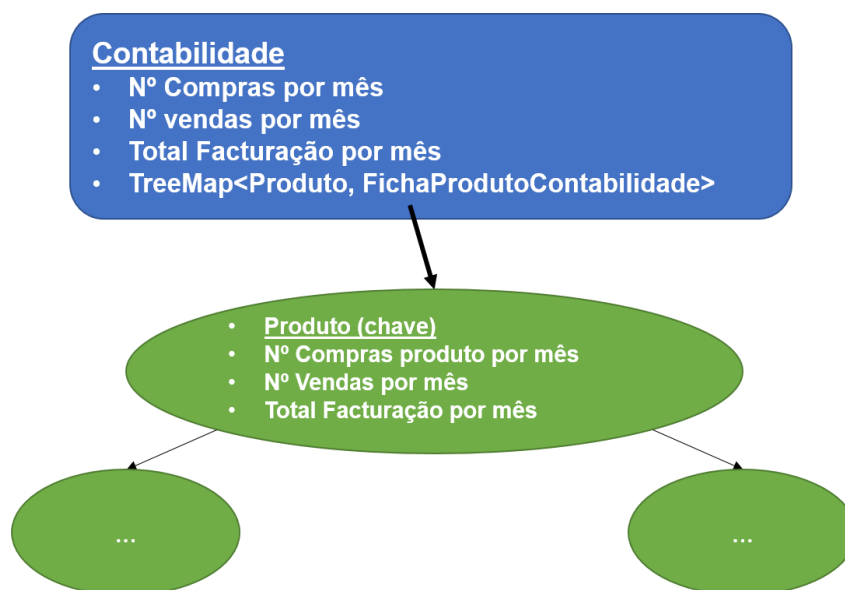


Figura 4 - Estrutura do módulo contabilidade

2.4 MÓDULO COMPRAS

O principal desafio no planeamento do módulo das compras foi a forma como iriam ser associados produtos a clientes e vice-versa. Optou-se por resolver essa questão tendo uma estrutura em que a chave é o cliente, e o valor da chave contém informação relativas aos produtos que esse cliente comprou:

```
private TreeMap<Cliente, FichaClienteCompras> arvoreClientes;
```

Onde a **FichaClienteCompras**, que guarda a informação sobre quantidades e nº de produtos comprados pelo cliente, além do dinheiro total gasto pelo cliente:

```
private Matriz_Int_12x2 numUnidadesCompradasClientePorMes;  
private Matriz_Int_12x2 numComprasClientePorMes;  
private Matriz_Double_12x2 dinheiroGastoClientePorMes;  
private TreeMap<Produto, FichaProdutoDeClienteCompras> produtosCliente;
```

Nesta estrutura, além da informação global sobre compras, vendas e unidades compradas do cliente, é possível verificar que a informação dos produtos comprados é também guardada num *TreeMap* *produtosCliente*.

Os valores deste map contém informação relativa às vendas do produto, para o cliente a que está associado. Assim, a **FichaProdutoDeClienteCompras** é constituída por:

```
private Matriz_Int_12x2 numUnidadesCompradasProdutoClientePorMes;  
private Matriz_Int_12x2 numComprasProdutoClientePorMes;  
private double totalGastoClienteProduto;
```

A decisão de se usar *TreeMap* tanto no acesso principal como no acesso secundário, tal como na Contabilidade, deve-se mais uma vez à necessidade de ordenação dos resultados de forma a que possam ser apresentados mais facilmente.

A estrutura apresentada anteriormente corresponde ao “core” do módulo das compras. Embora a decisão de ter acesso principal por Cliente e acesso secundário por Produto tenha sido tomada por ser a que melhor respondia à maioria das queries, a verdade é que não ter um acesso principal por produto piora algumas queries, nomeadamente a query onde é pedido que para cada produto é necessário saber quantos clientes distintos esse produto teve. Tendo em vista a melhoria dessa query, o módulo das compras tem ainda uma estrutura secundária:

```
private HashMap<Produto, ParProdutoNClientes>  
arvoreParesProdutoNClientes;
```

Neste Map, para cada produto é guardada a informação do número de clientes por mês e totais que esse produto teve. Assim, o *ParProdutoNClientes* tem a seguintes variáveis de instância:

```
private int[] numeroClientesDistintosPorMes;  
private int numeroTotalClientesDistintos;
```


Para esta estrutura secundária tomou-se a decisão de usar um HashMap. No programa, esta estrutura apenas vai ser acedida para procuras e inserções de elementos individuais, tarefas que um HashMap realiza de forma rápida. Este Map não precisa de ser iterado e os resultados tirados não precisam de sofrer qualquer tipo de ordenação, daí se ter usado um HashMap em detrimento de um TreeMap por exemplo. Esta escolha revelou-se acertada depois de feitas as medições de tempos que se apresentaram posteriormente.

Além das estruturas apresentadas, o módulo de Compras tem ainda informação global sobre o número de clientes por mês e totais, bem como do número de compras de valor zero.

As variáveis de instância completas do módulo de **Compras** são por isso:

```
private int numeroComprasValorZero;
private int numeroClientesDistintosPorMes[];
private int numeroTotalClientesDistintos;
private TreeMap<Cliente, FichaClienteCompras> arvoreClientes;
private HashMap<Produto, ParProdutoNClientes> arvoreParesProdutoNClientes;
```

O módulo de compras pode ser representado pelo seguinte esquema:

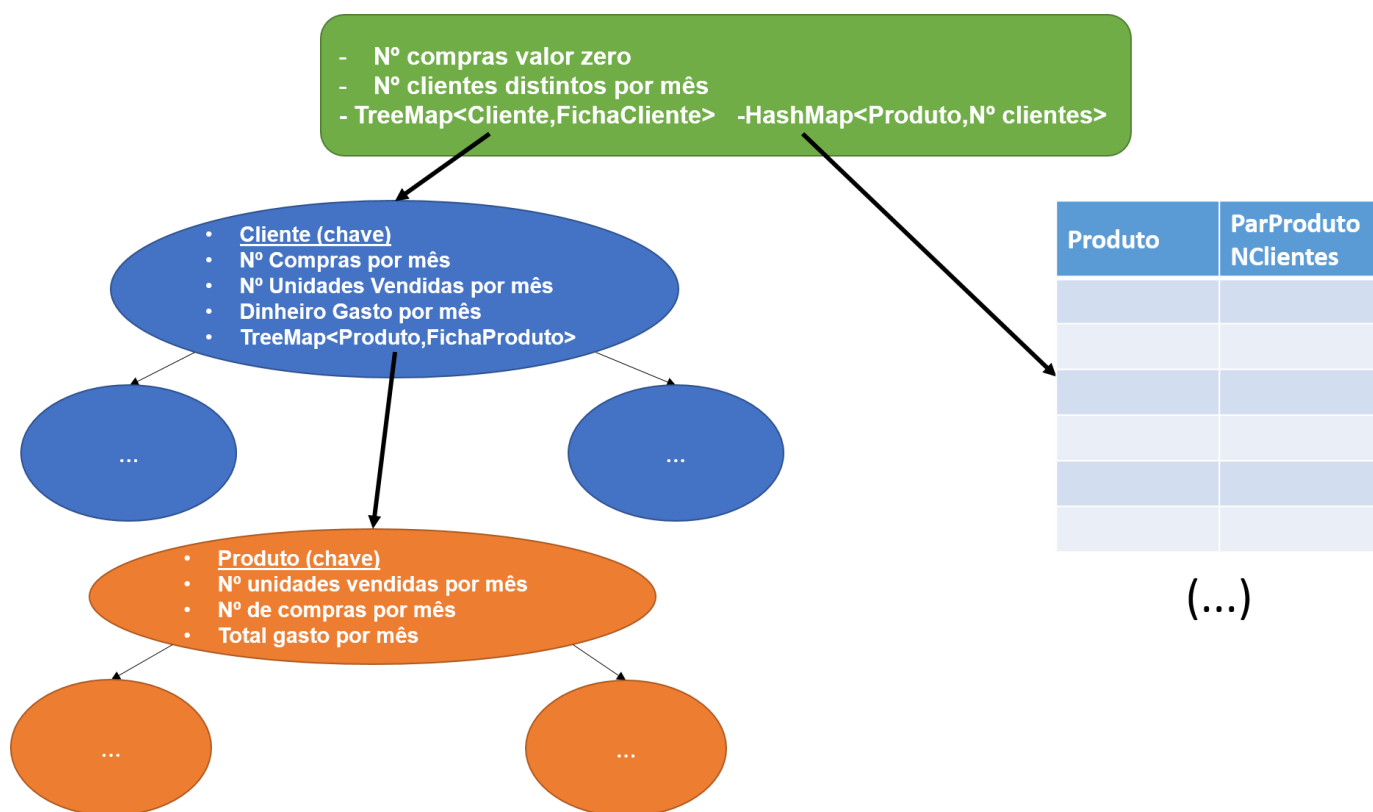


Figura 5 - Estrutura do módulo de compras

2.5 CLASSES MATRIZ_INT_12x2 / MATRIZ_DOUBLE_12x2

Da apresentação anterior feita aos módulos, é possível verificar que as informações de vendas, nº de unidades compradas e facturação são representadas por uma variável do tipo `Matriz_Int_12x2` ou `Matriz_Double_12x2`.

De facto, desde o início do projecto foi perceptível que se iriam efectuar muitas operações repetidas sobre matrizes nos módulos, visto que a maior parte da informação útil do módulo de contabilidade e do módulo das compras se encontra guardada em matrizes. Além disso, todas estas matrizes têm a particularidade de se destinarem a associar valores a um mês e tipo de compra. Ou seja, todas elas têm o mesmo tamanho e o tipo de operações a realizar sobre elas é o mesmo.

Percebendo isso, e de forma a reutilizar e melhorar a legibilidade do código, decidimos criar classes que representassem essas matrizes e nos permitissem trabalhar sobre elas. Estas classes representam então matrizes 12x2, em que a cada linha corresponde um mês e cada coluna corresponde a um tipo de compra. A 1ª coluna representa valores para os tipo de compra NORMAL, e a 2ª valores para o tipo de compra PROMOCAO. Assim, a posição `[2][0]` de uma matriz em que são guardados o nº de compras, representa o nº de compras NORMAIS feitas em Março (3º mês, índice = 2).

As variáveis de instância e classe são por isso:

```
public static int MAX_ROW = 12;
public static int MAX_COL = 2;

private int[][] matriz = new int[MAX_ROW][MAX_COL];
private int total;
```

Chama-se aqui também a atenção para a variável *total*. Esta variável contém a soma total dos valores da matriz. Sempre que um valor na matriz é alterado, esta variável é actualizada para reflectir o novo total. Esta variável trata-se de uma medida de optimização feita para que, sempre que seja necessário calcular a soma dos valores da matriz, não se tenha que somar as 24 posições da matriz, mas sim apenas a consultar o valor de uma variável.

Isto é particularmente relevante em queries em que é necessário iterar várias “Fichas”, seja de produto ou cliente, e se pretenda saber o total de forma rápida. Um exemplo concreto disto é a query que pede uma lista de produtos não comprados. Ora, saber se o produto não foi comprado, equivale a saber se o nº de compras é igual a zero. A função que devolve os produtos não comprados tem por isso que percorrer todos os produtos e para cada um deles ver o total de compras. O programa seria bem menos eficiente se, ao iterar todos os produtos, para cada um deles se tivesse que somar 24 valores para saber se o nº de compras foi zero.

Sendo certo que esta variável torna mais lentas operações de actualização de valores, feitas na leitura dos ficheiros de compras, foi deliberada a decisão de tornar a leitura dos ficheiros ligeiramente mais lenta para que as queries fossem mais eficientes.

2.6 CLASSES HIPERMERCADO E COMPRA

Um dos objectivos deste projecto foi fazer uma aplicação com classes que respeitem os princípios da modularidade e encapsulamento. Isto motivou a ter todas as inserções nas estruturas a inserir um clone da informação passada como argumento e todos os get's a essas estruturas a devolver um clone do objecto da estrutura.

Estes princípios foram respeitados em todos os módulos de dados feitos, em todas as classes em que tinham que ser garantidos. Há no entanto duas classes – Hipermercado e Compra – em que deliberadamente os métodos setters não fazem clone da informação passada como argumento e os getters não devolvem um clone da informação pedida. Visto a modularidade e encapsulamento serem pontos cruciais do projecto, justificaremos esta decisão agora.

A classe Hipermercado é uma classe agregadora de módulos. No caso concreto deste programa apenas temos uma variável do tipo Hipermercado. Visto que várias partes do programa precisam de aceder a esta variável, esta foi posta como “global a vários ficheiros”, sendo declarada com os atributos *public static* no main e fornecidos métodos de acesso. Sendo uma variável de uma classe global agregadora de módulos, várias partes do programa podem e devem alterar os módulos do hipermercado, pelo que não se justifica fazer clones dos módulos. O encapsulamento é garantido ao nível dos módulos, não ao nível da classe hipermercado. Esta classe foi por isso vista por isso como uma classe auxiliar ao main, não como um módulo.

À semelhança da classe Hipermercado, a classe Compra é uma classe cujas variáveis de instância podem e devem ser alteradas livremente na leitura dos ficheiros, sem que haja necessidade de clonar a informação. Serve também como uma classe agregadora auxiliar ao main e portanto também nesta classe os setters e os getters não fazem clones. O encapsulamento é garantido ao nível dos módulos, em que na inserção de uma compra, a sua informação é copiada, não sendo por isso guardados os apontadores da Compra passados como argumento.

3. INTERFACE COM UTILIZADOR

Nesta secção apresenta-se a interface com o utilizador, fazendo algumas considerações sobre as decisões tomadas.

Ao iniciar o programa, o utilizador tem um menu que lhe permite escolher 4 opções para ler os ficheiros:

```
=====
GESTHIPER >> Leitura Ficheiros
=====
1) Ler ficheiro objecto
2) Ler ficheiros genericos
3) Procura automatica ficheiro objecto
4) Procura automatica ficheiro generico
-----
0 - Sair
=====
Escolha uma opção: █
```

Figura 6 - Menu de leitura de ficheiros

- **Ler ficheiro objecto** – Permite ao utilizador indicar o nome de um ficheiro objecto para ser lido.
- **Ler ficheiros genericos** – Permite ao utilizador indicar o nome de 3 ficheiros: de produtos, de clientes e de compras que pretende ler.
- **Procura automática de ficheiro objecto** – O programa procura todos os ficheiros .obj na pasta. Caso apenas encontre um ficheiro, começa-o a ler imediato. Caso encontre vários, apresenta-os ao utilizador, que pode escolher o ficheiro que pretende indicando um número apenas.
- **Procura automática de ficheiro genérico** – O programa tenta encontrar automaticamente os ficheiros de produtos, clientes e compras. Tal como na opção anterior, caso apenas encontre 1 ficheiro começa-o a ler de imediato. Caso encontre vários ficheiros, apresenta uma opção ao utilizador.

De seguida mostra-se um exemplo da execução do programa em que se escolheu a opção 4:

```
=====
GESTHIPER >> Leitura Ficheiros
=====
1) Ler ficheiro objecto
2) Ler ficheiros genericos
3) Procura automatica ficheiro objecto
4) Procura automatica ficheiro generico
-----
0 - Sair
=====
Escolha uma opção: 4
1 ficheiro de clientes encontrado: datasets/FichClientes.txt
1 ficheiro de produtos encontrado: datasets/FichProdutos.txt
4 ficheiros de compras encontrados
Escolha que ficheiro quer ler:
1) datasets/Compras.txt
2) datasets/Compras1.txt
3) datasets/Compras3.txt
4) datasets/miniCompras.txt
Insira nº do ficheiro: █
```

Figura 7 - Escolha de ficheiro de compras

Como se pode ver, o programa encontrou apenas 1 ficheiro de clientes e de produtos e por isso seleccionou-os automaticamente para leitura. Visto ter encontrado 4 ficheiros possíveis para as compras, apresentou uma opção ao utilizador. Depois do utilizador escolher a opção, os ficheiros começam a ser lidos, sendo apresentado os tempos de leitura:

```
=====
GESTHIPER >> Leitura Ficheiros
=====
A ler datasets/FichClientes.txt...
Ficheiro datasets/FichClientes.txt lido com sucesso.
A ler datasets/FichProdutos.txt...
Ficheiro datasets/FichProdutos.txt lido com sucesso.
A ler datasets/miniCompras.txt...
Ficheiro datasets/miniCompras.txt lido com sucesso.
=====
Tempo de leitura Clientes: 0.195058388 segs.
Tempo de leitura Produtos: 5.115044885 segs.
Tempo de leitura Compras: 2.263342242 segs.
Tempo total leitura: 7.5734455149999995 segs.
=====
Pressione qualquer tecla para continuar: 
```

Figura 8 - Ficheiros e tempos de leitura

Assim que o utilizador insira algo e carregue enter, é apresentado o menu das queries, onde o utilizador por escolher que query deseja realizar:

```
=====
GESTHIPER >> Menu Queries
=====
QUERIES:
 1) Estatísticas último ficheiro
 2) Dados gerais
 3) Produtos não comprados
 4) Clientes que não compraram
 5) Nº compras / clientes num mês
 6) Compras cliente num mês
 7) Compras / clientes /facturacao produto
 8) Compras produto N e P
 9) Produtos mais comprados cliente
10) Produtos mais vendidos
11) Clientes com mais produtos diferentes comprados
12) Clientes que mais compraram produto
OPERAÇÕES:
13) Guardar em ficheiro objecto
14) Carregar ficheiro objecto
15) Mudar ficheiro compras
=====
0 - Sair
=====
Escolha o nº opção: 
```

Figura 9 - Menu Queries

A interface das queries é muito semelhante em todas. Em todas, o utilizador tem a opção de voltar ao menu anterior ou sair do programa. Nas queries que o justificam o utilizador pode também sempre voltar a repetir a query sem ter que voltar ao menu principal. A interface da query 6 é um bom exemplo disso.

```
=====
GESTHIPER >> QUERIE 6
Compras de cliente
=====
Código Cliente: CN220
=====
|  | Número | Produtos |  |
| Mes | Compras | Distintos | € Gasto |
=====
| Jan | 0 | 0 | 0.00 |
| Fev | 1 | 1 | 135.24 |
| Mar | 2 | 2 | 48.32 |
| Abr | 1 | 1 | 16.08 |
| Mai | 0 | 0 | 0.00 |
| Jun | 0 | 0 | 0.00 |
| Jul | 1 | 1 | 82.80 |
| Ago | 0 | 0 | 0.00 |
| Set | 0 | 0 | 0.00 |
| Out | 1 | 1 | 382.66 |
| Nov | 0 | 0 | 0.00 |
| Dez | 0 | 0 | 0.00 |
=====
| Tot | 6 | 6 | 665.10 |
=====
Tempo query: 0.0014 segundos.
=====
0 - Sair | 1 - Menu Principal | 2 - Procurar outro cliente
=====
Escolha opção: 
```

Figura 10 - Query 6

Além das opções de sair ou voltar ao menu principal, o utilizador tem a opção de repetir a query, que neste caso envolve a introdução de um novo código de cliente.

Nas queries que envolvem paginação de resultados, além destas opções, o utilizador pode ainda ir para a primeira página (4), recuar uma página (5), avançar uma página (6) ou ir directamente para a última página (7). Além disso o utilizador pode ainda dar o número de uma página para a qual queira ir (2):

```
=====
GESTHIPER >> QUERIE 10
N Produtos mais vendidos
=====
Página 1/19559
-----
| # | Codigo | Unidades | Clientes |
|   | Produto | Vendidas | Distintos |
|---|---|---|---|
| 1 | LB3741 | 96 | 7 |
| 2 | OL7932 | 85 | 6 |
| 3 | QX9126 | 80 | 5 |
| 4 | FH9726 | 77 | 5 |
| 5 | AB4121 | 72 | 5 |
| 6 | YG6379 | 69 | 4 |
| 7 | EN9171 | 68 | 4 |
| 8 | WP2000 | 68 | 4 |
| 9 | OU9509 | 67 | 5 |
|10 | CM7549 | 66 | 4 |
|---|---|---|---|
A mostrar 1-10 de 195584 resultados.
Tempo query: 0.4168 segundos. (Produtos: 0.3435, Clientes: 0.0733)
=====
0 - Sair | 1 - Menu Principal | 3 - Escolher novo N
[<] 4 [<] 5 ### 6 [>] 7 [>>] | 2 - Pag...
=====
Insira nº da opção > 
```

Figura 11 - Query 10, exemplo de query com paginação

Nas queries em que é pedido algum tipo de input ao utilizador, por exemplo, um código de cliente ou de produto, o utilizador tem sempre a oportunidade de “cancelar” a execução da query e voltar atrás ou sair do programa.

```
=====
GESTHIPER >> QUERIE 6
Compras de cliente
=====
q - sair | b - voltar
-----
Indique o cliente que quer procurar: 
```

Figura 12 - Possibilidade de sair do programa ou voltar ao menu das queries

4. NAVEGAÇÃO NOS RESULTADOS

Visto serem várias as queries com elevado número de resultados, um dos desafios deste projecto foi decidir a melhor forma de apresentar esse elevado número de resultados ao utilizador. A forma como o utilizador vê esses resultados e navega pelas páginas foi apresentada na secção anterior. Nesta secção pretende-se explicar a forma como no código, esta forma de paginar os resultados acontece. Visto que a operação de paginação é comum e o código da paginação em todas as queries seria praticamente o mesmo, decidiu-se criar uma classe **Paginador<E>** que auxilia o main na tarefa de paginar os resultados. O paginador pode ser criado sobre uma lista de elementos do tipo E (um qualquer tipo) e divide o array em várias páginas, pelo simples cálculo de índices. Além de dividir o array por várias páginas, a qualquer momento “sabe” mais informação sobre as páginas, nomeadamente, quantos elementos tem cada uma, que posição do array deve ser acedida para se consultar a página X etc. Revela-se assim um auxiliar útil para o tratamento de paginação. Veja-se um exemplo, retirado de uma das queries:

```
List<Cliente> listaClientesSemCompras = moduloCompras.getClientesSemCompras();

Paginador<Cliente> paginador = new Paginador<>(listaClientesSemCompras, 10,
1);
```

Na 1ª instrução, o módulo de compras é consultado para se obter a lista dos clientes sem compras. Dado esta lista poder conter muitos clientes, queremos ver esta lista de resultados paginada. Isso é feito pela 2ª instrução, em que é criado um paginador sobre o array listaClientesSemCompras. Os restantes parâmetros do constructor dizem respeito ao número de elementos por página que se quer ver e em que página se quer começar. Neste exemplo escolhemos mostrar 10 clientes por página, a começar na primeira página. Logo a seguir, o main consulta este paginador para saber qual o total de páginas que tem a lista:

```
total_paginas = paginador.getNumPaginas();
```

As funções que tratam das queries de paginação têm uma variável *numero_pagina* que contém em cada momento a página que o utilizador quer ver. É a função da query responsável por actualizar o valor a esta variável e garantir que nunca toma um valor abaixo de zero, nem acima de *total_paginas*. Tendo esta variável, no ciclo for que controla a apresentação das páginas executa-se a seguinte sequência de instruções.

```
paginador.gotoPagina(numero_pagina);
inicio_pagina = paginador.getPosInicialPagActual();
num_elems_pag_actual = paginador.getNumElemsPagActual();
fim_pagina = inicio_pagina + num_elems_pag_actual;
```

Em primeiro lugar, é pedido ao paginador para “ir para a página *numero_pagina*”. Em termos práticos, o que o paginador faz é actualizar o seu estado interno com os índices que caracterizam aquela página. Depois dessa actualização ao paginador, o utilizador pede a posição da lista que deve consultar para ter o 1º elemento dessa página e guarda esse resultado em *inicio_pagina*. Pede ainda ao paginador qual o número de elementos da página actual e por fim calcula qual o último índice da página.

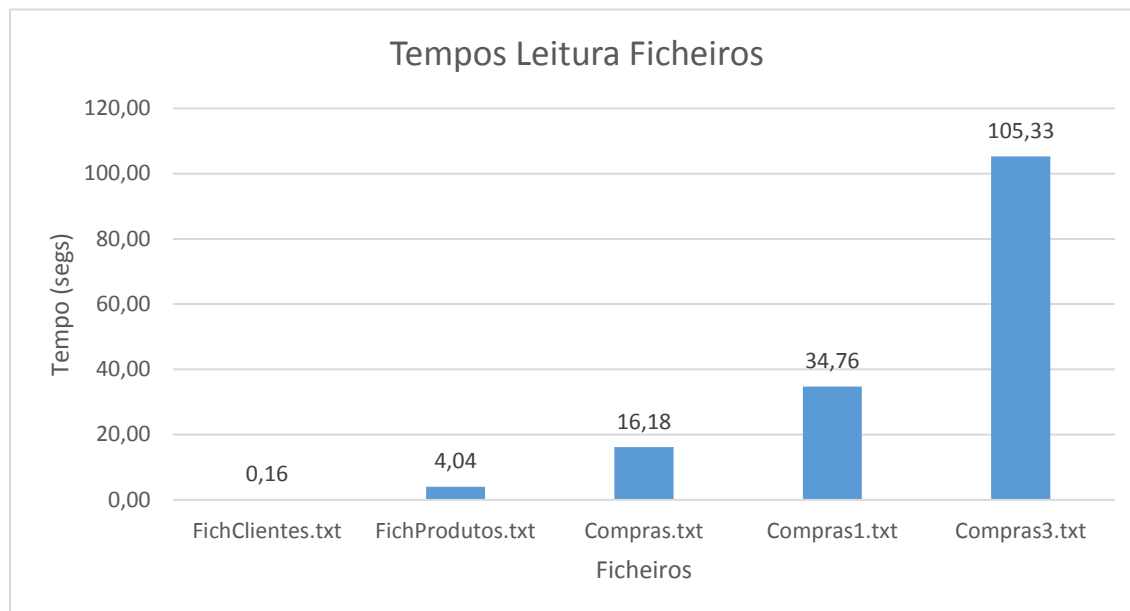
Tendo estes valores, mostrar uma página agora é uma tarefa simples, bastando fazer o seguinte ciclo para apresentar todos os seus elementos:

```
for (int i = 0; i < num_elems_pag_actual; i++)
{
    cliente = listaClientesSemCompras.get(inicio_pagina+i);
    System.out.printf("| %5d | %9s |\n", inicio_pagina+i+1,
    cliente.getCodigoCliente());
}
```

5. PERFORMANCE LEITURA E QUERIES

5.1 PERFORMANCE LEITURA FICHEIROS

De seguida apresentam-se sob forma gráfica os tempos de leitura dos ficheiros.



Tempos Leitura	
Ficheiro	Tempo (secs)
FichClientes.txt	0.16
FichProdutos.txt	4.04
Compras.txt	16.18
Compras1.txt	34.76
Compras3.txt	105.33

Gráfico 1- Tempos de leitura para os vários ficheiros de input

A leitura do ficheiro de clientes é quase instantânea. Isto explica-se facilmente pelo facto do número de clientes ser pequeno (16.384) e de um cliente apenas ser inserido nos catálogos de clientes e nas compras.

O ficheiro de produtos apresenta um tempo significativamente maior que o ficheiro de clientes (4 segundos), o que é explicado pelo facto do input ter um tamanho 116 vezes maior que o input dos clientes (195.584 produtos) e pelo facto da leitura dos produtos colocar informação em 3 módulos: catálogo, contabilidade e compras. No módulo das compras o produto tem que ser colocado no TreeMap da ficha de cliente correspondente e ainda no HashMap que associa cada produto ao seu número de clientes, enquanto que os clientes apenas são colocados na estrutura principal das compras.

Relativamente aos ficheiros de compras, note-se que a leitura do ficheiro de Compras.txt demora metade do tempo do ficheiro Compras1.txt. De facto o ficheiro Compras.txt tem 500.000 compras, que são metade dos registos de compras do ficheiro Compras1.txt. Trata-se de um crescimento de tempo linear, em que o duplicar o tamanho do output leva a que os tempos de leitura também dupliquem. Note-se ainda que quando o tamanho do input triplica, do ficheiro Compras1.txt para o ficheiro Compras3.txt o programa parece não acusar a carga, e o tempo mantém-se também linear, o que confirma uma boa escolha para as estruturas.

5.2 PERFORMANCE QUERIES

Nesta secção apresenta-se as medições ao tempo de resposta às queries que foram efectuadas. Para obter resultados mais conclusivos, o tempo das queries apresentado, reflecte o tempo que as queries demoram a obter **todos** os resultados que precisam. Seria fácil otimizar ainda mais estas queries de modo a que alguns valores apenas fossem calculados quando fossem precisos i.e, quando fosse preciso mostra-los ao utilizador. Esta optimização não foi feita propositadamente, com vista à obtenção de resultados mais conclusivos sobre a performance intrínseca às queries.

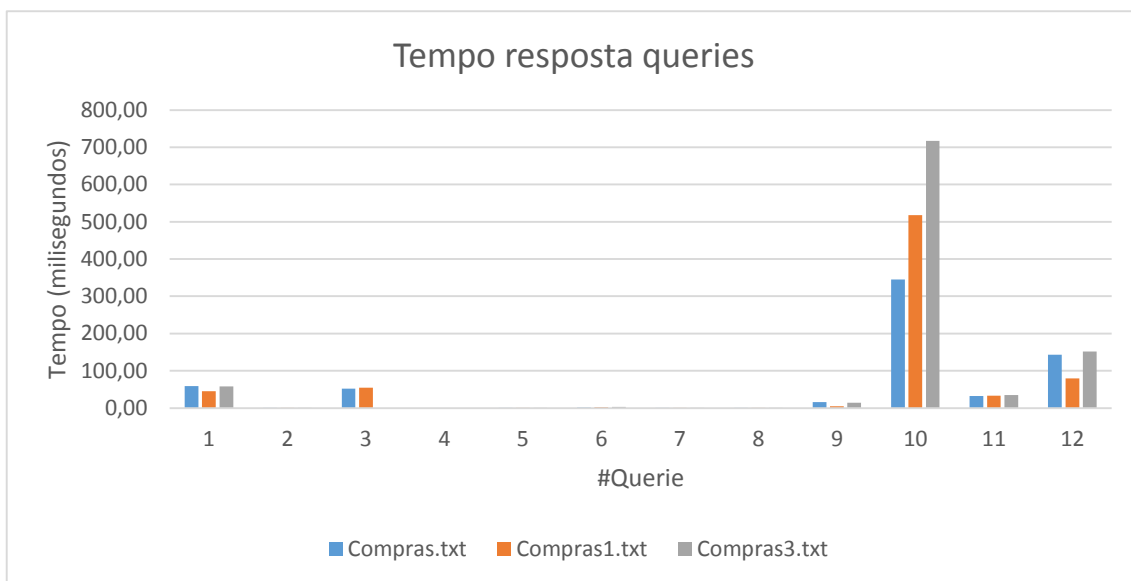


Gráfico 2 - Tempos de resposta às queries

Tempos Queries (ms)			
#	Compras.txt	Compras1.txt	Compras3.txt
1	58.80	45.40	58.00
2	0.40	0.50	0.50
3	52.20	54.90	-
4	-	-	-
5	0.30	0.20	0.20
6	1.10	2.30	2.70
7	0.40	0.30	0.40
8	0.05	0.05	0.08
9	16.00	4.70	14.10
10	345.20	518.20	717.40
11	32.20	32.70	35.10
12	142.80	79.50	151.90

Como se pode ver pela análise do gráfico e tabela acima, os tempos para as queries são bastante bons, sendo todos abaixo de 1 segundo. Visto que nas nossas estruturas optamos por guardar apenas informação global, as queries em geral apenas precisam de consultar essas informações, eventualmente fazendo uma travessia a um dos TreeMap, o que de facto são operações rápidas e explicam os tempos obtidos.

A excepção a esta linha geral é indiscutivelmente a querie 10, de longe a querie mais demorada e a mais complexa. Nesta querie é necessário ir buscar informação das quantidades vendidas de cada produto, ordenar de acordo com a vendas, e posteriormente, para cada um desses produtos saber o número de clientes distintos. Envolve criação e ordenação de listas de pares e consultas aos módulos da contabilidade e das compras.

Como nota final destaca-se o facto do tamanho de input não parecer interferir no tempo de resposta às queries. Embora na querie 10 os tempos possam dar essa ideia, convém referir que para essa querie os tempos foram bastante irregulares, o que dificulta a retirada de conclusões. Os tempos obtidos foram retirados todos na mesma altura, o que pareceu a metodologia mais adequada para este tipo de resultados. No entanto é preciso notar que embora para o ficheiro de Compras3.txt a querie 10 tenha demorado 700 milissegundos, a verdade é que, para outras execuções do mesmo programa, com o mesmo ficheiro, a mesma querie chegou a demorar 300 milissegundos apenas. De facto, não se conseguiu encontrar motivo para que na querie 10 o tempo de resposta aumente com o aumento do tamanho do input, o que parece ser confirmado pela irregularidade dos tempos observados.

6. MEDIDAS PERFORMANCE

6.1 LEITURA

Para a medida de performance da leitura dos ficheiros de entrada, foram comparadas duas classes do java: Scanner e BufferedReader. Para cada uma destas classes foram testados os tempos com parsing e sem parsing. Os tempos de parsing dizem respeito à leitura do ficheiro, armazenamento da linha numa variável String e tratamento dessa linha para a criação de uma instância da classe Cliente, Produto ou Compra (dependendo do ficheiro). Nenhuma operação é realizada com a instância criada. Os tempos sem parsing correspondem apenas à leitura simples do ficheiro, em que as linhas são armazenadas numa variável, mas nada é feito com a informação da linha.

Os tempos obtidos foram os seguintes:

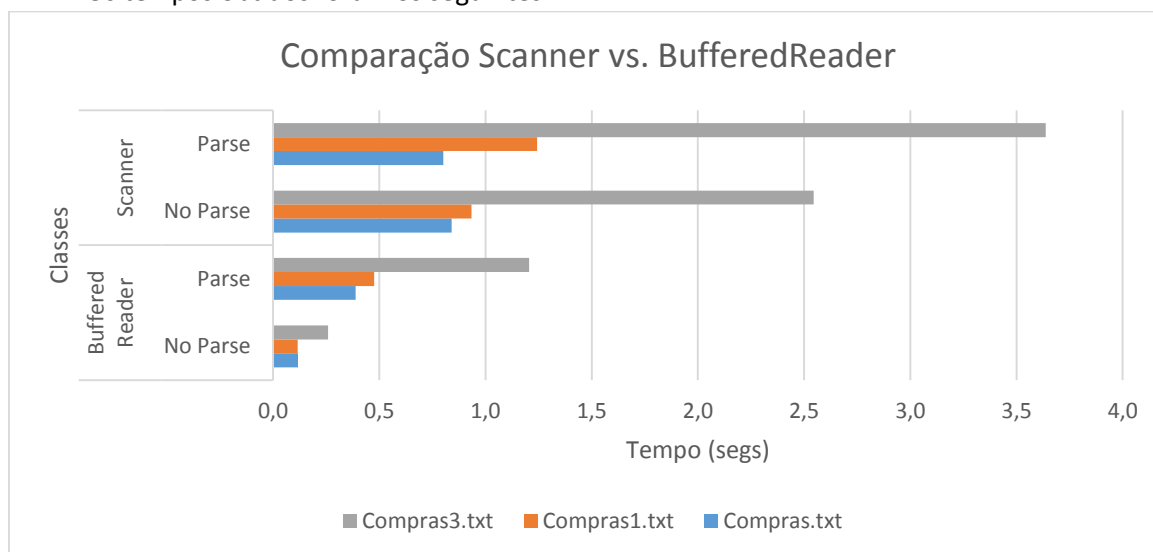


Gráfico 3 - Análise comparativa aos tempos de leitura das classes Scanner e BufferedReader

Tempos Leitura (secs)				
Ficheiros	Buffered Reader		Scanner	
	No Parse	Parse	No Parse	Parse
Compras.txt	0.117	0.389	0.841	0.802
Compras1.txt	0.116	0.476	0.933	1.243
Compras3.txt	0.259	1.205	2.545	3.637

A classe BufferedReader revelou-se de facto a mais eficiente para proceder à leitura dos ficheiros. Sem surpresa, a leitura com ambas as classes foi mais demorada quando foi feito o parsing da informação da linha.

6.2 ESTRUTURAS

Sendo as principais estruturas usadas no programa TreeMaps e um HashMap, para realizar testes relativos à performance das estruturas decidiu-se verificar qual os tempos para as queries tendo todos os Maps como TreeMaps em primeiro lugar, e como HashMaps em segundo lugar.

Para isso, começou-se por ver quais os tempos de leitura dos ficheiros de input. Para o ficheiro de compras, usamos como referência apenas o ficheiro Compras1.txt. Os resultados para os outros ficheiros de compras foram semelhantes.

Por uma questão de simplificação, nos gráficos e tabelas seguintes “Tree+HashMap” servirá para referenciar o programa entregue, que tem todos os módulos principais como TreeMaps e nas compras um HashMap na estrutura secundária das compras que a cada produto associa o nº de clientes distintos, conforme apresentado em secções anteriores. “TreeMap” diz respeito ao programa com todas essas estruturas dos módulos substituídas por TreeMaps e “HashMap” refere-se ao programa com as mesmas estruturas substituídas por HashMaps.

Os tempos de leitura dos ficheiros de input foram os seguintes:

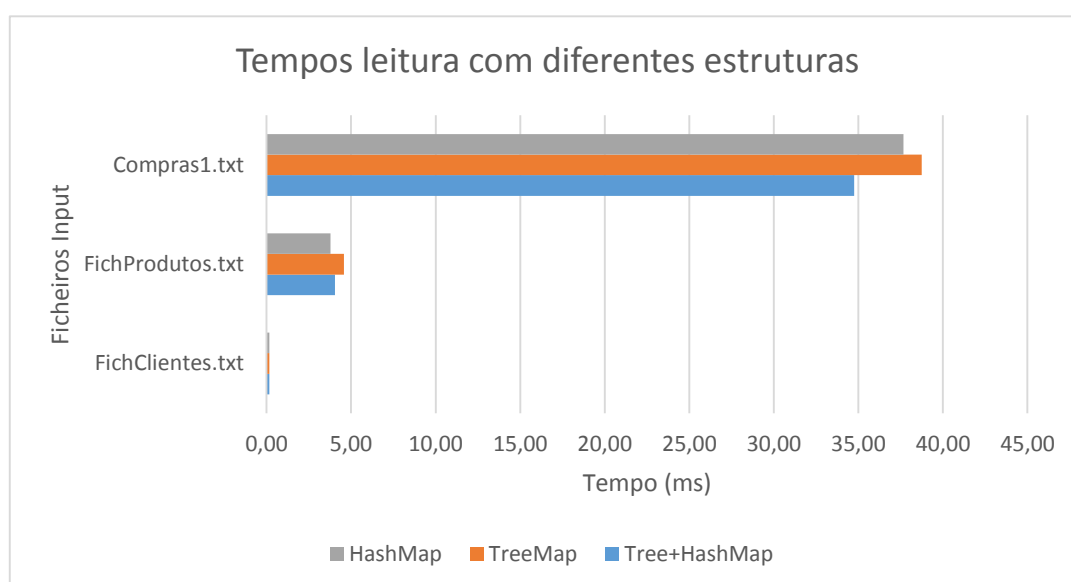


Gráfico 4 - Tempos de leitura dos ficheiros de input usando diferentes estruturas

Tempos Leitura (segs)			
Ficheiro	Tree+HashMap	TreeMap	HashMap
FichClientes.txt	0.16	0.16	0.17
FichProdutos.txt	4.04	4.57	3.78
Compras1.txt	34.76	38.74	37.66

Os tempos das queries foram os seguintes:

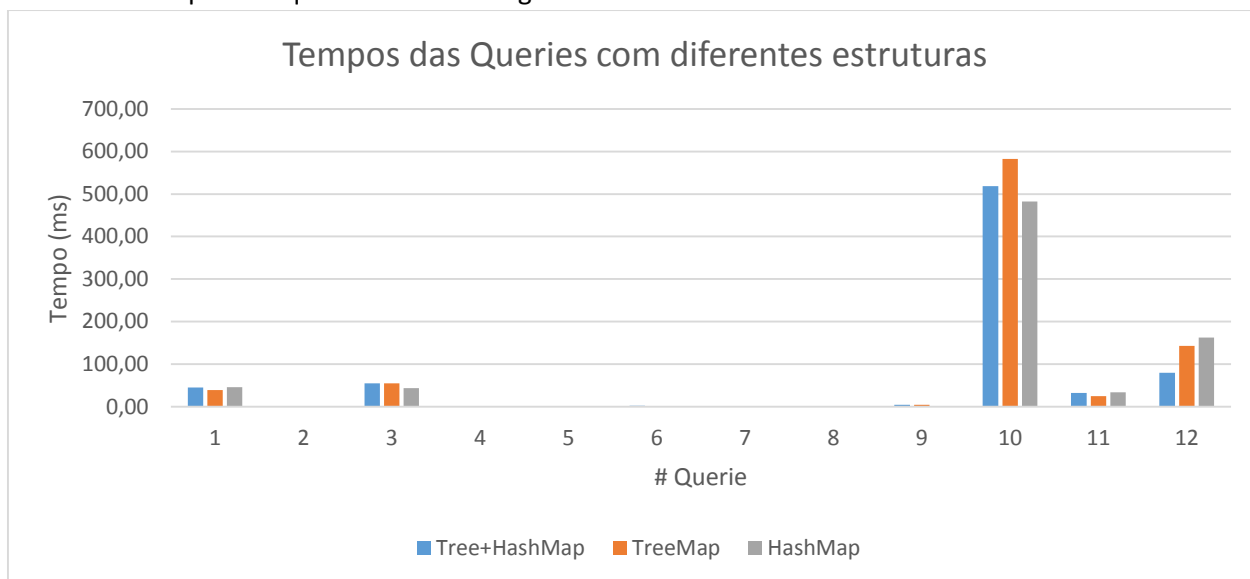


Gráfico 5 - Tempos das execuções das queries com as diferentes estruturas

Tempos Queries (ms)			
Querye	Tree+HashMap	TreeMap	HashMap
1	45.40	39.50	46.10
2	0.50	0.50	0.20
3	54.90	54.70	43.40
4	-	-	-
5	0.20	0.20	0.10
6	2.30	1.90	1.70
7	0.30	0.50	0.50
8	0.05	0.05	0.04
9	4.70	4.20	0.60
10	518.20	582.30	482.10
11	32.70	24.70	34.00
12	79.50	143.30	162.70

Tanto na leitura como nos resultados das queries não houve diferenças significativas em relação aos tempos registados. Nas queries notou-se uma tendencia ligeiramente positiva a favor do uso de HashMaps, no entanto esta pequena diferença de tempo é insignificante se se pensar que os resultados dos HashMaps não estão ordenados, quando o deveriam estar. Isto é importante para a apresentação de resultados ao utilizador, uma vez que depois da obtenção de resultados é necessário um “esforço” adicional para os ordenar. Isto torna os HashMaps pouco competitivos em relação aos TreeMaps nesse aspecto, e em algumas queries este factor foi fundamental para a decisão de ter um TreeMap e não um HashMap.

Sendo a querie 10 a pior querie em termos de tempos, decidiu-se investiga-la mais a fundo e ver qual o impacto destas diferentes estruturas nessa querie.

A querie 10 faz duas coisas: pede ao módulo da contabilidade uma lista de pares (Produto, Unidades Vendidas) dos produtos da contabilidade, estando esta lista ordenada pela segunda componente dos pares e pelo código de produto caso a segunda componente tome o mesmo valor. Numa segunda fase, para cada um dos produtos dessa lista, é pedido ao módulo compras para indicar o número de clientes distintos de cada um desses produtos, guardando os resultados desses pedidos num ArrayList.

Assim sendo, para esta querie fez sentido não apenas analisar o seu tempo global, mas analisar o tempo de cada uma destas tarefas.

Os resultados foram os seguintes:

Tempos Querie 10 (secs)			
	Tree+HashMap	TreeMap	HashMap
Obter Lista Produtos	0.442	0.326	0.401
Array nº Clientes únicos	0.076	0.256	0.081
Total	0.518	0.582	0.482

Estes resultados mostram que de facto, a decisão inicial de se usar um HashMap na estrutura secundária do módulo das compras para associar produtos ao seu número de clientes foi, de facto, uma boa escolha. Com um HashMap nessa estrutura, a tarefa de para cada produto saber seu número de clientes fica mais de duas vezes mais rápida.

7. CONCLUSÃO

O projecto entregue cumpre os requisitos propostos.

A modularidade foi garantida através da criação de classes com API completa, com constructores apropriados e métodos que permitem um bom número de operações com as classes e módulos do programa. Nesse espírito, foram ainda incluídos não só nas classes dos módulos como em todas as que tal se justificava, métodos essenciais a qualquer classe “bem comportada” nomeadamente `hashCode()`, `toString()`, `equals()` e `clone()`.

O encapsulamento dos módulos foi garantido através do uso de `clone's` ao inserir nas estruturas e ao serem procurados elementos nas mesmas.

Além da modularidade e encapsulamento, uma boa estruturação do programa e legibilidade do mesmo também foram aspectos tidos em conta, o que levou à criação de tipos enumerados e métodos e variáveis com nomes sugestivos, ainda que isso tenha levado a que esses nomes fossem por vezes longos.

Os tempos de resposta às queries e de leitura dos ficheiros são também bastante satisfatórios na nossa opinião. O tempo de leitura dos ficheiros é melhor que linear no tamanho dos ficheiros de compras e nenhuma query demora mais que um minuto a ser executada, sendo que a maioria demora menos de 1 milissegundo. Estes resultados reflectem um bom planeamento da arquitectura do programa e boa escolha das estruturas usadas.

Entre várias possíveis melhorias ao programa, destaca-se a melhoria que poderia ser feita relativamente aos tempos de serialização. Os tempos de leitura e escrita de ficheiros objecto são longos. No caso do nosso programa, demora menos tempo ler dos ficheiros de texto do que ler de um ficheiro objecto com a mesma informação. A título experimental conseguimos com sucesso melhorar estes tempos usando a biblioteca `Kryo`¹, no entanto alguns erros de última hora levaram-nos a “jogar pelo seguro” e não incluir a serialização feita com esta biblioteca no programa entregue.

¹ Disponível em <https://github.com/EsotericSoftware/kryo>

8. BIBLIOGRAFIA

- [1] F. M. J. Martins, JAVA6 e Programação Orientada aos Objectos, FCA, 2009.