|  |  |
| --- | --- |
| EENG | **Escola de Engenharia**  Departamento de Engenharia Informática  Mestrado Integrado em Engenharia Informática  Laboratórios de Informática III |

Relatório de Laboratórios de Informática III

**GereVendas**

**(Trabalho Prático de Java)**

Grupo 1

João Martins (A68646)



João Pereira (A75273)



Manel Castro (A71646)



Índice

[1. Introdução 3](#_Toc453705088)

[2. Arquitetura de classes 4](#_Toc453705089)

[2.1 Constantes 5](#_Toc453705090)

[2.2 LStrings 5](#_Toc453705091)

[2.3 EstatisticasFicheiro 6](#_Toc453705092)

[2.4 EstatisticasGerais 7](#_Toc453705093)

[2.5 CatalogoProdutos 9](#_Toc453705094)

[2.6 CatalogoClientes 10](#_Toc453705095)

[2.7 FatProdMes 11](#_Toc453705096)

[2.8 FatMes 13](#_Toc453705097)

[2.9 FatAnualProd 14](#_Toc453705098)

[2.10 Faturacao 15](#_Toc453705099)

[2.11 ComprasDoProduto 17](#_Toc453705100)

[2.13 Filial 19](#_Toc453705101)

[2.14 Filiais 20](#_Toc453705102)

[2.15 Pares e triplos 21](#_Toc453705103)

[2.16 Excepções 22](#_Toc453705104)

[2.17 Venda 22](#_Toc453705105)

[2.18 Hipermercado 23](#_Toc453705106)

[2.19 Menu 24](#_Toc453705107)

[2.20 HipermercadoApp 25](#_Toc453705108)

[3. Interface de utilizador e navegação 26](#_Toc453705109)

[3.1 Menus 26](#_Toc453705110)

[3.2 Navegação 28](#_Toc453705111)

[3. Medidas de *performance* 29](#_Toc453705112)

[3.1 Análise de tempos de leitura: Scanner vs BufferedReader 29](#_Toc453705113)

[3.2 Análise de desempenho das diferentes coleções 30](#_Toc453705114)

[3.3 Outros dados de execução para o ficheiro Vendas\_3M.txt 36](#_Toc453705115)

[4. Conclusão 39](#_Toc453705116)

[Figura 1-Arquitetura de classes 4](#_Toc453705124)

[Figura 2-Esquema da estrutura de dados de EstatisticasGerais 9](#_Toc453705125)

[Figura 3-Esquema de estrutura de dados de CatalogoProdutos 10](#_Toc453705126)

[Figura 4-Esquema da estrutura de dados de CatalogoClientes 11](#_Toc453705127)

[Figura 5-Esquema da estrutura de dados de FatProdMes 12](#_Toc453705128)

[Figura 6-Estrutura de dados de FatMes 14](#_Toc453705129)

[Figura 7-Esquema da estrutura de dados de FatAnualProd 15](#_Toc453705130)

[Figura 8-Esquema de estrutura de dados da faturação 17](#_Toc453705131)

[Figura 9-Esquema da estrutura de dados de ComprasPorCliente 19](#_Toc453705132)

[Figura 10-Esquema da estrutura de dados de Filial 20](#_Toc453705133)

[Figura 11-Esquema da estrutura de dados utilizada em Filiais 21](#_Toc453705134)

[Figura 12-Menu principal 26](#_Toc453705135)

[Figura 13-Menu de leitura de dados 26](#_Toc453705136)

[Figura 14-Menu de estatísticas 27](#_Toc453705137)

[Figura 15-Menu das queries interativas 27](#_Toc453705138)

[Figura 16-Menu apresentado à saída 28](#_Toc453705139)

[Figura 17-Opções da navegação 28](#_Toc453705140)

[Figura 18-Exemplo de cabeçalho apresentado na navegação 28](#_Toc453705141)

[Figura 19-Gráfico de comparação dos tempos de leitura: Scanner (+FileReader) vs. BufferedReader 29](#_Toc453705142)

[Figura 20-Gráfico de performance da query5, para diferentes coleções 31](#_Toc453705143)

[Figura 21-Gráfico de performance da query 6, para diferentes coleções 32](#_Toc453705144)

[Figura 22- Gráfico de performance da query 7, para diferentes coleções 33](#_Toc453705145)

[Figura 23- Gráfico de performance da query 8, para diferentes coleções 34](#_Toc453705146)

[Figura 24-Gráfico de performance da query 9, para diferentes coleções 35](#_Toc453705147)

[Figura 25 – Utilização de CPU e da heap ao implementar Lists com ArrayList (Vendas\_3M) 36](#_Toc453705148)

[Figura 26 - Utilização de CPU e da heap ao implementar Lists com Vector (Vendas\_3M) 36](#_Toc453705149)

[Figura 27 - Utilização de CPU e da heap ao implementar Maps com HashMap (Vendas\_3M) 37](#_Toc453705150)

[Figura 28 - Utilização de CPU e da heap ao implementar Maps com TreeMap (Vendas\_3M) 37](#_Toc453705151)

[Figura 29 - Utilização de CPU e da heap ao implementar Sets com HashSet (Vendas\_3M) 38](#_Toc453705152)

[Figura 30 - Utilização de CPU e da heap ao implementar Sets com TreeSet (Vendas\_3M) 38](#_Toc453705153)

[Tabela 1-Análise dos tempos de leitura: Scanner vs. BufferedReader 29](#_Toc453705154)

[Tabela 2-Ganhos conseguidos nas leituras com BufferedReader, comparativamente com Scanner (+FileReader) 29](#_Toc453705155)

[Tabela 3-Performance da query 5 com diferentes coleções 31](#_Toc453705156)

[Tabela 4- Performance da query 6 com diferentes coleções 32](#_Toc453705157)

[Tabela 5-Performance da query 7 com diferentes coleções 33](#_Toc453705158)

[Tabela 6-Performance da query 8 com diferentes coleções 34](#_Toc453705159)

[Tabela 7- Performance da query 9 com diferentes coleções 35](#_Toc453705160)

# Introdução

O trabalho desenvolvido consistiu na 2ª e última fase do projeto **GereVendas**, em que se procedeu ao desenvolvimento de uma aplicação Java capaz de responder de forma eficiente a questões relacionadas com os clientes, produtos e vendas de um hipermercado com 3 filiais. O carregamento dos dados do hipermercado tanto pode ser realizado a partir de um ficheiro de estado, como através de 3 ficheiros de texto: um ficheiro de clientes, um ficheiro de produtos e um ficheiro de vendas. Depois de carregados os dados, passa a ser possível utilizar todas as funcionalidades do programa, nomeadamente: guardar em ficheiro o estado da aplicação, consultar estatísticas do último ficheiro de vendas lido, obter estatísticas gerais sobre os dados carregados e realizar qualquer uma de 9 *queries* interativas de forma a obter diversas informações sobre produtos, clientes, faturação e vendas. Em termos de estruturas armazenadas temos: estatísticas sobre o último ficheiro de vendas lido, estatísticas gerais, catálogo de produtos, catálogo de clientes, faturação e filiais. Todos estes dados dizem respeito ao hipermercado em questão, pelo que são armazenados numa instância da classe Hipermercado, que constitui uma unidade lógica agregadora de todos os dados consultados na aplicação. Além da aplicação acima descrita, foram também desenvolvidos testes de *performance* da leitura, com e sem *parsing*, dos vários ficheiros de vendas (Vendas\_1M.txt, Vendas\_3M.txt e Vendas\_5M.txt), bem como comparações do desempenho das *queries* 5 a 9 entre adaptações da aplicação principal, nas quais se alterou a implementação das coleções e maps utilizados na versão original.

# Arquitetura de classes

A arquitetura de classes utilizada neste projeto de Java é até certo ponto uma “tradução” da organização dos módulos do projeto de C, ainda que tenhamos efetuado algumas alterações e adaptações dos vários tipos de dados para reduzirmos os tempos de leitura dos ficheiros de texto. Além das alterações que visam reduzir o tempo de leitura, também acrescentámos:

* pares e triplos para agrupar os resultados das várias *queries*;
* estatísticas relativas ao último ficheiro lido e estatísticas gerais, relativas aos dados do hipermercado;
* uma classe de medição de tempos (a classe **Crono**, disponibilizada pelo professor Mário Martins);

A arquitetura de classes utilizada pelo grupo tem então a seguinte forma:

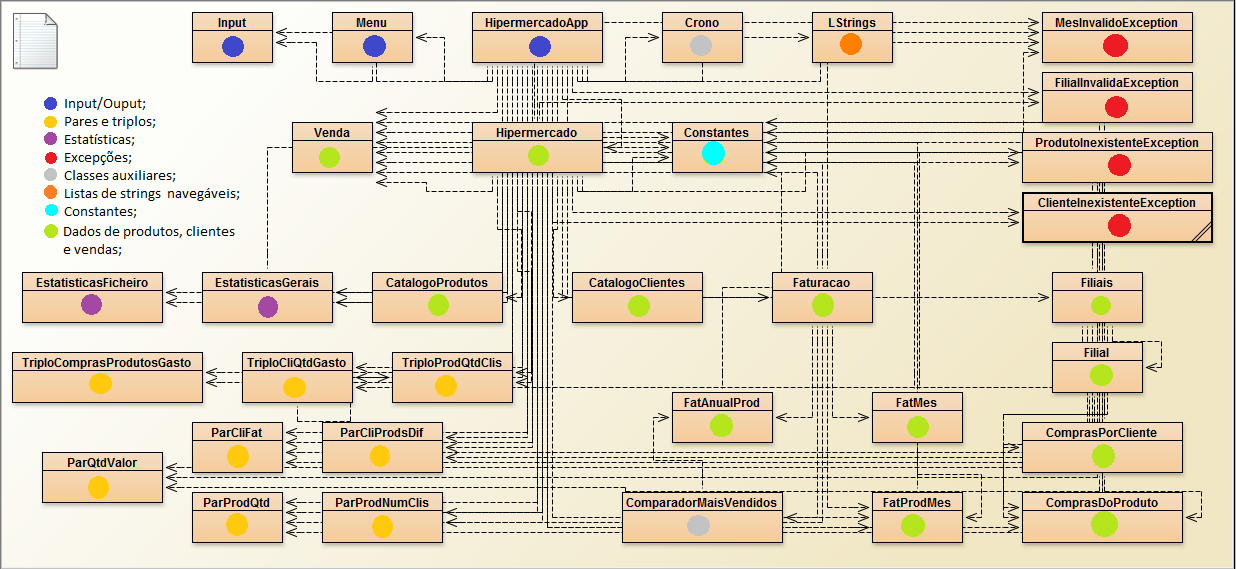


Figura -Arquitetura de classes

## Constantes

A classe **Constantes** tem apenas 2 variáveis, que definem as constantes utilizadas na aplicação **GereVendas**. Assim, no corpo desta classe temos as declarações:



As variáveis apresentadas foram declaradas como:

* **public** para podermos ter acesso às mesmas nas restantes classes do projeto, sem necessitarmos de definir *getters*;
* **static** para não ser necessária uma instância para aceder aos seus valores;
* **final** para que sejam de facto constantes e as restantes classes não as possam modificar inadvertidamente;

De referir que as restantes variáveis de instância e variáveis de classe deste projeto foram declaradas como **private**, de forma a evitar o acesso direto às mesmas, esconder a estrutura interna das várias classes e instâncias e preservar o encapsulamento.

## LStrings

A classe LStrings representa listas de Strings com noção de página. As instâncias de LStrings permitem a navegação de grandes volumes de dados (representados na forma de Strings), página a página. Por omissão, uma LStrings tem 20 Strings por página, no entanto a API de LStrings permite a criação listas com um número diferente de Strings por página, através do construtor:

**LStrings**([Collection](http://download.oracle.com/javase/8/docs/api/java/util/Collection.html?is-external=true)<[String](http://download.oracle.com/javase/8/docs/api/java/lang/String.html?is-external=true)> strings, int tamanhoPag) (para mais informação, ver *javadoc* do projeto).

Em termos de funcionalidade, cada instância LStrings permite:

* Obter a página atual;
* Ir para a página seguinte;
* Ir para a página anterior;
* Ir para a primeira página;
* Ir para a última página;
* Selecionar uma página;

Além destas operações, cada instância desta classe pode também devolver informações como o seu número total de páginas, o número de Strings por página e o número da página em que se encontra.

### Atributos

Na declaração de v.i. de LStrings, temos:



, onde:

* **strings** é uma List com todas as Strings de uma instância;
* **numPag** é o número da página atual;
* **tamanhoPag** é o número de Strings por página;
* **totalPags** é o número total de páginas;

**Nota:** o índice da primeira String da página atual é facilmente calculado, através da multiplicação .

### Variáveis de classe

O número de Strings por página por omissão é constante e não faria sentido colocá-lo em todas as instâncias criadas, pelo que optámos por declará-lo da seguinte forma:



## EstatisticasFicheiro

Cada instância da classe **EstatisticasFicheiro** contém informações sobre o último ficheiro de vendas lido pelo utilizador, tais como: nome do ficheiro, número total de registos de venda errados, número total de produtos, total de diferentes produtos comprados, total de produtos não comprados, número total de clientes, total dos que realizaram compras, total de clientes que nada compraram, total de compras de valor total igual a 0.0 e faturação global.

Na aplicação desenvolvida, as estatísticas do último ficheiro lido são criadas sempre que é efetuada a leitura de um novo ficheiro de vendas.

### Atributos

Na declaração de v.i. de EstatisticasFicheiro temos:



, onde:

* **ficheiro** guarda o nome do último ficheiro de vendas lido pelo utilizador;
* **totalVendasErr** representa o total de vendas erradas/inválidas;
* **totalProdutos** indica o número total de produtos no sistema;
* **totalDifProdsCompr** representa o número total de diferentes produtos comprados;
* **totalClientes** indica o número total de clientes no sistema;
* **totalCliCompr** guarda o número total de clientes que realizaram compras;
* **totalComprasZero** indica o número total de compras de valor total igual a 0.0;
* **faturacaoTotal** representa o valor total faturado;

**Nota:** não é necessário guardar os totais de clientes que não realizaram compras e de produtos não comprados, uma vez que estes valores podem ser facilmente obtidos pelas subtrações:

* **totalClientes** – **totalCliCompr**;
* **totalProdutos** – **totalDifProdCompr**;

Por fim, é importante mencionar que declarámos todas as v.i. de EstatisticasFicheiro como **final** e não disponibilizámos *setters* para garantirmos que as instâncias de EstatisticasFicheiro são imutáveis. Tomámos esta decisão porque cada instância de EstatisticasFicheiro é criada no final da leitura de um ficheiro de vendas e a partir daí as estatísticas relativas ao ficheiro de vendas lido não sofrem quaisquer alterações. Se o utilizador ler um novo ficheiro de vendas, criamos uma nova instância de EstatisticasFicheiro com os dados desse ficheiro.

## EstatisticasGerais

Cada instância de **EstatisticasGerais** guarda informações gerais respeitantes aos dados atuais registados nas estruturas, tais como,

* Número total de compras por mês;
* Faturação total por mês, para cada filial e valor total global;
* Número de clientes distintos que compraram em cada mês;

Como as estatísticas gerais dizem respeito aos dados registados nas estruturas, estas podem ser calculadas só quando o utilizador as pede pela 1ª vez. Assim, no contexto da aplicação desenvolvida optámos por não criar as estatísticas gerais aquando da leitura dos ficheiros, criando-as em vez disso quando o utilizador tenta consultá-las pela 1ª vez.

**Nota:** Não disponibilizámos *setters* para garantir que as instâncias de EstatisticasGerais são imutáveis. A razão desta decisão prende-se com o facto de depois de lidos os vários ficheiros, os dados gerais relativos às estruturas em memória não se alterarem até à leitura de novos ficheiros. Assim, a cada conjunto de 3 ficheiros (produtos, clientes e vendas) corresponde uma instância de EstatisticasGerais distinta e imutável.

### Atributos

Na declaração de v.i. de EstatisticasGerais temos:



, onde:

* **totalCompras** é um *array* de inteiros que na posição de indice i tem o número total de compras no mês i;
* **totalFaturado** é um *array* bi-dimensional de *doubles*, que na posição de índices (i,j) armazena o total faturado no mês i, na filial j;
* **totalCliDistintos** é um *array* de inteiros que na posição de indice i guarda o número total de clientes distintos que compraram no mês i;

### 1.4.2 Esquema da estrutura de dados

private int[] totalCompras;

private int[] totalCliDistintos;

(*padding*)

Janeiro

…

Dezembro

Fevereiro

[0]

[1]

…

[12]

[2]

…

Meses

Jan

Dez

1

2

3

(*padding*)

(*padding*)

Filiais

Fev

private double[][] totalFaturado;

Figura -Esquema da estrutura de dados de EstatisticasGerais

## CatalogoProdutos

Cada instância da classe **CatalogoProdutos** permite o registo de códigos de produto (valores da classe String), bem como a verificação da existência de um determinado código de produto e a consulta do número total de produtos registados nessa instância.

No contexto da aplicação desenvolvida, utilizámos uma instância de CatalogoProdutos nas v.i. de Hipermercado (secção 2.18.1) para registar os códigos de todos os produtos disponíveis, obter o total de produtos existentes no hipermercado, verificar a existência dos códigos de produto que surgem nas linhas do ficheiro de vendas e validar os códigos de produto que o utilizador introduz em algumas das *queries* interativas.

### Atributos

De forma a conseguirmos inserções e consultas em tempo tendencialmente constante, optámos por usar um HashSet<String> para guardar os códigos dos produtos de cada instância de CatalogoProdutos. No entanto, para podermos mudar facilmente a implementação para outro tipo de Set, declaramos a única v.i. de CatalogoProdutos cf. :



Assim, se pretendermos alterar a implementação do conjunto de códigos de produtos basta modificarmos a expressão codigosProdutos = new HashSet<>(), nos construtores de CatalogoProdutos, de acordo com o tipo de Set que pretendermos utilizar.

**Nota:** de referir que utilizámos um **Set** (e não uma **List**) porque cada código de produto é único e, por conseguinte, é necessário garantir a inexistência de elementos repetidos em cada instância de CatalogoProdutos. Ora, um **Set** permite-nos precisamente alcançar esse fim.

### Esquema da estrutura de dados

…

“NR1091”

“AF1184”

“DV1252”

\*

\*

\*

“AF1184”

\*

\*

…

“DV1252”

\*

“NR1091”

Figura -Esquema de estrutura de dados de CatalogoProdutos

## CatalogoClientes

As instâncias e API de **CatalogoClientes** são semelhantes às de CatalogoProdutos, diferindo apenas no conteúdo dos catálogos, já que num caso temos clientes e no outro temos produtos. Assim, cada instância de CatalogoClientes permite-nos: inserir códigos de cliente, obter o número total de clientes registados na mesma e testar se um dado código de cliente faz parte dessa instância.

No caso da aplicação desenvolvida, utilizámos uma instância de CatalogoClientes nas v.i. de Hipermercado (secção 2.18.1), de forma a armazenar os códigos dos vários clientes registados no hipermercado, obter o total de clientes registados, comprovar a existência dos códigos de cliente que surgem nas linhas do ficheiro de vendas e validar códigos de cliente introduzidos pelo utilizador em algumas das *queries* interativas da aplicação.

### Atributos

Cada instância de CatalogoClientes tem uma única v.i. que, tal como a v.i. de CatalogoProdutos, é um Set<String> implementado com um HashSet<String>, permitindo-nos a realização inserções e consultas em tempo tendencialmente constante.

Assim, na declaração de v.i. de CatalogoClientes temos:



**Nota:** tal como os códigos de produtos, os códigos de cliente também são únicos. A utilização de um **Set** (em vez de uma **List**) garante a inexistência de códigos de cliente repetidos em cada instância de CatalogoClientes.

### Esquema da estrutura de dados

…

“Z5000”

“B3304”

“S4262”

\*

\*

\*

“Z5000”

\*

\*

…

“S4262”

\*

“B3304”

Figura -Esquema da estrutura de dados de CatalogoClientes

## FatProdMes

A classe **FatProdMes** representa a faturação de um produto num mês. Cada instância desta classe permite-nos obter informações relativas às vendas de um produto num mês, como:

* Número de unidades vendidas em cada filial e globalmente;
* Valor faturado em cada filial e globalmente;

De referir que não disponibilizámos *setters* para FatProdMes, porque qualquer manipulação e alteração dos valores armazenados numa instância de FatProdMes pode ser realizada através do método adiciona().

(para obter mais informação acerca deste método, consultar o *javadoc* do projeto).

### Atributos

Na declaração de v.i. de FatProdMes, temos:



, onde:

* **mes** é um inteiro que indica o mês do ano a que uma instância de FatProdMes diz respeito;
* **codigoProduto** é o código do produto ao qual se referem os dados guardados na instância;
* **unidsVendFilial** é um *array* de inteiros que na posição de índice i tem o número de unidades vendidas na filial i, para o produto e mês em questão;
* **faturacao** – é um *array* de doubles que na posição i tem a faturação obtida na filial i, para o produto e mês em questão;

**Nota:** embora no âmbito da nossa aplicação não fosse necessário guardar o mês nas v.i. de FatProdMes, para garantirmos que a classe FatProdMes é independente do contexto em que está inserida optámos por ter o mês como variável de instância.

### Esquema da estrutura de dados

private int[] unidsVendFilial;

(*padding*)

Unids vend. na filial 1

Unids. vend na filial 3

Unids. vend. na filial 2

[0]

[1]

[3]

[2]

private final int mes;

private String codigoProduto;

(*padding*)

Faturado na filial 1

Faturado na filial 3

Faturado na filial 2

[0]

[1]

[3]

[2]

private double[] faturacao;

Figura -Esquema da estrutura de dados de FatProdMes

## FatMes

A classe FatMes representa a faturação de um dado mês. Cada instância de FatMes permite-nos obter dados como:

* Total de vendas do mês a que diz respeito;
* Total faturado nesse mês;
* Faturação de cada um dos produtos vendidos nesse mês;

### Atributos

Na declaração das v.i. de FatMes temos:



, onde:

* **mes** é um valor inteiro que representa o mês a que uma instância diz respeito;
* **nfiliais** é o número de filiais a considerar, aquando da criação da faturação dos produtos no mês (FatProdMes);
* **totalVendas** é o total de linhas de venda do ficheiro de vendas;
* **totalFaturado** é o total de dinheiro faturado no mês;
* **fatProds** é um mapeamento de código de produto para a sua faturação no mês da instância de FatMes considerada;

Relativamente ao Map<String, FatProdMes> fatProds, decidimos implementá-lo com um HashMap porque verificámos experimentalmente (através de medições de tempo) que esta implementação permite uma redução do tempo de leitura dos ficheiros de texto, comparativamente com uma implementação que faça uso de um TreeMap.

**Nota:** O número de filiais (**nfiliais**) é guardado para que aquando da construção da faturação de um produto num mês (FatProdMes – secção 2.7) saibamos para quantas filiais temos que guardar informação. Poderíamos ter usado a constante 3, mas nesse caso as classes FatMes e FatProdMes não seriam independentes do contexto da nossa aplicação. Relativamente a guardarmos o mês nas v.i. de FatMes, fizemo-lo porque embora no contexto das instâncias de Faturacao saibamos a que mês diz respeito cada instância de FatMes (através da posição que essa instância ocupa no array FatMensal[], a apresentar na secção 2.10 - Faturacao), noutro contexto poderíamos não saber. Assim, ao guardarmos o mês numa v.i. de FatMes garantimos também a independência de contexto.

### Esquema das estruturas de dados

private Map<String, FatProdMes> todosProdutos; (implementado com HashMap)

“VI1834”

“IT1158”

“QK1900”

\*

\*

\*

<”IT1158”, FatProdMes>

\*

\*

…

<”QK1900”, FatProdMes>

\*

<”VI1834”, FatProdMes>

…

private final int mes;

private final int nfiliais;

private int totalVendas;

private double totalFaturado;

Figura -Estrutura de dados de FatMes

## FatAnualProd

A classe **FatAnualProd** representa a faturação anual de um produto. Cada instância de FatAnualProd guarda o código do produto a que diz respeito e a quantidade vendida desse produto, no ano todo, para cada uma das filiais.

De referir que não disponibilizamos *setters* porque qualquer alteração e atualização dos campos de uma instância de FatAnualProd pode ser realizada através do método adicionaUnidades(), da API de FatAnualProd.

No caso da aplicação desenvolvida, usámos instâncias de FatAnualProd no Map<String, FatAnualProd> todosProdutos das v.i. de Faturacao (secção 2.10.1), onde mapeamos os códigos dos vários produtos (incluindo os não vendidos) na respetiva faturação anual. Iterando os valores do Map acima referido conseguimos obter os códigos dos produtos que nunca foram vendidos. Ordenando a coleção dos valores desse Map (devolvida pelo método values() ) por ordem decrescente de quantidade total vendida e limitando em seguida o tamanho da coleção ordenada a um inteiro X, conseguimos obter os X produtos mais vendidos ao longo do ano.

### Atributos

Na declaração das v.i. de FatAnualProd temos:



, onde

* **codigoProduto** é o código do produto a que a instância diz respeito;
* **totalUnids** é um *array* de inteiros que na posição de índice i guarda o total de unidades vendidades na filial i, no ano todo;

### Esquema da estrutura de dados

private int[] totalUnids;

(*padding*)

Unids vend. na filial 1

Unids. vend na filial 3

Unids. vend. na filial 2

[0]

[1]

[3]

[2]

private String codigoProduto;

Figura -Esquema da estrutura de dados de FatAnualProd

## Faturacao

Cada instância de **Faturacao** guarda informação sobre as vendas mensais e do ano todo, filial a filial e globalmente. Nas vendas de um dado mês temos apenas dados relativos aos produtos vendidos nesse mês. Na informação do ano todo temos dados sobre todos os produtos, mesmo os que nunca foram vendidos, para que a faturação possa indicar quais produtos nunca foram comprados. É também importante mencionar que não é feita qualquer referência a clientes, já que toda a informação sobre quem realizou cada compra é mantida pelas respetivas filiais.

No contexto da aplicação desenvolvida, temos uma instância de Faturacao nas v.i. da classe de Hipermercado (secção 2.18.1), que nos permite responder a questões como:

* Quais produtos nunca foram comprados e o seu respetivo total (*query* 1);
* Qual foi o número global de vendas realizadas num dado mês (*query* 2);
* Quantas vezes um produto foi comprado e qual foi o total faturado com esse produto, para cada um dos meses (*query* 4);
* Quais foram os X produtos mais vendidos em todo o ano (*query* 6);

**Nota:** na *query* 6, X é um inteiro dado pelo utilizador.

### Atributos

Nas v.i. de instância de Faturacao temos:



, onde:

* **nfiliais** é o número de filiais para as quais a instância guarda informação de faturação (guardamos este valor porque embora no caso da nossa aplicação conheçamos o número de filiais, se mudássemos para outro contexto o número de filiais poderia ser diferente);
* **fatMensal** é um *array* que na posição de índice i tem uma instância de FatMes, com informação relativa à faturação do mês i;
* **todosProdutos** mapeia códigos de produtos em instâncias de FatAnualProd com informação relativa à faturação anual do produto utilizado como chave de pesquisa;

### Esquema das estruturas de dados

private final int nfiliais;

private FatMes[] fatMensal;

(*padding*)

FatMes (Jan)

…

FatMes (Fev)

FatMes (Dez)

[0]

[1]

…

[2]

[12]

…

<”NR1076”, FatAnualProd>

<”AF1182”, FatAnualProd>

<”QQ1149”, FatAnualProd>

…

…

…

private Map<String, FatAnualProd> todosProdutos; (implementado com TreeMap)

Figura -Esquema de estrutura de dados da faturação

## ComprasDoProduto

Cada instância de **ComprasDoProduto** guarda informação acerca das vendas de um determinado produto. Esta classe não fornece *setters* e a única maneira de alterar a informação desta classe é registando uma venda com o método **registaVenda().** Cada instância de ComprasDoProduto permite-nos obter informações relativas às vendas de um produto como:

* Número de unidades compradas desse produto a que diz respeito;
* Total gasto nas compras desse produto;

### Atributos

Na declaração de v.i. de ComprasDoProduto, temos:



, onde:

* **codigoProduto** guarda o código do produto ao qual correspondem os dados das vendas;
* **unidadesCompradas** guarda o número de unidades do produto compradas;
* **faturacao** indica o total gasto nas compras do produto;
  1. ComprasPorCliente

Cada instância de **ComprasPorCliente** guarda informação acerca das compras que um cliente fez, mês a mês. Note-se que esta classe não fornece *setters*, pelo que a única maneira de alterar (adicionar) informação desta classe é registando uma venda com o método **registaVenda()**.

### Atributos

Na declaração de v.i. de ComprasPorCliente, temos:



, onde:

* **codigoCliente** corresponde ao código do cliente para o qual guardamos informação sobre as compras;
* **quantasComprasPorMes** é um array de 13 elementos cujo elemento de índice 0 serve de *padding* e que no índice i, para , contém o número de compras que o cliente fez no mês i;
* **comprasPorMes** é uma lista de 13 Maps, 1 por mês, que a cada produto comprado nesse mês faz corresponder a respetiva instância de ComprasDoProduto. O elemento de índice 0 serve apenas de *padding*, para podermos aceder à informação dos meses pelo número do mês.

### Esquema da estrutura de dados

“NR1091”

“DV1252””

“IT1197”

\*

\*

\*

\*

\*

\*

<”NR1091”, ComprasDoProduto>

…

<”IT1197”, ComprasDoProduto>

<”DV1252”, ComprasDoProduto>

…

(*padding*)

Compras do mês de Janeiro

…

Compras do mês de Dezembro

[0]

[1]

…

[25]

private List<Map<String, ComprasDoProduto>> comprasPorMes; (12 Maps + 1 para *padding*)

private String codigoProduto;

private int[] quantasComprasPorMes;

(*padding*)

Nº compras(Jan)

…

Nº compras (Dez)

Nº compras(Fev)

[0]

[1]

…

[12]

[2]

Map<String, ComprasDoProduto>, com as compras de um mês (implementado com HashMap)

Figura -Esquema da estrutura de dados de ComprasPorCliente

## Filial

Cada instância de **Filial** guarda informações relativas a uma filial e às compras que nela foram feitas ao longo do ano, incluindo dados relativos aos clientes que compraram na filial, nomeadamente: que produtos compraram, em que quantidades, quanto gastaram neles e quantas compras fizeram na filial.

### Atributos



Esta única variável de instância representa uma lista com 26 mapeamentos, um por cada letra do alfabeto. O índice 0 da lista contém o mapeamento que aos clientes cujo código começa por 'A' faz corresponder a informação sobre as suas compras (guardada num objeto da classe ComprasPorCliente – 2.12), o índice 1 contém o Map que aos clientes cujo código começa por 'B' faz corresponder as suas compras e assim por diante.

### Esquema da estrutura de dados utilizada

Clientes cujo código começa por ‘A’

Clientes cujo código começa por ‘B’

…

Clientes cujo código começa por ‘Z’

[0]

[1]

…

[25]

“A2657”

“A3975””

“A1338”

\*

\*

\*

\*

\*

\*

<”A2657”, ComprasPorCliente>

…

<”A1338”, ComprasPorCliente>

<”A3975”, ComprasPorCliente>

…

Map<String, ComprasPorCliente>, implementado com HashMap

(exemplo para clientes cujo código começa por ‘A’)

private List<Map<String, ComprasPorCliente>> clientesOrdenados; (26 Maps)

Figura -Esquema da estrutura de dados de Filial

## Filiais

A classe **Filiais** guarda informação relativa às várias filiais que constituem uma cadeia de hipermercados e permite fazer consultas globais na mesma, i.e.: consultas de informações relativas todas as suas filiais.

### Atributos

Na declaração de v.i. de Filiais temos:



Pela única variável instância facilmente se percebe que um objeto Filiais é, essencialmente, um “conjunto” de objetos da classe Filial (secção 2.13), guardados num *array* de forma a que. ao indice i-1, estejam associados os dados da filial i.

### Esquema da estrutura de dados

Filial 1

Filial 3

Filial 2

[0]

[2]

[1]

private Filial[] filiais;

Figura -Esquema da estrutura de dados utilizada em Filiais

## Pares e triplos

Algumas das consultas interativas da aplicação pedem a devolução de coleções de pares ou até triplos de elementos, como por exemplo:

* *Query* 3 - número de compras realizadas, número de produtos distintos comprados e total gasto por um cliente, mês a mês (triplo);
* *Query* 5 – códigos dos produtos que um cliente mais comprou e quantidade comprada (par);
* *Query* 6 - códigos dos X produtos mais vendidos no ano, quantidade vendida a cada um deles e número de clientes distintos que compraram cada um (triplo);
* *Query* 7 - lista dos 3 maiores compradores, em termos de dinheiro faturado (par);
* *Query* 8 - códigos dos X clientes que mais produtos diferentes compraram e número de produtos distintos que cada um comprou (par);
* *Query* 9 - códigos dos X clientes que mais compraram um produto, quantidade comprada e valor gasto por cada um desses clientes (triplo);

Assim, tornou-se necessário organizar a informação a devolver em cada uma destas *queries*, pelo que optámos por definir as seguintes classes de pares/triplos auxiliares:

* **TriploComprasProdutosGasto** – triplo (int, int, double) auxiliar da *query* 3, que armazena o número de compras, o número produtos distintos comprados e o total gasto por um cliente, num dado mês;
* **ParProdQtd** - par (String, int), auxiliar da *query* 5, que armazena um código de produto e a quantidade desse produto que o cliente selecionado na *query* comprou;
* **TriploProdQtdClis** – triplo (String, int, int), auxiliar da *query* 6, que guarda um código de produto, a quantidade comprada desse produto e o número de clientes distintos que o compraram;
* **ParCliFat** – par (String, double), auxiliar da *query* 7, que guarda um código de cliente e o valor total faturado com esse cliente, numa dada filial;
* **ParCliProdsDif** – par (String, int), auxiliar da *query* 8, que armazena um código de cliente e o número de produtos distintos por ele comprados;
* **TriploCliQtdGasto** – triplo (String, int, double), auxiliar da *query* 9, que armazena um código de cliente, a quantidade que ele comprou de um dado produto e o total gasto no mesmo;

## Excepções

Todas as classes de exceções definidas pelo grupo são subclasses da classe Exception. Na

declaração das classes de exceções criadas neste projeto temos, de uma forma geral:



, onde <nome> é um *placeholder* que assinala o local onde escrevemos a designação da exceção definida.

### ClienteInexistenteException

A exceção **ClienteInexistenteException** é atirada sempre que um método da classe Hipermercado que deveria receber um código de cliente registado, recebe um código que não consta no catálogo de clientes.

### ProdutoInexistenteException

Analogamente a ClienteInexistenteException, a exceção **ProdutoInexistenteException** é atirada sempre que um método de Hipermercado que deveria receber um código de produto registado, recebe um código que não faz parte do catálogo de produtos.

### MesInvalidoException

A exceção **MesInvalidoException** é atirada sempre que um método de Hipermercado que deveria receber um número de mês válido, recebe um número de mês inválido (i.e. não pertencente ao intervalo ).

### FilialInvalidaException

A exceção **FilialInvalidaException** é atirada quando um dado método de Hipermercado que deveria receber um número de filial válido, recebe um número de filial inválido (i.e. : não pertencente ao intervalo ).

## Venda

Cada instância de Venda guarda os dados de uma venda necessários para a adicionar ao hipermercado. Numa instância de venda temos:

* Código do produto vendido;
* Preço unitário do produto vendido;
* Total de unidades vendidas;
* Código do cliente que efetuou a compra;
* Mês em que se sucedeu a venda;
* Filial onde foi registada a venda;

## Hipermercado

### Atributos

A classe **Hipermercado** é a classe agregadora do projeto, na medida em que resulta da composição das principais classes até agora descritas, nomeadamente:

* CatalogoProdutos;
* CatalogoClientes;
* Faturacao;
* Filiais;
* EstatisticasFicheiro;
* EstatisticasGerais;

Assim sendo, na declaração das v.i. de Hipermercado temos:



### Funcionalidades

Os métodos de instância de Hipermercado permitem:

* Registar uma compra;
* Registar um cliente;
* Registar uma venda;
* Verificar se um produto ou cliente faz parte dessa instância;
* Criar as estatísticas do último ficheiro lido;
* Criar as estatísticas gerais;
* Devolver as estatísticas de ficheiro ou as estatísticas gerais;
* Obter os resultados de cada uma das *queries* interativas (ou auxiliar a obtenção dos mesmos) através da invocação dos métodos de Faturacao e Filiais;

Além das funcionalidades referidas, cada instância de Hipermercado tem ainda o método gravaObj() que possibilita o armazenamento de toda a sua estrutura numa *ObjectStream*. Para que tal fosse possível, bastou acrescentarmos **implements Serializable** no cabeçalho da classe de Hipermercado e nos cabeçalhos de cada uma das classes das v.i. de Hipermecado.

A operação de leitura e reconstrução de um Hipermercado previamente guardado com gravaObj() é realizada através da invocação do método de classe leObj(), que recebe o nome do ficheiro do ficheiro de estado a partir do qual se pretende realizar a leitura e, em caso de sucesso, devolve o Hipermercado lido.

## Menu

A classe **Menu** permite a criação de menus genéricos, com título e um número arbitrário de opções.

A construção de um Menu é tipicamente realizada através do construtor parametrizado

public Menu(String titulo, String[] opcoes, boolean temOpcaoSair), onde **título** é uma String com o título do Menu, **opções** é um *array* de Strings com as opções de Menu e **temOpcaoSair** é um valor do tipo *boolean* que indica se o Menu a criar deverá ter opção de sair.

As opções de uma instância de Menu são numeradas desde 1 até à *length* do array de Strings que as armazena e, no caso de o Menu ter a opção “Sair”, essa opção é sempre a opção 0.

Depois de criado um Menu, podemos enviar-lhe a mensagem executa() para este apresentar o seu título, as suas opções e uma *prompt* (“>>>”), de forma a indicar ao utilizador que deverá selecionar uma opção. Sempre que a opção introduzida pelo utilizador é inválida, é apresentada uma mensagem de erro e prossegue-se com a leitura de uma nova opção. Uma vez introduzida uma opção válida, esta pode ser obtida através do envio da mensagem getOpcao() à instância de Menu que leu essa opção.

### Atributos

Na declaração de v.i. de Menu temos:



, onde:

* **titulo** é o título da instância;
* **separador** é uma String de carateres ‘\*’, que é calculada aquando da criação de um Menu, com base no comprimento das opções e do título, e que separa o título da primeira opção e a última opção da *prompt*;
* **opcoes** é um *array* de Strings com as opções da instância;
* **temOpcaoSair** é um valor do tipo *boolean* que indica se um Menu tem opção de sair;
* **op** – é um valor inteiro que representa o número da opção atual (quando nunca foi lida uma opção válida, op tem valor -1);

**Nota**: é possível alterar o título de uma instância de Menu, através do método setTitutlo().

## HipermercadoApp

A classe **HipermercadoApp** é a classe principal da aplicação **GereVendas**. É lá que está definido o método o método main() e que, através da criação de uma instância de HipermercadoApp e do envio da mensagem executa() a essa instância, podemos realizar:

* A leitura dos dados (a partir de ficheiros de texto ou de *ObjectStreams*);
* As consultas estatísticas;
* As consultas interativas;
* A gravação do estado do hipermercado;

**Nota:** como é natural, os 3 últimos pontos só podem ser executados depois de realizada a leitura dos dados.

### Atributos

Na declaração de v.i. de HipermercadoApp, temos:



, onde:

* **hipermercado** é o Hipermercado utilizado numa instância de HipermercadoApp;
* **menuPrincipal** é o menu apresentado quando uma instância de HipermercadoApp inicia a sua execução;
* **menuSair** corresponde ao menu apresentado ao sair da aplicação, sempre que esta se encontra populada, para que o utilizador indique se tenciona salvaguardar o estado da mesma (i.e.: guardar o conteúdo da v.i. hipermercado numa *ObjectStream*);
* **menuLeitura** é o menu apresentado para que o utilizador selecione se pretende ler os dados a partir de ficheiros de texto ou de um ficheiro de estado;
* **menuEstatisticas** é um menu que, sempre que a aplicação está populada, pode ser acedido a partir do menu principal e que permite que o utilizador realize consultas estatísticas;
* **menuQueries** é outro menu que pode ser acedido a partir do menu principal, quando a aplicação está populada, e que possibilita a execução das várias *queries* interativas da aplicação;

**Nota:** o conteúdo dos menus será descrito na secção seguinte.

1. Interface de utilizador e navegação

## Menus

Tal como já foi brevemente referido na descrição dos atributos de HipermercadoApp (secção 2.20.1), a aplicação desenvolvida tem 5 menus:

* Menu principal;
* Menu de leitura de dados;
* Menu de estatísticas;
* Menu das *queries* interativas;
* Menu apresentado à saída (apenas quando a aplicação está populada);

### Menu principal

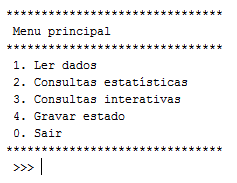


Figura -Menu principal

O menu principal é apresentado quando a aplicação começa a executar e tem as opções:

1. “Ler dados”, que permite aceder ao menu de leitura, de forma a optar entre ler os dados dos ficheiros de texto, ou de uma *ObjectStream*;
2. “Consultas estatísticas”, para aceder ao menu de estatísticas;
3. “Consultas interativas”, para aceder ao menu das *queries* interativas;
4. “Gravar estado”, para guardar o estado da aplicação (i.e.: os conteúdos do hipermercado) numa *ObjectStream*;
5. “Sair” para sair da aplicação;

### Menu de leitura de dados

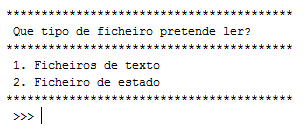


Figura -Menu de leitura de dados

O menu de leitura de dados (Figura 13) é apresentado quando o utilizador seleciona a opção 1. do menu principal (“Ler dados”) e permite que este opte entre ler os dados de ficheiros de texto (opção 1) e carregar um estado anterior (opção 2).

### Menu de estatísticas

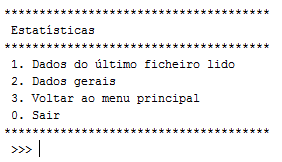


Figura -Menu de estatísticas

O menu de estatísticas (Figura 14) pode ser acedido através da opção 2. do menu principal, quando a aplicação já se encontra populada, e permite a obtenção de informações estatísticas sobre o último ficheiro lido (opção 1), bem como dados gerais relativos aos conteúdos das estruturas das estruturas armazenadas em memória (opção 2).

**Nota:** estando neste menu, o utilizador também pode sair diretamente da aplicação (opção 0) ou por voltar ao menu principal (opção 3).

### Menu das *queries* interativas

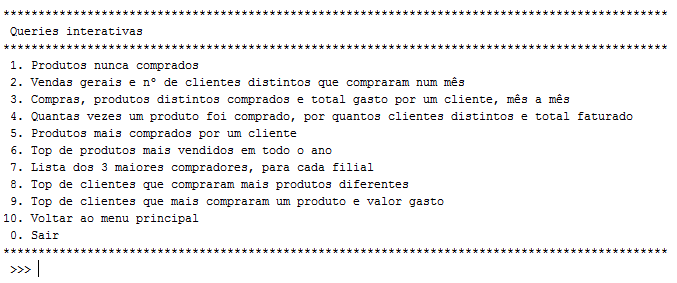


Figura -Menu das queries interativas

O menu das *queries* interativas (Figura 15) pode ser acedido através da opção 3. do menu principal, quando a aplicação já se encontra populada. Estando neste menu, o utilizador pode selecionar qualquer uma das 9 *queries* interativas disponibilizadas pela aplicação.

### Menu apresentado à saída

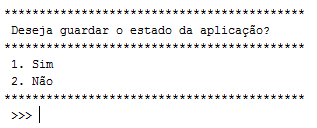


Figura -Menu apresentado à saída

Quando o utilizador opta por sair e a aplicação está populada, é-lhe apresentado o menu que pode ser observado na Figura 16, para que possa decidir se pretende guardar o estado da aplicação antes de sair.

## Navegação

Em todas as *queries* que produzem ou podem produzir grandes volumes de dados (*queries* 1, 5, 6, 8 e 9) optámos por apresentar 20 linhas de cada vez, seguidas do menu da navegação que pode ser observado na Figura 17:



Figura -Opções da navegação

Esta interface da navegação é semelhante à apresentada no projeto de C, contudo acrescentámos (internamente) a possibilidade de passar um cabeçalho (String) para o método **navega()** de HipermercadoApp, para que este seja apresentado antes do início de cada página. Podemos observar um cabeçalho, por exemplo, nos resultados da *query* 5, tal como ilustrado na Figura 18:

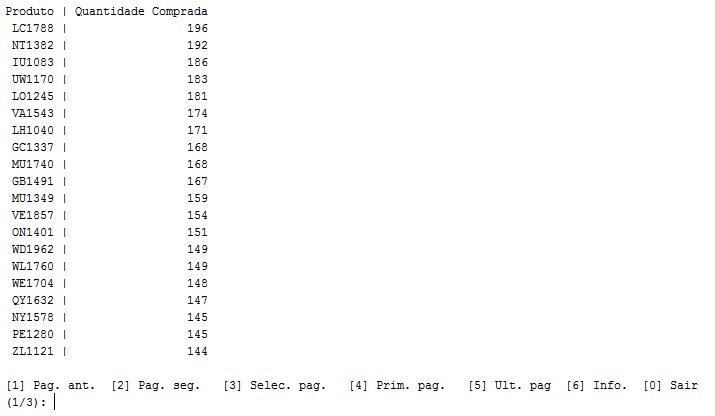


Figura -Exemplo de cabeçalho apresentado na navegação

# 3. Medidas de *performance*

## 3.1 Análise de tempos de leitura: Scanner vs BufferedReader

Medindo os tempos de leitura (com e sem *parsing*) dos ficheiros de 1M, 3M e 5M, com a classe Scanner(+ FileReader) e em seguida com BufferedReader, chegámos aos seguintes resultados:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Ficheiro | | |
| Vendas\_1M.txt | Vendas\_3M.txt | Vendas\_5M.txt |
| Leitura sem *Parsing* (Scanner e FileReader) | 1,43 | 5,66 | 9,61 |
| Leitura com *Parsing* (Scanner e FileReader) | 2,39 | 7,47 | 11,9 |
| Leitura sem *Parsing* (BufferedReader) | 0,12 | 0,27 | 2,67 |
| Leitura com *Parsing* (BufferedReader) | 0,59 | 3,6 | 5,21 |

Tabela -Análise dos tempos de leitura: Scanner vs. BufferedReader

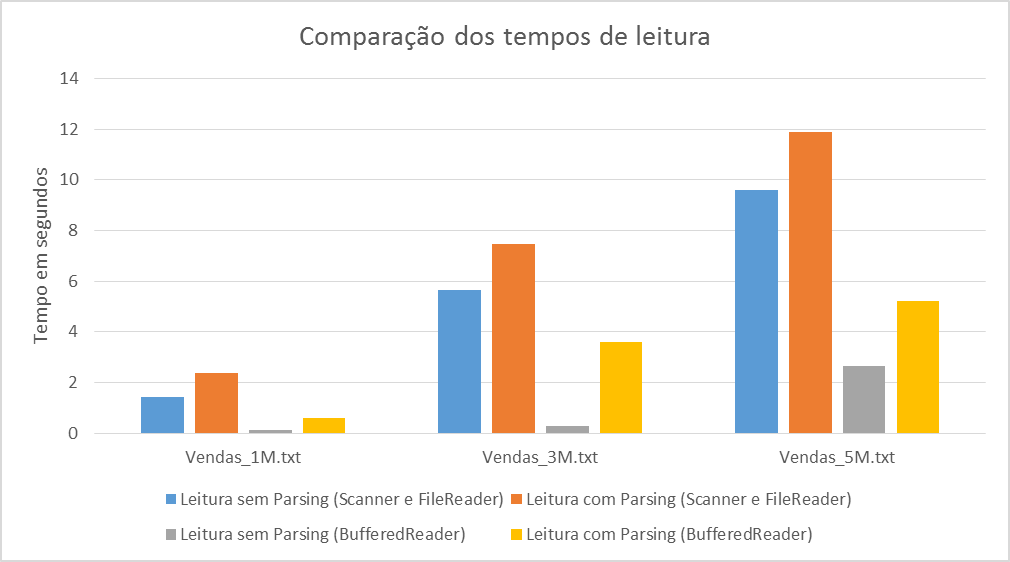


Figura -Gráfico de comparação dos tempos de leitura: Scanner (+FileReader) vs. BufferedReader

Pela análise dos resultados apresentados na Tabela 1 e na Figura 19, rapidamente verificamos que existe uma diferença abismal entre os tempos de leitura conseguidos com a classe BufferedReader, relativamente aos resultados obtidos com uma combinação de Scanner com FileReader. Esta diferença bastante significativa, verificada para todos os ficheiros de vendas e para a leitura com e sem *parsing*, traduz-se nos ganhos que podem ser observados na Tabela 2:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Ficheiro | | |
| Vendas\_1M.txt | Vendas\_3M.txt | Vendas\_5M.txt |
| Ganho da Leitura sem *Parsing* | 11,92 | 20,96 | 3,60 |
| Ganho da Leitura com *Parsing* | 4,05 | 2,08 | 2,28 |

Tabela -Ganhos conseguidos nas leituras com BufferedReader, comparativamente com Scanner (+FileReader)

Estes ganhos foram calculados dividindo, para cada um dos casos testados, o tempo de execução obtido com a classe Scanner pelo tempo de execução conseguido com a classe BufferedReader, pelo que indicam quantas vezes mais rápida a leitura com BufferedReader é, relativamente à leitura com a classe Scanner.

É de salientar que no caso da leitura sem *parsing*, os ganhos para o ficheiro Vendas\_1M.txt e Vendas\_3M.txt estão na ordem das dezenas! No caso do ficheiro Vendas\_5M.txt, esperávamos que o ganho fosse muito maior, mais próximo dos ganhos que obtivemos com os outros ficheiros, mas tal não se verificou empiricamente.

Já nas leituras com *parsing*, observámos que os ganhos são muito menores, o que, a nosso ver, é justificado por as operações de *parsing* das linhas terem um peso mais significativo do que o da leitura das linhas de venda. Assim, os ganhos são menores, já que as operações de leitura têm um peso relativo (ao resto do programa) menor.

## Análise de desempenho das diferentes coleções

Para testarmos a *performance* das diferentes estruturas de dados, decidimos criar 6 versões do trabalho final, em que todo o código é idêntico ao original, exceto **uma** alteração por versão:

* Numa das versões, todas as listas são implementadas com **ArrayList**’s (o código é o original).
* Uma das versões implementa todos os elementos do tipo **List** com instâncias da classe **Vector**.
* Outra versão apenas usa **Map**’s implementados com **HashMap**’s.
* Noutra versão, todos os **Map**’s são **TreeMap**’s.
* Uma versão tem todos os seus **Set**’s implementados com instâncias de **TreeSet**.
* Outra versão cujos **Set**’s são implementados unicamente com **HashSets**.

De forma a que os testes fossem o mais uniformes possíveis, definimos que todas as *queries* que necessitassem do código de um cliente iriam ser executadas para o código Z5000, que sempre que fosse necessário um argumento numérico, este seria 15 e sempre que fosse necessário introduzir o código de um produto, seria introduzido o código NR1091.

Dadas estas condições de teste, obtivemos os resultados que passaremos a apresentar nas seguintes páginas (todos os tempos apresentados estão em segundos).

### Query 5:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | Query 5 | | |
| Vendas\_1M | Vendas\_3M | Vendas\_5M |
| List | ArrayList | 0,006 | 0,006 | 0,006 |
| Vector | 0,031 | 0,016 | 0,011 |
| Map | Hashmap | 0,045 | 0,013 | 0,010 |
| TreeMap | 0,029 | 0,013 | 0,009 |
| Set | HashSet | 0,046 | 0,012 | 0,008 |
| TreeSet | 0,028 | 0,007 | 0,005 |

Tabela -Performance da query 5 com diferentes coleções

Figura -Gráfico de performance da query5, para diferentes coleções

Observando a tabela e gráfico apresentados, constatamos que:

* A implementação das Lists com ArrayList proporcionou um melhor desempenho do que a versão em que utilizamos a classe Vector;
* O desempenho da *query* 5 foi, na maioria das implementações, pior com o ficheiro Vendas\_1M do que com os restantes ficheiros. Este resultado vai contra a nossa intuição, contudo, ao fim de sucessivas medições chegámos sempre a resultados idênticos aos que aqui apresentamos. Embora não consigamos precisar o que possa ter levado a que a *performance* obtida com o ficheiro Vendas\_1M fosse em geral pior do que a obtida com os restantes ficheiros, é possível que a disposição dos dados nas estruturas tenha sido tal, que no caso de Vendas\_3M e Vendas\_5M o acesso ao cliente Z5000 acabou por ser mais rápido/eficiente do que para Vendas\_1M;

### Query 6:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | Query 6 | | |
| Vendas\_1M | Vendas\_3M | Vendas\_5M |
| List | ArrayList | 0,952 | 1,179 | 3,647 |
| Vector | 1,877 | 1,341 | 1,583 |
| Map | Hashmap | 1,041 | 3,265 | 3,981 |
| TreeMap | 1,928 | 3,242 | 2,821 |
| Set | HashSet | 0,929 | 2,824 | 1,590 |
| TreeSet | 2,004 | 1,354 | 1,740 |

Tabela - Performance da query 6 com diferentes coleções

Figura -Gráfico de performance da query 6, para diferentes coleções

Pela observação dos resultados obtidos, constatamos que:

* A implementação das Lists com ArrayList trouxe uma melhor *performance* do que a implementação com Vector no caso de Vendas\_1M, contudo, no caso de Vendas\_5M obtivemos um tempo bastante melhor com Vector do que com ArrayList;
* A *performance* do HashMap superou a do TreeMap no caso de Vendas\_1M e foi inferior para Vendas\_5M;
* A *performance* do HashSet foi melhor do que a do TreeSet para Venda\_1M, mas foi bastante pior no caso de Vendas\_3M;

### Query 7:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | Query 7 | | |
| Vendas\_1M | Vendas\_3M | Vendas\_5M |
| List | ArrayList | 0,077 | 0,201 | 0,293 |
| Vector | 0,172 | 0,201 | 0,319 |
| Map | Hashmap | 0,110 | 0,186 | 0,302 |
| TreeMap | 0,084 | 0,186 | 0,389 |
| Set | HashSet | 0,111 | 0,203 | 0,286 |
| TreeSet | 0,116 | 0,183 | 0,394 |

Tabela -Performance da query 7 com diferentes coleções

Figura - Gráfico de performance da query 7, para diferentes coleções

Analisando os resultados apresentados, concluímos que:

* A implementação das Lists com ArrayList levou, no caso de Vendas\_1M, a uma melhor *performance* da *query* 7, do que a implementação com Vector;
* Para o ficheiro Vendas\_3M, a *performance* das várias implementações é praticamente igual;
* No caso de Vendas\_5M as implementações baseadas em árvores tiveram um pior desempenho do que as restantes;
* Em qualquer uma das implementações, o tempo de execução da *query* subiu com o aumento do tamanho do ficheiro (tal como seria de esperar);

### Query 8:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | Query 8 | | |
| Vendas\_1M | Vendas\_3M | Vendas\_5M |
| List | ArrayList | 0,475 | 1,454 | 2,510 |
| Vector | 0,482 | 1,397 | 5,588 |
| Map | Hashmap | 0,529 | 1,385 | 2,477 |
| TreeMap | 0,452 | 1,298 | 2,251 |
| Set | HashSet | 0,350 | 0,832 | 4,024 |
| TreeSet | 0,507 | 1,340 | 2,522 |

Tabela -Performance da query 8 com diferentes coleções

Figura - Gráfico de performance da query 8, para diferentes coleções

Pela análise da tabela e gráfico apresentados, observamos que:

* Para os ficheiros Vendas\_1M, o desempenho das várias implementações é praticamente o mesmo;
* No caso de Vendas\_3M, a implementação Sets com HashSets conduziu a um melhor desempenho do que com TreeSets. Contudo, passando para o ficheiro Vendas\_5M já obtivemos um melhor desempenho com TreeSets, do que com HashSets;
* Para o ficheiro Vendas\_5M, a implementação de Lists com Vectors levou a um tempo de execução bastante elevado, o que nos leva a concluir que para esta *query* a utilização de ArrayLists como implementação de List é uma melhor opção;

### Query 9:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | Query 9 | | |
| Vendas\_1M | Vendas\_3M | Vendas\_5M |
| List | ArrayList | 0,067 | 0,082 | 0,110 |
| Vector | 0,109 | 0,091 | 0,123 |
| Map | Hashmap | 0,091 | 0,086 | 0,138 |
| TreeMap | 0,120 | 0,159 | 0,232 |
| Set | HashSet | 0,099 | 0,087 | 0,093 |
| TreeSet | 0,103 | 0,084 | 0,100 |

Tabela - Performance da query 9 com diferentes coleções

Figura -Gráfico de performance da query 9, para diferentes coleções

Os resultados das medições realizadas nesta *query*, levaram-nos a concluir que:

* Em qualquer um dos ficheiros, a implementação dos maps com TreeMap leva a um pior desempenho do que a implementação com TreeSets, o que faz do TreeSet uma melhor opção para esta *query* em particular;
* A implementação das Lists com ArrayList levou, no caso de Vendas\_1M, a um desempenho claramente melhor do que a implementação com Vectors;
* Ao nível das 2 implementações da interface Set não existem diferenças significativas de *performance*, pelo que qualquer uma se adequa à *query* 9;

## Outros dados de execução para o ficheiro Vendas\_3M.txt

Abaixo apresentam-se dados obtidos com a ferramenta *Java VisualVirtualMachine*, que mostram o uso da *heap* e do CPU ao longo do tempo, para o ficheiro Vendas\_3M.txt, quando realizamos a leitura dos dados e em seguida executamos as *queries* de 5 a 9, nas diferentes implementações apresentadas na secção anterior:

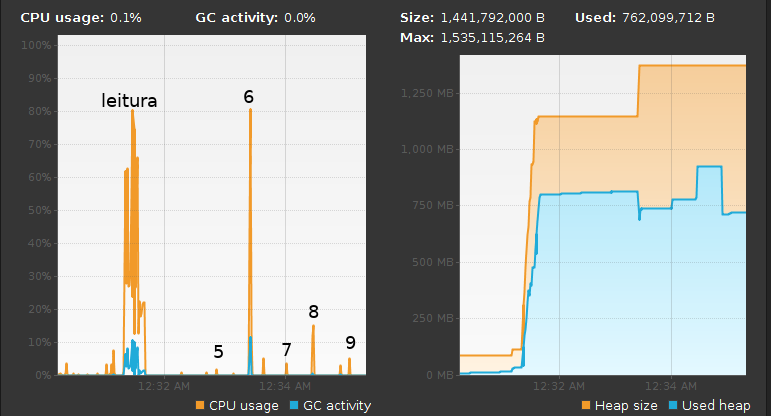


Figura – Utilização do CPU e da heap ao implementar Lists com ArrayList (Vendas\_3M)

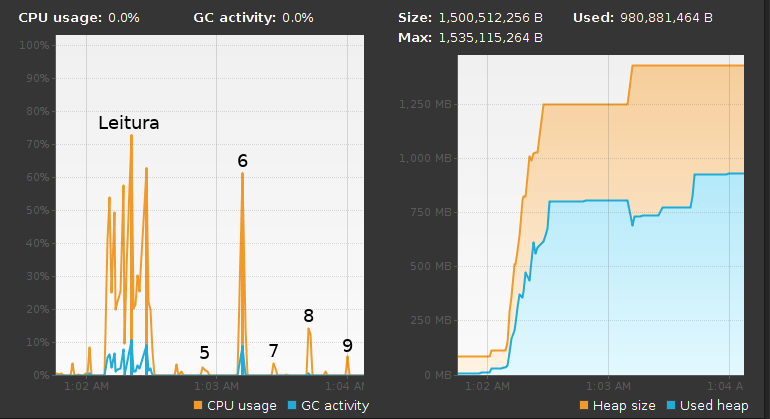


Figura - Utilização do CPU e da heap ao implementar Lists com Vector (Vendas\_3M)

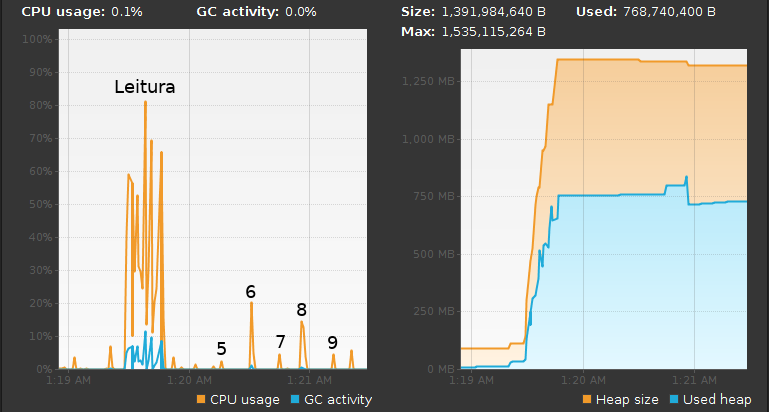


Figura - Utilização do CPU e da heap ao implementar Maps com HashMap (Vendas\_3M)

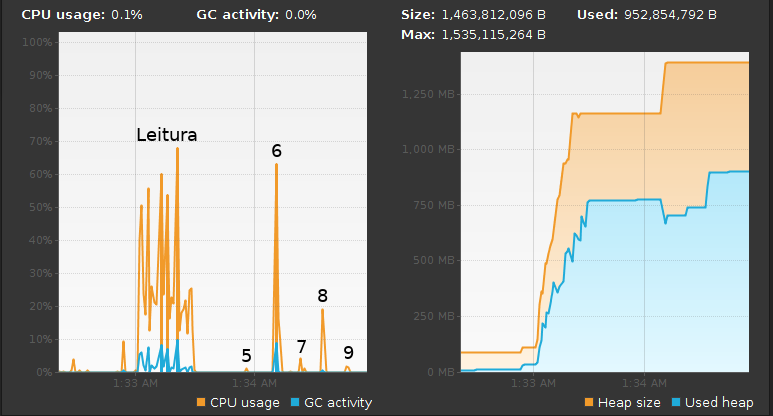


Figura - Utilização do CPU e da heap ao implementar Maps com TreeMap (Vendas\_3M)

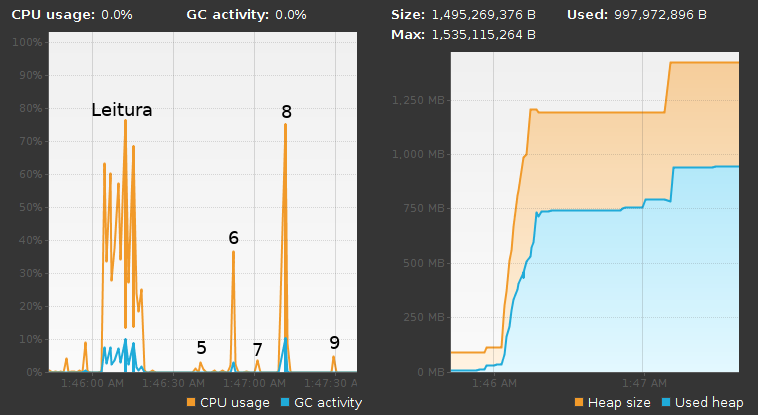


Figura - Utilização do CPU e da heap ao implementar Sets com HashSet (Vendas\_3M)

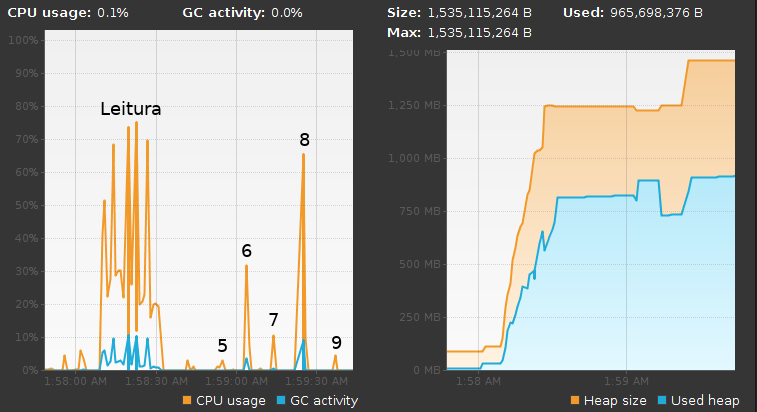


Figura - Utilização do CPU e da heap ao implementar Sets com TreeSet (Vendas\_3M)

# 4. Conclusão

Sendo este trabalho a continuação de um trabalho desenvolvido em C, é impossível deixar de comentar as diferenças que notamos ao desenvolver nas duas linguagens.

Ao longo do desenvolvimento deste projeto, constatámos que é muito mais fácil desenvolver e manter um programa desta escala em Java do que em C, principalmente devido à grande quantidade de bibliotecas “padrão” disponíveis no JSE e ao paradigma de POO, que nos permite estruturar mais facilmente o programa e conceber código mais simples e percetível.

Outra das razões para acharmos que este trabalho foi mais interessante e para termos gostado de trabalhar mais em Java foi o facto de que pudemos testar várias implementações para o mesmo tipo abstrato de dados, modificando muito pouco código, graças à declaração das variáveis e dos métodos usando interfaces e não classes concretas. Deste modo, obtivemos resultados palpáveis para as diferentes implementações, o que não se deu em C.

Utilizámos, sempre que pudemos, construções disponíveis em Java 8, nomeadamente as operações em *Streams* e expressões lambda, de forma a ganharmos experiência no uso das mesmas. A nosso ver, estas adições são de grande utilidade, uma vez que nos permitem escrever, normalmente com menos código, operações a realizar sobre um “conjunto” de dados num estilo composicional/funcional.

O desempenho da aplicação foi uma surpresa muito agradável. Inicialmente, pensámos que a nossa aplicação iria sofrer imenso porque o código Java compilado em *Bytecode* corre numa *VirtualMachine*, o que poderia ser um fator de grande *overhead* mas, nos nossos testes, os tempos de leitura foram significativamente melhores em Java do que em C (que é uma linguagem muito mais “próxima do metal”).

Houve também uma grande diferença no tipo de *bugs* que encontramos (e corrigimos) nos dois projetos. Enquanto que o trabalho de C foi marcado por *bugs* extremamente subtis, relacionados com pormenores de baixo nível, que nos levaram a uma experiência verdadeiramente penosa e a horas de *debugging* a observar *stack frames*, os *bugs* em Java foram bastante mais fáceis de encontrar e corrigir.

O facto de termos a memória gerida pelo ambiente de *runtime* (obviamente não deixamos de adotar as melhores práticas para facilitar o trabalho do *garbage collector*) permitiu-nos focar mais no “sumo” da aplicação, i.e. no fim que pretendíamos que ela atingisse e não tanto nos detalhes mais “chatos” da implementação.

Mais uma vez, fizemos sempre por manter o encapsulamento de forma a preservar a pureza da aplicação e evitar a partilha de dados entre as diferentes instâncias das classes do programa. Todas as classes implementam o método clone() e sempre que um método precisa de devolver uma v.i. mutável e privada a um objeto, é devolvida uma cópia.

Em suma, foi um trabalho muito enriquecedor porque ganhámos uma nova visão e abertura para as vantagens da programação em linguagens de alto nível, principalmente no que toca a projetos de média e grande escala.