|  |  |
| --- | --- |
| 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. Arquitetura de classes 3](#_Toc453616858)

[1.1 Constantes 4](#_Toc453616859)

[1.2 LStrings 4](#_Toc453616860)

[1.3 EstatisticasFicheiro 5](#_Toc453616861)

[1.4 EstatisticasGerais 6](#_Toc453616862)

[1.5 CatalogoProdutos 8](#_Toc453616863)

[1.6 CatalogoClientes 9](#_Toc453616864)

[1.7 FatProdMes 10](#_Toc453616865)

[1.8 FatMes 11](#_Toc453616866)

[1.9 FatAnualProd 13](#_Toc453616867)

[1.10 Faturacao 14](#_Toc453616868)

[1.11 ComprasDoProduto 16](#_Toc453616869)

[1.13 Filial 17](#_Toc453616870)

[1.14 Filiais 18](#_Toc453616871)

[1.15 Pares e triplos 18](#_Toc453616872)

[1.16 Excepções 19](#_Toc453616873)

[1.17 Venda 20](#_Toc453616874)

[1.18 Hipermercado 20](#_Toc453616875)

[1.19 Menu 21](#_Toc453616876)

[1.20 HipermercadoApp 22](#_Toc453616877)

[2. Interface de utilizador e navegação 23](#_Toc453616878)

[2.1 Menus 23](#_Toc453616879)

[2.2 Navegação 25](#_Toc453616880)

[3. Medidas de *performance* 26](#_Toc453616881)

[Figura 1-Arquitetura de classes 3](#_Toc453608532)

[Figura 2-Esquema da estrutura de dados de EstatisticasGerais 8](#_Toc453608533)

[Figura 3-Esquema de estrutura de dados de CatalogoProdutos 9](#_Toc453608534)

[Figura 4-Esquema da estrutura de dados de CatalogoClientes 10](#_Toc453608535)

[Figura 5-Esquema da estrutura de dados de FatProdMes 11](#_Toc453608536)

[Figura 6-Estrutura de dados de FatMes 13](#_Toc453608537)

[Figura 7-Esquema da estrutura de dados de FatAnualProd 14](#_Toc453608538)

[Figura 8-Esquema de estrutura de dados da faturação 16](#_Toc453608539)

[Figura 9-Menu principal 23](#_Toc453608540)

[Figura 10-Menu de leitura de dados 23](#_Toc453608541)

[Figura 11-Menu de estatísticas 24](#_Toc453608542)

[Figura 12-Menu das queries interativas 24](#_Toc453608543)

[Figura 13-Menu apresentado à saída 25](#_Toc453608544)

[Figura 14-Opções da navegação 25](#_Toc453608545)

[Figura 15-Exemplo de cabeçalho apresentado na navegação 25](#_Toc453608546)

# 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 (e que irão ser discutidas mais adiante), 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:

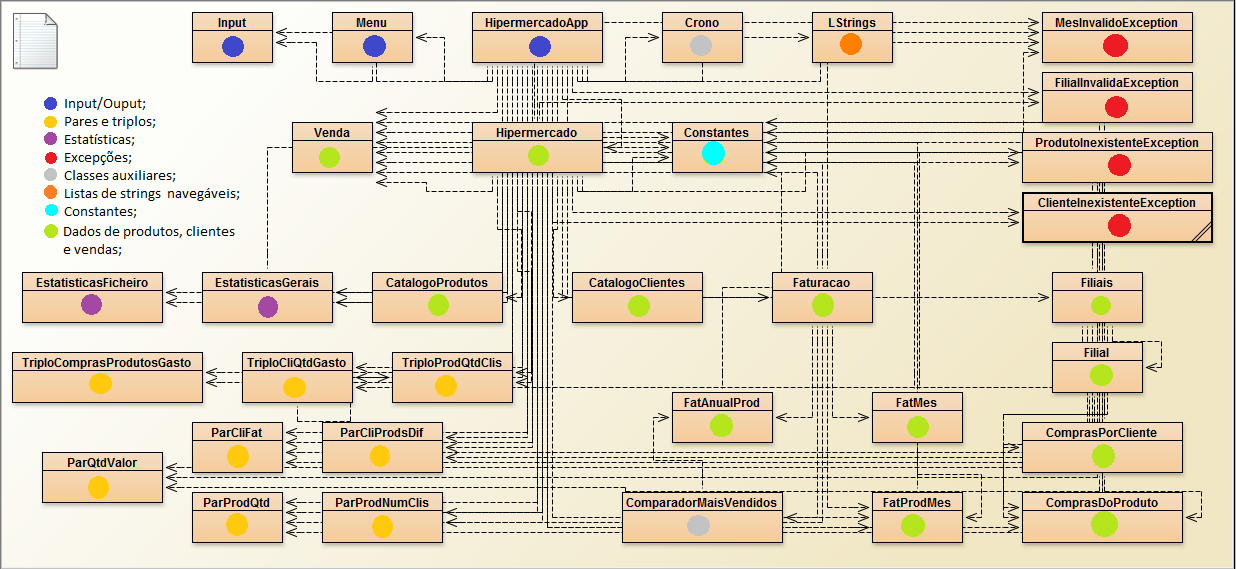


Figura 1-Arquitetura de classes

## 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 necessário 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" \o "class or interface in java.util)<[String](http://download.oracle.com/javase/8/docs/api/java/lang/String.html?is-external=true" \o "class or interface in java.lang)> 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 todos 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 e 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 facturado;

**Nota:** não é necessário guardar os totais de clientes que não realizaram compras e de produtos não comprados, uma vez que estes 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 2-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 1.18.1) 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 tencionarmos utilizar.

**Nota:** de referir que utilizamos 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 3-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 existe nesse catálogo.

No caso da aplicação desenvolvida, utilizámos uma instância de CatalogoClientes nas v.i. de Hipermercado (secção 1.18.1), 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 4-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 *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 a quem os dados guardados na instância dizem respeito;
* **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 double[] faturacao;

(*padding*)

Faturado na filial 1

Faturado na filial 3

Faturado na filial 2

[0]

[1]

[3]

[2]

private final int mes;

private String codigoProduto;

Figura 5-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 verificamos 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 1.7) 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 1.10 - 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 6-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 1.10.1), 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 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 7-Esquema da estrutura de dados de FatAnualProd

## Faturacao

Cada instância de **Faturacao** guarda informação relativa às vendas mensais e do ano todo, filial a filial e globalmente. Na parte das 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, 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 e 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 8-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;
* Total gasto na compra do 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 na compra 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 em que o elemento de índice 0 serve de *padding* e 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 o respetivo 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

## 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 – secção 1.12), 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

## Filiais

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

### Atributos

Na declaração de v.i. de Filiais temos:



Pela única variável instância se percebe que um objeto Filiais é, essencialmente, um “conjunto” de objetos da classe Filial (secção 1.13), guardados num *array* de forma a que. ao indice i-1, estejam associados os dados da filial I.

### Esquema da estrutura de dados

## 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, produtos distintos comprados e total gasto por um cliente, em cada 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, a quantidade vendida para cada um deles e o número de clientes distintos que o compraram (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 o número de produtos distintos que cada um comprou (par);
* *Query* 9 - códigos dos X clientes que mais compraram um produto, a quantidade que cada um comprou e o valor gasto por cada um deles (triplo);

De forma organizar a informação a devolver em cada uma destas *queries*, 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 comprada por um cliente;
* **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 diferentes 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 definidas 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 referidas, nomeadamente:

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

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



### Functionalidades

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 a criar, **opções** é um *array* de Strings com as opções de Menu a criar 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 opção 0.

Uma vez 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 (antes de ter sido lido uma opção válida, op tem valor -1);

**Nota**: também 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 que é apresentado quando uma instância de HipermercadoApp inicia a sua execução;
* **menuSair** corresponde ao menu apresentado ao sair da aplicação, sempre que a aplicação está populada, para que o utilizador indique se tenciona salvaguardar o estado da mesma (i.e.: guardar o conteúdo do Hipermercado numa *ObjectStream*);
* **menuLeitura** é o menu apresentado para que o utilizador selecione se pretende ler os dados de ficheiros de texto ou de um ficheiro de estado;
* **menuEstatisticas** é um menu que pode ser acedido a partir do menu principal, sempre que a aplicação está populada, 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 com detalhe na secção seguinte.

1. Interface de utilizador e navegação

## Menus

Tal como já foi brevemente referido na descrição dos atributos de HipermercadoApp (secção 1.20.1), a aplicação desenvolvida tem 5 menus:

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

### Menu principal

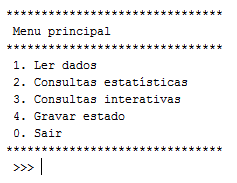


Figura 9-Menu principal

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”, que permite guardar o estado da aplicação (i.e.: os conteúdos do hipermercado) numa *ObjectStream*;
5. “Sair” para sair da aplicação;

### Menu de leitura de dados

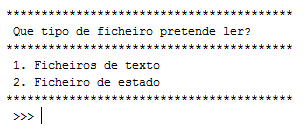


Figura 10-Menu de leitura de dados

O menu de leitura de dados (Figura 10) 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 ler um estado anterior (opção 2).

### Menu de estatísticas

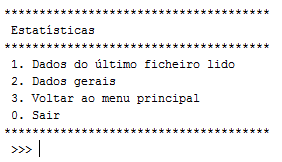


Figura 11-Menu de estatísticas

O menu de estatísticas (Figura 11) pode ser acedido através da opção 2. do menu principal, quando a aplicação já se encontra populada, e permite a obtenção 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

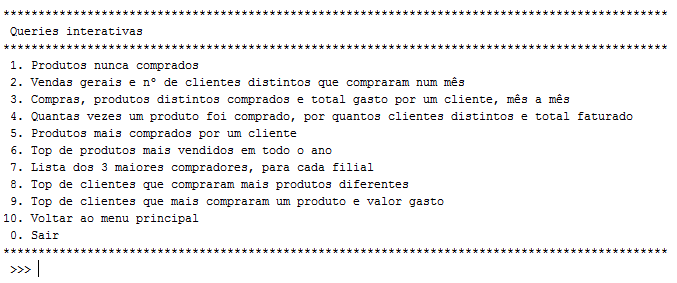


Figura 12-Menu das queries interativas

O menu das *queries* interativas (Figura 12) 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 de 9 *queries* interativas da aplicação desenvolvida.

### Menu apresentado à saída

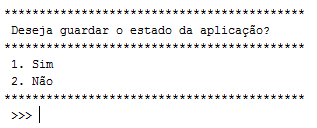


Figura 13-Menu apresentado à saída

Quando o utilizador opta por sair e a aplicação está populada, é-lhe apresentado o menu que pode ser observado na Figura 13, 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 14:



Figura 14-Opções da navegação

Como se pode observar, a interface da navegação é semelhante à apresentada no projeto de C, contudo acrescentámos 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 pela Figura 15:

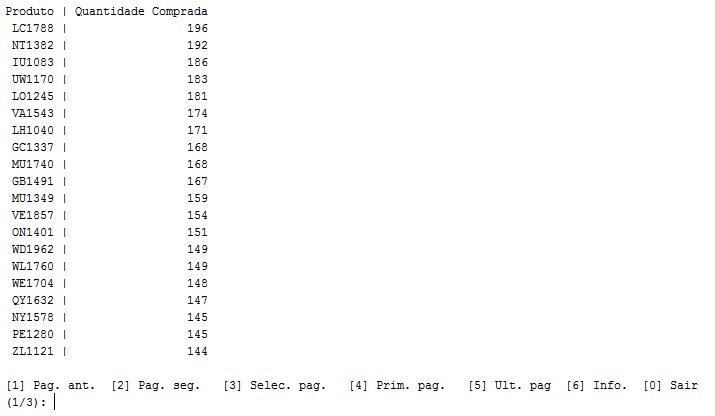


Figura 15-Exemplo de cabeçalho apresentado na navegação

# 3. Medidas de *performance*