|  |  |
| --- | --- |
| 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](#_Toc453586201)

[1.1 Constantes 4](#_Toc453586202)

[1.2 LStrings 4](#_Toc453586203)

[1.2.1 Atributos 4](#_Toc453586204)

[1.2.2 Variáveis de classe 5](#_Toc453586205)

[1.3 EstatisticasFicheiro 5](#_Toc453586206)

[1.3.1 Atributos 5](#_Toc453586207)

[1.4 EstatísticasGerais 6](#_Toc453586208)

[1.4.1 Atributos 6](#_Toc453586209)

[1.4.2 Esquema da estrutura de dados 6](#_Toc453586210)

[1.5 CatalogoProdutos 7](#_Toc453586211)

[1.5.1 Atributos 7](#_Toc453586212)

[1.5.2 Esquema da estrutura de dados 8](#_Toc453586213)

[1.6 CatalogoClientes 8](#_Toc453586214)

[1.6.1 Atributos 8](#_Toc453586215)

[1.6.2 Esquema da estrutura de dados 9](#_Toc453586216)

[1.7 Faturacao 9](#_Toc453586217)

[1.7.1 Atributos 10](#_Toc453586218)

[1.7.2 Esquema das estruturas de dados 11](#_Toc453586219)

[1.8 FatMes 11](#_Toc453586220)

[1.8.1 Atributos 12](#_Toc453586221)

[1.8.2 Esquema das estruturas de dados 13](#_Toc453586222)

[1.9 FatProdMes 13](#_Toc453586223)

[1.9.1 Atributos 13](#_Toc453586224)

[1.9.2 Esquema da estrutura de dados 14](#_Toc453586225)

[1.10 FatAnualProd 14](#_Toc453586226)

[1.10.1 Atributos 15](#_Toc453586227)

[1.10.2 Esquema da estrutura de dados 15](#_Toc453586228)

[1.11 Filiais 15](#_Toc453586229)

[1.11.1 Atributos 15](#_Toc453586230)

[1.11.2 Esquema da estrutura de dados 15](#_Toc453586231)

[1.12 Filial 16](#_Toc453586232)

[1.12.1 Atributos 16](#_Toc453586233)

[1.12.2 Esquema da estrutura de dados utilizada 16](#_Toc453586234)

[1.13.1 Atributos 16](#_Toc453586235)

[1.13.2 Esquema da estrutura de dados 16](#_Toc453586236)

[1.14 ComprasDoProduto 17](#_Toc453586237)

[1.14.1 Atributos 17](#_Toc453586238)

[1.15 Pares e triplos 17](#_Toc453586239)

[1.16 Excepções 18](#_Toc453586240)

[1.16.1 ClienteInexistenteException 18](#_Toc453586241)

[1.16.2 ProdutoInexistenteException 18](#_Toc453586242)

[1.16.3 MesInvalidoException 18](#_Toc453586243)

[1.16.4 FilialInvalidaException 18](#_Toc453586244)

[1.17 Venda 19](#_Toc453586245)

[1.18 Hipermercado 19](#_Toc453586246)

[1.18.1 Atributos 19](#_Toc453586247)

[1.18.2 Functionalidades 19](#_Toc453586248)

[1.19 Menu 20](#_Toc453586249)

# 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:

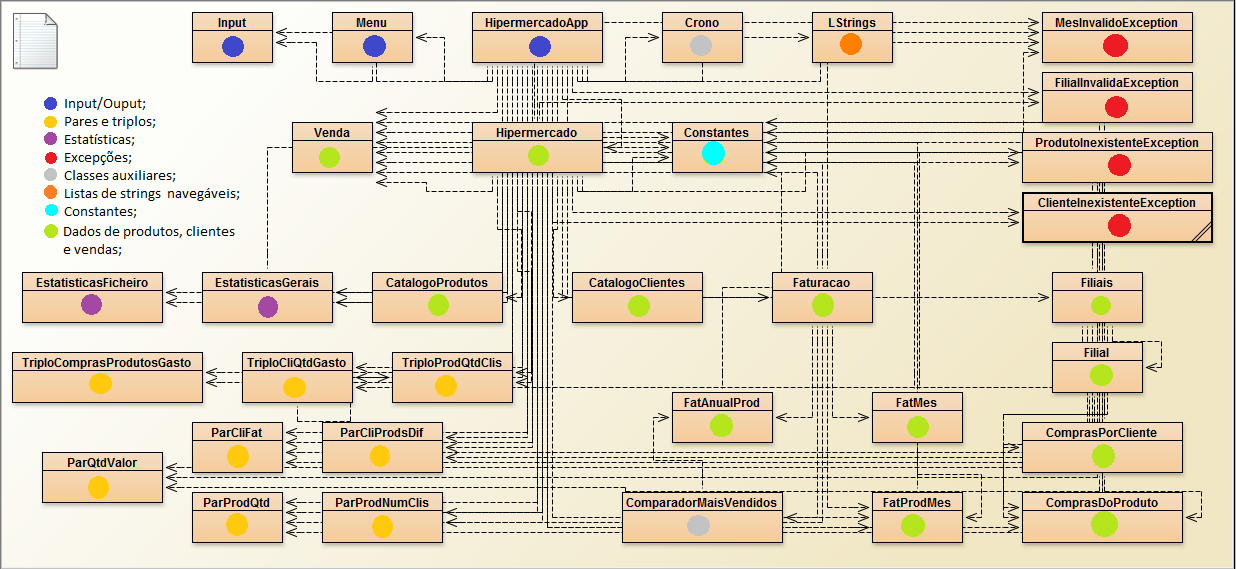


Figura -Arquitetura de classes

## Constantes

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



Estas variáveis 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 navegáveis, com noção de página, e permite a navegação em grandes volumes de dados (na forma de Strings). O número de Strings por página de uma LStrings é, por omissão, igual a 20. No entanto, se o utilizador da API de LStrings pretender criar listas de Strings com um número diferente de Strings por página, tem um construtor que lhe permite fazê-lo, cuja assinatura é:

**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 operações, 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, uma LStrings também pode 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 obtido, através da multiplicação .

### Variáveis de classe

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



## EstatisticasFicheiro

Cada instância da classe **EstatisticasFicheiro** contém várias 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.

No caso da 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**;

É também importante referir 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.

## EstatísticasGerais

Cada instância de **EstatisticasGerais** guarda dados informações gerais respeitantes aos dados atuais já 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 e por só as criarmos quando o utilizador tenta consultá-las pela 1ª vez.

Não disponibilizámos *setters* para garantir que as instâncias de EstatisticasGerais são imutáveis. A razão dessa 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 em 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

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, utilizamos uma instância de CatalogoProdutos nas v.i. de Hipermercado para registar os códigos de todos os produtos disponíveis, obter o total de produtos existentes no hipermercado e verificar a existência dos códigos de produto que surgem nas linhas do ficheiro de vendas e dos 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 armazenar os códigos dos produtos de cada instância de CatalogoProdutos. Contudo, para podermos mudar facilmente a implementação para outro tipo de Set, declaramos a única v.i. de CatalogoProdutos da seguinte forma:



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 pretendamos 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 e um **Set** permite-nos precisamente alcançar esse fim.

### Esquema da estrutura de dados

…

“NR1091”

“AF1184”

“DV1252”

\*

\*

\*

“AF1184”

\*

\*

…

“DV1252”

\*

“NR1091”

Figura -HashSet (esquema da estrutura de dados utilizada no catálogo de produtos)

## 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, utilizamos uma instância de CatalogoClientes nas v.i. de Hipermercado, 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 -HashSet (esquema da estrutura de dados do catálogo de clientes)

## 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 uma das compras 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 (a classe FatMes encontra-se descrita na secção 1.5);
* **todosProdutos** mapeia códigos de produtos em instâncias de FatAnualProd com informação relativa à faturação anual do produto utilizado como chave e pesquisa (a classe FatAnualProd encontra-se descrita na secção 1.7);

### Esquema das estruturas de dados

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

private final int nfiliais;

private FatMes[] fatMensal;

(*padding*)

FatMes (Jan)

…

FatMes (Fev)

FatMes (Dez)

[0]

[1]

…

[2]

[12]

…

<”NR1076”, FatAnualProd>

<”AF1182”, FatAnualProd>

<”QQ1149”, FatAnualProd>

…

…

…

Figura -Esquema de estrutura de dados da faturação

## 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, optámos por 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, em comparação com uma implementação que utilize um TreeMap.

**Nota:** O número de filiais (**nfiliais**) é guardado para que, aquando da construção da faturação de um produto num mês (FatProdMes – secção 1.6) 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[] ), noutro contexto poderíamos não saber. Assim, ao guardarmos o mês numa v.i. de FatMes garantimos 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 utilizada em FatMes

## FatProdMes

A classe **FatProdMes** representa a faturação de um produto num mês. Cada instância de FatProdMes 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;

### 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 contexto da nossa aplicação não fosse necessário guardar o mês nas v.i. de FatProdMes, para garantirmos que a classe FatProdMes é independente do contexto 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 -Esquema da estrutura de dados utilizada em FatProdMes

## 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.

No caso da aplicação desenvolvida, utilizamos instâncias de FatAnualProd no Map<String, FatAnualProd> todosProdutos das v.i. de Faturacao (secção 1.4.1), para guardar a faturação anual de todos os produtos, incluindo a dos não vendidos. 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 -Esquema da estrutura de dados de FatAnualProd

## 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.

### 1.11.1 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 ), guardados num *array* de forma a que. ao indice i-1, estejam associados os dados da filial I.

### 1.11.2 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.

### 1.12.1 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 ), o índice 1 contém o Map que aos clientes cujo código começa por 'B' faz corresponder as suas compras e assim por diante.

### 1.12.2 Esquema da estrutura de dados utilizada

* 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

## 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;

## 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. Assim, na

declaração das classes de exceções definidas pelo grupo 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 que são necessários para a adicionar ao hipermercado. Numa instância de venda temos então:

* 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