

Escola de Engenharia

Departamento de Informática

Universidade do Minho

**Gesthiper**

**Laboratórios de Informática III**

Abril 2015

**Grupo nº 50:**

André Santos (a61778) Bruno Pereira (a72628)

# Arquitectura da aplicação e Estruturação Main

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

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

public static Hipermercado **hipermercado**;

(...)

public static Hipermercado **getHipermercado**() {

return **hipermercado**;

}

public static void **setHipermercado**(Hipermercado novo\_hiper) {

**hipermercado** = novo\_hiper;

}

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

public static void **main**(String[] args) {

**hipermercado** = new Hipermercado();

MenuLeitura.menuLeitura();

MenuQueries.menuPrincipal();

}

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

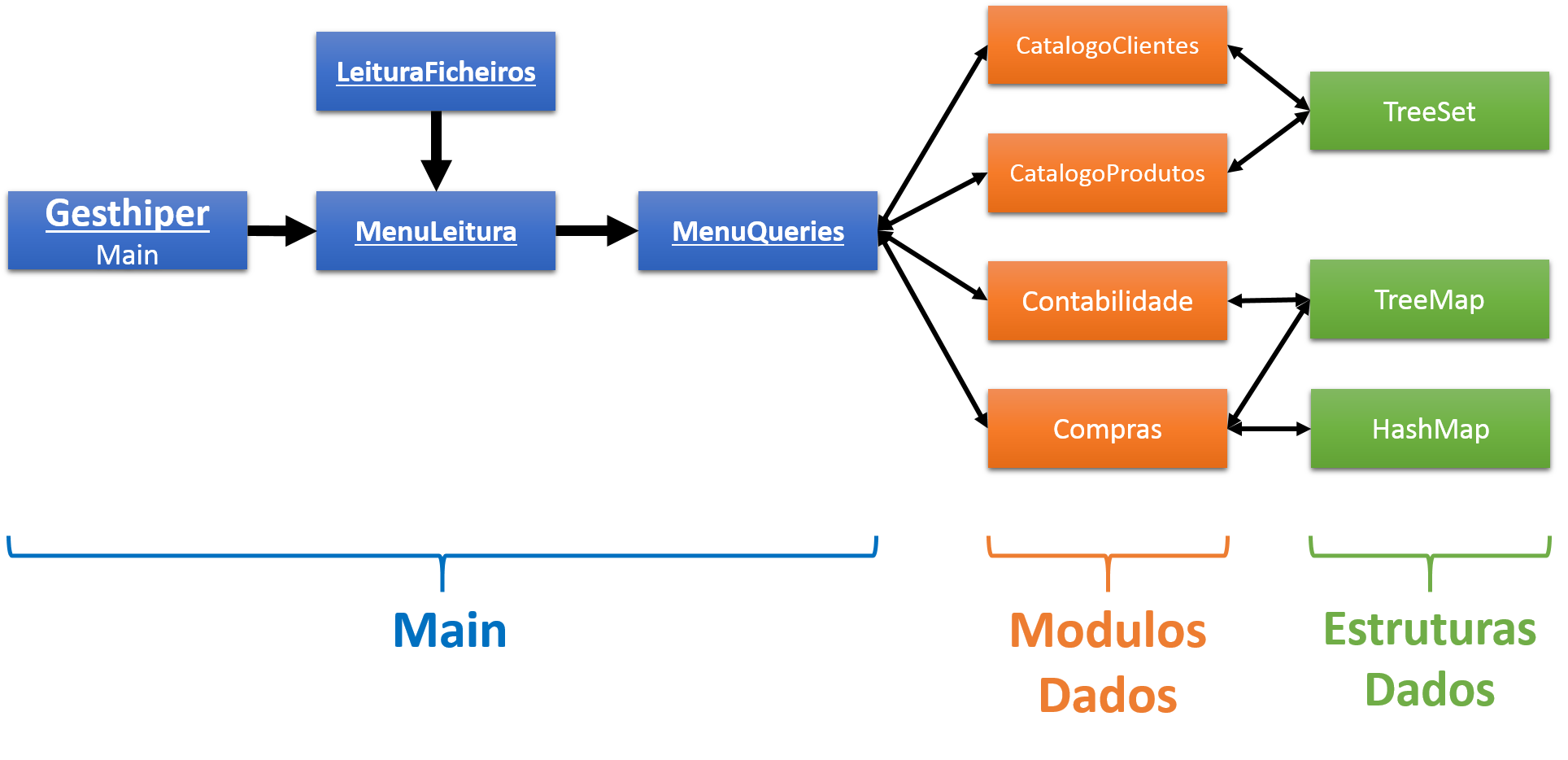


Figura 1 - Arquitectura geral da aplicação

Na figura 1 cada rectângulo corresponde ao nome de uma classe do programa. Visto o esquema pretender representar apenas a arquitectura geral, omitem-se algumas classes secundárias. Como se pode ver o “main” na verdade é constituído por 4 ficheiros/classes:

* **Gesthiper** – Onde se localiza a função main, apresentada anteriormente.
* **LeituraFicheiros** - Funções que tratam da leitura de ficheiros.
* **MenuLeitura** – Parte da interface onde o utilizador escolhe de que forma, e que ficheiros pretende ler para inicializar o programa. Relativamente à leitura, esta parte trata apenas da interface e tratamento do input do utilizador, chamando as funções de LeituraFicheiros para ler os ficheiros efectivamente.
* **MenuQueries** – Parte da interface responsável por mostrar ao utilizador as queries disponíveis e as executar quando seleccionadas, fazendo pedidos aos módulos.

A decisão de dividir o main nestes 4 ficheiros teve como objectivo principal evitar ter um ficheiros com elevado número de linhas de código co as consequentes dificuldades de navegação e procura de partes específicas de código que isso traria. Como objectivo secundário, visa dividir o código de forma lógica, de acordo com a funcionalidade, o que por sua vez também facilita a navegação pelo código. A divisão da parte computacional da parte de I/O na leitura dos ficheiros veio-se a revelar bastante útil numa fase posterior do projecto, em que para mudar o ficheiro de compras a meio da execução do programa foi necessário chamar uma das funções de LeituraFicheiros sem ter associada a interface do MenuLeitura.

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

# Módulos de Dados

## Catálogo Produtos

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

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

O motivo para se ter um ArrayList de TreeSets e não apenas um único TreeSet visa facilitar procuras no catálogo. Embora este projecto não necessite de consultar os módulos dos catálogos, assume-se que os objectivos deste módulo são os mesmos do projecto de C, em que é útil ver o catálogo como vários conjuntos em vez de um único.

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

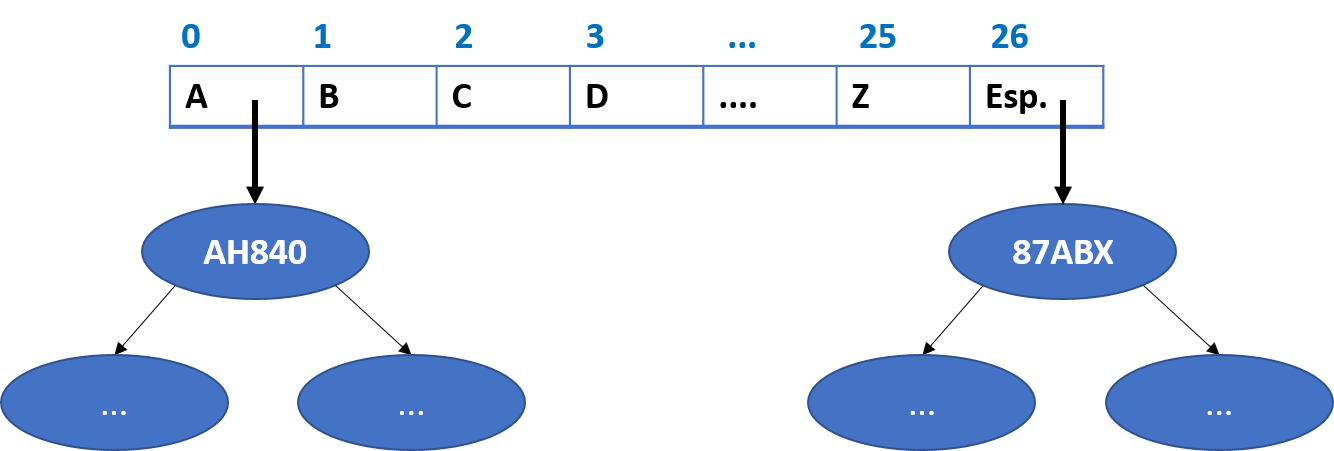


Figura 2- Estrutura do catálogo de produtos

## Catálogo Clientes

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

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

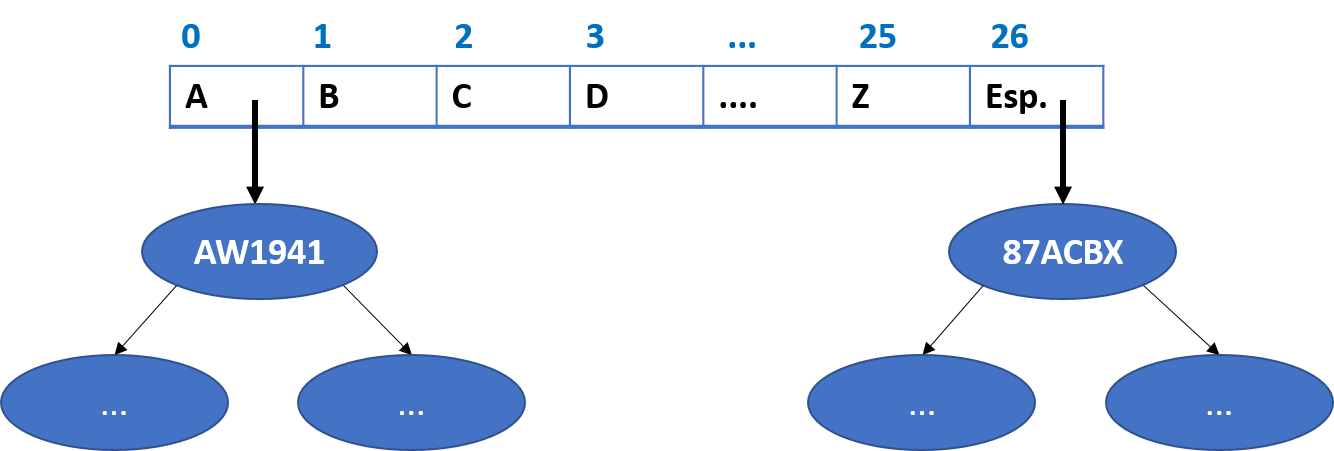


Figura 3- Estrutura do catálogo de clientes

## Contabilidade

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

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

private Matriz\_Int\_12x2 **totalComprasPorMes**;

private Matriz\_Int\_12x2 **totalUnidadesVendidasPorMes**;

private Matriz\_Double\_12x2 **totalFacturadoPorMes**;

**TreeMap**<Produto, FichaProdutoContabilidade> **arvoreProdutos**;

Em que a **FichaProdutoContabilidade** consiste em:

private Matriz\_Int\_12x2 **numUnidadesProdutoVendidasPorMes**;

private Matriz\_Int\_12x2 **numComprasProdutoPorMes**;

private Matriz\_Double\_12x2 **facturacaoProdutoPorMes**;

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



Figura - Estrutura do módulo contabilidade

## Módulo Compras

O principal desafio no planeamento do módulo das compras foi a forma como iriam ser associados produtos a clientes e vice-versa. Optamos por resolver esta questão tendo uma estrutura em que o acesso principal é feito por cliente, onde na informação de cada cliente há a dados relativos aos produtos que esse cliente comprou:

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

Onde a **FichaClienteCompras**, que guarda a informação dos produtos que o cliente comprou, é constituida por:

private Matriz\_Int\_12x2 **numUnidadesCompradasClientePorMes**;

private Matriz\_Int\_12x2 **numComprasClientePorMes**;

private Matriz\_Double\_12x2 **dinheiroGastoClientePorMes**;

private **TreeMap**<Produto, FichaProdutoDeClienteCompras> **produtosCliente**;

Nesta estrutura , além da informação global sobre compras, vendas e unidades compradas do cliente, é possível verificar que a informação dos produtos comprados pelo cliente é também guardada num TreeMap produtosCliente. Os valores deste map contêm informação relativa às vendas do produto, para o cliente a que está associado. Assim, a **FichaProdutoDeClienteCompras** é constituída por:

private Matriz\_Int\_12x2 **numUnidadesCompradasProdutoClientePorMes**;

private Matriz\_Int\_12x2 **numComprasProdutoClientePorMes**;

private double **totalGastoClienteProduto**;

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

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

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

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

private int[] **numeroClientesDistintosPorMes**;

private int **numeroTotalClientesDistintos**;

Para esta estrutura secundária tomou-se a decisão de usar um HashMap. No programa, esta estrutura apenas vai ser acedida para procuras e inserções de elementos individuais, tarefas que um HashMap realiza de forma rápida. Este Map não precisa de ser iterado e os resultados tirados não precisam de sofrer qualquer tipo de ordenação, dai se ter usado um HashMap em deterimento de um TreeMap por exemplo.

Além das estruturas apresentadas anteriormente, o módulo de Compras tem ainda informação global sobre o número de clientes por mês e totais, bem como do número de compras de valor zero. As variáveis de instância completas do módulo de **Compras** são por isso:

private int **numeroComprasValorZero**;

private int **numeroClientesDistintosPorMes**[];

private int **numeroTotalClientesDistintos**;

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

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

O módulo de compras pode ser representado pelo seguinte esquema:

[Inserir aqui esquema do modulo compras]

## Classes Matriz\_Int\_12x2 / Matriz\_Double\_12x2

Da apresentação anteiror feita aos módulos, é possível verificar que as informações de vendas, nº de unidades compradas e facturação é representada por uma variável do tipo Matriz\_Int\_12x2 ou Matriz\_Double\_12x2.

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

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

As variáveis de instância e classe são por isso:

public static int **MAX\_ROW** = 12;

public static int **MAX\_COL** = 2;

private int[][] **matriz** = new int[MAX\_ROW][MAX\_COL];

private int **total**;

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

Isto é particularmente relevante em queries em que é necessário iterar várias “Fichas” seja de produto ou cliente e se pretenda saber o total de forma rápida. Um exemplo concreto disto é a querie que pede uma lista de produtos não comprados. Ora, saber se o produto não foi comprado, equivale a saber se o nº de compras é igual a zero. A função que devolve os produtos não comprados tem por isso que percorrer todos os produtos e para cada um deles ver o total de compras. Ora, o programa seria bem menos eficiente se ao iterar todos os produtos, para cada um deles tivesse que se somar 24 valores para saber se o nº de compras foi zero. Sendo certo que esta variável torna mais lentas operações de actualização de valores, foi deliberada a decisão de tornar a leitura ligeiramente mais lenta para que as queries fossem eficientes.