Universidade do Minho

Departamento de Informática

Laboratórios de Informática III

Mestrado Integrado em Engenharia Informática

2015/2016

GereVendas – Projeto em Java

Grupo 11

Carlos Pereira – 61887

Diogo Silva – 76407

Rafael Braga – 61799

1. GereVendas

Esta classe é responsável pelo arranque e inicialização do programa. Contém todas as variáveis auxiliares para efeitos de estatística, assim como a definição de todos os menus utilizados na comunicação entre o utilizador e o programa. Denote-se que também se encontra em *GereVendas* uma instância da classe *Hipermercado*.

- Consultas Estatísticas

A apresentação das consultas estatísticas assim como a definição dos respectivos métodos é definida em *GereVendas.* A primeira consulta estatística é calculada à medida se adicionam dados para a memória. Ou seja, é calculada em conjunto com o *parse* das linhas de vendas. De forma que seja possível, estão definidas nesta classe variáveis auxiliares. O método que permite a sua inicialização é chamado pela *main*. A segunda consulta estatística é efectuada após os dados estarem todos em memória, isto é, depois do preenchimento de todas as estruturas de dados. No fim do carregamento de todos os ficheiros, são apresentadas estas duas consultas estatísticas.

- Carregamento de Ficheiros

Ao utilizador é permitido decidir quais os ficheiros correspondentes aos clientes, produtos e vendas que pretende carregar. Também é dada a escolha de carregar o programa com um conjunto de ficheiros predefinido. Este é o primeiro menu que é apresentado ao utilizador no início do programa. Além destas duas opções, também é dada a opção de carregar um estado do programa anterior (neste caso a instância *Hipermercado* anteriormente referida). Por predefinição este ficheiro em disco é denominado por *hipermercado.dat*. Contudo, o utilizador pode inserir o nome de outro ficheiro. Por fim, caso o nome do ficheiro que o utilizador insira não exista, é pedido novamente que insira outro nome.

- Menu Principal

Após o carregamento de dados com sucesso é apresentado ao utilizador um menu com as consultas interactivas que pode fazer. *GereVendas* encarrega-se de medir os tempos que cada consulta demora a calcular o resultado. Para tal ser possível, é utilizada a classe auxiliar *Crono*. Algumas consultas apresentam um número elevado de dados no *terminal* o que requer a utilização de uma classe auxiliar encarregue pela paginação. Ao fim de cada consulta, o resultado é convertido para uma *List<String>* pelo método respectivo em *GereVendas* e enviado para o método responsável pela navegação.

- Ler *Input* do Utilizador

Para facilitar a comunicação entre o utilizador e *GereVendas* em diversos contextos, foram definidos vários métodos que pedem *inputs* utilizando a classe auxiliar *Input.* Por exemplo, se é pedido ao utilizador para inserir um mês do ano, caso este forneça um inteiro que não esteja dentro do intervalo permitido é apresentada uma mensagem de erro e é pedido um novo mês. Esta prática é análoga para os outros contextos no decorrer do programa.

- Paginação:

Para apresentar os resultados de diversas consultas é utilizada uma classe auxiliar para paginação. Utilizam-se funções *lambda* na apresentação dos resultados para efeitos de abstracção. Deste modo torna-se possível apresentar cada *String* no *terminal* ou noutro componente qualquer (por exemplo: Página *Web).*

- Leitura e Escrita de Dados:

A recuperação do estado do programa assim como o processo de salvaguardar o mesmo em disco é efectuado a partir de *GereVendas* com o auxílio da classe *Ficheiros* (a definição dos métodos necessários encontra-se presente nesta classe). Por estado do programa, entenda-se a instância *Hipermercado* presente em *GereVendas*.

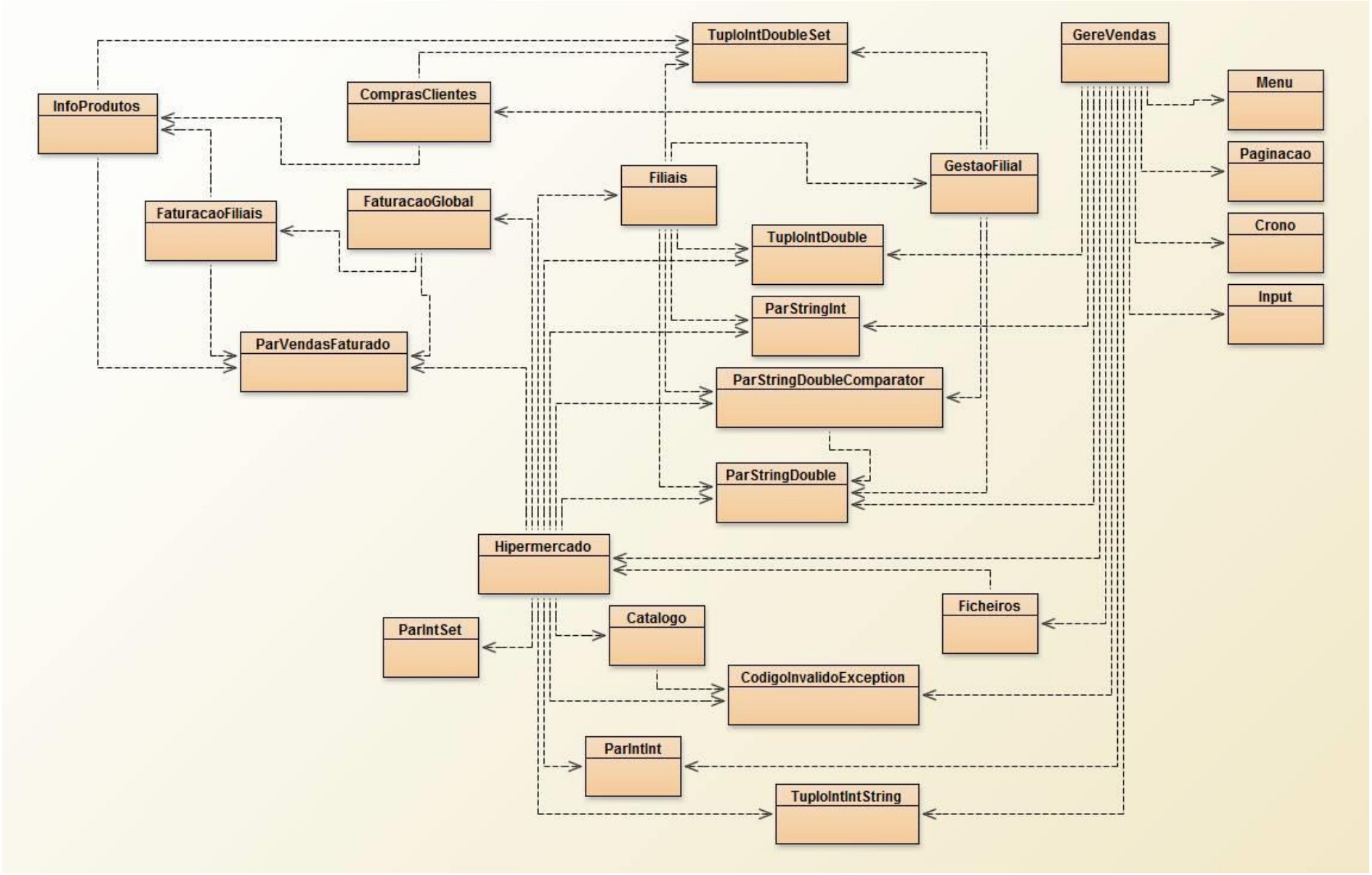

Figura 1 Esquema da classe *GereVendas*

Classe *Hipermercado*

Esta é a classe principal que contém as estruturas necessárias para o funcionamento de *GereVendas*. Contém estruturas referente à facturação global, à gestão de filiais, bem como catálogos de clientes e de produtos. Guarda informação acerca dos produtos mais vendidos e também é responsável pelo cálculo de todas as *queries* iterativas do programa *GereVendas*.

As estruturas principais desta classe são:

- *fGlobal*: dados relativos à facturação global guardados na classe *FaturacaoGlobal*

- *filiais*: dados relativos à gestão de todas as filiais guardados na classe *Filiais*

- *produtos*: catálogo de produtos guardados na classe *Catalogo*

- *clientes*: catálogo de clientes guardados na classe *Catalogo*

- *xMaisVendidos*: *HashMap* com as unidades totais vendidas e facturação global de todos os produtos, sendo uma estrutura auxiliar da *querie* 6

Para resolver as *queries, GereVendas* chama os métodos em *Hipermercado* que lidam com as estruturas responsáveis pelos dados a processar. Depois, devolve os resultados na estrutura adequada. Quando é necessário devolver mais do que um parâmetro, *Hipermercado* devolve esses resultados com a ajuda das classes auxiliares que foram criadas para esse efeito.

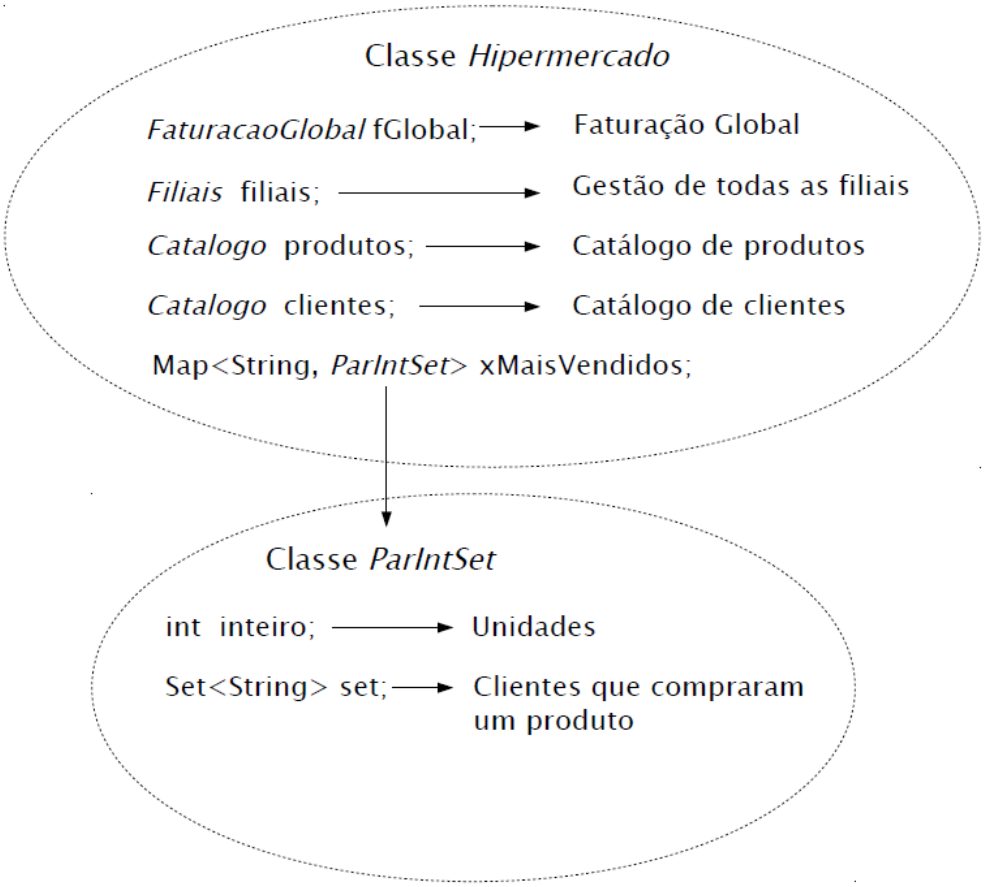

Figura 2 Esquema da classe *Hipermercado*

Classe *Catalogo*

Esta classe é responsável por guardar os códigos dos clientes e dos produtos. A estrutura principal é um *TreeMap* em que as chaves (*keys*) correspondem ao primeiro caractere do código e o valor (*value*) de cada chave é um *TreeSet* com os códigos que começam por esse caractere. Desta forma já temos os códigos separados pela primeira letra, facilitando assim uma possível pesquisa.

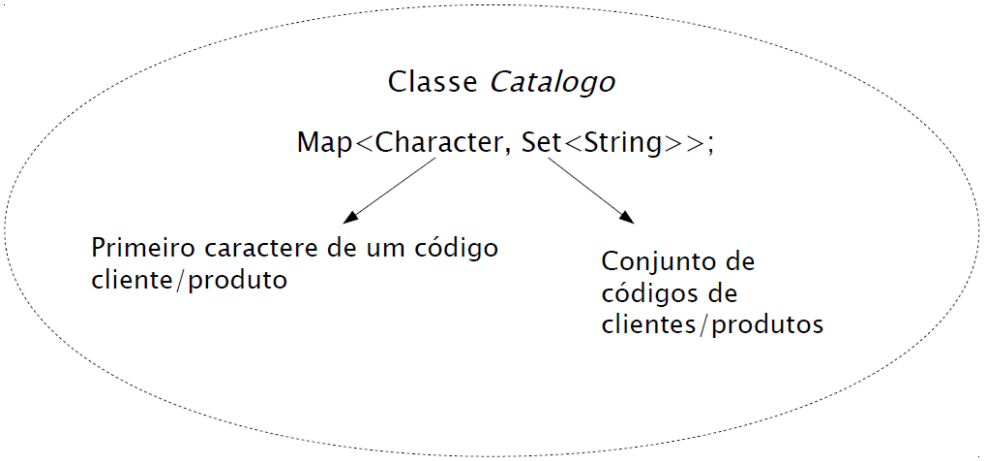

Figura 3 Esquema da classe *Catalogo*

Para além dos construtores e dos métodos principais comuns, esta classe também tem um método que permite aplicar uma função a todos os valores da *TreeMap*, como também verificar se um código existe na estrutura.

O encapsulamento é garantido porque só é possível alterar elementos da estrutura através de métodos que foram criados para esse efeito. Quando é necessário retornar um elemento, é devolvida uma cópia do valor.

Classe *FaturacaoGlobal*

Esta classe é responsável por guardar os dados relativos à facturação. Tem duas estruturas principais: *fatGlobal* e *nuncaComprados*.

A facturação global é guardada no *HashMap fatGlobal*, onde as chaves correspondem ao mês e os valores à facturação das filiais para esse mês. As informações sobre a facturação das filiais ficam ao cargo da classe *FaturacaoFiliais*. A facturação nas filiais é um *HashMap* em que as chaves correspondem ao número da filial e os seus valores às informações das vendas de produtos (códigos de produtos, número de unidades vendidas e total facturado). Desta forma é possível saber, por exemplo, quantas vezes um produto foi vendido numa dada filial e num dado mês.

Todos os métodos que retornam elementos desta estrutura devolvem uma cópia do valor. Apenas se consegue aceder aos dados na estrutura através dos métodos criados para esse efeito. Desta forma mantém-se o encapsulamento dos dados.

Os códigos dos produtos que nunca foram comprados são guardados em *nuncaComprados*, sendo este um *TreeSet* de *Strings.* No início do programa, todos os produtos em *Produtos.txt* são adicionados a esta estrutura. Quando é feita a leitura do ficheiro relativo às vendas, é removido de *nuncaComprados* o produto que é lido de cada linha.

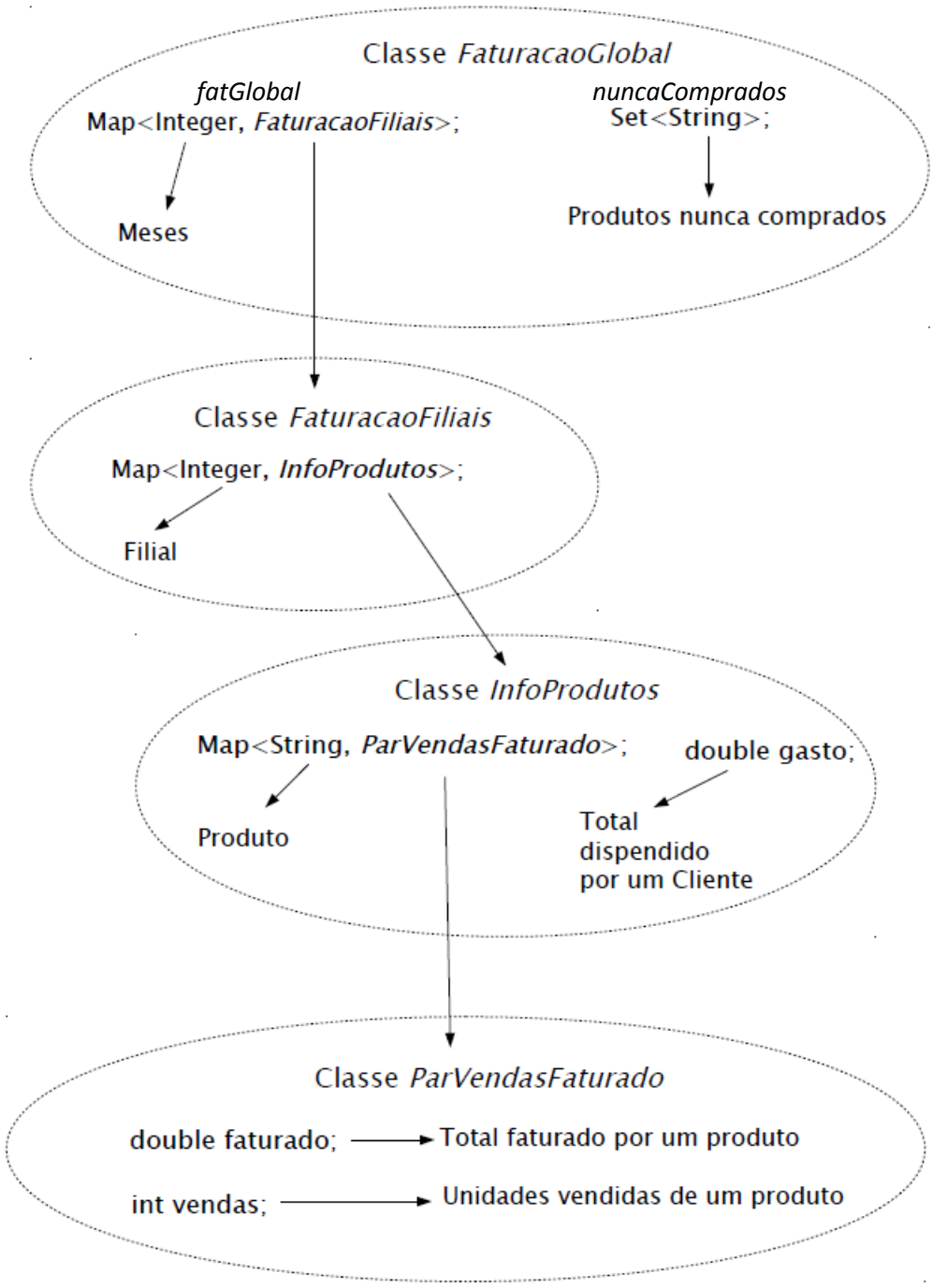

Figura 4 Esquema da classe *FaturacaoGlobal*

Classe *Filiais*

Esta classe é responsável por guardar os dados relativos às filiais. Tem duas estruturas principais: *gestaoFiliais* e *clientesProdutos*.

gestaoFiliais é um *HashMap* em que as chaves correspondem aos números das filiais e os valores à classe *GestaoFilial*. Por sua vez, a *GestaoFilial* contém informações, por mês, sobre as compras de cada Cliente. *GestaoFilial* é um *HashMap* em que as chaves correspondem aos meses e os valores às informações sobre as vendas realizadas: para cada cliente tem-se a lista dos produtos, as unidades e quanto gastou, numa filial e num mês. Desta forma é possível saber, por exemplo, quantas vezes um produto foi comprado por um cliente numa dada filial e num dado mês.

Todos os métodos que retornam elementos desta estrutura devolvem uma cópia do valor. Apenas se consegue aceder aos dados na estrutura através dos métodos criados para esse efeito. Desta forma mantém-se o encapsulamento dos dados.

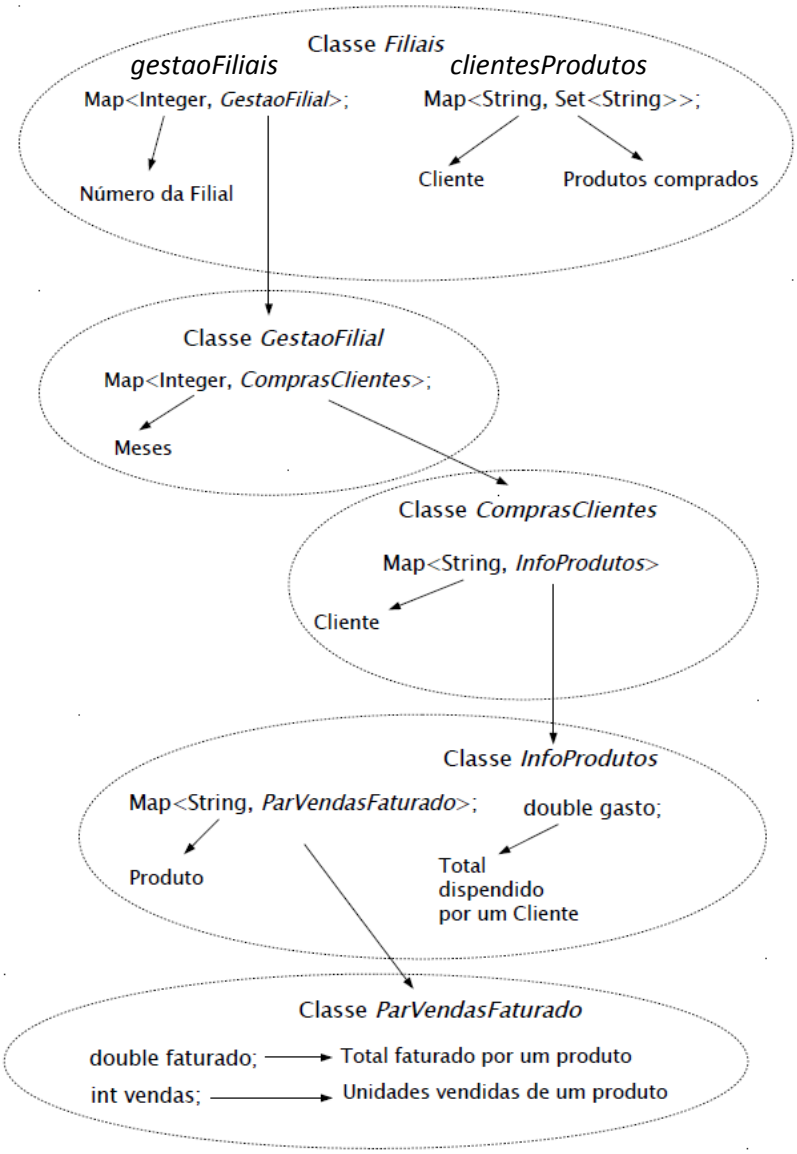

Figura 5 Esquema da classe *Filiais*

Classes auxiliares comuns

Para além das classes principais já referidas (e suas auxiliares), também foram criadas classes que são utilizadas por todas as outras classes. Foram feitas quando surgiu a necessidade de devolver mais do que um parâmetro. Assim, para resolver um pedido de *Hipermercado*, evita-se fazer mais do que uma travessia nas estruturas. As classes principais (e por sua vez as suas auxiliares) recorrem a estas classes auxiliares comuns para guardarem os vários valores que pretendem devolver. A estas classes foram dados nomes genéricos para que pudessem ser utilizados noutros programas.

Sendo as classes auxiliares comuns:

- *ParIntSet*

- *ParStringDouble*

- *ParStringInt*

- *TuploIntDouble*

- *TuploIntDoubleSet*

- *TuploIntIntString*

2. GereVendasEstatisticas

GereVendasEstatisticas é responsável pela realização de testes de performance do programa *GereVendas.* Realiza testes aos ficheiros de um, três e cinco milhões de vendas, nomeadamente a leitura com e sem *parsing* através da classe *Scanner* e da classe *BufferedReader.* O programa pede ao utilizador para escolher um dos três ficheiros de vendas para se realizar testes adicionais. Estes testes consistem na medição de tempos das *queries* mais complexas do programa *GereVendas*, nomeadamente as *queries* 5 a 9. As medições de tempos destas *queries* são efetuadas para vários tipos de coleções. Por exemplo, uma *query* que tenha sido implementada com um *TreeSet* será testada com um *HashSet* e um *LinkedHashSet.*

O programa *GereVendasEstatisticas* usa apenas as classes e as funções necessárias para a execução dos testes descritos acima. Para cada *query*, procedeu-se à sua replicação invertendo o tipo das coleções utilizadas no seu algoritmo para a medição de performance. Como o programa *GereVendas* utilizou meramente interfaces para a resolução das *queries,* este trabalho não foi muito moroso já que não se teve que alterar nada nos parâmetros de entrada e de saída das funções. Por exemplo, para uma *query* que utilizou um *HashMap,* para se substituir por um *TreeMap* apenas se teve que alterar a linha que continha a instrução "*var x = new HashMap<>();"* por "*var x = new TreeMap<>();"*.

O programa começa por apresentar ao utilizador um menu para a escolha do ficheiro de vendas para se realizar os testes das *queries.* De seguida são calculados todos os testes. Sempre que se obtém o resultado de um teste este é apresentado numa tabela em modo de texto no ecrã. Os testes são realizados pela seguinte ordem:

1. Leitura dos ficheiros de um, três e cinco milhões de vendas sem *parsing* através das classes *Scanner* e *BufferedReader.*
2. Leitura dos ficheiros de um, três e cinco milhões de vendas com *parsing* através das classes *Scanner* e *BufferedReader.* Os resultados são convertidos para instâncias da classe *Venda* e guardados num *ArrayList.*
3. Carregamento do ficheiro de vendas escolhido pelo utilizador e salvaguarda dos seus dados nas estruturas implementadas pela classe *Hipermercado.* É realizado um teste de *profiling* simples a este ficheiro. Basicamente efetuam-se as medições de tempos a cada 50.000 inserções dos dados lidos do ficheiro à classe *Hipermercado.* Os resultados dos tempos são guardados num *ArrayList* para uma posterior análise.
4. Medição dos tempos das *queries* 5 a 9 implementadas com vários tipos de coleções.

À medida que se vão obtendo os resultados dos testes, para além de serem apresentados no ecrã, são adicionados aos gráficos. Estes gráficos são visualizados quando todos os resultados dos testes tiverem sido obtidos. Para a criação destes gráficos usaram-se as classes *javafx.scene.chart.BarChart* e *javafx.scene.chart.LineChart* (para o teste de *profiling*) e algumas classes auxiliares a estas. Estas classes fornecem métodos que tornam a construção dos gráficos relativamente simples, bastando criar-se várias categorias de medições e adicionarem-se os valores a estas. Os gráficos são agrupados numa *Frame* através de um *GridLayout.* Este *layout* é bastante flexível pois permite o agrupamento de vários componentes dispostos em várias linhas e colunas de iguais dimensões.

Performance

Apresentam-se de seguida os resultados dos testes de performance (em segundos) para os computadores utilizados na construção dos programas *GereVendas* e *GerevendasEstatisticas.* Para cada computador é descrito o modelo do processador e da memória *RAM.*

	Intel® Core™ i5 4670K 3.4 GHz (Turbo 3.8 GHz) 16 GB RAM DDR3	Intel® Core™ i7 6700HQ 2.6 GHz 32 GB RAM DDR4	Intel® Core™ i7 Q740 1.73 GHz 8 GB RAM DDR3
Scanner sem *Parsing*	1.042387687	1.041991045	3.299493342
Scanner com *Parsing*	1.214549306	1.16632112	3.549936482
BufferedReader sem *Parsing*	0.090278106	0.124005335	0.313109448
BufferedReader com *Parsing*	0.232190512	0.16435253	1.034004436

Tabela 1 Resultados dos testes para o ficheiro Vendas_1M.txt

	Intel® Core™ i5 4670K 3.4 GHz (Turbo 3.8 GHz) 16 GB RAM DDR3	Intel® Core™ i7 6700HQ 2.6 GHz 32 GB RAM DDR4	Intel® Core™ i7 Q740 1.73 GHz 8 GB RAM DDR3
Scanner sem *Parsing*	3.330912304	2.692830905	9.667654122
Scanner com *Parsing*	5.268661355	4.711099716	11.905586861
BufferedReader sem *Parsing*	0.20817528	0.211659913	0.430961614
BufferedReader com *Parsing*	2.541439143	2.15444786	9.11247161

Tabela 2 Resultados dos testes para o ficheiro Vendas_3M.txt

	Intel® Core™ i5 4670K 3.4 GHz (Turbo 3.8 GHz) 16 GB RAM DDR3	Intel® Core™ i7 6700HQ 2.6 GHz 32 GB RAM DDR4	Intel® Core™ i7 Q740 1.73 GHz 8 GB RAM DDR3
Scanner sem *Parsing*	4.770564618	5.181685442	12.207151254
Scanner com *Parsing*	10.974097919	6.606192819	46.719110579
BufferedReader sem *Parsing*	0.589372328	0.334168755	1.790072342
BufferedReader com *Parsing*	5.826325289	3.999091359	28.015065901

Tabela 3 Resultados dos testes para o ficheiro Vendas_5M.txt

Pela análise das tabelas apresentadas verifica-se claramente a superioridade de performance ao carregar ficheiros através da classe *BufferedReader.*

Apresenta-se de seguida os resultados dos testes (em segundos) das *queries* 5 a 9 para diferentes coleções. Os valores apresentados correspondem ao computador com menor performance (Intel® Core™ i7 Q740 1.73 GHz 8 GB RAM DDR3). Os valores dos testes das *queries* 5 a 9, bem como do teste de *profiling* correspondem ao ficheiro de três milhões de vendas.

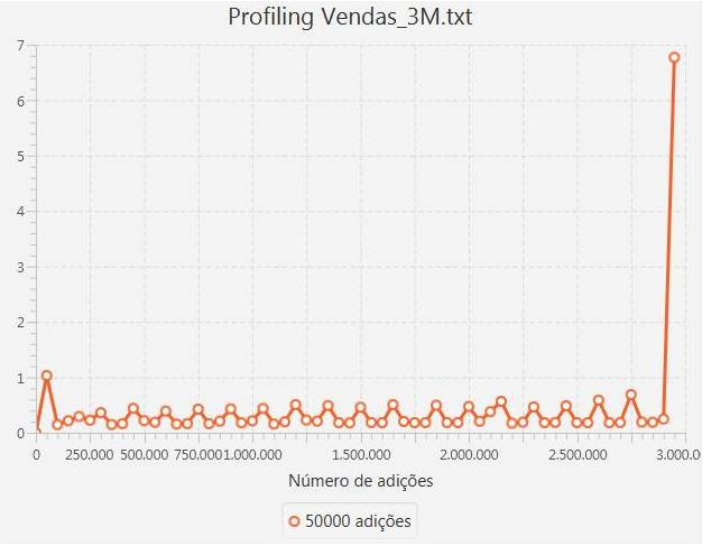

Figura 6 – *Profiling* do ficheiro *Vendas_3M.txt*

Como se pode ver na **Figura 6**, os tempos de inserção de cada 50.000 dados nas estruturas da classe *Hipermercado* são praticamente constantes (exceto na primeira e última medição). Quando o tempo da inserção aumenta, significa que foi realizado o *rehashing* e o redimensionamento da tabela de *hash* do *HashMap.* Após ser feita uma operação destas, verifica-se que o tempo de inserção seguinte é menor. Isto significa que a tabela de *hash* tem menos colisões.

Relativamente ao pico na primeira medição, este deve-se ao redimensionamento ser feito com mais frequência no início. Como o tamanho por defeito de um *HashMap* é de 16 chaves, ao fazer a inserção dos primeiros 50.000 elementos, é esperado que seja necessário fazer mais vezes o redimensionamento. Tudo isto contribui para um impacto negativo de performance.

Quando a tabela de *hash* contém muitos elementos, o tempo de procura para a inserção de um novo valor correspondente a uma chave é elevado. A isto poderá dever-se o pico na última medição.

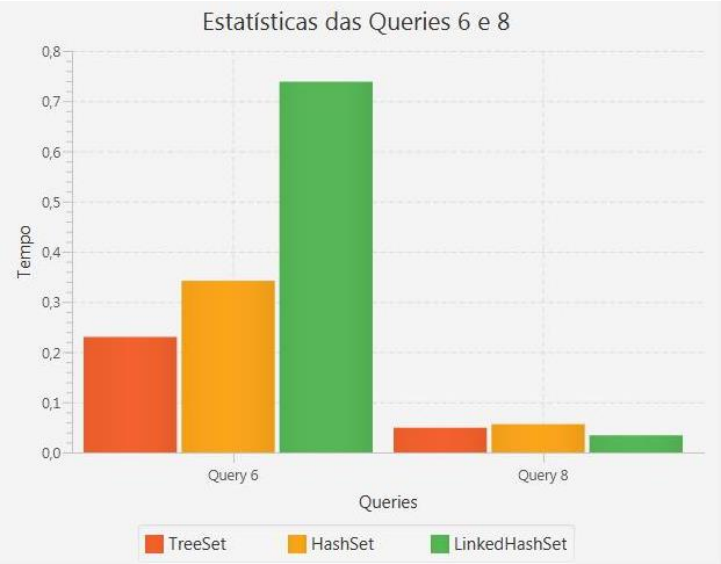

Figura 7 Tempos de execução das *queries* 6 e 8 para diferentes coleções.

Os gráficos apresentados na **Figura 7** representam os tempos de execução das *queries* 6 e 8 para diferentes coleções do tipo *Set*.

Para a *query* 6 a coleção mais eficiente é o *TreeSet*. Isto poderá dever-se ao facto do *TreeSet* ordenar-se após cada inserção, ao contrário do *HashSet* e do *LinkedHashSet* que têm de ser ordenados após a inserção de todos os elementos. Como *HashSet* e o *LinkedHashSe*t se baseiam em tabelas de *hash*, é esperado que seja feito o redimensionamento das tabelas. Tudo isto contribui para uma pior *performance*.

Na *query* 8, pode-se verificar que a utilização de diferentes estruturas resulta em tempos praticamente iguais (ao contrário do verificado na *query* 6). O *LinkedHashSet* apresenta o melhor resultado. Isto poderá dever-se ao facto de uma iteração nesta estrutura ser mais eficiente.

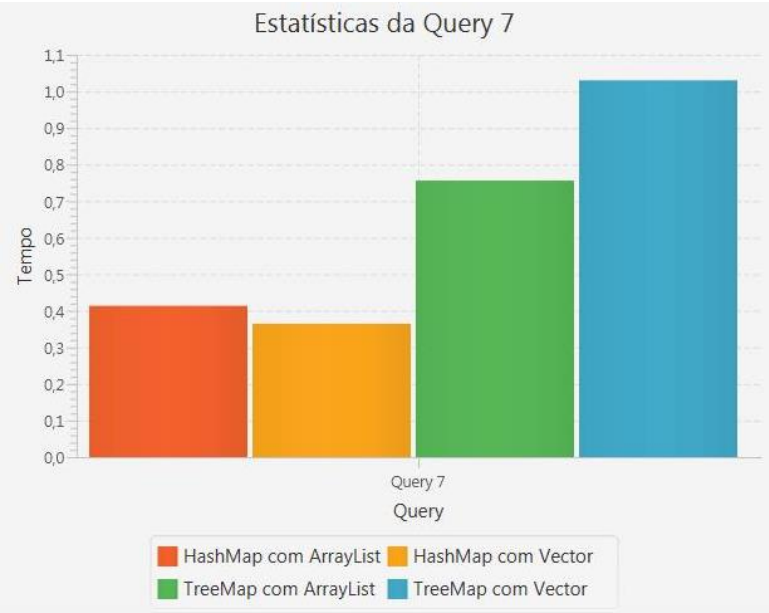

Figura 8 Tempos de execução da *query* 7 para diferentes coleções

Na **Figura 8**, Verifica-se que o *HashMap* é superior ao *TreeMap*. O *TreeMap* perde tempo ao fazer a ordenação dos elementos, contribuindo para uma penalização na performance.

Também se pode verificar que o *Vector* é uma melhor opção que um *ArrayList* quando combinado com um *HashMap*.

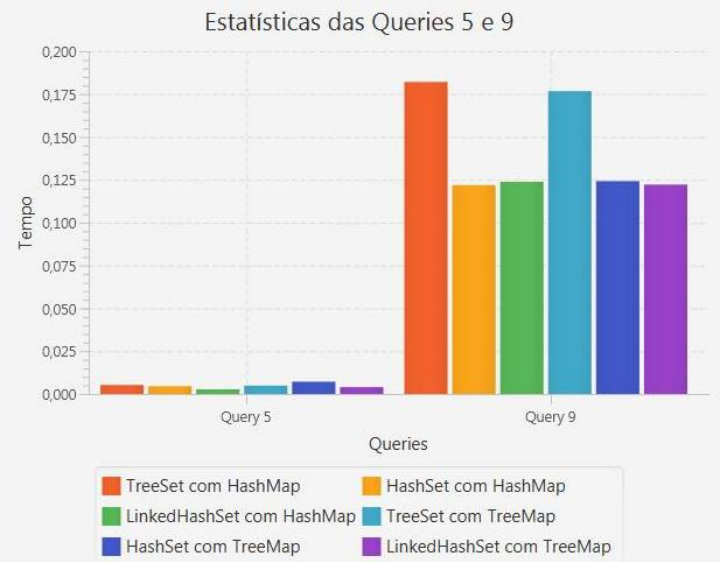

Figura 9 Tempos de execução das *queries* 5 e 9 para diferentes coleções.

Olhando para a **Figura 9** verifica-se que para a *query* 5 qualquer estrutura é eficiente e todas elas fornecem tempos muito semelhantes. A estrutura mais adequada dependerá da execução.

Na *query* 9 o uso de *TreeSet* é claramente o mais dispendioso. Pode dever-se ao facto da ordenação ser feita à inserção de um elemento. Ao não usar-se esta estrutura, observa-se que qualquer outra combinação é apropriada para esta *query*.

Outras considerações

Para melhorar os tempos de população da instância *Hipermercado* tentou recorrer-se ao paralelismo intrínseco de *Java* 8. Para tal utilizou-se uma *Stream<Venda>* depois de ter sido efetuado todo o *parsing* de um ficheiro de vendas. O código da estratégia utilizada é o seguinte:

```
public static void addHipermercado(ArrayList<Venda> lst) {
lst.parallelStream.forEach(v -> {
    hipermercado.addFGlobal(v.getProduto(),v.getPreco(),v.getUnidades(),
                            v.getMes(), v.getFilial());
    hipermercado.addFiliais(v.getProdutos(), v.getPreco(), v.getUnidades(),
                            v.getCliente(), v.getMes(), v.getFilial();
    hipermercado.addXMaisVendidos(v.getProduto(), v.getUnidades(),
                                  v.getCliente());
}
```

Figura 10 Código utilizado

O tempo de população diminuiu significativamente. Para o ficheiro de três milhões de vendas diminuiu dos 12.880645813 segundos para 8.731498513 segundos (Intel Core™ i7 6700HQ 2.6 GHz 32 GB RAM DDR4). No entanto, esta solução traz alguns problemas. Como as estruturas utilizadas pela classe *Hipermercado* se baseiam em *Map<K, V>* na inserção de um novo elemento é necessário obter o mesmo e atualizar o novo valor. Como isto acontece, há a possibilidade de se aceder ao mesmo elemento ao mesmo tempo. Esta solução não foi utilizada pois verificou-se que os resultados das *queries* variavam de execução para execução.

Conclusão

Com a realização deste trabalho conclui-se que a linguagem *Java* facilita muito a construção de aplicações em larga escala e fornece vários mecanismos eficientes para o armazenamento de grandes quantidades de dados. Esta linguagem permite a construção de várias classes facilmente reutilizáveis e a divisão de um programa muito complexo em componentes independentes e de pequenas funções. Também fornece mecanismos que facilitam o encapsulamento, aumentando assim a segurança.

Java Collections Framework oferece vários tipos de estruturas eficientes para múltiplos tipos de consultas. De todas as coleções a mais utilizada foi a *HashMap<K, V>* que permite o agrupamento de inúmeros valores organizados por chaves com uma grande eficiência. Uma grande funcionalidade consiste no uso de interfaces em vez de classes mais concretas das coleções. Isto permitiu a realização de inúmeros testes para se verificar qual o tipo de coleção mais eficiente para uma determinada *query.* Relativamente aos testes de eficiência, *Java* dispõe de inúmeras ferramentas que permitem uma melhor visualização dos resultados nomeadamente os gráficos de barras e os gráficos de linhas.

Neste projeto recorreu-se aos novos componentes provenientes de *Java 8*. A nova atualização dispõe de inúmeras funcionalidades que tornam a construção de código muito mais organizada e simples. As expressões *lambda* foram muito utilizadas e permitiram que *queries* muito mais complexas que o projeto em *C* fossem implementadas numa maneira muito mais simples e com muito menos linhas de código. Para além disso, recorreu-se à passagem de expressões *lambda* por argumentos de funções. Esta funcionalidade permitiu a construção de classes mais genéricas e independentes entre si.