Universidade do Minho
Escola de Engenharia

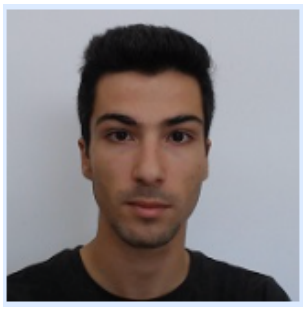
Sistema de Gestão de Vendas

Trabalho Prático de Java

Grupo 26

a84610 – Gonçalo Almeida

a86788 – Lázaro Pinheiro



a80960 – Rúben Rodrigues

Laboratórios de Informática III
2º Ano, 2º Semestre
2018/2019

Índice

Introdução

O presente trabalho prático desenvolve-se no âmbito da Unidade Curricular Laboratórios de Informática III, lecionada no 2º semestre do 2º ano do curso de Engenharia Informática.
Este trabalho tem como objetivo aumentar os conhecimentos na linguagem Java e, fundamentalmente, a apresentação dos desafios que se colocam a quem concebe e programa aplicações *software* com grandes volumes de dados e com a mais elevada complexidade algorítmica e estrutural.

Procura-se a criação de uma aplicação de uma aplicação Desktop em Java, baseada na utilização das interfaces e das coleções de JCF (*"Java Collections Framework"*), cujo objetivo é a realização de consultas interativas e estatísticas de informações relativas à gestão de uma cadeia de distribuição, com vista a facilitar a organização e aumentar o rendimento.

O mesmo trabalho prático reveste-se de uma mais valia para colocar em prática conhecimentos, fomentar e consolidar aprendizagens e desbravar terreno no mundo da linguagem Java e do paradigma dos objetos.

Modelo Visão Controlador (MVC)

Para a realização deste projeto, seguiu-se a recomendação dada pelos docentes da Unidade Curricular. Implementou-se a arquitetura de software MVC, que consiste na separação entre a informação da interação do utilizador com ela.

Tem como vantagens a fácil reutilização de código, melhor performance – fruto da separação das camadas -, melhor desempenho e produtividade - devido à estrutura de pacotes modulares - permitindo que os programadores possam trabalhar em paralelo e que as partes da aplicação possam ser alteradas sem que haja necessidade de alterar outras.

Tem como principal desvantagem a necessidade de mais tempo para explorar e modelar o sistema.

Modelo

A principal função do modelo é o armazenamento e tratamento de dados.
Assim sendo, dividimos o nosso modelo em dois submodelos, os ***ModelosBase*** (ver Anexo – Figura 1) que estruturam e validam a informação de toda a aplicação e os ***Catalogos*** (ver Anexo – Figura 2) que armazenam e tratam toda a informação.
Neste package possuímos uma classe agregadora, denominada SGV, com a finalidade de incluir todos os catálogos com o intuito de preservar o encapsulamento. Nesta classe, é realizada a leitura (carregamento da informação dos ficheiros de texto).

Modelos Base

Cliente

A classe Cliente (ver Anexo - Figura 7) é uma simples estrutura de dados, utilizada para validar uma linha lida do ficheiro "Clientes.txt", contendo uma *String* como variável de instância a qual representa um código de cliente.

Produto

A classe Produto (ver Anexo – Figura 8) é uma simples estrutura de dados, utilizada para validar uma linha lida do ficheiro "Produtos.txt", contendo uma *String* como variável de instância a qual representa um código de produto.

Venda

A classe Venda (ver Anexo – Figura 10), que a sua principal função é validar uma linha de venda, lida de um ficheiro de texto "Vendas_.txt", para que posteriormente os dados validados possam ser armazenados nos catálogos.

Fatura

A classe Fatura (ver Anexo – Figura 9) tem como funcionalidade armazenar os totais de unidades vendidas e o faturado, diferenciando-os em tipo 'N', normal, ou 'P', em promoção.

Módulos de Dados

Catálogo de Produtos

(ver Anexo – Figura 4 e Figura 12)

Para a realização deste módulo de dados, julgou-se de relevante interesse a utilização de um *TreeSet* de Produtos, ordenado alfabeticamente utilizando um *Comparator* denominado compProdutoCodigo, de modo a que a validação da existência de um produto na estrutura seja mais eficiente e que cada elemento é único.

Catálogo de Clientes

(ver Anexo – Figura 3 e Figura 11)

Para a realização deste módulo de dados, julgou-se pertinente a utilização de um TreeSet de Clientes, ordenado alfabeticamente utilizando um Comparator denominado compClienteCodigo, de modo a que a validação da existência de um produto na estrutura seja mais eficiente e que cada elemento é único.

Faturação Global

(ver Anexo – Figura 5 e Figura 13)

Esta classe tem como principal função determinar o total faturado para todos os produtos existentes, para tal utilizou-se um *HashMap* onde a chave é um código de produto e o valor uma Matriz, onde as linhas representam os meses e as colunas as filiais, responsável por armazenar instâncias da classe Fatura. Assim para todos os produtos, é possível determinar o total faturado e o número de quantidades vendidas, tendo em conta os meses e as filiais.

Gestão de Filial

(ver Anexo – Figura 6 e Figura 14)

Esta classe tem como variáveis de instância um *HashMap,* cujas chaves são os códigos de clientes que realizaram compras, e os valores são *HashMap's,* cujas chaves são os códigos dos produtos que o cliente comprou e o valor é uma instância da classe Matriz. Esta Matriz tem como dimensão o número de meses por número de filiais e armazena objetos do tipo Fatura. Sempre que uma venda é acrescentada é acedido a matriz através dos códigos de cliente e produtos desta, e a fatura na posição mês-filial é atualizada com a quantidade de unidades vendidas e o total faturado dessa venda, dependendo se é do tipo N ou P (inicialmente a fatura é inicializada com quantidade e faturado a 0).

Visão

(ver Anexo – Figura 15 a Figura 18)

A visão gera uma representação dos dados presentes no Modelo solicitado, exibindo-os ao utilizador.

Aqui, temos implementados todos os menus e diferentes modos de Listagem (de uma *String* e de uma lista de *String's*).

Controlador

A missão do controlador é enviar comandos para o Modelo, com a finalidade de atualizar o seu estado, e para a Visão, com o intuito de alterar a visão da informação no Modelo.

Assim, toda a lógica de negócio é implementada neste módulo.

Performance

Como critérios para a escolha das melhores estruturas, usou-se o tempo despendido e o espaço em memória gasto.

Como tal fizemos dez testes (ver Anexo – Figura 19) para cada estrutura e realizou-se a média entre os diferentes tempos obtidos e a média entre os diferentes espaços ocupados.

É de realçar que nem sempre o melhor tempo significa que seja a melhor estrutura. A título exemplificativo, temos o catálogo de produtos e de clientes. O *TreeSet* é a estrutura mais demorada na sua povoação, em contrapartida é a mais rápida ao validar as vendas.

Conclusão

Com a realização deste trabalho desenvolvemos os nossos conhecimentos na linguagem de Java e no tratamento de grandes quantidades de dados, que nos criou dificuldades em termos de espaço na memória.

O facto de validar os códigos existentes nas vendas tornou-se bastante demorado e deve ser implementada uma forma mais otimizada. Outra alteração a fazer seria o facto de o total faturado de uma venda não estar a multiplicar pelas unidades vendidas, isto por erro de compreensão nosso, que entendemos que o faturado era da venda de todas as unidades.

A implementação MVC mostrou-se bastante útil na divisão de tarefas entre membros e mostrou-se ser uma boa técnica de programação devido às vantagens anteriormente referidas.

Em forma de conclusão, o trabalho em geral foi concretizado, faltando apenas a validação, a questão do faturado e uma melhor gestão de memória, aspetos esses que deveriam ter sido melhorados e solucionados.

Anexos

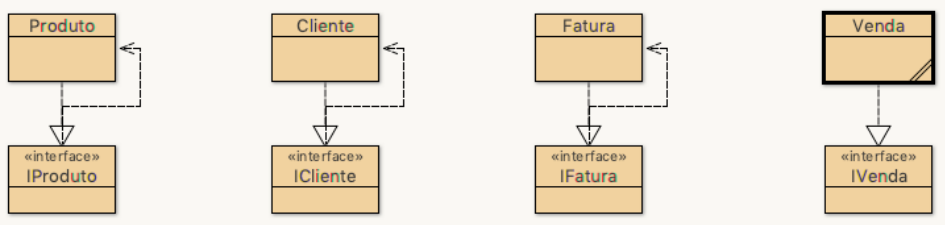


Figura 1 - Representação do Package ModelosBase.

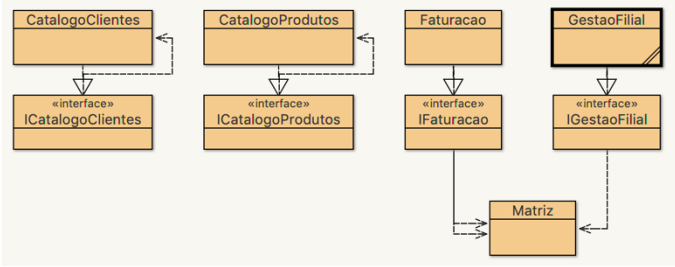


Figura 2 - Representação do Package Catalogos.

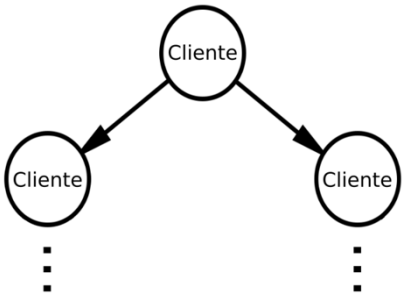


Figura 3 - Representação da estrutura Catálogo de Clientes.

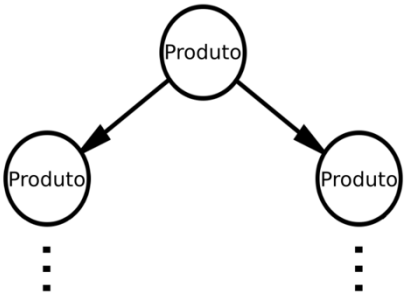


Figura 4 - Representação da estrutura Catálogo de Produtos.

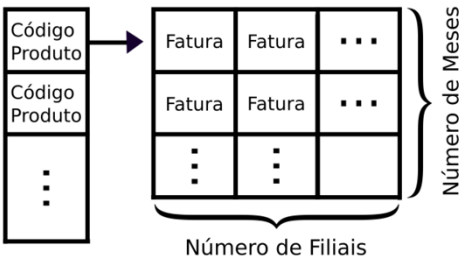


Figura 6 - Representação da estrutura Faturação.

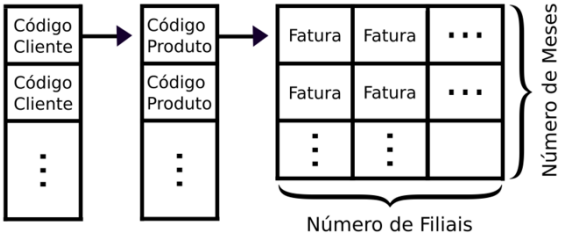


Figura 5 - Representação da estrutura Gestão de Filial.

```java
public interface ICliente{
    /**
      * Devolve o código de Cliente
      * @return codigoCliente
      */
    public String getCodigoCliente();
    /**
     * Verifica a igualdade de dois objectos
     * @param objecto
     * @return boolean
     */
    public boolean equals(Object o);
    /**
     * Metodo que devolve a representação em String do Cliente
     * @return codigoCliente
     */
    public String toString();
    /**
     * Cria uma cópia de Cliente
     * @return Cliente
     */
    public Cliente clone();
}
```

Figura 7 – Interface de Cliente

```java
public interface IProduto{
    /**
     * Devolve o código do produto
     * @return String
     */
    public String getCodigoProduto();
    /**
     * Verifica a igualdade de dois objectos
     * @param objecto
     * @return boolean
     */
    public boolean equals(Object o);
    /**
     * Retorna uma representação textual do objecto
     * @return String
     */
    public String toString();
    /**
     * Cria uma cópia do objecto
     * @return objecto
     */
    public Produto clone();
}
```

Figura 8 - Interface de Produto

```java
public interface IFatura{
    /**
      * Devolve a quantidade de tipo Promoção
      * @return quantidadeP
      */
    public int getQuantidadeP();
    /**
      * Devolve a quantidade de tipo Normal
      * @return quantidadeN
      */
    public int getQuantidadeN();
    /**
      * Devolve o preço de tipo Promoção
      * @return precoP
      */
    public double getPrecoP();
    /**
      * Devolve o preço de tipo Normal
      * @return precoN
      */
    public double getPrecoN();
    /**
      * Devolve o total faturado
      * @return totalFaturado
      */
    public double getFaturado();
    /**
      * Devolve a quantidade total.
      * @return quantidadeTotal
      */
    public int getQuantidade();
    /**
     * Verifica a igualdade de dois objectos
     * @param objecto
     * @return boolean
     */
    public boolean equals(Object o);
    /**
     * Método que devolve a representação em String da Fatura
     * @return String com quantidadeN, precoN, quantidadeP e precoP
     */
    public String toString();
    /**
     * Cria uma cópia de Fatura
     * @return Fatura
     */
    public Fatura clone();
    /**
     * Verifica se uma Fatura é vazia(não existem quantidades).
     * @return boolean
     */
    public boolean faturaVazia();
    /**
     * Atualiza a Fatura(preço e quantidade) tendo em conta o tipo.
     * @param quantidade, preco, tipo
     */
    public void atualizaFatura(int quantidade, double faturado, String tipo);
}
```

Figura 10 - Interface de Fatura

```java
public interface IVenda{
    /**
      * Devolve o código do produto vendido
      * @return string
      */
    public String getCodProd();
    /**
      * Devolve o código do cliente
      * @return string
      */
    public String getCodCli();
    /**
      * Devolve a quantidade de unidades vendidas
      * @return int
      */
    public int getQuantidade();
    /**
      * Devolve o preço de cada unidade
      * @return double
      */
    public double getPreco();
    /**
      * Devolve o tipo de venda
      * @return string
      */
    public String getTipo();
    /**
      * Devolve o mês da realização da venda
      * @return integer
      */
    public int getMes();
    /**
      * Devolve a filial onde foi realizada a compra
      * @return integer
      */
    public int getFilial();
    /**
     * Devolve o total faturado de uma venda
     * @return double
     */
    public double faturado();
}
```

Figura 9 - Interface de Venda

```java
public interface ICatalogoClientes{
    /**
     * Devolve o TreeSet do CatalogoClientes
     * @return TreeSet de Clientes
     */
    public Collection<Cliente> getClientes();
    /**
     * Devolve o número de clientes lidos
     * @return int
     */
    public int getNumeroClientesLidos();
    /**
     * Adiciona um Cliente ao CatalogoClientes
     * @param Cliente
     */
    public void addCliente(Cliente cliente);
    /**
     * Liberta o CatalogoClientes da memória
     */
    public void libertaCatalogoClientes();
    /**
     * Valida as string de clientes e, caso sejam válidas, cria
     * para cada string um Cliente e adiciona ao CatalogoClientes¢
     * @param coleção de strings
     * @return CatalogoClientes
     */
    public CatalogoClientes validaClientes(Collection<String> clientes);
    /**
     * Determina se um cliente existe no CatalogoClientes
     * @param código do cliente
     * @return boolean
     */
    public boolean clienteExiste (String codCliente);
    /**
     * Verifica a igualdade de dois objectos
     * @param objecto
     * @return boolean
     */
    public boolean equals(Object o);
    /**
     * Metodo que devolve a representação em String do CatalogoClientes
     * @return string
     */
    public String toString();
    /**
     * Cria uma cópia de CatalogoClientes
     * @return CatalogoClientes
     */
    public CatalogoClientes clone();
}
```

Figura 11 – Interface de CatalogoClientes

```java
public interface ICatalogoProdutos{
    /**
     * Devolve o TreeSet do CatalogoProdutos
     * @return TreeSet de Produtos
     */
    public Collection<Produto> getProdutos();
    /**
     * Devolve o número de produtos lidos
     * @return int
     */
    public int getNumProdutosLidos();
    /**
     * Cria uma cópia de CatalogoProdutos
     * @return CatalogoProdutos
     */
    public CatalogoProdutos clone();
    /**
     * Verifica a igualdade de dois objectos
     * @param objecto
     * @return boolean
     */
    public boolean equals(Object o);
    /**
     * Metodo que devolve a representação em String do CatalogoProdutos
     * @return string
     */
    public String toString();
    /**
     * Valida as string de produtos e, caso sejam válidos, cria para cada
     * string um Produto e adiciona ao CatalogoProdutos
     * @param coleção de strings
     * @return CatalogoProdutos
     */
    public CatalogoProdutos validaProdutos (Collection<String> produtos);
    /**
     * Determina se um produto existe no CatalogoProdutos
     * @param código do produto
     * @return boolean
     */
    public boolean existeProduto(String codigoProduto);
    /**
     * Adiciona um Produto ao CatalogoProdutos
     * @param Produto
     */
    public void adicionaProduto(Produto produto);
    /**
     * Liberta o CatalogoProdutos da memória
     */
    public void libertaCatalogoProdutos();
    /**
     * Determina se um produto existe no CatalogoProdutos
     * @param código do produto
     * @return boolean
     */
    public boolean produtoExiste (String codProduto);
}
```

Figura 12 – Interface de CatalogoProdutos

```java
public interface IFaturacao{
    /**
     * Devolve as faturas
     * @return faturasAux
     */
    public Map<String,Matriz> getFaturas();
    /**
     * Método que insere um protudo
     * @param produto
     */
    public void insereProduto(String produto) throws ProdutoJaExisteException;
    /**
     * Método que adiciona uma nova venda
     * @param vendas
     */
    public void adicionaVendas(Collection<Venda> vendas);
    /**
     * Metodo que devolve a representação em String da Faturação
     * @return sb.toString().
     */
    public String toString();
}
```

Figura 13 – Interface de Faturacao

```java
public interface IGestaoFilial{
    /**
     * Metodo que devolve a representação em String do GestaoFilial
     * @return String gestor
     */
    public String toString();
    /**
     * Insere todas as vendas de uma Collection<Venda>
     * @param Collection<Venda>
     */
    public void adicionaVendas (Collection<Venda> vendas);
    /**
     * Liberta o catálogo GestaoFilial da memória
     */
    public void clear();
    /**
     * Retorna os clientes que existem na GestaoFilial
     * @return Set<String>
     */
    public Set<String> getClientes();
    /**
     * Retorna os produtos que existem na GestaoFilial
     * @return Set<String>
     */
    public Set<String> getProdutos(String codCliente);
    /**
     * Retorna uma Fatura de um produto que um cliente comprou num dado mês e numa dada filial
     * @param Strig, String, int, int
     * @return Fatura
     */
    public Fatura getFatura (String codCliente, String codProduto, int mes, int filial);
    /**
     * Determina se um cliente existe na GestaoFilial
     * @param String
     * @return boolean
     */
    public boolean clienteExiste (String codCliente);
    /**
     * Determina se um produto foi comprado por um clientes
     * @param String, String
     * @return boolean
     */
    public boolean produtoExiste (String codCliente, String codProduto);
    /**
     * Determina se um produto existe na GestaoFilial
     * @param String
     * @return boolean
     */
    public boolean produtoExiste(String codProduto);
}
```

Figura 14 – Interface de GestaoFilial

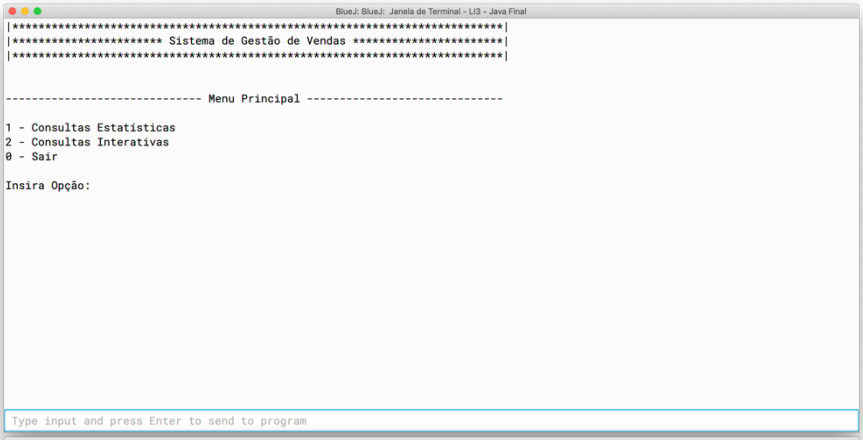


Figura 15 – Menu Principal

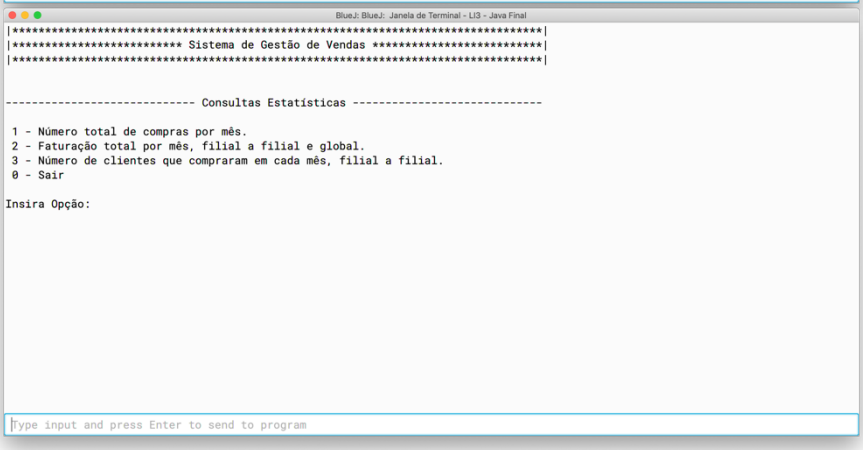


Figura 16 – Menu Consultas Estatísticas

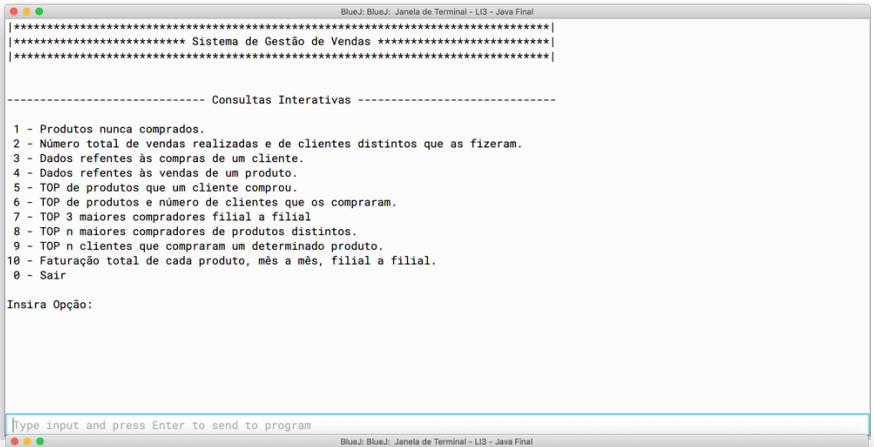


Figura 17 – Menu Consultas Interativas

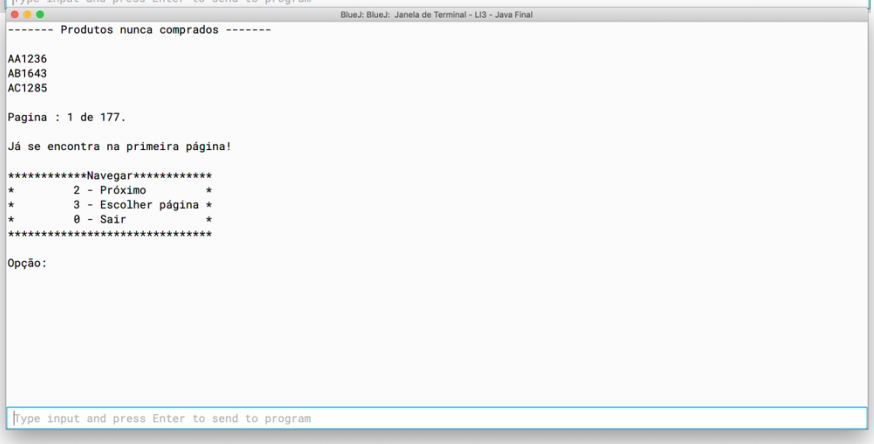


Figura 18 - Navegador

Vendas_1M.txt	Global	Faturacao		GestaoFilial	
		HashMap	TreeMap	HashMaps	TreeMaps
Tempo de Leitura (s)	4.362313419	2.313535266	3.682112316	4.039456385	5.649248815
Espaço Ocupado	1,28 Gb	748,7 Mb	778,7 Mb	970,7 Mb	1,04 Gb

Vendas_3M.txt	Global	Faturacao		GestaoFilial	
		HashMap	TreeMap	HashMaps	TreeMaps
Tempo de Leitura (s)	11.630810797	4.729875781	8.984553029	10.459462206	8.411617217
Espaço Ocupado	3,18 Gb	1,84 Gb	1,76 Gb	3,08 Gb	3 Gb

Vendas_5M.txt	Global	Faturacao		GestaoFilial	
		HashMap	TreeMap	HashMaps	TreeMaps
Tempo de Leitura (s)	20.267162168	7.328850102	13.746023391	17.818617381	31.142259702
Espaço Ocupado	4,30 Gb	2,80 Gb	2,85 Gb	3,94 Gb	4,06 Gb

Clientes.txt	CatalogoClientes		
	HashSet	TreeSet	ArrayList
Tempo de Leitura (s)	0.119218759	0.129544143	0.101828559
Espaço Ocupado	69,1 Mb	78,1 Mb	68,2 Mb

Produtos.txt	CatalogoProdutos		
	HashSet	TreeSet	ArrayList
Tempo de Leitura (s)	0.352287295	0.450247518	0.290276487
Espaço Ocupado	163,4 Mb	139,9 Mb	201,8 Mb

Figura 19 - Medições de Performance