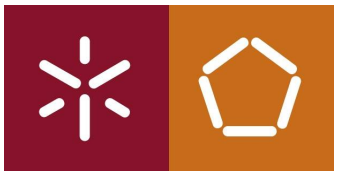


Universidade do Minho

Mestrado Integrado em Engenharia Informática

Laboratórios de Informática III

Relatório do Projeto em Java

Grupo 60:

Ana Rita Rosendo
A84475

Gonçalo Esteves
A85731

Rui Oliveira
A83610

Contents

1 Introdução

No âmbito da Unidade Curricular de Laboratórios de Informática 3, foi-nos proposta a elaboração de um projeto em linguagem de programação Java, cuja finalidade seria a leitura e análise de diversos ficheiros de dados, fornecidos pelos docentes. Estes ficheiros estão divididos em três tipos: ficheiros com códigos de produtos, ficheiros com códigos de clientes, e ficheiros com códigos de vendas. O objetivo do trabalho passa pela leitura e validação dos códigos dados nos ficheiros, e a sua inserção em coleções diversas que permitirão a análise da informação através das 10 *queries* sugeridas pelos docentes, cuja implementação também está ao nosso encargo.

De modo a elaborar o nosso projeto, debruçamo-nos sobre a matéria lecionada na Unidade Curricular de Programação Orientada aos Objetos, aproveitando os conhecimentos adquiridos para elaborar coleções eficientes, capazes de suportar a informação pretendida. Deste modo, foram definidas classes necessárias para o funcionamento do projeto.

2 Classes Definidas

2.1 Produto

```
public class Produto implements IProduto, Serializable{
    private String codigo;
}
```

A classe **Produto** foi criada para guardar a informação de um produto. Esta classe contém uma variável de instância do tipo String que representa o código de um produto.

```
public interface IProduto extends Serializable{

    public String getCodigo();
    public void setCodigo(String cod);
    public String toString();
    public boolean equals(Object o);
    public IProduto clone ();
    public int hashcode();
    public int compareTo(IProduto c);
    public boolean prodValido(String s);
}
```

A interface **IProduto** foi implementada de forma a conter todos os métodos necessários para tratar a informação que define um produto.

2.2 Cliente

```
public class Cliente implements ICliente, Serializable{
    private String codigo;
}
```

A classe **Cliente** foi criada para guardar a informação de um cliente. Esta classe contém uma variável de instância do tipo String que representa o código de um cliente.

```
public interface ICliente extends Serializable{

    public String getCodigo();
    public void setCodigo(String cod);
    public String toString();
    public boolean equals(Object o);
    public ICliente clone ();
    public int hashcode();
    public int compareTo(ICliente c);
    public boolean cliValido(String s);
}
```

A interface **ICliente** foi implementada de forma a conter todos os métodos necessários para tratar a informação que define um cliente.

2.3 CatProd

```
public class CatProd implements ICatProd, Serializable{
       private Set<IProduto> catProd;
}
```

A classe **CatProd** foi implementada com o intuito de guardar a informação de um catálogo de produtos. Esta classe dispõe de uma variável de instância, um Set de **IProduto**, que representa o catálogo de produtos.

```
public interface ICatProd extends Serializable{

    public Set<IProduto> getCatProd();
    public void setCatProd(Set<IProduto> pro);
    public CatProd clone();
    public boolean equals(Object o);
    public String toString();
    public int hashCode();
    public void addProduto(IProduto pro);
    public boolean existeProduto(IProduto pro);
    public int numeroProdutos();
}
```

A interface **ICatProd** foi implementada de forma a conter todos os métodos necessários para tratar a informção que define um catálogo de produtos.

2.4 CatCli

```
public class CatCli implements ICatCli, Serializable{
    private Set<ICliente> catCli;
}
```

A classe **CatCli** foi criada de modo a guardar a informação de um catálogo de clientes sendo que, para tal, foi usado um Set de **ICliente**.

```
public interface ICatCli extends Serializable{

    public Set<ICliente> getCatCli();
    public void setCatCli(Set<ICliente> cli);
    public CatCli clone();
    public boolean equals(Object o);
    public String toString();
    public int hashCode();
    public void addCliente(ICliente cli);
    public boolean existeCliente(ICliente cli);
    public int numeroClientes();
}
```

A interface **ICatCli** foi implementada de forma a conter todos os métodos necessários para tratar a informação que define um catálogo de clientes.

2.5 Fatura

```
public class Fatura implements IFatura, Serializable{
    IProduto codProd;
    double preco;
    int quantidade;
    String tipo;
}
```

A classe **Fatura** foi criada de modo a guardar a informação de uma fatura. Sendo assim, esta classe contém uma varável de instância que representa o código do produto, duas variáveis de instância que representam o preço e a quantidade

de produto vendido, e uma variável de instância que representa o tipo de venda (em promoção ou normal).

```
public interface IFatura extends Serializable{

    public IProduto getCodProd();
    public double getPreco();
    public int getQuantidade();
    public String getTipo();
    public void setCodProd(IProduto codigo);
    public void setPreco(Double preco);
    public void setQuantidade(int quantos);
    public void setTipo(String tipo);
    public String toString();
    public boolean equals(Object o);
    public IFatura clone ();
    public int hashcode();
    public int compareTo(IFatura f);
}
```

A interface **IFatura** foi implementada de forma a conter todos os métodos necessários para tratar a informação que define uma fatura.

2.6 Faturacao

```
public class Faturacao implements IFaturacao, Serializable{
    private Map<IProduto, List<IFatura>> faturacao;
}
```

A classe **Faturacao** foi criada para guardar a informação de uma faturação. Sendo assim, esta classe contém uma variável de instância, um Map, que a um determinado produto associa a sua lista de faturas.

```
public interface IFaturacao extends Serializable{
    public Map<IProduto, List<IFatura>> getFaturacao();
    public void setFaturacao(Map<IProduto, List<IFatura>> fat);
    public Faturacao clone();
    public boolean equals(Object o);
    public String toString();
    public int hashCode();
    public void addValues(IProduto key, IFatura value);
    public List<IFatura> determinadaFaturas(IProduto p);
    public boolean existeFatura(IProduto pro);
    public List<IParProdQuantos> topXMaisVendidos(int x);
    public int prodsVendidos();
    public SimpleEntry<Integer, Double> comprasAOEFaturacaoTotal
    ();
}
```

A interface **IFaturacao** foi implementada de forma a conter todos os métodos necessários para tratar a informação que define uma faturação.

2.7 FilialAux

```
public class FilialAux implements IFilialAux, Serializable{
    private IProduto codProd;
    private double preco;
    private int quantidade;
    private String tipo;
    private ICliente codCli;
    private int mes;
}
```

A classe **FilialAux** foi criada de modo a guardar toda a informação de uma venda necessária à filial. Sendo assim, esta classe, contém várias variáveis de instância que representam o código do produto, o preço do produto, a quantidade do produto, o tipo de venda, o código do cliente que efetuou a compra e o mês em que o mesmo realizou a compra.

```
public interface IFilialAux extends Serializable{
    public IProduto getCodProd();
    public double getPreco();
    public int getQuantidade();
    public String getTipo();
    public ICliente getCodCli();
    public int getMes();
    public void setCodProd(IProduto codigo);
    public void setPreco(Double preco);
    public void setQuantidade(int quantos);
    public void setTipo(String tipo);
    public void setCodCli(ICliente codigo);
    public void setMes(int mes);
    public String toString();
    public boolean equals(Object o);
    public IFilialAux clone ();
    public int hashcode();
    public boolean contemProduto(IProduto p, List<IFilialAux> x);
    public boolean contemCliente(ICliente c, List<IFilialAux> x);
    public List<IFilialAux> produtosIguais(IProduto p, List<
    IFilialAux> x);
}
```

A interface **IFilialAux** foi implementada de forma a conter todos os métodos necessários para tratar a informação que define uma venda com a informação necessária à filial.

2.8 Filial

```
public class Filial implements IFilial, Serializable{
    private Map<ICliente, List<IFilialAux>> filial;
}
```

A classe **Filial** foi criada para guardar a informação de todas as vendas de uma filial. Sendo assim, esta classe contém uma variável de instância, um Map, que a um determinado cliente associa a sua lista de vendas efetuadas relativas a uma determinada filial.

```
public interface IFilial extends Serializable{

    public Map<ICliente, List<IFilialAux>> getFilial();
    public void setFilial(Map<ICliente, List<IFilialAux>> f);
    public Filial clone();
    public boolean equals(Object o);
    public String toString();
    public int hashCode();
    public void addValues(ICliente key, IFilialAux value);
    public List<IFilialAux> determinadaFilial(ICliente c);
    public SimpleEntry<Integer, List<ICliente>> vendasPorMes(int
    mes);
    public List<ICliente> todosCliComprProd(IProduto pro);
    public Map<ICliente, Double> getTop3();
    public ICliente cliMenosGasto(Map<ICliente, Double> map);
    public List<IProduto> numeroProdsDiferentes(ICliente cli);
    public ITrioComprasProdGasto determinaEstatisticas();
    public Set<ICliente> clientesCompradores();
    public Map<IProduto, double[]> todosProdsEFat();
}
```

A interface **IFilial** foi implementada de forma a conter todos os métodos necessários para tratar a informação que define todas as vendas de uma filial.

2.9 Venda

```
public class Venda implements IVenda, Serializable{
    private IProduto codProd;
    private double preco;
    private int quantidade;
    private String tipo;
    private ICliente codCli;
    private int mes;
    private int filial;
}
```

A classe **Venda** foi criada para guardar a informação de uma venda. Sendo assim, esta classe contém várias varáveis de instância que representam o código do produto vendido, o preço e a quantidade de produto vendido, o tipo de venda (Promoção ou Normal), o código do cliente que efetuou a compra e o mês e a filial em que a venda foi realizada.

```
public interface IVenda extends Serializable{

    public IProduto getCodProd();
    public double getPreco();
    public int getQuantidade();
    public String getTipo();
    public ICliente getCodCli();
    public int getMes();
    public int getFilial();
    public void setCodProd(IProduto codigo);
    public void setPreco(Double preco);
    public void setQuantidade(int quantos);
    public void setTipo(String tipo);
    public void setCodCli(ICliente codigo);
    public void setMes(int mes);
    public void setFilial(int filial);
    public String toString();
    public boolean equals(Object o);
    public IVenda clone ();
    public int hashcode();
    public IVenda linhaToVenda(String linha, ICatCli catCli,
    ICatProd catProd);
    public IFatura vendaToFatura();
    public IFilialAux vendaToFilial();
}
```

A interface **IVenda** foi implementada de forma a conter todos os métodos necessários para tratar a informação que define uma venda.

2.10 GereVendasModel

```
public class GereVendasModel implements InterGereVendasModel,
    Serializable{
    private ICatProd catProd;
    private ICatCli catCli;
    private IFaturacao faturacao;
    private IFilial filial1;
    private IFilial filial2;
    private IFilial filial3;
    String fichVendas;
    int vendasInvalidas;
}
```

A classe **GereVendasModel** foi criada para realizar as querys. Esta classe é consituída por todos os algoritmos e é responsável por realizar os métodos correspondentes às querys e enviar os resultados para a classe **GeraVendas-Controller**.

```
public interface InterGereVendasModel extends Serializable{

    public ICatCli getCatCli();
    public void setCatCli(ICatCli cli);
    public ICatProd getCatProd();
    public void setCatProd(ICatProd pro);
    public IFaturacao getFaturacao();
    public void setFaturacao(IFaturacao fat);
    public IFilial getFilial1();
    public void setFilial1(IFilial  f);
    public IFilial  getFilial2();
    public void setFilial2(IFilial  f);
    public IFilial  getFilial3();
    public void setFilial3(IFilial  f);
    public InterGereVendasModel clone();
    public boolean equals(Object o);
    public String toString();
    public int hashCode();

    /*
         ***** METODOS DE LEITURAS DE FICHEIROS *****
    */
```

```
    public void lerFichWithBuff_Produtos(String fichtxt);
    public void lerFichWithBuff_Clientes(String fichtxt);
    public void lerFichWithBuff_Vendas(String fichtxt);

    /*
         ***** METODOS AUXILIARES DE CARREGAMENTO DE DADOS *****
    */

    public void carregarDados(String fichVendas);

    /*
         ***** METODOS QUERIES *****
    */

    public List<IProduto> query1();
    public int[] query2(int mes);
    public ITrioComprasProdGasto query3(ICliente c);
    public ITrioComprasProdGasto query4(IProduto p);
    public List<IParProdQuantos> query5(ICliente c);
    public List<IParProdQuantos> query6(int x);
    public List<Map<ICliente, Double>> query7();
    public List<IParCliQuantos> query8(int x);
    public List<IParCliQuantos> query9(IProduto p, int x);
    public List<Map<IProduto, double[]>> query10();
    public String estatistica1();
    public List<ITrioComprasProdGasto> estatistica2();
    public List<IProduto> catProdEmLista();
}
```

A interface **InterGereVendasModel** foi implementada de forma a conter todos os métodos necessários para tratar a informação das diversas querys.

2.11 GereVendasView

```
public class GereVendasView implements InterGereVendasView,
    Serializable{
    private List<String> opcoes;
}
```

A classe **GereVendasView** foi criada para servir de "conversor" dos resultados das querys. Isto é, depois do utilizador escolher a opção que deseja, a classe

GeraVendasController envia a esta classe o resultado da opção escolhida e esta classe mostra ao utilizador o resultado.

```
public interface InterGereVendasView extends Serializable{

    public List<String> getOpcoes();
    public void setOpcoes(String[] opcoes);
    public GereVendasView clone();
    public int menu_Inicial();
    public int menu_Queries();
    public int menu_Estatisticas();
    public int menu_Ficheiros();
    public void query1(List<IProduto> l)
    public void query2(int mes, int[] total);
    public void query3(ITrioComprasProdGasto t, String cliente);
    public void query4(ITrioComprasProdGasto t, String produto);
    public void query5(List<IParProdQuantos> p);
    public void query6(List<IParProdQuantos> aux);
    public void query7(List<Map<ICliente, Double>> lis);
    public void query8(List<IParCliQuantos> lis);
    public void query9(List<IParCliQuantos> aux);
    public void query10(List<Map<IProduto, double[]>> aux, List<
    IProduto> s);
    public void estatisticas2(List<ITrioComprasProdGasto> trio,
    int n);
    public void estatisticas1(String s);
}
```

2.12 GereVendasController

```
public class GereVendasController implements
    InterGereVendasController, Serializable{
    private InterGereVendasModel model;
    private InterGereVendasView view;
}
```

A classe **GereVendasController** é responsável por fazer uma ligação entre a classe **GeraVendasView** e a classe **GeraVendasModel** contribuindo para o bom funcionamento do projeto. Sendo assim, esta classe é constituída pelas interfaces das classes antes referidas.

Depois do utilizador ecolher a opção que deseja, a classe **GeraVendasView** envia à classe **GeraVendasController** a opção solicitada, esta classe informa à classe **GeraVendasModel** o que o utilizador deseja, a classe **GeraVendasModel** calcula o resultado e envia à classe **GeraVendasController** o resultado. Posto isto, a mesma envia o resultado à classe **GeraVendasView** que mostra ao utilizador o mesmo.

```
public interface InterGereVendasController extends Serializable{

    public void setModel(InterGereVendasModel x);
    public void setView(InterGereVendasView y);
    public InterGereVendasModel getModel();
    public InterGereVendasView getView();
    public void startController();
    public void lerFicheiros();
    public void consultarQueries();
    public void abreEstadoGuardado();
    public void guardaEstadoFicheiro();
}
```

2.13 GeraVendasAppMVC

```
public class GeraVendasAppMVC{
    public static void main(String[] args){
        InterGereVendasModel model = new GereVendasModel();
        model.carregarDados("Vendas_1M.txt");  //metodo que vai
    ler os ficheiros de dados e guardar no model

        if(model == null){
            out.println("Erro a ler os ficheiros.\n");
            System.exit(-1);
        }

        InterGereVendasView view = new GereVendasView();

        InterGereVendasController control = new
    GereVendasController();

        control.setModel(model);
        control.setView(view);
        control.startController();
        //control.exit(0);
    }
}
```

A classe **GeraVendasAppMVC** é a classe principal. Nesta classe, são inicializados os métodos necessários ao bom funcionamento do projeto. É criada uma interface **InterGeraVendasModel**, os dados são carregados, é criada uma interface **InterGeraVendaView** e uma interface **InterGereVendasController**.

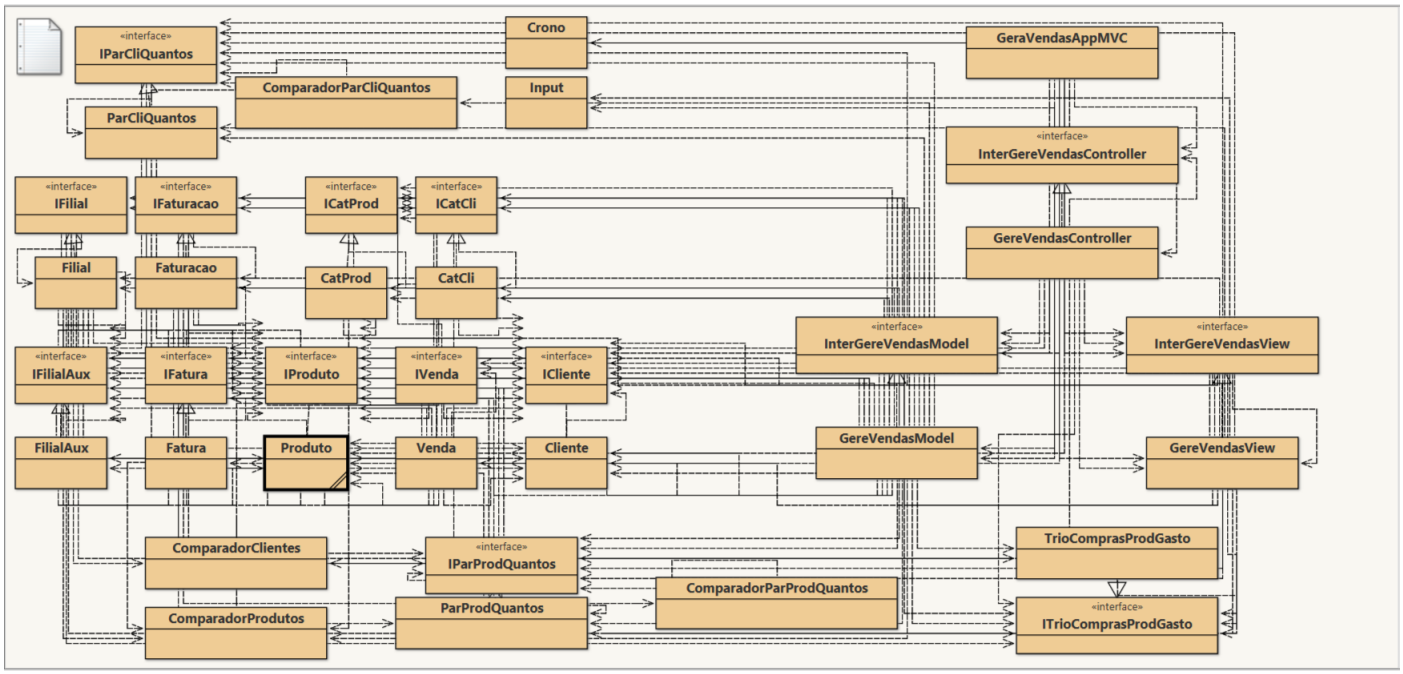


Figure 1: Esquema geral do projeto

3 Queries

Como foi referido anteriormente, a classe **GeraVendasModel** é responsável por definir as queries. Sendo assim, todas as seguintes queries foram definidas na classe **GeraVendasModel**.

3.1 Query 1

A query 1 determina a lista ordenada alfabeticamente com os códigos dos produtos nunca comprados e o seu respectivo total.

Nesta *query* foi criado um método que devolve uma lista de **IProduto** que possui os produtos nunca comprados. Para tal, a partir do **CatProd**, foi criado um Set de **IProduto**. Após termos o conjunto dos produtos, percorremos o mesmo e verificamos se existe uma fatura correspondente a esse produto na faturação. Caso exista, adicionamos esse produto à lista de **IProduto**.

3.2 Query 2

A query 2, dado um mês válido, determina o número total global de vendas realizadas e o número total de clientes distintos que as fizeram. O utilizador pode aceder aos resultados filial a filial ou para todas as 3.

Nesta *query*, foi criado um método que devolve um array de inteiros que possui na primeira, terceira e quinta posição o número de vendas nas filiais 1, 2 e 3, respetivamente; que possui na segunda, quarta e sexta posição o número de clientes que compraram nas filiais 1, 2 e 3, respetivamente; e que possui na sétima posição o número de clientes que fizeram compras em pelo menos uma filial.

Primeiramente, foram criadas 3 coleções auxiliares, para cada filial, do tipo SimpleEntrySet onde o número total de vendas e os clientes que as realizaram num determinado mês foram guardados, recorrendo a um método definido na classe **Filial**. Seguidamente, foi criada uma lista de **ICliente** onde foram inseridos os clientes das coleções auxiliares não permitindo repetições.

Por fim, de modo a preencher o array, para as posições zero, dois e quatro fomos buscar as keys das coleções auxiliares de cada filial, para as posições um, três e cinco fomos buscar o tamanho dos values das coleções auxiliares de cada filial e para a posição seis fomos buscar o tamanho da lista com todos os cliente que efetuaram as vendas sem repetições.

3.3 Query 3

A query 3, dado um código de um cliente, determina, para cada mês, quantas compras o cliente fez, quantos produtos distintos comprou e quanto gastou no total.

Nesta *query* foi criado um método que devolve uma interface **ITrioComprasProdGasto** que representa a informação correspondente ao número de compras, ao número de produtos comprados e ao valor total gasto. Esta é constituída por um array de inteiros que guarda o número de compras efetuadas em cada mês, um array de inteiros que guarda os produtos comprados em cada mês e um array de doubles que guarda o valor gasto em cada mês, sendo que cada índice de um array corresponde ao mês.

Primeiro, foram definidos dois arrays de inteiros e um array de doubles, todos com um tamanho igual a 12, arrays esses que guardam o número de compras efetuadas, número de produtos comprados e o valor total gasto em cada mês. De seguida, foi definida uma lista de **IFilialAux** denominada produtosDistintos. Posteriormente, criamos uma lista de **IFilialAux** recorrendo ao método da classe **Filial** chamado determinadaFilial(String codCliente). Depois percorremos essa lista e incrementamos os valores que estão na posição equivalente ao mês tanto no array de compras efetuadas como no array de valor gasto sendo que obtemos essa posição recorrendo ao método getMes() da classe **IFilialAux**. Posteriormente, inserimos na lista produtosDistintos os produtos que ainda não estão presentes na lista e incrementamos o valor da posição equivalente ao mês do array de produtos comprados. O processo repete-se para cada filial.

Por fim, definimos uma interface **ITrioComprasProdGasto** recorrendo ao construtor por parâmetros sendo que os parâmetros são a lista com o número de compras efetuadas, a lista com o número de produtos comprados e a lista com o valor gasto.

3.4 Query 4

A query 4, dado um código de um produto existente, determina, mês a mês, quantas vezes foi comprado, por quantos clientes diferentes e o total facturado.

Nesta *query* foi criado um método que devolve uma interface **ITrioComprasProdGasto** que representa a informação correspondente ao número de vezes que o produto foi comprado, ao número de clientes distintos que compraram esse produto e ao valor total faturado. Esta é constituída por um array de inteiros que guarda o número de compras efetuadas em cada mês, um array de inteiros que guarda os produtos comprados em cada mês e um array de doubles que guarda o valor gasto em cada mês, sendo que cada índice de um array corresponde ao mês.

Primeiro, foram definidos dois arrays de inteiros e um array de doubles, todos com um tamanho igual a 12, arrays esses que guardam o número de vezes que o produto foi comprado, número de clientes distintos que compraram o produto e o valor total faturado em cada mês. De seguida, foi definida uma interface **IFilialAux** auxiliar, uma lista de **IFilialAux** denominada clientesDistintos e uma lista de **IFilialAux** denominada lista. Depois percorremos os values da filial em questão e se o produto p estiver aí presente , colocamos em lista esse produto recorrendo ao método produtosIguais(IProduto p, List¡**IFilialAux**) definido na classe **FilialAux**. Depois percorremas a lista denominada lista e incrementamos

os valores que estão na posição equivalente ao mês tanto no array que guarda o número de vezes que o produto foi comprado como no array que guarda o valor total faturado sendo que obtemos essa posição recorredno ao método getMes() da classe **IFilialAux**. Posteriormente, inserimos na lista clientesDistintos os produtos que ainda não estão presentes na lista e incrementamos o valor da posição equivalente ao mês do array de clientes que compraram o produto. O processo repete-se para cada filial.

Por fim, definimos uma interface **ITrioComprasProdGasto** recorrendo ao construtor por parâmetros sendo que os parâmetros são a lista com o número de vezes que o produto foi comprado, a lista com o número de clientes que compraram o produto e a lista com o valor total faturado.

3.5 Query 5

A query 5, dado o código de um cliente, determina a lista de códigos de produtos que mais comprou (e quantos), ordenada por ordem decrescente de quantidade e, para quantidades iguais, por ordem alfabética dos códigos.

Para esta *query*, inicialmente, criamos uma lista de **IProduto's** e uma lista de pares do tipo **IParProdQuantos**, que guardam o código de um produto e a sua quantidade vendida.

Em primeiro lugar, acedemos às listas de **IFilialAux's** do cliente em questão(ou seja, para a filial 1,2 e 3) e concatenamo-las numa só, já ordenada de forma alfabética, recorrendo a uma **stream** e, mais precisamente, ao seu método **sorted**, usando a ordem natural como comparação de **IFilialAux's**. Após isso, percorremos essa lista concatenada de **IFilialAux's** e elemento a elemento, verificamos se o produto correspondente existe na nossa lista inicial de **IProduto's**. Caso não exista, inserimo-lo nessa lista e também na lista de pares, bem como a sua quantidade vendida. Se pelo contrário, ele já estiver na nossa lista de **IProduto's**, quer dizer que já o temos guardado também nalista de pares, logo, só precisamos de atualizar a sua quantidade vendida, somando a que já temos guardada à dada pela **IFilialAux** que estamos a analisar.

Por fim, tendo já a nossa lista de pares com todos os produtos comprados pelo cliente dado, vamos ordenar a mesma por ordem decrescente de quantidade, recorrendo novamente a uma **stream** e ao seu método **sorted**, mas desta vez usando o método **comparing** da classe **Comparator**.

3.6 Query 6

A query 6 determina o conjunto dos X produtos mais vendidos em todo o ano (em número de unidades vendidas) indicando o número total de distintos clientes que o compraram (X é um inteiro dado pelo utilizador);

Por forma a elaborarmos esta *query*, comecamos inicialmente por determinar a lista dos X produtos mais comprados. Para tal, utilizamos o método **topX-**

MaisVendidos da nossa **IFaturacao**, que para um dado número X, retorna uma lista com os X produtos mais comprados e a sua quantidade, armazenados num par **IParProdQuantos** (neste caso não será necessária, mas é assim mais fácil ordenar a lista tendo em conta o número de compras).

Posto isto, precisamos agora de determinar quantos clientes diferentes compraram cada um dos produtos. Assim sendo, para cada **IParProdQuantos** que retiramos da lista recebida previamente, vamos buscar a lista de *ICliente* que compraram o *IProduto* em cada uma das filiais e intersetar as listas, originando a lista com todos os *ICliente* que compraram esse produto. Após isto, só temos de gerar um novo *IParProdQuantos* que será adicionado a uma lista a retornar, em que o *IProduto* será o que foi agora analisado, e a quantidade será , para este caso, o número total de **ICliente** que compraram o produto (bastando, para tal, simplesmente obter o tamanho da lista de **ICliente** determinada).

Por fim, retornamos uma lista de **IParProdQuantos** que associa cada *IProduto* à quantidade de **IClientes** que o compraram.

3.7 Query 7

A query 7 determina, para cada filial, a lista dos três maiores compradores em termos de dinheiro facturado.

Para a resolução desta *query*, foi criada uma lista de **Map**'s que guardava como chave um **ICliente** e como valor um double.

De seguida criamos um método auxiliar, que se encontra na classe **Filial**(**getTop3()**), que para cada filial, percorre-a, e para cada uma delas devolve um map do género que implementamos inicialmente. Esse map contém os 3 clientes que mais gastaram nessa filial e, além disso, a quantidade de dinheiro que a filial faturou com as compras dos mesmos.

Por fim, após percorrermos todas as filiais através do método auxiliar, adicionamos cada map retornado à nossa lista inicial, obtendo assim uma lista que contém os 3 maiores clientes de cada filial, bem como os 3 valores monetários faturados correspondentes.

3.8 Query 8

A query 8 determina os códigos dos X clientes (sendo X dado pelo utilizador) que compraram mais produtos diferentes (não interessa a quantidade nem o valor), indicando quantos, sendo o critério de ordenação a ordem decrescente do número de produtos.

Nesta *query*, criamos 3 listas de **IProduto's** e uma lista de pares do tipo **IParCliQuantos**.

Percorremos então o catálogo de clientes e para cada cliente, com ajuda de um método auxiliar que se encontra na classe **Filial**(**numeroProdsDiferentes(ICLiente cli)**), conseguimos obter uma lista com todos os diferentes produtos que o cliente que estamos a analisar comprou, para cada filial.

Percorremos então as duas listas devolvidas para a filial 2 e 3 e caso os produtos das mesmas não se encontram na lista devolvida para a filial 1, adicionamo-los à mesma, de modo a obter a lista de todos os produtos difentes.

Por conseguinte, adicionamos à nossa lista de pares o cliente que estamos a analisar e o tamanho da lista de todos os produtos diferentes. Fazemos este processo para todos os clientes do catálogo.

Por fim, devolvemos a nossa lista de pares, limitando o seu tamanho ao valor X dado pelo utilizador.

3.9 Query 9

A query 9, dado o código de um produto que deve existir, determina o conjunto dos X clientes que mais o compraram e, para cada um, qual o valor gasto (ordenação cf. 5).

Com o objetivo de resolução desta *query*, foi construído um método tal que, dado um código de produto, devolve o conjunto dos X **IClientes** que mais o compraram e, para cada um deles, qual o valor gasto. Para a ordenação foi usada a mesma utilizada que na *query* 5. Inicialmente, criamos então duas listas, uma de **ICliente's** e outra de pares do tipo **IParCliQuantos**.

O algoritmo usado foi o mesmo que na *query* 5, com a diferença de que em vez de concaternarmos as 3 listas de **IFilialAux's** do cliente dado para as filiais 1, 2 e 3, fizemos o mesmo processo, porém, para cada uma dessas listas de forma individual. Ou seja, à medida que vemos cada lista, verificamos se existe ou não o cliente correspondente à **IFilialAux** que estamos a analisar e, conforme a sua validade, inserimos o mesmo e a sua quantidade(atualizando-a caso o cliente já exista na nossa lista inicial de **ICliente's**) na nossa lista de pares a devolver no final.

Após termos a nossa lista de pares do tipo **IParCliQuantos** com todos os clientes que compraram determinado produto, ordenamo-la conforme a ordenação da *query* 5, limitando o seu tamanho ao X dado pelo utilizador.

3.10 Query 10

A query 10 determina mês a mês, e para cada mês filial a filial, a facturação total com cada produto.

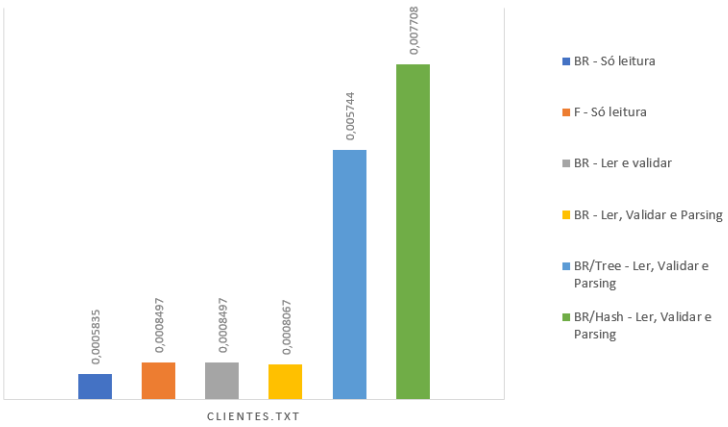
Nesta *query* foi criado um método que devolve uma lista de **Map's** que a um determinado **IProduto** associa um array de doubles que corresponde ao valor total da faturação em cada mês.

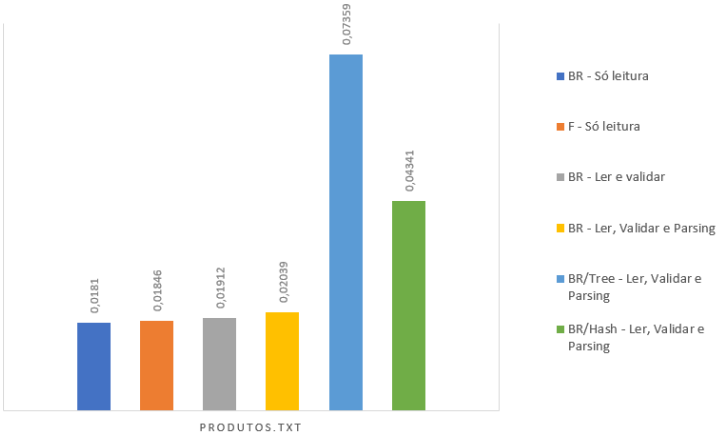
Em primeiro lugar, foi definido um Map, para cada filial cujos nomes variam entre fil1, fil2 e fil3, cuja chave é um **IProduto** e o valor é um array de doubles correspondente ao valor total da faturação de cada mês sendo que esse **Map** é obtido recorrendo ao método todosProdsEFat() da classe **Filial**. Em segundo lugar, foi definido uma nova lista de **Map's** cuja chave é um **IProduto** e o

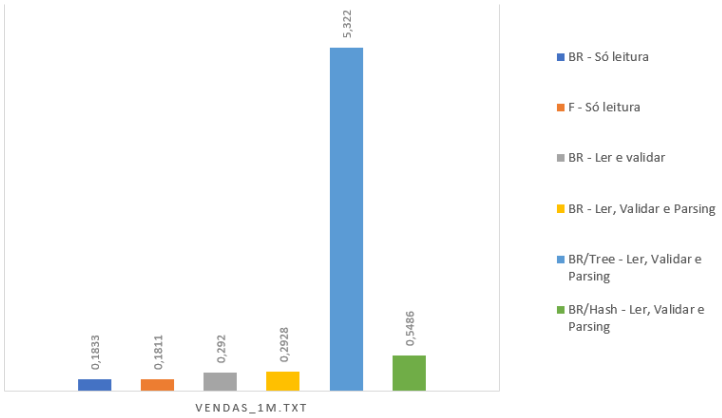
valor é um array de doubles correspondente ao valor total da faturação de cada mês, denominado ret. Por último, é adicionado à lista ret todos os **Map's** fil1, fil2 e fil3 definidos no início.

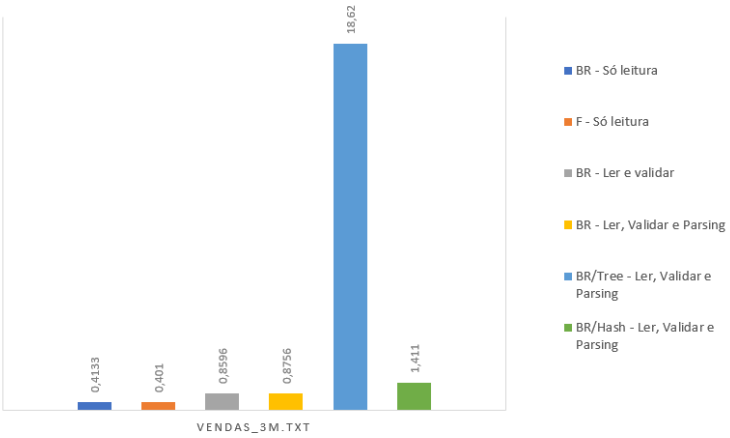
4 Medidas de Performance

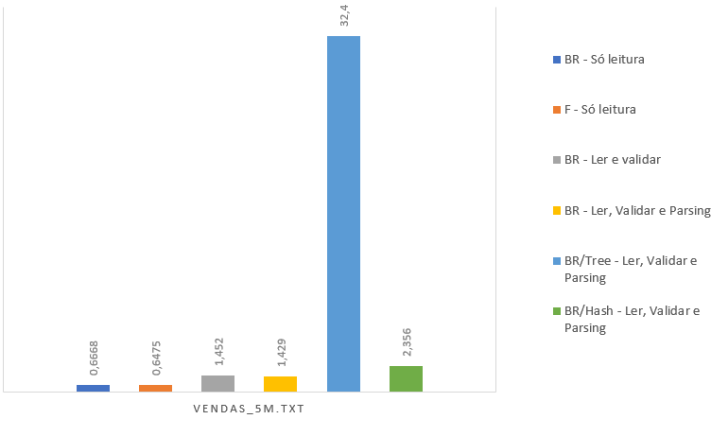
Com o intuito de comparar os resultados obtidos com diferentes métodos, não só de leitura de ficheiros, mas também de armazenamento de dados, elaboramos os seguintes gráficos. Uma vez que os tempos obtidos por **Buffered Readers** e **Files** são bastante parecidos, realizamos os restantes testes usando apenas **Buffered Readers**. Para além dos tempos de leitura com **Buffered Readers** e **Files**, analisamos também os tempos de: leitura e validação da informação; leitura, validação da informação e *parsing* da mesma; e leitura, validação da informação, *parsing* da mesma e inserção da informação nas estruturas respetivas. Quanto à inserção da informação também analisamos dois casos: a inserção em *TreeMaps* e *TreeSets*, e a inserção em *HashMaps* e *HashSets*. Para a determinação dos tempos, foi calculada a média para cada situação a partir de 10 testes diferentes. Como tal, elaboramos os seguintes gráficos:


0,0005835
0,0008497
0,0008497
0,0008067
0,005744
0,007708
CLIENTES.TXT
BR - Só leitura
F - Só leitura
BR - Ler e validar
BR - Ler, Validar e Parsing
BR/Tree - Ler, Validar e Parsing
BR/Hash - Ler, Validar e Parsing


0,0181
0,01846
0,01912
0,02039
0,07359
0,04341
PRODUTOS.TXT
BR - Só leitura
F - Só leitura
BR - Ler e validar
BR - Ler, Validar e Parsing
BR/Tree - Ler, Validar e Parsing
BR/Hash - Ler, Validar e Parsing


0,1833
0,1811
0,292
0,2928
5,322
0,5486
VENDAS_1M.TXT
BR - Só leitura
F - Só leitura
BR - Ler e validar
BR - Ler, Validar e Parsing
BR/Tree - Ler, Validar e Parsing
BR/Hash - Ler, Validar e Parsing


18,62
0,4133
0,401
0,8596
0,8756
1,411
VENDAS_3M.TXT
BR - Só leitura
F - Só leitura
BR - Ler e validar
BR - Ler, Validar e Parsing
BR/Tree - Ler, Validar e Parsing
BR/Hash - Ler, Validar e Parsing


32,4
0,6668
0,6475
1,452
1,429
2,356
VENDAS_5M.TXT
BR - Só leitura
F - Só leitura
BR - Ler e validar
BR - Ler, Validar e Parsing
BR/Tree - Ler, Validar e Parsing
BR/Hash - Ler, Validar e Parsing

Através da interpretação dos gráficos, podemos, tal como afirmado em cima, concluir que os tempos de leitura para **Buffered Reader** e **Files** são similares. Tendo isto por base, e sabendo à partida que teriamos de validar a informação nos ficheiros linha a linha, optamos por usar o **Buffered Reader**, uma vez que, deste modo, podemos validar e instanciar (quando as validações dão resultados positivos) cada linha, logo a seguir a esta ter sido lida, poupando o esforço de memória de guardar todas as linhas num *array*, e só posteriormente as validar e instanciar.

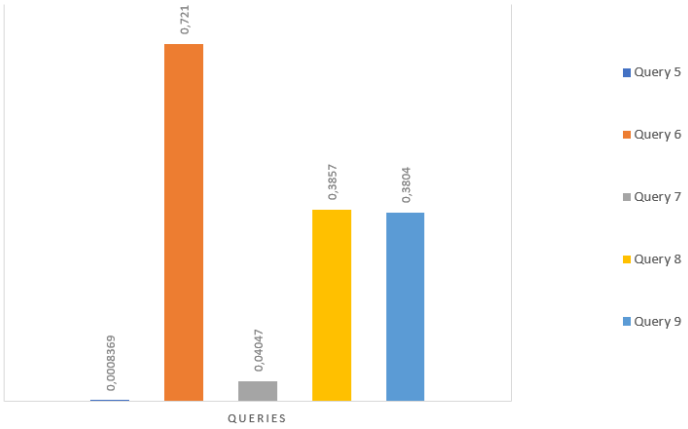
Posto isto, podemos também afirmar que as validações e instanciações da informação não interferem em muito nos tempos de execução, uma vez que os tempos obtidos com e sem validação/instanciação são praticamente iguais.

Por fim, debruçamo-nos na interpretação dos resultados obtidos aquando da inserção da informação em estruturas de dados, podendo, logo à partida, concluir que estão aqui resididas as maiores diferenças temporais, não só em relação aos tempos previamente analisados, mas também comparando tempos entre **Hash** e **Tree** Maps/Sets.

Começando por comparar os tempos sem inserção com os tempos com inserção, encontramos logo diferenças bastantes consideráveis. Podemos, assim, concluir logo que a grande maioria do tempo utilizado no processamento da informação dos ficheiros é na inserção dos dados em estruturas.

No entanto, são ainda mais consideráveis as variações de tempo utilizado quando comparado o tempo utilizado na inserção em **TreeMaps/Sets** e o tempo utilizado na inserção em **HashMaps/Sets**. Tal como é indicado nos gráficos, o tempo de inserção em estruturas de **Hash** é consideravelmente mais reduzido, permitindo o armazenamento mais rápido de informação em memória.

Apesar disto, esta aparente diferença positiva das estruturas de **Hash** em deterimento das **Tree** é bastante atenuada na invocação das *queries* que o programa suporta. Isto deve-se ao facto de que como grande parte destas queries envolvem pesquisas nas árvores, sabemos à partida de que estas serão mais eficientes usando **TreeMaps** e **TreeSets**, que permitem o uso de *Comparators*, que tornam possível a inserção ordenada da informação, agilizando, posteriormente, a pesquisa. Assim sendo, chegamos aos seguintes tempos para as *queries* 5 a 9:



5 Conclusão

Após a elaboração deste projeto, podemos afirmar que aumentamos o nosso nível de conhecimento, não só no que toca ao domínio da linguagem de programação Java, mas também à compreensão de determinados conceitos, nomeadamente o encapsulamento de dados e a otimização de código. Com o desenvolvimento deste trabalho, fomo-nos deparando com diversos obstáculos que se colocaram à nossa frente, sendo necessário não só fazer uso de conhecimentos previamente adquiridos, mas também de outros novos, conseguidos através de uma procura nas fontes disponíveis.