Projeto de Laboratórios de Informática 3

Grupo 52

Carlos Miguel Luzia de Carvalho A89605

Francisco Correia Franco A89458

José Pedro Carvalho Costa A89519

A89605

A89458

A89519

Índice

Introdução

Nesta unidade curricular foi-nos proposta a implementação de um Sistema de Gestão de Vendas (SGV) com base na leitura de ficheiros disponibilizados pelos docentes.

A segunda fase deste projeto consistiu no desafio de implementar este sistema na linguagem Java. Apesar do intuito de executar as funcionalidades do programa de forma rápida, a modularidade e o encapsulamento das estruturas por nós utilizadas eram prioridade.

Estas preocupações vão de encontro à aprendizagem realizada neste semestre no âmbito da programação orientada aos objetos.

Uma vez que a primeira fase deste projeto fora a implementação deste sistema em C, já tínhamos em mente as estruturas a utilizar para guardar as informações necessárias, só tínhamos de definir as classes.

Descrição dos Módulos

A arquitetura do software é definida por 4 módulos principais: Catálogo de Clientes, Catálogo de Produtos, Faturação global e Vendas por filial, cujas fontes de dados são os 3 ficheiros de texto Produtos.txt, Clientes.txt e Vendas_1M.txt e uma interface que permita a comunicação com o cliente.

Cada linha do ficheiro Produtos.txt representa um código de produto e é formado por 2 letras maiúsculas seguidas de 4 dígitos.

Cada linha do ficheiro Clientes.txt representa um código de cliente e é formado por 1 letra maiúscula seguida de 4 dígitos.

Cada linha do ficheiro Vendas_1M.txt representa uma venda efetuada e é formada por: código de produto, preço unitário decimal, número inteiro de unidades compradas, tipo de compra, código de cliente, mês da compra e filial.

Catálogo de Produtos

Módulo de dados onde são guardados todos os produtos do ficheiro Produtos.txt.

Primeiro criamos uma classe que conteria a informação relativa a um só produto, neste caso uma string. Esta classe serve para caso no futuro seja adicionada mais informação a um produto, só precisamos de alterar esta classe.

Os dados são guardados numa HashMap com 26 posições referentes às 26 letras do alfabeto. Cada índice contém um apontador para um Set de Produto de uma dada letra associada a esse índice. Deste modo procuramos tornar a procura de um certo produto mais eficiente.

Tipos de dados criados

```
public class Produtos3 implements InterfaceProdutos, Serializable
{
    // instance variables - replace the example below with your own
    private Map<Integer,Set<Produto>> produtos;
```

```
public class Produto implements Serializable
{
    private String produto;
```

Catálogo de Clientes

Módulo de dados onde são guardados todos os de produtos do ficheiro Clientes.txt.

Primeiro criamos uma classe que conteria a informação relativa a um só cliente, neste caso uma string. Esta classe serve para caso no futuro seja adicionada mais informação a um cliente, só precisamos de alterar esta classe.

Os dados são guardados numa Tabela de Hash com 26 posições referentes às 26 letras do alfabeto. Cada índice contém um apontador para um array com os códigos de clientes de uma dada letra associada a esse índice. Deste modo procuramos tornar a procura de um certo cliente mais eficiente.

A estrutura deste módulo é quase igual á do catálogo de produtos, só que em vez de guardar produtos guarda clientes, não achamos necessário criar uma estrutura diferente para estes dois módulos devido á sua semelhança.

```
public class Clientes3 implements InterfaceClientes, Serializable
{
    // instance variables - replace the example below with your own
    private Map<Integer,Set<Cliente>> clientes;
```

```
public class Cliente implements Serializable
{
    private String cliente;
```

Faturação global

Módulo de dados que irá conter as estruturas de dados responsáveis pela resposta eficiente a questões quantitativas que relaciona os produtos às suas vendas mensais ou globais, em modo Normal (N) ou em Promoção (P). Este módulo deve referenciar todos os produtos, mesmo os que nunca foram vendidos. Tal como nos produtos iniciamos a estrutura Faturação com uma classe Faturação que contém uma HashMap com 26 posições correspondentes a uma letra do alfabeto. Para cada uma dessas posições esta associada uma List<ProdFat que contém uma string de produto e uma HashMap de filiais. Esta HashMap tem tamanho 3 e cada umas das posições esta associada a uma classe FilFat, que incorpora uma HashMap de 12 posições que correspondem aos meses e um inteiro que verifica se está ocupado. Dentro de cada mês,

representado pela lasse MesFat temos então a informação necessária para efetuar a faturação de um produto.

Com esta organização conseguimos detetar certas vantagens tais como a eficaz procura de informação referente a um produto, graças ao fato da lista de produtos se encontrar ordenado, porém certas desvantagens devido ao grande número de estruturas usadas.

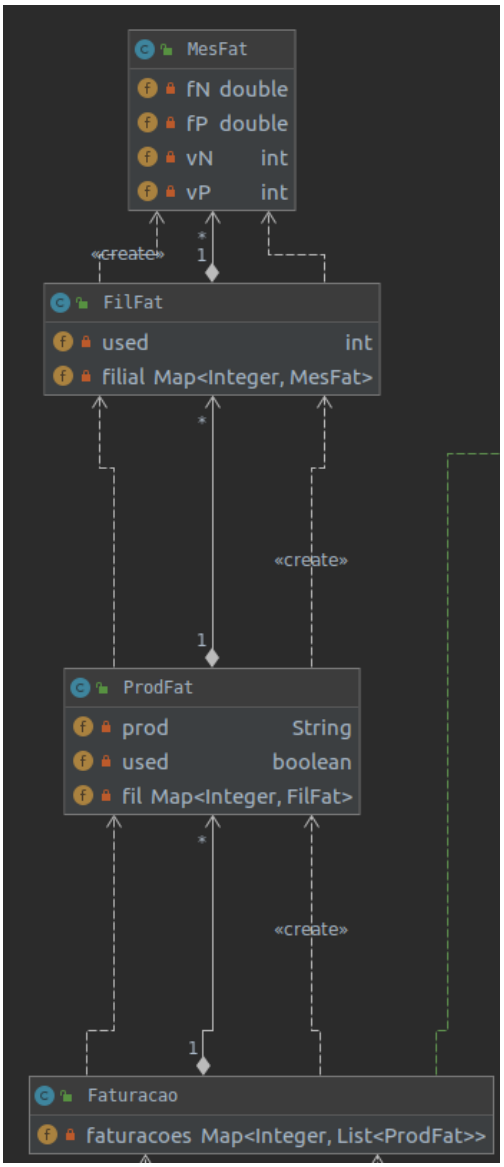
```
public class Faturacao implements InterfaceFaturacao, Serializable {
    private Map<Integer, List<ProdFat>> faturacoes;
```

```
public class ProdFat implements Serializable {

    private String prod;
    private boolean used;
    private Map<Integer, FilFat> fil;
```

```
public class FilFat implements Serializable {
    private int used;
    private Map<Integer, MesFat> filial;
```

```
public class MesFat implements Serializable {
    private double fN;
    private double fP;
    private int vN;
    private int vP;
```



Double fN – Faturação em N;

Double fP – Faturação em P;

Int vN – Vendas em N;

Int vP – Vendas em P

Filial

Módulo de dados que, a partir das leituras, conterá estruturas de dados adequadas á representação das relações entre produtos e clientes, ou seja, para cada cliente, saber quais os produtos que compraram, indicando quantidades, e tipo de compras em cada mês e filial. Esta estrutura assemelhasse bastante á estrutura faturação com a única diferença de em vez de receber produtos recebe clientes e é lhe acrescentada mais uma classe dentro de cada mês, sendo esta uma HashMap que contem a informação de compras realizadas por um certo cliente.

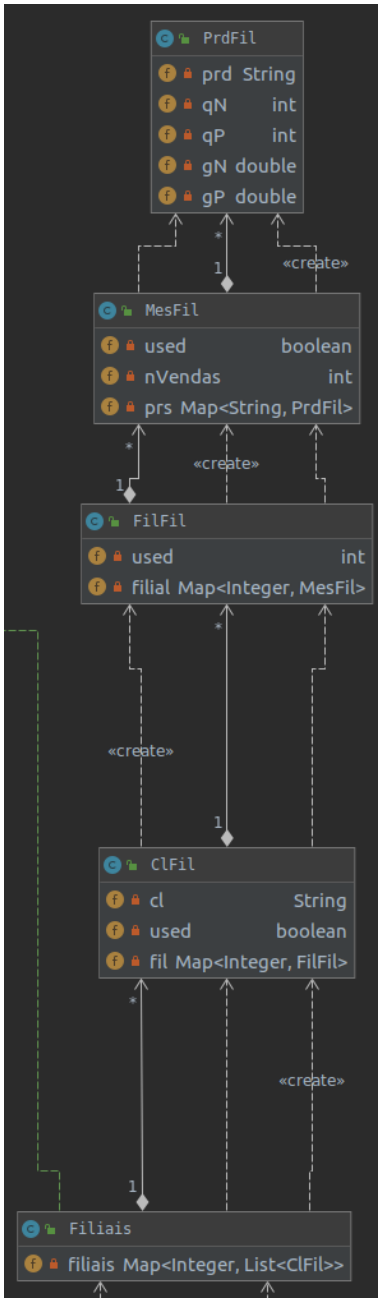
```
public class Filiais implements InterfaceFiliais, Serializable {
    private Map<Integer, List<ClFil>> filiais;
```

```
public class ClFil implements Serializable {
    private String cl;
    private boolean used;
    private Map<Integer, FilFil> fil;
```

```
public class FilFil implements Serializable {
    private int used;
    private Map<Integer, MesFil> filial;
```

```
public class MesFil implements Serializable {
    private boolean used;
    private int nVendas;
    private Map<String,PrdFil> prs;
```

```
public class PrdFil implements Serializable {
    private String prd;
    private int qN;
    private int qP;
    private double gN;
    private double gP;
```



Double gN –Gasto em N;

Double gP – Gasto em P;

Int qN – Quantidade em N;

Int qP – Quantidade em P;

String pr – Código de produto;

Modelo, Fluxo e Apresentação

O programa é controlado pela função main na classe GestVendas, a qual chama o interpretador (controlador), onde são implementadas os métodos respetivas ao modelo, ou seja, todos os métodos que são usadas na classe Queries incluindo as próprias. São estes métodos que leem, executam e armazenam toda a informação necessária para uso da api.

Relativamente á apresentação esta é moderada pelo controlador (o qual recebe informação necessária para dar resposta aos pedidos) e dá display no ecrã o output da informação pedida pelo utilizador.

De forma a exemplificar o contato entre o modelo o fluxo e a apresentação temos então, numa fase inicial a interface começa por imprimir no ecrã todas as opções de consulta dos módulos de acordo com as queries, sendo que não é possível escolher nenhuma sem antes ler os ficheiros (Querie 1). Ao selecionar uma opção é pedido ao utilizador que insira os dados necessários para a consulta dos módulos e então são chamados os métodos de consulta que recebem os dados inseridos como parâmetros. Estes métodos guardam os resultados Maps e Sets e depois enviam cópias dos resultados de volta para a interface para que esta imprima os resultados da procura no ecrã.

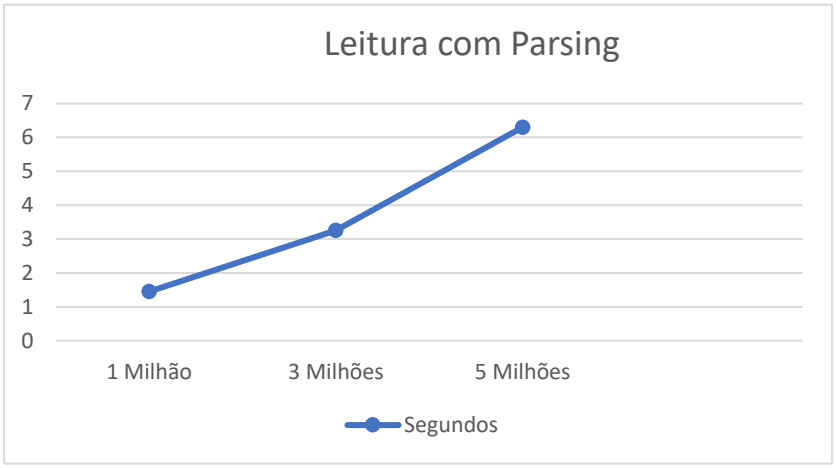
Testes de performance

Posteriormente ao desenvolvimento e codificação de todo o projeto, foi-nos proposto realizar alguns testes de performance relativamente á leitura dos ficheiros de vendas 1M, 3M e 5M e ao tempo de execução de cada querie para diferentes módulos.

1. Tempos de leitura sem parsing;

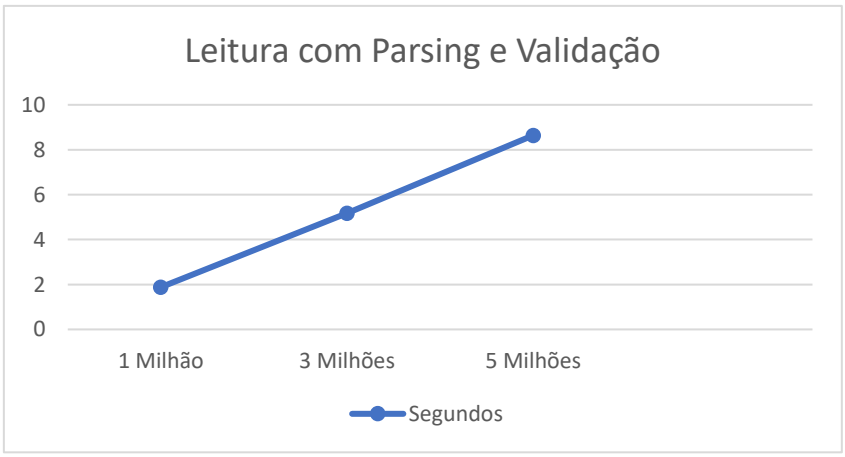
Vendas_1M.txt	0,78 Segundos
Vendas_3M.txt	1,45 Segundos
Vendas_5M.txt	3,9 Segundos

2. Tempos de leitura com parsing;



Vendas_1M.txt	1,45 Segundos
Vendas_3M.txt	3,25 Segundos
Vendas_5M.txt	6,3 Segundos

3. Tempos de leitura com parsing e validação;

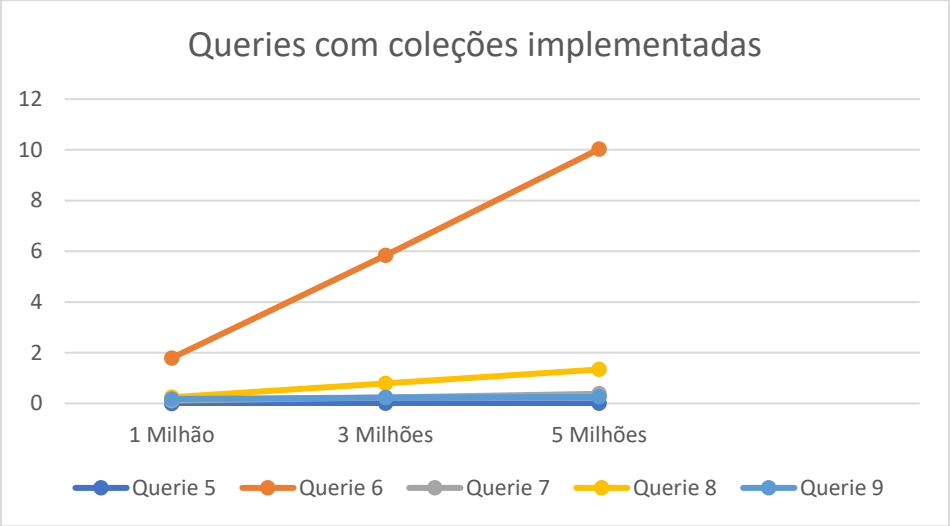


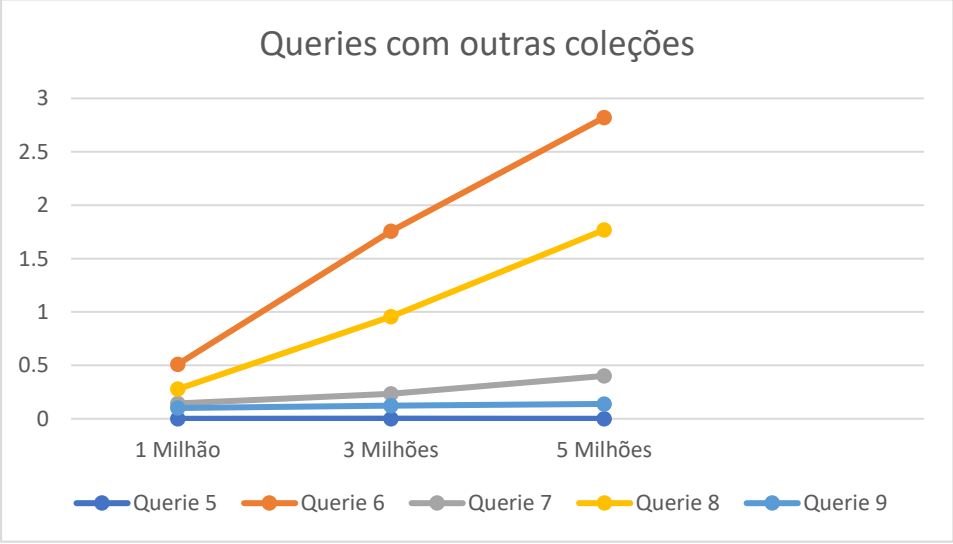
Vendas_1M.txt	1,88 Segundos
Vendas_3M.txt	5,17 Segundos
Vendas_5M.txt	8,64 Segundos

Em primeiro lugar é de notar que estes tempos foram obtidos numa máquina com um processador de 1.8GHz e 8GB de RAM, pelo que em computadores com especificações diferentes destas, é de se esperar resultados diferentes.

Com a análise destes três gráficos concluímos que o tempo que demoramos a ler os ficheiros e a executar estas 3 queries tem tendência a aumentar, o que consideramos aceitável uma vez que a quantidade de vendas vai aumentando de ficheiro para ficheiro.

4. Testes de Performance das Queries;

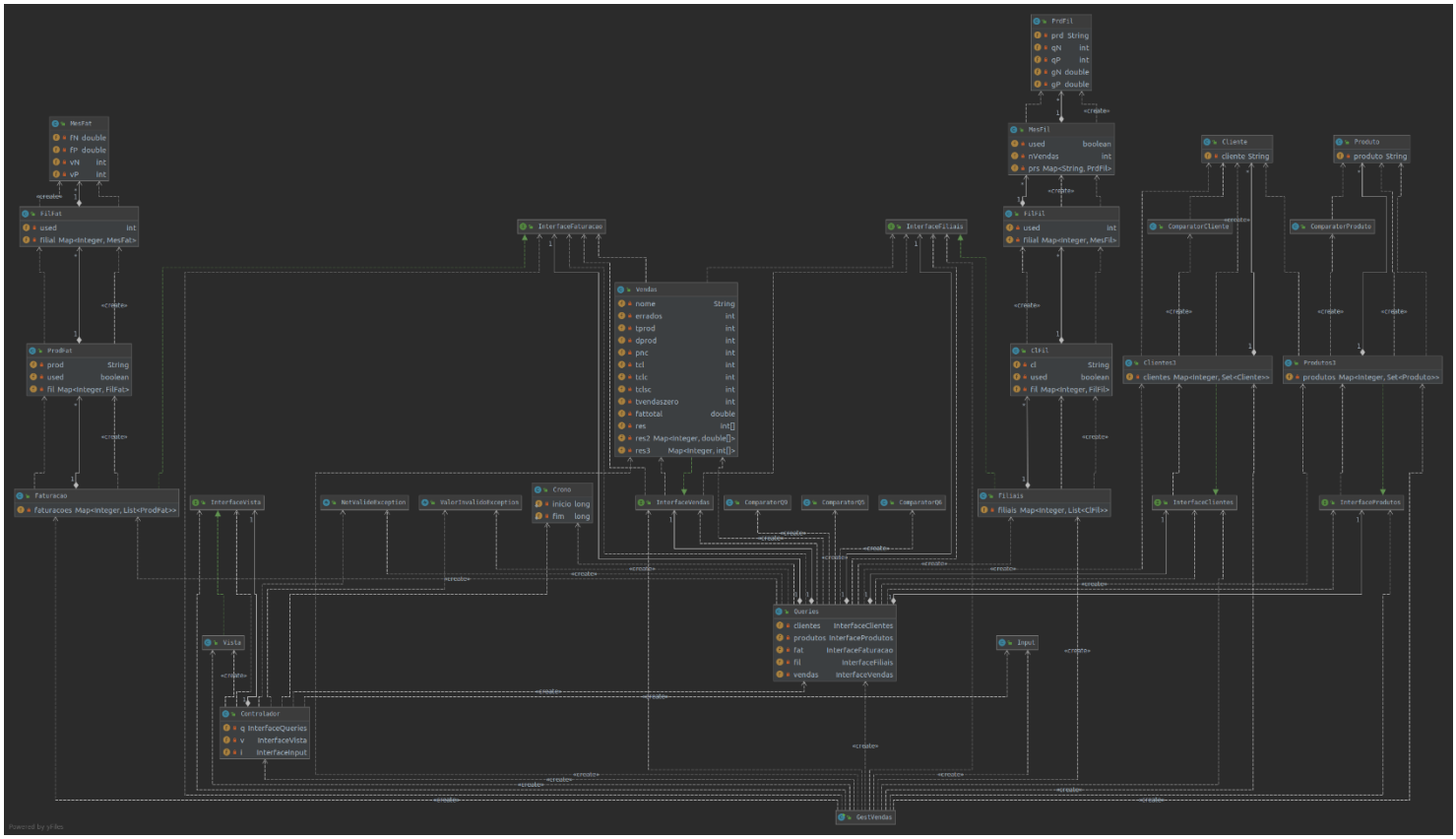




É de notar que os testes de performance das queries foram realizados numa máquina com um processador de 1.8GHz e 16GB de RAM, pelo que em computadores com especificações diferentes destas, é de se esperar resultados diferentes.

Com a observação destes tempos podemos concluir que HashMaps e HashSets são mais rápidas do que TreeMaps e TreeSets, respetivamente, mas não é possível responder ás queries utilizando a estrutura Hash pois estas não permitem ordenação, pelo que as estruturas finais das queries são Trees.

Diagrama de Classes



Conclusão

A realização deste projeto em java por um lado mostrou-se mais desafiante pois é uma linguagem que aprendemos este semestre, mas por outro lado as questões do encapsulamento, modularidade e definição de estruturas, não foram problema por já termos trabalhado com elas na primeira fase em C. Trabalhar em java facilita a parte da gestão de memória que é sempre uma preocupação em C, já que não nos temos de preocupar em alocar e libertar memória e evitam-se os memory leaks e segmentation faults.

Em retrospetiva, com esta segunda fase concluímos que é mais rápido fazer o programa em Java, já que temos uma variedade de bibliotecas prontas á disposição, mas é mais eficiente em C.