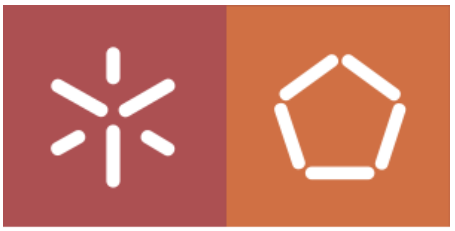


Universidade Do Minho
Departamento de Informática

GestReviewsAppMVC
Grupo nº 23

Alexandre Martins (A93242)
Francisco Neves (A93202)
Miguel Martins (A89584)

19 de Junho de 2021

Índice

1. Introdução

O projeto realizado tem como objetivo a gestão de dados em larga escala sem o apoio de bases de dados externas na linguagem, Orientada aos Objetos, *Java*, sendo que, para tal ser possível, deverá ser necessária a introdução de vários princípios de programação adequados à programação em larga escala, designadamente:

- ➔ Modularidade e encapsulamento de dados usando construções da linguagem;
- ➔ Criação de código reutilizável;
- ➔ Escolha otimizada das estruturas de dados e reutilização;
- ➔ Teste de performance e até profiling.

De forma a obter os dados necessários aos requisitos do projeto, foi necessária a implementação de diferentes estruturas de dados consideradas ideais para os problemas em questão.

2. Interpretação dos problemas apresentados

O projeto desenvolvido tem como fonte de dados três ficheiros *.csv* que podem ser fornecidos pelo utilizador, ou, poderão ser utilizados os ficheiros *.csv* predefinidos, ou seja, os ficheiros fornecidos pela equipa docente. Estes ficheiros são os seguintes:

- ➔ *users.csv* - Ficheiro que contém os utilizadores, é dividido em três parâmetros divididos por ‘;’;
- ➔ *business.csv* - Ficheiro que contém os negócios, é dividido em cinco parâmetros divididos por ‘;’;
- ➔ *reviews.csv* - Ficheiro que contém as avaliações dos negócios, dividido em 9 parâmetros divididos por ‘;’.

Tendo em conta estes ficheiros, o programa deve ser capaz de responder a 10 diferentes questões levantadas pela equipa docente:

1. Lista ordenada alfabeticamente com os identificadores dos negócios nunca avaliados e o seu respetivo total;
2. Dado um mês e um ano (válidos), determinar o número total global de reviews realizadas e o número total de *users* distintos que as realizaram;
3. Dado um código de utilizador, determinar, para cada mês, quantas *reviews* fez, quantos negócios distintos avaliou e que nota média atribuiu;
4. Dado o código de um negócio, determinar, mês a mês, quantas vezes foi avaliado, por quantos *users* diferentes e a média de classificação;
5. Dado o código de um utilizador determinar a lista de nomes de negócios que mais avaliou (e quantos), ordenada por ordem decrescente de quantidade e, para quantidades iguais, por ordem alfabética dos negócios;
6. Determinar o conjunto dos X negócios mais avaliados (com mais *reviews*) em cada ano, indicando o número total de distintos utilizadores que o avaliaram (X é um inteiro dado pelo utilizador);
7. Determinar, para cada cidade, a lista dos três mais famosos negócios em termos de número de reviews;
8. Determinar os códigos dos X utilizadores (sendo X dado pelo utilizador) que avaliaram mais negócios diferentes, indicando quantos, sendo o critério de ordenação a ordem decrescente do número de negócios;
9. Dado o código de um negócio, determinar o conjunto dos X users que mais o avaliaram e, para cada um, qual o valor médio de classificação (ordenação cf. 5);
10. Determinar para cada estado, cidade a cidade, a média de classificação de cada negócio.

Além destas questões, o programa deve ser também capaz de responder às seguintes questões estatísticas:

1. Apresenta ao utilizador os dados referentes aos últimos ficheiros lidos, designadamente, nome do ficheiro, número total de registos de *reviews* errados, número total de negócios, total de diferentes negócios avaliados (nº *reviews* > 0), total de não avaliados, número total de *users* e total dos que realizaram *reviews*, total de *users* que nada avaliaram, total de *users* inativos (sem *reviews*) e total de *reviews* com 0 impacto (0 valores no somatório de *cool*, *funny* ou *useful*).
2. Apresenta em ecrã ao utilizador os números gerais respeitantes aos dados actuais já registados nas estruturas, designadamente:
    a. Número total de *reviews* por mês;

b. Média de classificação de *reviews* por mês e o valor global (média global de *reviews*);
c. Número de distintos utilizadores que avaliaram em cada mês (não interessa quantas vezes avaliou).

Conforme a nossa interpretação, o objetivo principal deste projeto seria a aprendizagem de boas técnicas de desenvolvimento de software reutilizável para a manipulação de grandes volumes de dados, sem esquecer o encapsulamento e a modularidade.

3. Arquitetura do Projeto

Este projeto adota a estrutura *MVC* (*Model*, *View*, *Controller*) recomendada pela equipa docente.

1. Model

O *model* é a parte do *software* responsável pela inserção, manipulação e remoção da informação do programa. As diversas classes do *model* são:

1. *Business.java*
   Responsável por guardar a informação e responder a questões simples (por exemplo, a informação de um campo) de um negócio;
2. *Businesses,java*
   Responsável por guardar todos os negócios. A estrutura de dados utilizada para este propósito foi uma *Hashtable* da API *java.util*;
3. *User.java*
   Responsável por guardar a informação e responder a questões simples (por exemplo, a informação de um campo) de um utilizador;
4. *Users.java*
   Responsável por guardar todos os utilizadores. A estrutura de dados utilizada para este propósito foi uma *Hashtable* da API *java.util*;
5. *Review.java*
   Responsável por guardar a informação e responder a questões simples (por exemplo, a informação de um campo) de uma avaliação;
6. *Reviews.java*
   Responsável por guardar todas as avaliações. A estrutura de dados utilizada para este propósito foi uma *Hashtable* da API *java.util*;
7. *Query2Pair.java*
   Classe auxiliar de forma a guardar o resultado da *query 2* de forma mais legível;
8. *Query3Triple.java*
   Classe auxiliar de forma a guardar o resultado da *query 3* de forma mais legível;
9. *Query4Triple.java*
   Classe auxiliar de forma a guardar o resultado da *query 4* de forma mais legível;
10. *GestReviews.java*
    Classe principal que tem como objetivo estabelecer o *model* geral do programa.

2. View

O *view* é a parte do *software* responsável pela criação dos outputs do programa. O único módulo do *view* é:

1. *Outputs.java*
   Responsável pela criação dos outputs para o utilizador (por exemplo, a formatação de tabelas).

3. Controller

O *controller* é a parte do *software* responsável pelo controlo do programa. Os módulos do *controller* são:

1. *Controller.java*
   Responsável pelo controlo do programa;
2. *Crono.java*
   Classe fornecida pela equipa docente que permite medir os tempos de execução das *queries*;

Além do projeto principal, foi também criado um projeto de testes que nos permitiu verificar qual a estrutura de dados mais adequada para cumprir os objetivos do programa e se alguns dos métodos implementados funcionavam corretamente. Nomeadamente:

- *BusinessesHashMap.java*
  *C*lasse criada com o intuito de verificar a eficiência de um catálogo de *Businesses* no formato de um *HashMap*.
- *BusinessesHashSet.java*
  Classe criada com o intuito de verificar a eficiência de um catálogo de *Businesses* no formato de um *HashSet*.
- *UsersHashMap.java*
  *C*lasse criada com o intuito de verificar a eficiência de um catálogo de *Users* no formato de um *HashMap*.
- *UsersHashSet.java*
  Classe criada com o intuito de verificar a eficiência de um catálogo de *Users* no formato de um *HashSet*.
- *ReviewsHashMap.java*
  *C*lasse criada com o intuito de verificar a eficiência de um catálogo de *Reviews* no formato de um *HashMap*.
- *ReviewsHashSet.java*
  *C*lasse criada com o intuito de verificar a eficiência de um catálogo de *Reviews* no formato de um *HashSet*.
- *BusinessTest.java*
  Classe de testes unitários de métodos da classe *Business*.
- *BusinessesTest.java*
  Classe de testes unitários de métodos da classe *Businesses*.
- *ReviewTest.java*
  Classe de testes unitários de métodos da classe *Review*.
- *ReviewsTest.java*
  Classe de testes unitários de métodos da classe *Reviews*.
- *UserTest.java*
  Classe de testes unitários de métodos da classe *User*.

- *UsersTest.java*
  Classe de testes unitários da classe *Users*.

4. Estruturas de Dados

Pela nossa interpretação, esta deveria ser uma das partes mais cruciais para a execução eficiente do programa, dessa forma, decidimos utilizar a estrutura que melhor se adequou aos nossos desejos nas classes de teste.

Para o carregamento dos ficheiros as estruturas utilizadas foram:

1. *Businesses*

Hashtable fornecida pela API *java.util* que associa a cada *business_id* uma classe *Business* sendo que esta é definida por:

   a. *String id*
   b. *String name;*
   c. *String city;*
   d. *String state;*
   e. *String categories;*

2. *Users*[1]

Hashtable fornecida pela API *java.util* que associa a cada *user_id* uma classe *User* sendo que esta é definida por:

   a. *String id;*
   b. *String name.*

3. *Reviews*

Hashtable fornecida pela API *java.util* que associa a cada *user_id* uma classe *User* sendo que esta é definida por:

   a. *String review_id;*
   b. *String user_id;*
   c. *String business_id;*
   d. *float stars;*
   e. *int useful;*
   f. *int funny;*
   g. *int cool;*
   h. *LocalDateTime data;*
   i. *String text;*

Para a estrutura utilizada, ou seja, a *HashTable* da API *java.util*, tanto o tempo de remoção como de inserção é dado por ***O***(1).

[1] Conforme indicado pela equipa docente, os *friends* de cada *user* são desprezados.

A estratégia utilizada para a resolução de cada uma das questões levantadas foi a seguinte:

- ➔ *Query1*

  Inicialmente, a partir das *reviews*, percorrendo todo o seu catálogo, obtemos um conjunto de *ids* dos *businesses* que tiveram avaliação. Em seguida, obtemos um conjunto de todos os *businesses ids* existentes no catálogo. A partir destes dois conjuntos filtramos os elementos que estão presentes no segundo e não no primeiro, ordenando os seus elementos conforme pedido através de uma *stream*, por fim, passa-se o conjunto para lista.

- ➔ *Query2*

  Percorre-se todo o catálogo das *reviews* e filtra-se aquelas que pertencem ao mês e ao ano desejado, adicionando-as a um conjunto. Em seguida, coloca-se o tamanho do conjunto no primeiro elemento do *Query2Pair* e, através de uma *stream*, conta-se o número de *user ids* distintos e coloca-se no segundo elemento do par.

- ➔ *Query3*

  Percorre-se todo o catálogo das *reviews* e filtra-se aquelas que pertencem ao *user* com o *id* passado por argumento. Em seguida, gera-se um *Map* que associa, a partir da lista anterior, as *reviews* por mês. Por fim, percorre-se todo este *Map* e vai-se gerando o *Query3Triple* de cada valor com os dados necessários.

- ➔ *Query4*

  Percorre-se todo o catálogo das *reviews* e filtra-se aquelas que pertencem ao *business* com o *id* passado por argumento. Em seguida, gera-se um *Map* que associa, a partir da lista anterior, as *reviews* por mês. Por fim, percorre-se todo este *Map* e vai-se gerando o *Query4Triple* de cada valor com os dados necessários.

- ➔ *Query5*

  Inicialmente, percorre-se todos os *businesses* e gera-se um *Map* que associa a cada *business id* o seu nome. Em seguida, percorre-se todo o catálogo de *reviews* e para aquelas que pertencerem ao *user* com o *id* passado por argumento, atualiza-se um *Map* auxiliar que associa a cada nome de *business* o número de vezes que foi avaliado. Em seguida, ordena-se a lista de nomes de negócios conforme o desejado.

- ➔ *Query6*

  Inicialmente, obtém-se um *Map* que associa a cada ano uma lista dos *ids* de *reviews* realizadas nesse ano. Além disso, obtém-se também percorrendo o catálogo das *reviews* um *Map* que associa a cada *business id* uma lista com os *ids* de *reviews* feitas a esse *business*. Em seguida vai-se percorrendo estes *Maps* e obtendo-se a informação desejada.

- ➔ *Query7*

  Inicialmente, a partir do catálogo de *businesses* obtém-se um *Map* que associa a cada cidade um conjunto dos *businesses ids* da cidade. Em seguida, através do catálogo das *reviews* obtém-se um *Map* que associa a cada *business id* o número de vezes que ele foi avaliado. Por fim, vai-se percorrendo o primeiro *Map* e, através da informação presente no segundo, obtém-se a informação pedida.

- ➔ *Query8*
  Inicialmente, obtém-se uma lista de todas as *reviews* presentes no catálogo. Em seguida, através de uma *stream* vai-se obtendo os dados desejados.
- ➔ *Query9*
  Percorre-se todo o catálogo das *reviews* e obtém-se uma lista com todas as *reviews* do *business* com o *id* passado por argumento. Em seguida, obtém-se um *Map* que associa a cada *id* de *user* o número de vezes que avaliou negócios e a nota média atribuída. Por fim, percorre-se estes dados e obtém-se a informação desejada.
- ➔ *Query10*
  Inicialmente, obtém, através do catálogo de *businesses* um *Map* que associa os *businesses* por cidade e estado. Em seguida, através do catálogo das *reviews* obtém-se um *Map* que associa a cada *id* de *business* a nota média das suas avaliações. Por fim, percorre-se estes dados e obtém-se a informação desejada.

A nível das questões estatísticas a técnica utilizada foi a seguinte:

- ➔ *Query Estatística 1.*
  Dividiu-se em métodos auxiliares mais pequenos e acrescentou-se, por vezes, algumas variáveis de instância à classe *GestReviews* de forma a ser possível ir atualizando os valores à medida do avanço do programa.
- ➔ *Query Estatística 2.*
  a. Através do catálogo das *reviews* obtém-se um *Map* que associa o número de *reviews* realizadas a cada mês.
  b. Através do catálogo das *reviews* obtém-se um par em que o primeiro elemento representa a média global de *reviews* e o segundo contém um par que associa a cada mês o valor médio das classificações das *reviews* desse mês.

5. Outras Funcionalidades

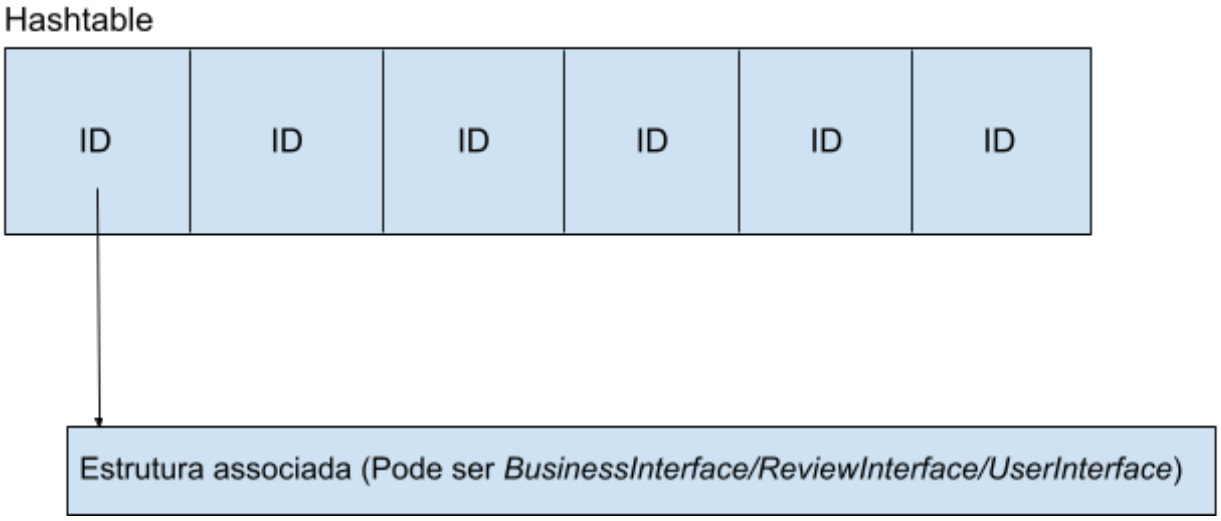
Além das respostas às queries, o programa é também capaz de ler ficheiros *.csv* que respeitem as normas dos ficheiros fornecidos pela equipa docente, a gravação do estado do programa num ficheiro de objetos (na diretoria *saves*, o nome do *save* pode ser fornecido pelo utilizador ou será gravado com o nome *default* "*gestReviews.dat"*) e a leitura de um ficheiro de objetos presente na diretoria *saves*.

6. Testes de Performance

Os testes foram executados num computador com as seguintes especificações: CPU - Intel i7-8750H @ 4.100GHz com 15863MiB de memória no OS *Manjaro Linux x86_64*.

Leitura dos ficheiros *default*	17.85 s
Query 1	0.36 s
Query 2 (mês 9 do ano 2010)	0.13 s
Query 3 (user “YoVfDbnISlW0f7abNQACIg“)	0.08 s
Query 4 (business “RA4V8pr014UyUbDvI-LW2A”)	0.08 s
Query 5 (user “YoVfDbnISlW0f7abNQACIg“)	0.14 s
Query 6 (top 100)	2.83 s
Query 7	0.34 s
Query 8 (top 100)	2.02 s
Query 9 (business “RA4V8pr014UyUbDvI-LW2A” - top 20)	0.06 s
Query 10	0.42 s
Estatísticas 1	0.80 s
Estatísticas 2 - a	0.22 s
Estatísticas 2 - b	0.23 s
Estatísticas 2 - c	0.24 s

7. Esboços das Estruturas Utilizadas



8. Diagrama de Classes[2]

a. Diagrama do Programa Principal

Diagrama de Classes gerado pelo *IntelliJ*

[2] Tendo em conta a quantidade de classes, o diagrama torna-se demasiado grande, o que poderá dificultar a leitura dele neste relatório. Dessa forma, foi criada a diretoria *images*, no *Git* do projeto, contendo as imagens deste relatório.

b. *Diagrama dos Testes*

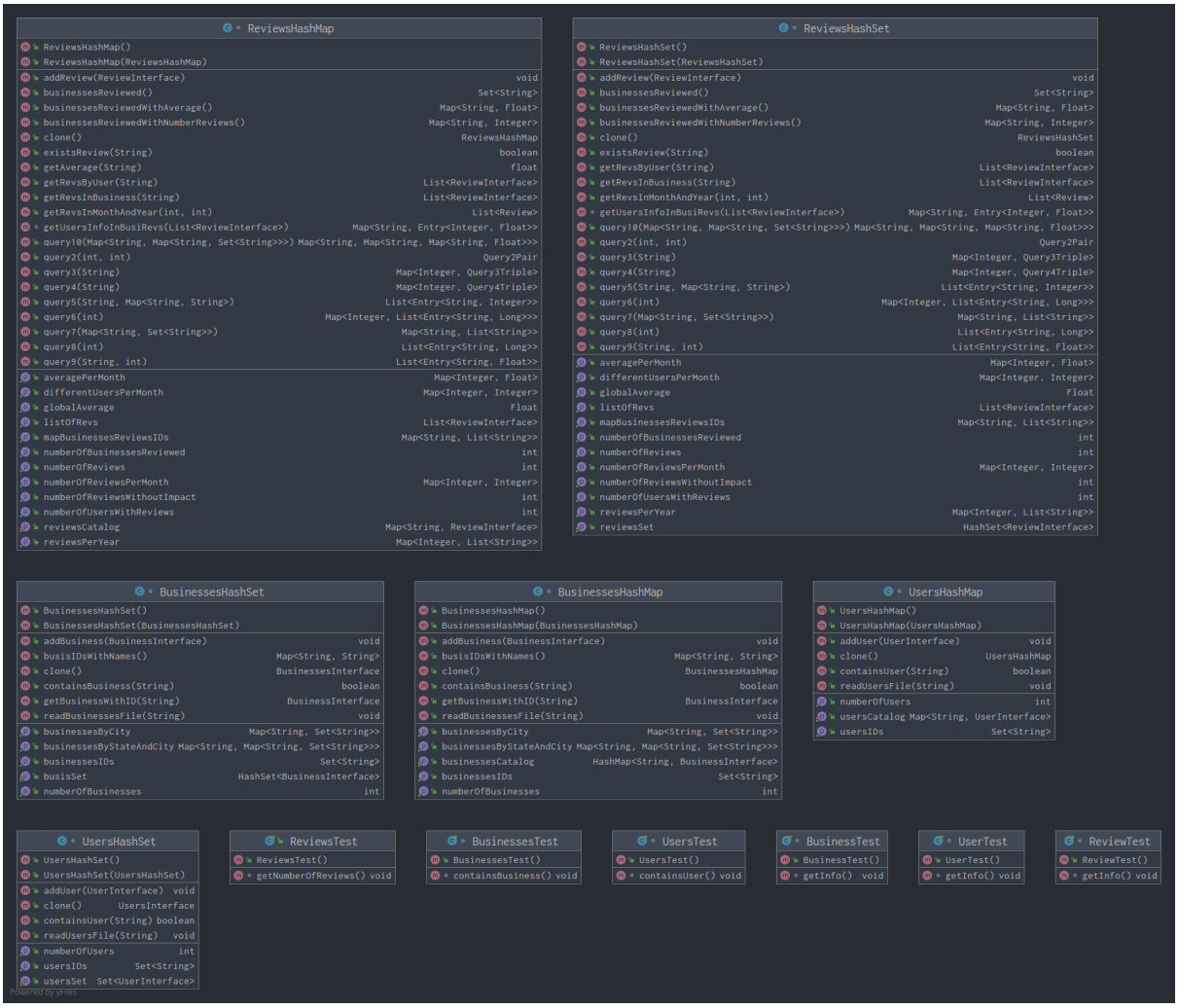


Diagrama de Classes gerado pelo *IntelliJ*

9. Conclusões Finais

Concluindo, somos da opinião que fomos capazes de cumprir com êxito todos os requerimentos da equipa docente, pelo que, consideramos este projeto bem sucedido.

Além disso, consideramos importante realçar que, através deste projeto e tendo em conta aquele que o precedeu, podemos concluir que, muitas vezes, o paradigma Orientado aos Objetos facilita o processo e revela-se muito útil para diversas tarefas de programação.