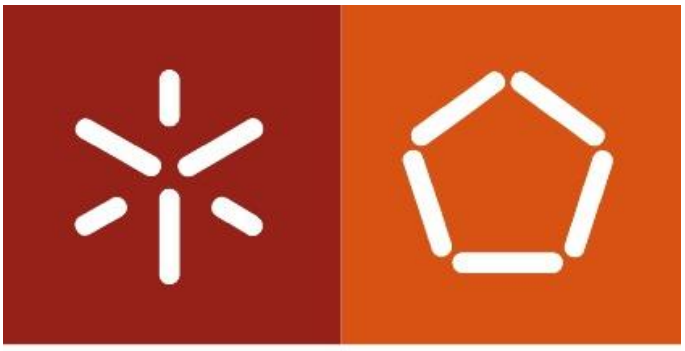


Universidade do Minho[3]

Departamento de Informática[1]

Projeto de Laboratórios de Informática III - Projeto de Java
Grupo 15

João Barbosa (a93270) Simão Cunha (a93262)
Tiago Silva (a93277)

19 de maio de 2021

Conteúdo

1 Introdução

O presente relatório descreve o projeto realizado no âmbito da disciplina de Laboratórios de Informática III (LI3), ao longo do segundo semestre, do segundo ano, do Mestrado Integrado em Engenharia Informática da Universidade do Minho. Tem como objetivo realizar consultas interativas de informações relativas à gestão básica de um sistema de recomendação/classificação de negócios tendo por base a linguagem de programação Java.

1.1 Descrição do Problema

Neste projeto foram utilizados os mesmos ficheiros em formato CSV do projeto de C, i.e., são ficheiros que contêm informação referente ao website *Yelp* [4], onde utilizadores de todo o mundo podem avaliar vários estabelecimentos, escrevendo as suas críticas.

Em concreto, o trabalho prendia-se em extrair a informação necessária dos vários ficheiros, de forma a conseguirmos responder a 10 querys obrigatórias e outras funcionalidades relacionadas com o conteúdo das mesmas da forma mais eficiente possível, como um módulo de estatisticas - que reúne e unifique os resultados relativos à informação contida em cada um dos catálogos - tendo especial atenção ao tempo de execução do programa, bem como ao encapsulamento dos dados.

1.2 Ficheiros CSV

Os 3 ficheiros necessários para a boa execução do programa são *business.csv*, *reviews.csv* e *users.csv* (nomes meramente ilustrativos), que continham as informações dos negócios, das críticas e dos utilizadores, respetivamente.

No ficheiro *business.csv*, extraímos o identificador(`business_id`), o nome, a cidade, o estado e as categorias em que cada estabelecimento se insere.

No ficheiro *reviews.csv*, recolhemos a informação relativa ao identificador da crítica (`review_id`), do identificador do utilizador que fez a classificação(`user_id`), do identificador do negócio que obteve a crítica (`business_id`), do valor das estrelas (0 a 5) atribuídas, do número de pessoas que acharam a review útil, do número de pessoas que acharam-na engraçada, do número de pessoas que a acharam *cool*, da data da publicação da review e do comentário feito pelo utilizador.

Por último, retiramos do ficheiros dos utilizadores, o identificador de um utilizador (`user_id`), do seu nome e da sua lista de amigos. Neste projeto, a equipa docente autorizou que o carregamento dos amigos do utilizador para a estrutura de dados fosse facultativo, uma vez que ocupa muito espaço em memória.

NB: Os vários parâmetros dos ficheiros vêm separados por ';'.

1.3 Conceção da Solução

Até obtermos a solução final, esta sofreu algumas alterações, nomeadamente ao nível do carregamento de dados. Numa primeira fase, líamos o ficheiro através do método `Files.readAllLines`, que colocava a informação numa lista de *strings* e, no final, procedíamos ao parsing lendo novamente o *Arraylist* de strings. Como vimos que tinhamos um tempo de execução pouco satisfatório e que estávamos a fazer um trabalho redundante, decidimos usar o método `reader.readLine()` que nos permite fazer o que queríamos de uma só vez. No entanto, o nosso projeto falha quando a opção de carregar os amigos de um utilizador é ativada, aparecendo a exceção `java.lang.OutOfMemoryError: Java heap space`.

O relatório está organizado da seguinte forma: a Secção 2 descreve o propósito e a estrutura de dados adotada em cada classe, a Secção 4 apresenta o como é que o modelo MVC foi implementado no nosso projeto, a Secção 5 apresenta imagens do nosso programa que efetua os testes de performance e a Secção 6 descreve as nossas conclusões do projeto. O relatório termina com 2 anexos: o desenho da estrutura de dados e o diagrama de classes.

Link do nosso repositório : `https://github.com/dium-li3/Grupo15` [2].

2 Classes do programa

Para desenvolvermos este trabalho construímos as seguintes classes:

2.1 Business

```
private String id;
private String name;
private String city;
private String state;
private List<String> categories;
```

A classe `Business` representa um negócio. Inclui o id do mesmo, o seu nome, a cidade onde se localiza, o seu estado e uma lista de categorias onde se inserem.

2.2 User

```
private String id;
private String name;
private String friends;
```

A classe `User` representa um utilizador. Inclui o id do mesmo, o seu nome e os seus amigos. Estes são representados por uma única string porque, uma vez que o seu carregamento não é obrigatório, decidimos escolher este tipo de dados porque ocupa menos espaço em memória, não são usados em querys e torna-se relativamente fácil guardar através da leitura de uma linha de um ficheiro.

2.3 Review

```
private String id;
private String userId;
private String businessId;
private double stars;
private int useful;
private int funny;
private int cool;
private LocalDateTime date;
private String text;
```

A classe `review` contém o ID da review (*review_id*), o id do utilizador que fez a criica (*user_id*) , o id do negócio que recebeu a review (*business_id*), o valor das estrelas (0 a 5) atribuídas, o número de pessoas que acharam a review "útil", o número de pessoas que acharam-na "engraçada", o número de pessoas que a acharam "cool", a data da publicação da review e o comentário feito pelo utilizador (*text*).

2.4 Estatísticas

```
private String usersFile;
private String businessesFile;
private String reviewsFile;
private ReviewScore[] reviews;
private int usersAmount;
private int businessesAmount;
private int reviewsInvalidAmount;
```

```
private Set<String> reviewedBusinesses;
private List<Set<String>> usersReviewers;
private int reviewsWithoutImpact;
```

A classe `Estatísticas` representa as estatísticas dos catálogos do projeto. Inclui o nome dos últimos ficheiros lidos de *users*, *reviews* e *businesses*, um array de *tuplos* com o número de *reviews* e o valor das *stars* atribuídas ao longo do ano, o número de *users* e *businesses* válidos, o número de *reviews* inválidas, um *Set* de *businesses* que foram avaliados, uma lista de 12 *Set* de *Users* que efetuaram uma *review* e o número de reviews sem impacto (i.e. 0 valores no somatório de cool, funny ou useful).

2.5 ReviewScore

```
private int reviewsAmount;
private double reviewsScore;
```

A classe *ReviewScore* representa um *tuplo*, que inclui o número de *reviews* realizadas e o score atribuído numa *review*.

2.6 PairInt

```
private int one;
private int two;
```

A classe *PairInt* representa um *tuplo* que auxilia na query 10. Irá incluir o número total global de *reviews* realizadas e o número total de *users* distintos que as realizaram.

2.7 BusinessReviewed

```
ReviewScore [] reviews;
List<Set<String>> usersReviewers;
String id;
```

Esta é a classe que representa um *business* através do seu id, duma lista de *Set's* dos utilizadores que o avaliou ao longo do ano e de um array de *tuplos* que inclui o número de *reviews* e o *score* atribuído por mês. Usamos *Set* em vez de *List* devido a um melhor tempo de execução no acesso à primeira *Collection*.

2.8 BusinessReviewsCount

```
String businessId;
String businessName;
int totalReviews;
```

Esta é a classe que representa um *business* através do seu id, do seu nome e do número total de *reviews* a esse negócio.

2.9 BusinessesPerState

```
Map<String, Map<String, Map<String,BusinessReviewed>>> businesses;
```

A classe `BusinessesPerState` auxilia a query 10. Serve para determinar para cada estado, cidade a cidade, a média de classificação de cada negócio. Para isto, associamos um *Map* de negócios que guardam as suas *reviews*, estando associada ao id de cada *business*. Este *Map* está associado a uma cidade, que por sua vez está associado a um estado.

Decidimos utilizar a *Collection* `Map` para um melhor acesso dos negócios de uma determinada cidade de um determinado estado (por exemplo).

2.10 TreeSetBusinesses

```
Set<Business> businesses;
int total;
```

A classe `TreeSetBusinesses` auxilia a query 1. Serve para determinar os negócios nunca avaliados e o seu total. Para isto, incluímos um *Set* de negócios e um inteiro que representa o total de *business* nunca avaliados. Decidimos usar um *Set* em vez de uma *List* por ser mais fácil de organizar.

2.11 BusinessComp

Esta classe implementa um *Comparator* para Business, onde compara-se 2 *business* através do seu id.

2.12 BusinessDistinctUsersComp

Esta classe implementa um *Comparator* para BusinessReviewed, onde compara-se 2 *business* através do número de utilizadores distintos que a avaliaram. Se existir empate, o critério passa a ser a ordem lexicográfica do id do negócio.

2.13 BusinessReviewedComp

Esta classe implementa um *Comparator* para BusinessReviewsCount, onde compara-se 2 *business* através do número de *reviews* efetuadas aos 2 negócios. Se existir empate, o critério passa a ser a ordem lexicográfica do id do negócio.

2.14 BusinessTotalReviewsComp

Esta classe implementa um *Comparator* para BusinessReviewed, onde compara-se 2 *business* através do número total de *reviews* efetuadas aos 2 negócios. Se existir empate, o critério passa a ser a ordem lexicográfica do id do negócio.

2.15 UserReviewer

```
ReviewScore reviews;
Set<String> businessesReviewed;
```

A classe `UserReviewer` extende a classe *User*. Inclui um *tuplo* com o número de reviews efetuadas e o score atribuído e um *Set* de id's de *business* avaliados por esse utilizador.

2.16 UserReviewerPerMonth

```
ReviewScore [] reviews;
List<List<String>> businessesReviewed;
```

A classe `UserReviewerPerMonth` extende a classe *User*. Inclui um array de *tuplos* com o número de reviews efetuadas e o score atribuído em cada mês e uma lista com 12 listas para cada mês de *List's* de id's de *business* avaliados por esse utilizador.

2.17 UserReviewerComp

Esta classe implementa um *Comparator* para UserReviewer, onde compara-se 2 *users* através do número de *business* avaliados. Se existir empate, o critério passa a ser a ordem lexicográfica do id do negócio.

2.18 UserReviewerTotalReviewsComp

Esta classe implementa um *Comparator* para UserReviewer, onde compara-se 2 *users* através do número total de *business* avaliados. Se existir empate, o critério passa a ser a ordem lexicogŕafica do id do negócio.

2.19 GestReviewsMVC

```
private Map<String, Business> businesses;
private Map<String, User> users;
private List<List<Review>> reviews;
private Statistics stats;
private final int buffer = 16*1024;
```

Esta classe é responsável por guardar os dados em memória em várias estruturas de dados. Os *businesses* são armazenados num `Map` onde a chave é o id do negócio, os *users* são armazenados num `Map` onde a chave é o id do utilizador e as *reviews* são armazenadas em `List` de 12 *List* de *reviews* efetuadas ao longo do ano. Também inclui as estatísticas dos catálogos e um inteiro para o tamanho de buffer que vai ser utilizado para a leitura e carregamento de dados.

Também é possivel ler e gravar dados num ficheiro objeto e inclui os algoritmos para as queries. Esta classe implementa também a interface SGR, já que esta extende todas as interfaces importantes como os catalogos; UsersCatalog, BusinessesCatalog e ReviewsCatalog; como as Estatisticas; StatisticsSGR, como a Leitura; ReadingSGR e por fim, as Queries.

2.20 ControlGestReviews

```
private GestReviewsMVC gr;
   private final Scanner scan;
   private final Errors errors;
   private final Messages messages;
   private final String[] menuPrincipal;
   private final String[] menuCarregarDado};
   private final String[] menuComAmigos;
   private final String[] menuCarregarCSV;
   private final String[] lerObjeto;
   private final String[] gravarObjeto;
   private final String[] analisarDados};
   private final String[] menuCatalogs;
   private final String[] menuQueries;
```

A classe `ControlGestReviews` controla o fluxo do programa. Contém os vários menus do programa, associando-os a uma funcionalidade da aplicação, as mensagens a aparecer no *stdout* e um *scanner*.

Dispomos várias funcionalidades na nossa aplicação:

1. *Manipular dados*: é possível carregar e gravar dados em ficheiros de texto e objeto.
2. *Efetuar queries*: é possível efetuar as 10 queries que temos disponíveis.
3. *Ver estatísticas*: é possível ver as estatísticas dos catálogos.
4. *Ver catálogos*: é possível observar o conteúdo dos ficheiros em forma de catálogo.

2.21 ViewQueries

Esta é a classe responsável pela paginação da aplicação e pela representação visual de alguns comandos do programa.

2.22 Messages

Esta é a classe responsável por apresentar mensagens de apoio no *stdout*.

2.23 Menu

Esta é a classe responsável pela estruturação dos menus. Foi inspirado num menu de um exercício prático da UC de Programação Orientada aos Objetos.

2.24 Errors

Esta é a classe responsável por apresentar mensagens de erro no *stdout*.

2.25 GestReviewsAppMVC

Esta classe é responsável pelo arranque do programa. Para isso, o método `main` invoca o método `run` da classe `ControlGestReviews` .

3 Interfaces do programa

O nosso programa inclui várias interfaces listadas abaixo:

- ANSIIColour: responsável por colorir *strings* do programa;
- BusinessesCatalog: responsável pelo catálogo de *businesses*;
- ReviewsCatalog: responsável pelo catálogo de *reviews*;
- UsersCatalog: responsável pelo catálogo de *users*;
- StatisticsSGR: responsável pelas estatísticas do programa;
- ReadingSGR: responsável pela leitura e armazenamento de dados;
- Queries: responsável pelas queries do programa;
- SGR: responsável pelas outras interfaces.

4 Estrutura do projeto

4.1 Modularização Funcional

O trabalho é composto pelas várias classes acima descritas encaixando-se do seguinte modo no modelo MVC:

- **Model**: Business.java, BusinessComp.java, BusinessDistinctUsersComp.java, BusinessReviewed.java, BusinessReviewedComp.java, BusinessReviewsCount.java, BusinessTotalReviewsComp.java, BusinessesPerState.java, GestReviewsMVC.java, PairInt.java, Review.java, ReviewScore.java, Statistics.java, TreeSetBusinesses.java, User.java, UserReviewer.java, UserReviewerComp.java, UserReviewerPerMonth.java, UserReviewerTotalReviewsComp.java.
- **Controller**: ControlGestReviews.java;
- **View**: Errors.java, Menu.java, Messages.java, ViewQueries.java.

4.2 Queries

Query 1

"Lista ordenada alfabeticamente com os identificadores dos negócios nunca avaliados e o seu respetivo total"

Através dos "stats"conseguimos obter os businesses "reviewed"guardados anteriormente durante a leitura dos ficheiros. Com isso, percorremos os businesses e aqueles que não constarem nessa lista, são os "não reviewed", adicionando à class auxiliar TreeSetBusinesses que é inicializada com o comparador BusinessComp.

Query 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"

Inicializamos uma variavel da class PairInt, ja que temos que devolver 2 informações. Com o mes e o ano valido, percorremos todas as reviews daquele mes, e as que forem daquele ano, adicionamos o primeiro valor do PairInt, e colocamos o user da review num set, para depois conseguirmos o numero de users (distintos) que deram review, colocando a informaçao no segundo valor do PairInt.

Query 3

"Dado um código de utilizador, determinar, para cada mês, quantas reviews fez, quantos negócios distintos avaliou e que nota média atribuiu"

Primeiro inicializamos uma variavel da class UserReviewerPerMonth atraves do User dado, e depois percorremos as reviews e adicionamos todas (já que dentro da classe ele verifica se a review é do user, caso seja adiciona a review ao mês especifico).

Query 4

"Dado o código de um negócio, determinar, mês a mês, quantas vezes foi avaliado, por quantos users diferentes e a média de classificação"

Inicializamos um BusinessReviewed através do id dado. Verificamos se o business é reviewed através do stats, e caso seja, percorremos as reviewes todas e adicionamos aquelas que forem daquele business.

Query 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"

Para esta query temos uma class auxilar BusinessReviewsCount, que conta quantos vezes um business foi reviewed. Depois usamos a query 3 com o user id dado e obtemos o UserReviewerPerMonth e com isso temos uma lista de todos os negócios que ele avaliou. Depois criamos um map para colocar os businesses em formato de BusinessReviewesCount, assim conseguindo contar os repetidos. Por fim é só adicionar a um TreeSet (inicializado com o comparador BusinessReviewedComp) todos os "values"do map e temos o resultado.

Query 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)"

Criamos um map em que a chave é o ano, e os valores são maps com BusinessReviewed e a sua chave o identificador para termos todos os negócios divididos por anos. Percorremos as reviewes,

verificamos o ano, o identificador de negócio e adicionamos a review ao Business especifico dentro do map. Depois usamos uma função auxiliar que tem a função de transformar os maps de cada ano numa lista ordenada (por um compador, neste caso BusinessDistinctUsersComp) e limitada a um top dado.

Query 7

"Determinar, para cada cidade, a lista dos três mais famosos negócios em termos de número de reviews"

Esta query usa o mesmo raciocino da anterior, só que em vez de um Integer a dizer o ano, temos uma String para a cidade, no final usamos antes o comparator BusinessTotalReviewsComp e o top não é dado e sim sempre 3.

Query 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"

Para esta query temos uma classe auxiliar UserReviewer que tem o método de adicionar uma review, e coleta as informações necessárias, como o número de negócios difrentes avaliados. Com isso criamos um map de UserReviewer em que a chave é o identificador do User, e por fim percorremos as reviews para criar os devidos UserReviewer no map, e caso já existissem só adicionar a review a estes. Por fim transfomamos o map numa lista ordenada pelo comparador UserReviewerComp limitada pelo top dado pelo utilizador.

Query 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)"

Esta query é praticamente igual à anterior, difere apenas que agora só adicionamos as queries do business especificado, e que como não nos interessa quantos negócios difrentes avaliou (pois como só adicionamos um business especifico vai ser sempre 1), ordenamos com o comparador UserReviewerTotalReviewsComp, que compara pelo o número total de reviews, e como só adicionamos as reviews daquele negócio especifico, vai comparar o numero total de reviews daquele negócio.

Query 10

"Determinar para cada estado, cidade a cidade, a média de classificação de cada negócio"

Para esta query temos uma class auxiliar BusinessesPerState que trata do funcionamento adicionar uma review. Esta class tem metodos, para criar um State novo, uma City nova, um Business novo, verificar se qualquer um destes antes existe, e atualizar um business com uma review. Tendo isto é só percorrer as reviews todas e atualizar o negócio da review.

5 Testes de performance

Especificações da máquina onde os testes foram executados:

- Nome: *MSI GL63 8SE*
- Sistema operativo: *Ubuntu*
- Processador: *Intel Coffeelake i7-8750H + HM370*
- RAM: *16GB DDR4 2666MHz*

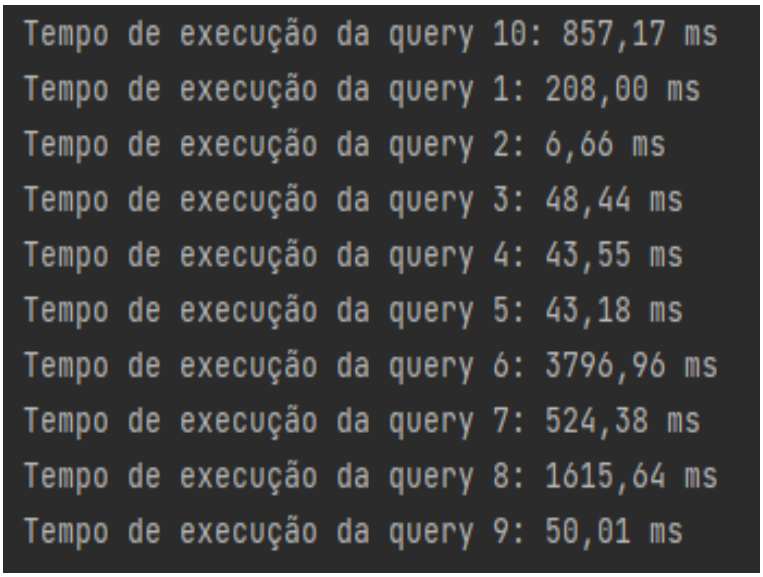
- Placa gráfica: *NVidea GeForce RTX 2060, com 6GB GDDR6*

Foi nos pedido que criássemos um programa de testes de performance independente do projeto. Este nosso programa testa o tempo de 2 formas de leitura dos dados de ficheiros de texto (linha a linha e caractere a caractere) e 2 formas de leitura de dados de ficheiros-objeto (com e sem buffer). Além disso, é medido o tempo de execução das nossas queries, onde executamos 10 vezes cada interrogação e imprimimos o tempo médio de cada query. Seguem abaixo imagens desses mesmos testes:

```
Leitura dos ficheiros objeto sem buffer:
Tempo de execução: 199978,04 ms
Leitura dos ficheiros .csv caractere a caractere:
Tempo de execução Business: 811,38 ms
Tempo de execução User: 52750,46 ms
Tempo de execução Review: 17681,33 ms
Leitura dos ficheiros objeto com buffer:
Tempo de execução: 6829,63 ms
Leitura dos ficheiros .csv linha a linha:
Tempo de execução Business: 515,60 ms
Tempo de execução User: 9817,71 ms
Tempo de execução Review: 4853,23 ms
```

Testes efetuados nas queries:

Query 1	TreeSetBusinesses x = businessesNotReviewed();
Query 2	PairInt x = allReviewsAndUsers(12, 2015);
Query 3	UserReviewerPerMonth x = getUserReviewer("rKmD1FKz-XXD7spAgMCKDg");
Query 4	BusinessReviewed x = allReviewsOfBusiness("8zehGz9jnxPqXtOc7KaJxA");
Query 5	Set⟨BusinessReviewsCount⟩x = businessesReviewedByUser("uUPTteweJvIe1qnwTxaGXg");
Query 6	Map⟨Integer, List⟨BusinessReviewed⟩x = topBusinessesMostReviewedByYear(10);
Query 7	Map⟨String, List⟨BusinessReviewed⟩x = topBusinessesMostReviewedByCity();
Query 8	List⟨UserReviewer⟩x = topUsersReviewers(10);
Query 9	List⟨UserReviewer⟩x = topUsersReviewersByBusiness("wZgUAuDuEGPEzKK-PsngKQ", 10);
Query 10	BusinessesPerState x = allBusinessesPerState();



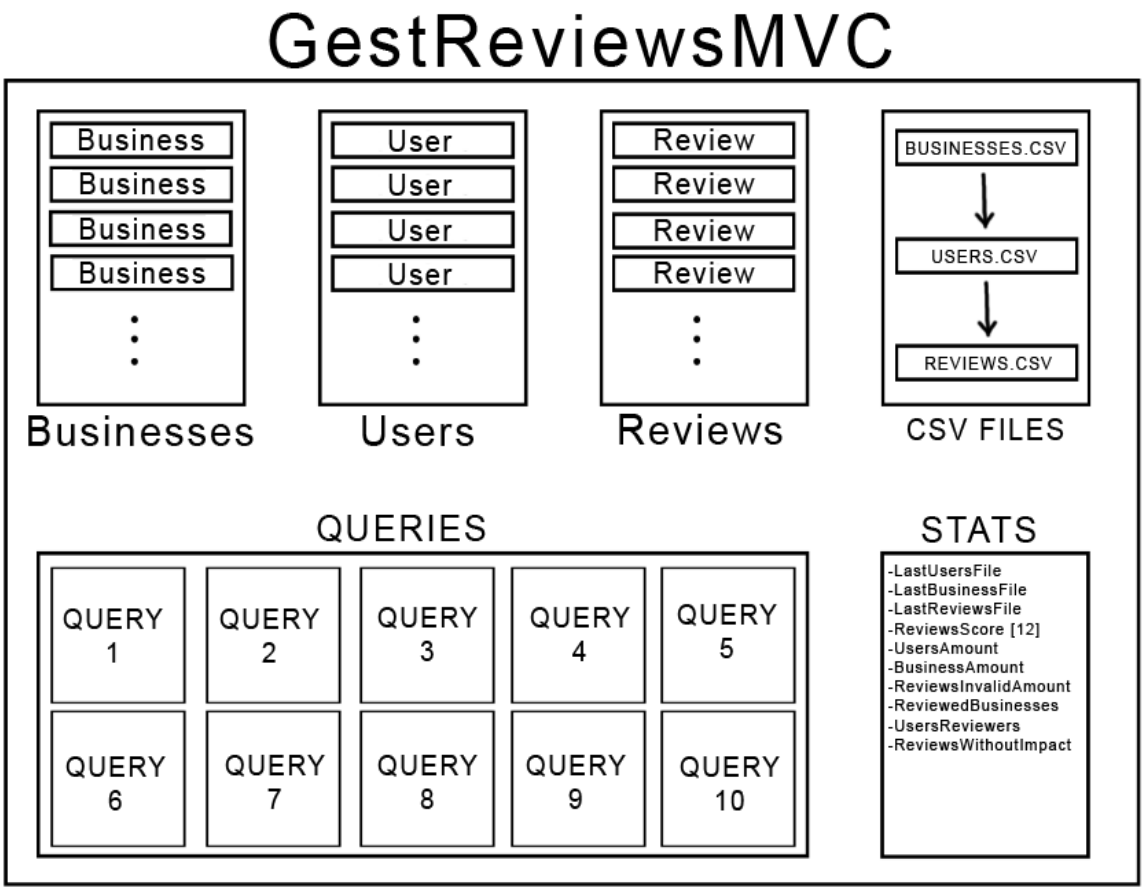
6 Conclusões

Tal como o projeto em C, sentimos que o nosso projeto está bem conseguido. O nosso programa é capaz de responder a todas as queries e mostrar os resultados de forma intuitiva, e fá-lo com tempos aceitáveis, embora haja queries que possam ser otimizadas, tais como as queries 6 e 7.

A Diagrama de Classes

Figura 1: Diagrama de classes do programa, gerado pelo *IntelliJ*

B Desenho da estrutura de dados



Referências

[1] Dpt informática. `https://www.di.uminho.pt/`.

[2] Github. `https://github.com/dium-li3/Grupo15`.

[3] Uminho. `https://www.uminho.pt/PT`.

[4] Yelp. `https://www.yelp.pt`.