Escola de Engenharia

Universidade do Minho

# Aplicação Desktop

2020/2021


**Grupo nº 9**

Rita Gomes, a87960
Leonardo Freitas, a93281
Joaquim Roque, a93310

# Indice

# Introdução

O presente relatório tem como objetivo descrever o desenvolvimento de uma aplicação desktop na linguagem de **programação** Java. Esta aplicação deve ser capaz de ler e armazenar em estruturas de dados adequadas as informações de vários ficheiros, para que, posteriormente, possam ser realizadas diversas queries.

Para procedermos à conceção da aplicação devemos, previamente, criar as classes das APIs de modo a seguir os bons **princípios** da programação com interfaces. Desta forma, deverá ser criado um Catálogo de Users, um Catálogo de Negócios, um Catálogo de Reviews e um Módulo de Estatísticas, que reúna e unifique resultados relativos à informação contida em cada um dos catálogos.

Para evitar que possa haver alterações indevidas por agentes externos será fundamental garantir o encapsulamento dos dados armazenados no sistema, **garantindo** assim a segurança do mesmo.

Ao longo deste relatório iremos descrever as abordagens adotadas pelo grupo para a resolução do problema proposto e, tal como é pedido, sustentaremos o mesmo com alguns testes de performance.

# Formalização do problema

De uma forma geral, pretende-se que seja possível ler e validar dados de memória secundária e popular estruturas de dados em memória central. Para além disso, é solicitado gravar as estruturas de dados em ficheiro de objetos e exportar, dos 3 ficheiros dados para análise na fase anterior (users.csv, businesses.csv e reviews.csv) as seguintes queries:

- Lista ordenada alfabeticamente com os identificadores dos negócios nunca avaliados e o seu **respectivo** total.
- Dado um mês e um ano (válidos), determinar o número total global de **avaliações** realizadas e o número total de users distintos que as realizaram.
- Dado um código de utilizador, determinar, para cada mês, quantas reviews fez, quantos negócios distintos avaliou e que nota média atribuiu.
- Dado o código de um negócio, determinar, mês a mês, quantas vezes foi avaliado, por quantos users diferentes e a média de classificação.
- 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.
- 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).
- Determinar, para cada cidade, a lista dos três mais famosos negócios em termos de número de reviews.
- 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.
- 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).
- Determinar para cada estado, cidade a cidade, a média de classificação de cada negócio.

# Conceção da Resolução

Nesta secção serão apresentadas as soluções usadas assim como a estruturação adotada pelo grupo para a resolução do problema.

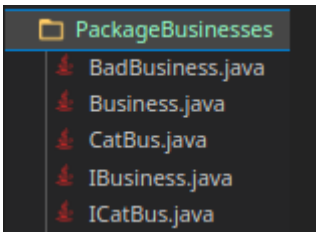
## 3.1. Estruturas de Dados

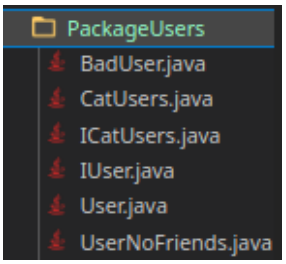
A informação lida será guardada, em *runtime*, numa variável de instância do tipo Map (da biblioteca do Java) do respetivo catálogo, que a cada identificador faz corresponder a instância da classe que o catálogo em questão deverá armazenar. Tal como no projeto de C, o grupo tomou a decisão baseando-se no facto dos tempos de procura e de inserção tenderem para constante neste tipo de estrutura.

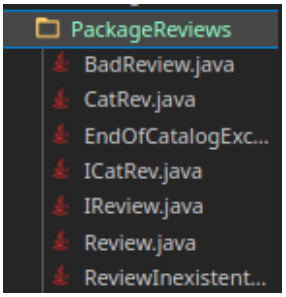
## 3.2. Classes e API

O projeto encontra-se dividido em *packages*:

### 3.2.1 PackageBusinesses, PackageUsers e PackageReviews




Em cada um destes, temos, para além das interfaces, a classe do tipo correspondente, a classe do catálogo onde este será agregado e uma exceção que será “atirada” se durante o carregamento das estruturas, a linha lida do ficheiro estiver incorretamente formatada.

Os catálogos seguem estruturas muito idênticas no que toca a métodos e variáveis de instância: além do *hashmap,* já referido, existe ainda um iterador que permite iterar pela estrutura, respeitando as boas práticas do paradigma OO, nomeadamente o encapsulamento, pois faz uso de métodos de clonagem de objetos. No que toca a métodos, implementam-se um construtor por omissão e os necessários à consulta da estrutura, bem como outros que serão úteis para efeitos estatísticos.

No ***PackageReviews*****,** existe ainda uma outra exceção que indicará que o catálogo não tem mais entradas a consultar, sendo portanto transversal aos 3 catálogos. Notar ainda, que no *package* dos utilizadores existem duas classes que implementam a mesma interface, onde a única diferença entre as duas é o facto do campo dos amigos ser ignorado ou não, particularmente útil para testes de performance. Criar código genérico torna-se uma das grandes vantagens de OOP.

Em cada uma das classes User, Business e Review são implementados métodos básicos como **getters, clone()** de modo a manter intacto o encapsulamento, bem como métodos de **parsing**, que funcionarão como construtores virtuais.

```
     * @return objeto clone do objeto que recebe a mensagem.
     */
    public IBusiness clone() {
        return new Business(this);
    }
```

```
    public static IBusiness parseBusiness(String s) throws BadBusiness{

        List<String> campos = Arrays.asList(s.split(";"));
        Set<String> categories;

        if(campos.size() == 5 && campos.get(0).length() == 22)
            categories = new TreeSet<>(Arrays.asList(campos.get(4).split(",")));

        else if(campos.size() == 4 && campos.get(0).length() == 22)
            categories = new TreeSet<>();

        else
            throw new BadBusiness();


        return new Business(campos.get(0), campos.get(1), campos.get(2), campos.get(3), categories)
    }
}
```

```
  public static IReview parseReview(String s) throws BadReview{

    List<String> campos = Arrays.asList(s.split(";"));
    IReview r;

    if(campos.size() == 9 && campos.get(0).length() == 22)
       r = new Review(campos.get(0), campos.get(1), campos.get(2), Float.parseFloat(campos.get(3)
                      Integer.parseInt(campos.get(4)), Integer.parseInt(campos.get(5)),
                      Integer.parseInt(campos.get(6)), campos.get(7), campos.get(8));

    else if(campos.size() == 8 && campos.get(0).length() == 22)
    r = new Review(campos.get(0), campos.get(1), campos.get(2), Float.parseFloat(campos.get(3)),
                   Integer.parseInt(campos.get(4)), Integer.parseInt(campos.get(5)),
                   Integer.parseInt(campos.get(6)), campos.get(7), "");

    else
       throw new BadReview();

    return r;
  }
}
```

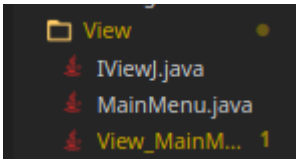
```
/**
 * Metodo que faz uma copia do objeto receptor da mensagem.
 * @return objeto clone do objeto que recebe a mensagem.
 */
public IUser clone() {
   return new User(this);
}
```

```
/**
 * Metodo que faz uma copia do objeto receptor da mensagem.
 * @return objeto clone do objeto que recebe a mensagem.
 */
public IReview clone() {
   return new Review(this);
}
```

```java
    public static IUser parseUser(String s) throws BadUser{

        List<String> campos = Arrays.asList(s.split(";"));
        IUser u;

        if(campos.size() == 3 && campos.get(0).length() == 22)
            u = new User(campos.get(0), campos.get(1), campos.get(2));
        else
            throw new BadUser();

        return u;
    }
}
```

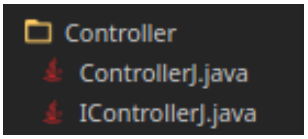
Como podemos ver por este print, seja qual for a classe User, Reviews e Businesses aplicam os mesmo métodos, ambas usam o clone para garantir que existe encapsulamento e ambas utilizam uma método de parse na leitura dos campos fornecidos pelo utilizador.

### 3.2.2 View



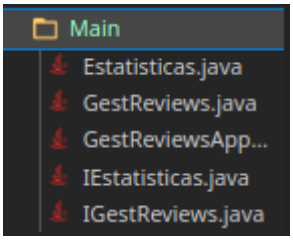

A classe *View_MainMenu* contém todos os métodos necessários para tratamento de input/output. Sempre que o controlador necessitar de informação do utilizador ou precisar de imprimir algo, chamará o método adequado desta classe.

### 3.2.3 Controller




O controlador será instanciado no início do programa, com as variáveis de instância do *model* e da *view* devidamente inicializadas. Cada um dos seus métodos irá ser chamado de acordo com o input do utilizador. Há um método adequado a cada query, pedindo ao *model* a informação necessária, fazendo depois uso da *view* para mostrar o *output*.

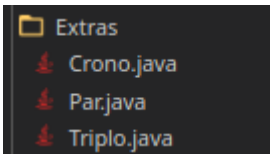
### 3.2.4 Main



A classe *GestReviewsAppMVC* contém o método main, cuja única função é instanciar o controlador, o modelo e a view. Já a *GestReviews* funciona como a agregação de todo o modelo, ou seja, os 3 catálogos e o módulo de estatísticas. Cada query tem o seu respetivo método, lendo o que for necessário dos catálogos e produzindo a lista das Strings a imprimir (mais detalhes na secção seguinte). Contém ainda os métodos de carregamento das estruturas, que, além de carregarem o respetivo catálogo, colocam a informação necessária no módulo de estatísticas. De notar que, para efeitos de validação, é necessário que as *reviews* sejam as últimas a serem lidas.

Neste package existe ainda a classe de Estatísticas, carregada imediatamente e automaticamente após a leitura dos ficheiros, se esta for bem sucedida. As variáveis contém toda a informação estatística necessária e o método **toString** devolve-a se tal for solicitado.

### 3.2.5 Extras




Aqui incluem-se as classes de tuplos Par e Triplo, úteis na manipulação de informação necessária nas queries, bem como a classe fornecida Crono.

```java
public class Par<A, B> {

    private A valor0;
    private B valor1;

    public Par(A value0, B value1) {
        this.valor0 = value0;
        this.valor1 = value1;
    }

    public A getValue0() {
        return this.valor0;
    }


    public B getValue1() {
        return this.valor1;
    }

    public void setValue0(A v){
        this.valor0 = v;
    }

    public void setValue1(B v){
        this.valor1 = v;
    }

    public static <A, B> Par<A, B> de(A value0, B value1){
        return new Par<A,B>(value0, value1);
    }
}
```

```java
public class Triplo<A, B, C> {

    private A valor0;
    private B valor1;
    private C valor2;

    public Triplo(A value0, B value1, C value2) {
        this.valor0 = value0;
        this.valor1 = value1;
        this.valor2 = value2;
    }

    public A getValue0() {
        return this.valor0;
    }


    public B getValue1() {
        return this.valor1;
    }

    public C getValue2() {
        return this.valor2;
    }

    public void setValue0(A v){
        this.valor0 = v;
    }

    public void setValue1(B v){
        this.valor1 = v;
    }

    public void setValue2(C v){
        this.valor2 = v;
    }

    public static <A, B, C> Triplo<A, B, C> de(A value0, B value1, C value2){
        return new Triplo<A,B,C>(value0, value1, value2);
    }
}
```

## 3.3. Desenvolvimento das Queries

Nesta secção serão mencionados os modos como foram implementadas as queries pedidas no enunciado e qual o algoritmo desenvolvido para atingir o resultado esperado.

Em todas elas existem alguns aspetos semelhantes, sendo que numa primeira fase são carregadas todas as *reviews* (ou negócios) para uma lista ou *set*, podendo estas ser filtradas ou não, tendo em atenção a exceção que indica o fim do catálogo. A informação a ser devolvida estará contida numa lista de *Strings* devidamente formatadas para impressão sob a forma de um tabela.

- **Query 1**

  Após a leitura dos negócios para um *HashSet,* iteramos pelo catálogo das *reviews* e removemos o negócio, se existente, cujo identificador se encontra especificado naquela avaliação. Daí o uso do *HashSet*, de modo a tornar este processo mais eficaz. Para garantir a ordenação, basta instanciar um *TreeSet*, usando o construtor por omissão e adicionar as Strings dos identificadores, devidamente formatadas.

- **Query 2**

  As avaliações são filtradas, durante a leitura, de acordo com o ano e o mês dados. Optamos por exigir que a classe que o invoque deve fornecer duas *Strings* para estes parâmetros pelo facto da variável data ser também ela uma *String.* Resta-nos apenas contar o número de utilizadores distintos, fazendo uso de *streams*, disponíveis a partir do Java 8.

- **Query 3**

  Agrupam-se as avaliações, previamente filtradas consoante o código dado, de acordo com o mês, numa estrutura auxiliar, através dos *collectors* do Java, que a cada mês faz corresponder uma lista de reviews. Para cada uma destas listas é calculado o número de negócios distintos, a média de estrelas e o tamanho da lista, agregando a informação numa *String*.

- **Query 4**

  As avaliações são filtradas de acordo com o código dado, e a informação é depois manipulada num processo semelhante à query 3.

- **Query 5**

Filtram-se as avaliações de acordo com o identificador dado. Estas são depois agrupadas de acordo com o nome do negócio, e teremos um *set,* ordenado por ordem alfabética e pelo número de *reviews,* com recurso a um *Comparator* de pares de Strings/inteiros.

- **Query 6**

As avaliações são inicialmente agrupadas por ano e cada uma destas listas é posteriormente mapeada num *map* que irá agrupá-las por id de negócio. Iterando sobre este último, recolhe-se a informação referente a id de negócio, número de avaliações e número de utilizadores distintos, armazenada num *set* de Triplos, ordenado consoante o número de avaliações. Daqui retiram-se os primeiros *n,* e colocam-se na lista a devolver, devidamente formatados.

- **Query 7**

Após agrupar por cidade, cada lista de *reviews* é convertida num *map* que agrupa por id de negócio. Daqui, tiram-se o número de reviews e o id de negócio para uma lista de Pares, ordenada pelo número, onde são apenas aproveitados os 3 primeiros, e adicionados à lista a devolver.

- **Query 8**

Na query 8, agrupam-se as *reviews* por id de utilizador; para cada entrada desse *map,* colocamos num par o id do utilizador e o número de negócios distintos, ordenando de acordo com este último, sendo depois aproveitados os primeiros *n.*

- **Query 9**

As avaliações, filtradas pelo id de negócio dado, são agrupadas por id de utilizador e convertidas em Triplos de id de negócio, número de reviews e média de estrelas. Apenas são colocadas na lista a devolver os primeiros *n,* após ordenação pelo número de avaliações.

- **Query 10**

Agrupadas as *reviews* por cidade/estado, para cada negócio, geramos um Par de número de *reviews* e média de estrelas.

# Testes de Performance

Os tempos de execução deste programa foram obtidos correndo o código num PC da marca **Lenovo Legion**, equipado com **16GB de memória RAM**, **processador I7**, no sistema operativo **Manjaro Distro**.
Foi usado openjdk na versão 15.0.2.

| Query Nº | Tempo 1 | Tempo 2 | Tempo 3 |
|---|---|---|---|
| **Load (Sem Amigos)** | **9.9897** | **9.8945** | **10.890** |
| **Load (Com Amigos)** | **13.483** | **13.143** | **12.709** |
| **Query 1** | **0.9363** | **0.5128** | **0.5082** |
| **Query 1e** | **-** | **-** | **-** |
| **Query 2** | **0.3493** | **0.2244** | **0.2674** |
| **Query 2e** | **-** | **-** | **-** |
| **Query 3** | **0.1017** | **0.1026** | **0.0989** |
| **Query 4** | **0.1400** | **0.0933** | **0.0957** |
| **Query 5** | **0.1112** | **0.0889** | **0.0948** |
| **Query 6** | **0.8663** | **0.8347** | **0.8152** |
| **Query 7** | **0.8670** | **0.8346** | **0.8577** |
| **Query 8** | **0.8632** | **0.9109** | **0.8677** |
| **Query 9** | **0.1040** | **0.0953** | **0.1116** |
| **Query 10** | **1.0857** | **1.0236** | **0.9279** |

**Observação: Foram feitos 3 testes para cada query o resultado é dado em segundos.**

# Conclusão

Com a realização deste trabalho foi possível consolidar e colocar em prática os conhecimentos obtidos ao longo não só da unidade curricular de Laboratórios de Informática III como também de Programação Orientada a Objetos. Para além disso, adquirimos um maior domínio da linguagem Java, destreza no uso da mesma e também uma maior espontaneidade no processo de tomar decisões lógicas para programar aplicações software.

Tal como sabemos, o intuito da Programação Orientada a Objetos é o de aproximar o manuseio das estruturas de um programa ao manuseio das coisas do mundo real, daí o nome objeto. Desta forma, para garantir o melhor desempenho possível da nossa aplicação recorremos a um dos principais conceitos ligados a esta linguagem, o encapsulamento.

No processo de desenvolvimento do projeto notamos algumas dificuldades nomeadamente no desenvolvimento da query estatística 2 a qual não conseguimos concluir e em arranjar outra maneira além da já usada para ler os ficheiros, talvez no caso em que tivéssemos mais tempo para dedicar a esta parte do trabalho o que faríamos de melhor seria tentar implementar essa mesma query e encontrar esse novo método de leitura para ficheiros. Admitimos ainda que a estruturação dos *packages* não seja a mais feliz, pelo que consideramos que a divisão dos 3 conceitos do padrão MVC deveria estar mais clara.

Apesar de tudo concluímos ter feito um bom trabalho face àquilo que nos foi proposto e consideramos que os conhecimentos adquiridos serão úteis em projetos futuros.