Universidade do Minho

Mestrado Integrado de Engenharia Informática
Laboratórios de Informática III

UMYELP - PROJETO JAVA

(MiEI-LI3 20/21)

GRUPO Nº25:

- Joana Maia Teixeira Alves (A93290)

- Maria Eugénia Bessa Cunha (A93264)

- Vicente Gonçalves Moreira (A93296)

Data de Entrega:

19-06-2021

Índice

Estratégia Inicial

Inicialmente, planeamos seguir uma estrutura muito semelhante à do projeto anterior (projeto_c), visto que obtivemos resultados satisfatórios. Como requerimento, e de forma a facilitar a futura expansibilidade da aplicação, decidimos criar várias interfaces dentro do modelo. Criamos uma interface por estrutura de dados (*IUser, IBusiness, IReview*) para permitir vários "géneros" destas (Ex: User Admin, Business Premium… etc.), no entanto, para este projeto apenas utilizamos uma classe básica por estrutura de dados (User, Business, Review).

A classe principal do modelo (gestReviews, que implementa *IgestReviews*) contém 3 listas principais, uma de utilizadores, outra de negócios e por último, uma de reviews. Contém também uma interface extra (*IStats*) onde guardamos várias estatísticas do modelo assim como Indexes que nos permitem consultas mais rápidas (mais informação na descrição da API).

Diagramas de classe da Aplicação

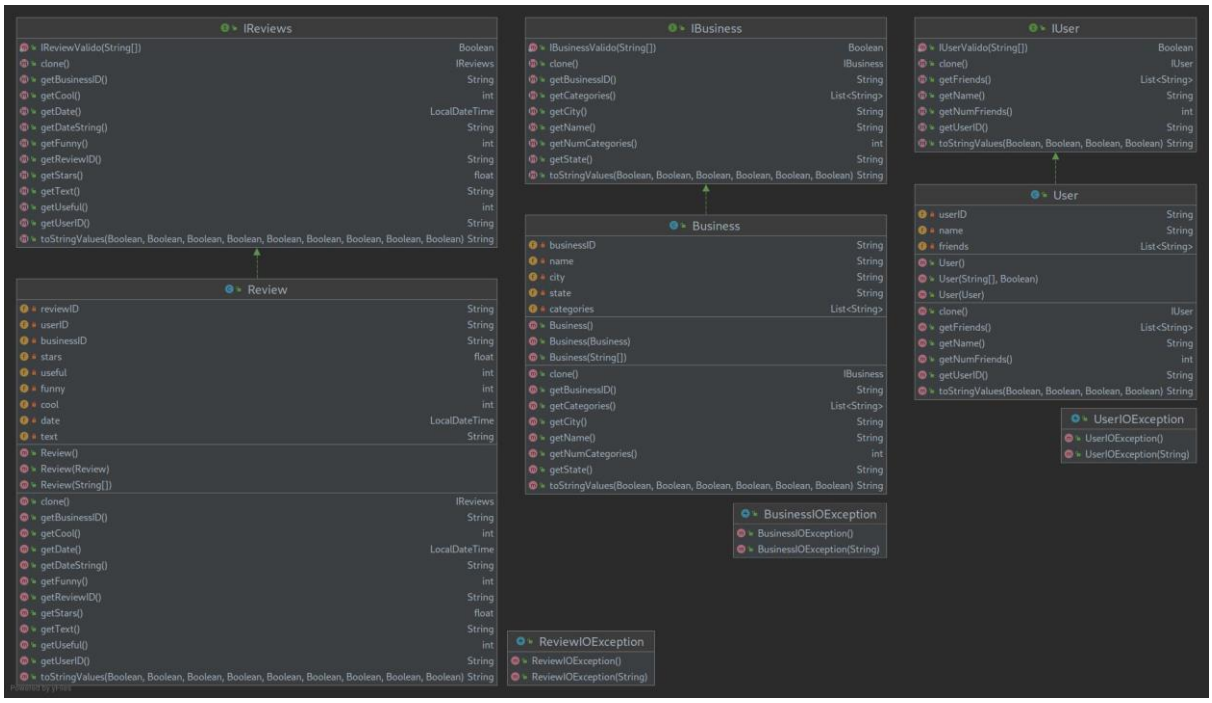

Figura 1 - Diagrama Extenso das estruturas de dados

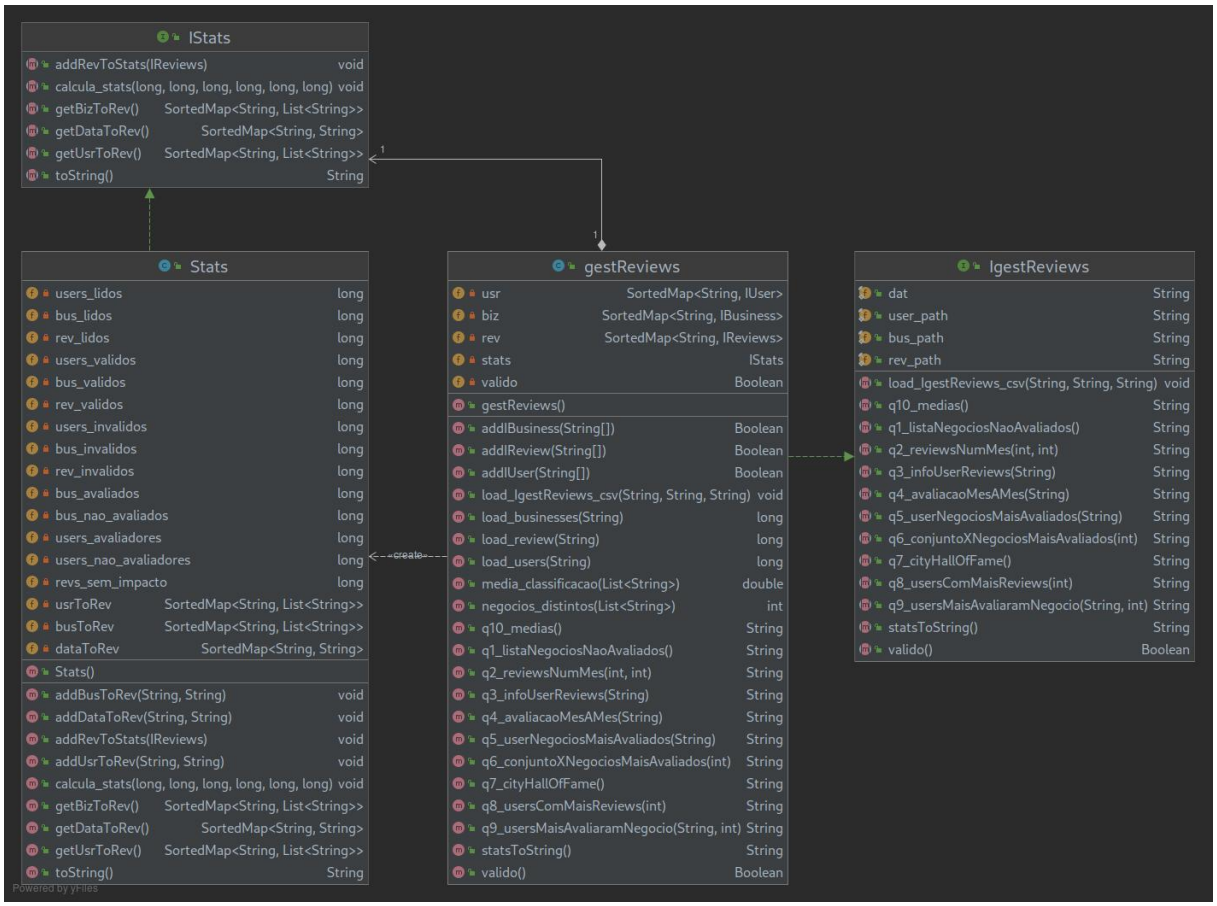

Figura 2 - Diagrama Extenso da relação das classes gestReviews e Stats

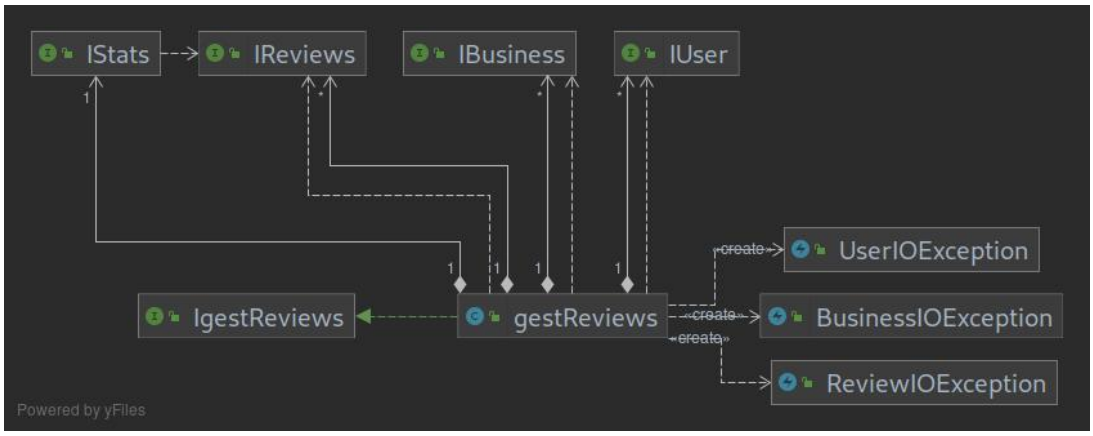

Figura 3 - Diagrama Simplificado do Modelo da aplicação

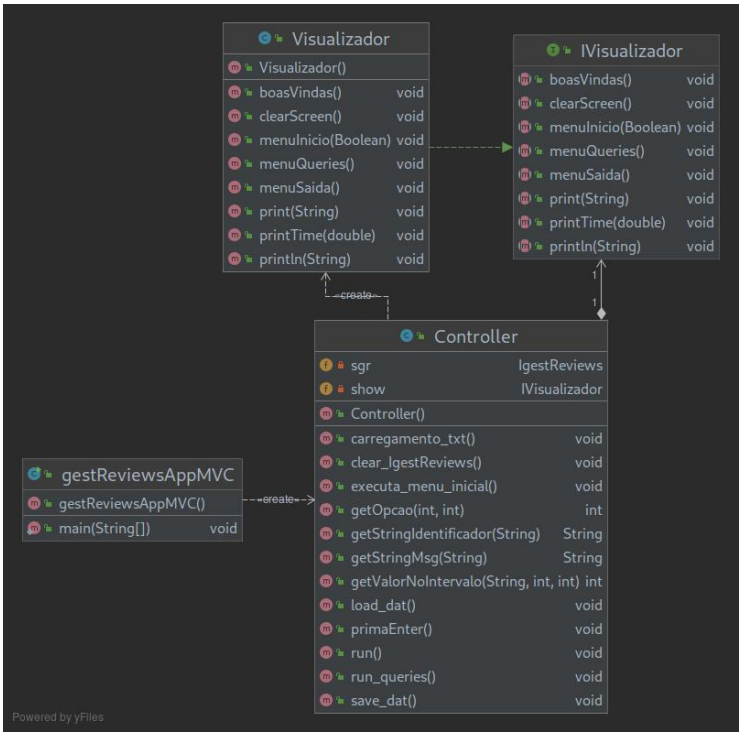

Figura 4 - Diagrama Extenso do MVC da aplicação

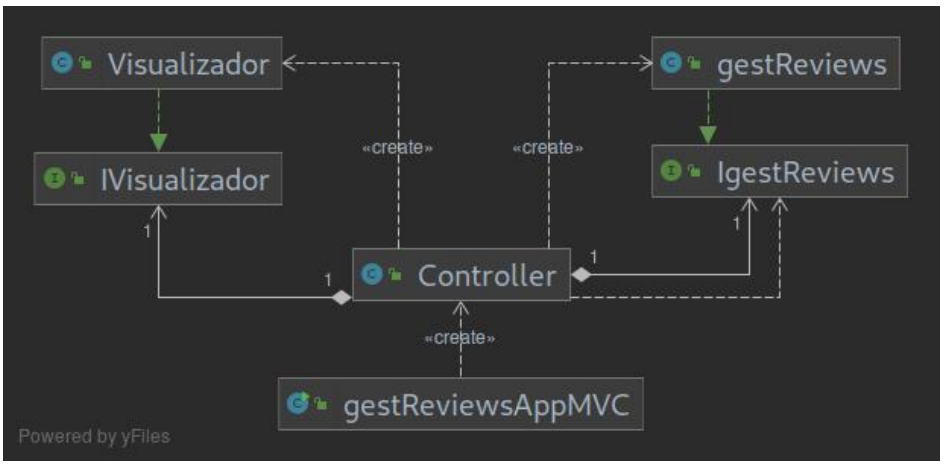

Figura 5 - Diagrama Simplificado do MVC da aplicação

Descrição das API's

Estruturas de dados (User, Business, Review)

Sendo estas três classes semelhantes entre si e cada uma implementando a sua interface respetiva, estas armazenam e gerem a informação individual dos dados da aplicação. Cada uma contém um construtor próprio que recebe como argumento uma lista de strings correspondentes aos seus campos. Também contêm métodos de validação de campos, leitura de variáveis de instância e de clonagem.

Stats

Esta classe contém as variáveis de instância necessárias para o cálculo das estatísticas do catálogo assim como indexes. Estes são constituídos de listas de identificadores de forma a acelerar as travessias e auxiliar na procura das estruturas de dados.

O index usrToRev contém a lista de utilizadores avaliadores e para cada um, a sua lista correspondente de reviews (IDs), no caso do index bizToRev, é a correspondência entre negócios avaliados e a sua lista de reviews. Por último, diferenciando-se do projeto C anterior, o index dataToRev organiza os IDs das reviews por ordem de data crescente, fazendo uma correspondência Data -> IDReview. Encontramos a necessidade de criar este index dado que esta informação se demonstrou ser útil na resolução rápida de algumas das queries.

Todos estes indexes utilizam a interface "*Sorted Map*" de forma a criar a correspondência entre os vários IDs por ordem alfabética. Já os dados das estatísticas são armazenados em Integers simples.

gestReviews

Para a classe principal do modelo, criamos métodos para o carregamento dos catálogos em .csv, métodos auxiliares de carregamento e criação de cada estrutura de dados, verificação da validade do catálogo e por fim, as várias queries requeridas.

Para armazenar as várias listas de estruturas de dados (como mencionado no tópico "Estratégia Inicial"), utilizamos a interface "*SortedMap*" de forma a armazenar os vários identificadores das estruturas por ordem alfabética.

Visualizador

Esta é uma classe simples implementando uma interface (*IVisualizador)*, e apenas contém métodos de impressão na consola. Contém também a aparência de cada Menu e suporta mensagens customizadas, para uso em caso de erros.

Controlador

Sendo esta classe a principal da nossa aplicação e contendo uma lógica muito específica, não encontramos a necessidade de criar uma interface para esta.

Esta classe contém todos os métodos de controlo da aplicação, como recolha de inputs de forma genérica (getOpcao, getStringIdentificador, getStringMsg e getValorNoIntervalo), funções da aplicação como o guardar e o carregar do catálogo em ficheiros binários e métodos sobre as lógicas dos vários menus.

Este também cronometra o tempo de execução de cada query e controla os vários erros que a aplicação possa encontrar durante o decorrer da mesma.

Otimizações das Estruturas

Ao contrário do nosso projeto anterior, as nossas estruturas não sofreram grandes alterações no decorrer do desenvolvimento. No início armazenávamos as listas das estruturas de dados usando a interface "*Map*", recorrendo a um *HashMap*. No entanto, quando passamos para o desenvolvimento das queries, encontramos a necessidade de ordenar estas listas por ordem alfabética para obter resultados rápidos. Assim modificamos a interface usada para "*SortedMap*" e passamos a usar TreeMap's para amazenar as estruturas de dados. Esta mudança foi extremamente fácil dado que a interface "*SortedMap*" é uma extensão da interface "*Map*", provando a conveniência de programação por interfaces.

Um dos desafios que encontramos foi no carregamento dos catálogos a partir dos ficheiros .csv. Este sofria de problemas de performance (~40s no primeiro PC no tópico de testes) e chegava ao limite máximo de memória permitida. Inicialmente este efetuava uma contagem de linhas em separado e de seguida efetuava várias operações sobre cada linha do ficheiro, de forma a construir as listas. Modificando o código de forma a efetuar estas operações em apenas uma iteração diminui o tempo de carregamento em cerca de 38% (~25s). Efetuando ainda mais modificações de forma que a contagem também fosse incluída na mesma operação ajudou novamente na performance de carregamento com uma redução extra de 40% (~15s).

```
public long load_users(String path) throws UserIOException {
    long total = 0;
    try {
        total = Files.lines(Paths.get(path)).skip(1).count();
        Map<String,IUser> antigo = Files.lines(Paths.get(path)).skip(1) Stream<String>
                                .map(s -> s.split( regex: ";")) Stream<String[]>
                                .filter(a -> IUser.IUserValido(a))
                                .map(a-> new User(a, load_friends: false)) Stream<User>
                                .collect(Collectors.toMap(u -> u.getUserID(),u -> u));
    }
    catch (IOException e){
        throw new UserIOException("Path do ficheiro de Users inválido e/ou erro de leitura.\nMore info:"+e);
    }
    return total;
```

Figura 6 - Código inicial do carregamento dos Users

```
public long load_users(String path) throws UserIOException {
    long total = 0;
    try {
        total = Files.lines(Paths.get(path)).skip(1).count();
        Files.lines(Paths.get(path)).skip(1).
                map(s -> s.split( regex: ";")).
                forEach(s -> addIUser(s));
    }
    catch (IOException e){
        throw new UserIOException("Path do ficheiro de Users inválido e/ou erro de leitura.\nMore info:"+e);
    }
    return total;
}
```

Figura 7 - Código após o primeiro desenvolvimento

```
public long load_users(String path) throws UserIOException {
    AtomicLong total = new AtomicLong();
    try {
        Files.lines(Paths.get(path)).skip(1).forEach(s-> {
                addIUser(s.split( regex: ";"));
                total.getAndIncrement(); });
    }
    catch (IOException e){
    throw new UserIOException("Path do ficheiro de Users inválido e/ou erro de leitura.\nMore info:"+e);
    }
    return total.get();
}
```

Figura 8 - Código final

Estratégias e Otimizações das Queries

Query 1: Percorrendo a lista de negócios, verificamos se o identificador existe no index bizToRev (negócios e as suas reviews), caso não exista, adiciona aos resultados. A ordem alfabética é garantida pelo SortedMap.

Query 2: Usando a data fornecida e o método "subMap", percorremos o index dataToRev e recolhemos as reviews feitas nesse mês, incrementado o contador e calculando a média.

Query 3: Utilizamos o auxílio de 3 estruturas/listas, uma que conta o número de reviews de cada mês, outra a média atribuída a cada mês e por fim a lista de negócios distintos. Dado o *User*, recolhemos a sua lista de reviews com auxílio ao index usrToRev e para cada review preenchemos a informação nas listas auxiliares.

Query 4: Semelhante à *query 3,* utilizamos 3 listas auxiliares sendo a lista de negócios distintos modificada para users distintos. Recolhemos a lista de reviews de um negócio através do index bizToRev e preenchemos a informação das 3 listas.

Query 5: Percorremos as reviews do *user* fornecido utilizando o index usrToRev, preenchendo uma lista de IDs de negócios e as suas ocorrências.

Query 6: Percorrendo a lista de reviews, preenchemos uma lista auxiliar de correspondência de negócios e o seu número de reviews por ano como também uma "blacklist", ou seja, uma lista de negócios com os users que já o avaliaram.

Query 7: Percorrendo o index bizToRev, recolhemos os vários negócios avaliados e contamos as suas reviews, guardando esta informação numa lista auxiliar de cidades e a sua lista de negócios/reviews correspondentes (Nested SortedMap).

Query 8: Percorrendo o index usrToRev e recorrendo ao auxílio do método "negócios_distintos" (devolve o número de negócios distintos dado uma lista de IDs de reviews) calculamos a lista de users e os seus negócios distintos.

Query 9: Percorrendo a lista de reviews do negócio fornecido (index bizToRev), recolhemos duas listas de users, uma contabiliza o número de reviews feitas a esse negócio e outra a média dada.

Query 10: Percorrendo o index bizToRev e recorrendo ao auxílio do método "calcula_media" (media a partir de uma lista de IDs de reviews) recolhemos as médias de cada negócio. À parte, também recolhemos uma lista completa de estados e cidades onde cada cidade tem por si uma outra lista de negócios correspondentes.

Nota: Para a recolha ordenada dos resultados das queries 5, 6, 7, 8 e 9, recorremos a um algoritmo rudimentar, percorrendo a lista auxiliar, identificando o maior resultado e de seguida retirando-o desta. Este processo repete o número de vezes necessário/pedido. Este processo tem algum impacto na performance das queries a longo prazo, mas visto que esta faz poucas iterações decidimos seguir por esta implementação.

Resultado dos Testes Realizados

Utilizando a nossa aplicação de teste (gestReviewsAppTEST) obtivemos os tempos médios e, para cada resultado apresentado, foram efetuados 10 testes de cada operação. O primeiro teste corresponde ao carregamento dos catálogos e os seguintes correspondem às respetivas queries.

Ex: Teste 2 -> Query 1 / Teste 11 -> Query 10

Sistema Operativo: Ubuntu 20.04.2 LTS (64bit)
Processador: Intel Core i5-8300H – 2.3GHz x 8
Memória: 7.6 GiB
Disco: Sandisk Corp SN520 NVMe SSD (256.1GB)
Placa gráfica: GeForce GTX 1050 Mobile
Estado: Ligado à corrente

```
RESULTADOS:
Teste 1: Tempo médio: 18,080 segundos  |  Memória utilizada: 1,584 gibibytes
Teste 2: Tempo médio: 47,160 milissegundos
Teste 3: Tempo médio: 318,144 milissegundos
Teste 4: Tempo médio: 5,690 milissegundos
Teste 5: Tempo médio: 758,789 microsegundos
Teste 6: Tempo médio: 522,529 microsegundos
Teste 7: Tempo médio: 4,663 segundos
Teste 8: Tempo médio: 27,953 milissegundos
Teste 9: Tempo médio: 1,140 segundos
Teste 10: Tempo médio: 394,576 microsegundos
Teste 11: Tempo médio: 967,728 milissegundos
```

Sistema Operativo: Manjaro 5.12.9-1 (64bit)
Processador: Intel Core i7-8550U – 1.8GHz x 4
Memória: 15.39 GiB
Disco: Western Digital SN720 NVMe SSD (476.94GiB)
Placa gráfica: Intel UHD Graphics
Estado: Ligado à corrente

```
RESULTADOS:
Teste 1: Tempo médio: 20,347 segundos  |  Memória utilizada: 1,734 gibibytes
Teste 2: Tempo médio: 40,933 milissegundos
Teste 3: Tempo médio: 328,201 milissegundos
Teste 4: Tempo médio: 4,315 milissegundos
Teste 5: Tempo médio: 780,077 microsegundos
Teste 6: Tempo médio: 461,307 microsegundos
Teste 7: Tempo médio: 6,448 segundos
Teste 8: Tempo médio: 32,693 milissegundos
Teste 9: Tempo médio: 1,484 segundos
Teste 10: Tempo médio: 567,697 microsegundos
Teste 11: Tempo médio: 1,068 segundos
```