Laboratórios de Informática III

MIEI/LEI

2020/2021

2º trabalho prático

Introdução

O projeto de Java da disciplina de LI3 tem por objetivo fundamental ajudar à consolidação experimental dos conhecimentos teóricos e práticos adquiridos na disciplina de Programação Orientada aos Objetos, procurando-se ainda capitalizar o mais possível o trabalho e os resultados obtidos no projeto de C. Pretende-se agora a criação de uma aplicação *D*esktop em Java, baseada na utilização das interfaces e das coleções de JCF (*Java Collections Framework*), cujo objetivo é a realização de consultas interativas de informações relativas à gestão básica de um sistema de recomendação/classificação de negócios.

Pré-requisitos

O projeto de Java tem como fontes de dados os mesmos ficheiros de texto usados no projeto C, pelo que a estrutura de cada linha de ficheiro de texto será a mesma.

Requisitos da aplicação a desenvolver

Pretende-se desenvolver uma aplicação desktop Java que seja capaz de, antes de mais, ler e armazenar em estruturas de dados (coleções de Java) adequadas às informações dos vários ficheiros, para que, posteriormente, possam ser realizadas diversas consultas (*queries*), algumas estatísticas e alguns testes de *performance*.

A classe agregadora de todo o projecto de Java será a classe `GestReviews` e a aplicação final, que seguirá o padrão MVC, designar-se-á por `GestReviewsAppMVC`.

A conceção da aplicação deverá seguir os princípios da **programação com interfaces**, ou seja, criando-se antes das classes as APIs, assim visando tornar a aplicação mais genérica e flexível. 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. Cada uma destas subestruturas de informação mantém as características e os requisitos que foram apresentados no projeto de C.

O desenho e a implementação do código da aplicação deverão ter em atenção que o mesmo deverá ter dois grandes grupos de funcionalidades (correspondentes a 70% do valor do trabalho):

- Leitura e validação de dados de memória secundária, a partir de ficheiros de texto, e população das estruturas de dados em memória central;
- Gravação da estrutura de dados (instância da classe `GestReviews`) em ficheiro de objetos;
- *Queries*: operações de consulta sobre as estruturas de dados.

Complementarmente, e correspondendo a 30% do valor do trabalho, pretende-se que seja desenvolvido um conjunto de pequenos programas (ou um que realize a globalidade dos testes), independentes da aplicação `GestReviewsAppMVC`, que realizem a funcionalidade designada por:

- Medidas de performance do código e das estruturas de dados.

Validação dos registos

Users: A lista de amigos poderá ser ignorada. Assim, um registo é válido se contiver 3 campos. No momento da leitura do registo, a informação do terceiro campo poderá ser descartada. Preferencialmente, por questões de expansibilidade, o programa deverá suportar a funcionalidade de opcionalmente carregar estes dados, através de uma simples mudança de configuração no projeto. Opcionalmente, podem ser realizados testes de *performance* entre leitura **com** *parsing* de amigos vs. leitura **sem** *parsing* de amigos.

Businesses: Um registo de *business* é válido se contém 5 campos, onde o último campo poderá conter texto vazio ou uma lista de categorias separadas por ",".

Reviews: Um registo de *review* é válido se contém 9 campos e os *id* de business e utilizador correspondem a um *business* e um utilizador registado. O último campo (*text*) poderá conter texto vazio.

Leitura, população das estruturas e persistência de dados

O programa deverá poder ler os ficheiros de texto a qualquer momento e carregar os respetivos dados para memória. Na primeira execução do programa a realização desta operação é obrigatória. Caso o utilizador não forneça nenhum caminho para os ficheiros de dados, o programa deve utilizar os nomes originais por omissão. A qualquer momento, deverá estar disponível uma opção que permita ao utilizador gravar toda a estrutura de dados de forma persistente usando `ObjectStreams`, criando por omissão o ficheiro *gestReviews.dat* ou um outro se indicado pelo utilizador. A qualquer momento também deverá o utilizador poder carregar os dados a partir de uma `ObjectStream` de nome dado, repopulando assim toda a informação da estrutura de dados até então existentes em memória. Tal poderá ser útil para evitar múltiplas validações.

Estatísticas

Sendo inúmeras as possíveis informações úteis a extrair da estrutura de dados, elas deverão ser agrupadas para o utilizador da seguinte forma:

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).

Consultas interativas

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.

A criação das consultas deve ser realizada de forma suficientemente estruturada de modo a que se torne simples alterar o identificador e texto da mesma no menu de consultas e invocar o método associado respetivo.

Todas as queries realizadas devem, antes mesmo da apresentação dos resultados, e qualquer que seja a forma como os resultados finais são apresentados, indicar ao utilizador os tempos de execução usando o método long `System.nanoTime()` que mede

diferenças de tempo em nanosegundos e que devem ser convertidas para segundos e milisegundos para apresentação (usar a classe `Crono`).

Apresentação do projeto e Relatório

Tal como o projeto de C, o desenvolvimento deste projeto deve ser feito colaborativamente com o auxílio de *Git/GitHub*. Os grupos de trabalho serão precisamente os mesmos do trabalho do projeto de C. Os docentes serão membros integrantes da equipa de desenvolvimento de cada trabalho e irão acompanhar semanalmente a evolução dos projetos.

A entrega do trabalho será efetuada através desta plataforma e os elementos do grupo serão avaliados individualmente de acordo com a sua contribuição para o repositório. Serão fornecidas algumas regras que os grupos deverão seguir para manter e estruturar o repositório, de forma a que o processo de avaliação e de execução dos trabalhos possa ser uniforme entre os grupos e possa ser efetuada de forma automática.

O projeto será extraído automaticamente do repositório de cada grupo imediatamente após a data de entrega. *Commits* posteriores à data de entrega com alterações efetuadas no código do projeto não serão consideradas para fins de avaliação. O projeto poderá ser desenvolvido com recurso ao *BlueJ* ou outros *IDEs* como *NetBeans*, *Eclipse* ou *IntelliJ*. O relatório do projeto de Java deverá ter a estrutura já conhecida, sendo de salientar agora a importância do diagrama de classes, do desenho da estrutura de dados usada e da apresentação dos resultados dos testes. Ainda que não seja para incluir no relatório, gere o javadoc do seu projeto e inclua-o na pasta docs do projeto.

O relatório final será entregue aquando da apresentação do projeto e servirá de guião para a avaliação do mesmo. Contudo, aquando da entrega do trabalho, o diagrama de classes e desenho da estrutura de dados deverá já estar disponível.

ANEXO

QUESTÕES E RECOMENDAÇÕES GERAIS

- Algumas das consultas devolvem coleções de “pares de coisas” ou mesmo triplos. Por exemplo, pares de (código de negócio, nº de *users*). É, portanto, aconselhável criar classes auxiliares que representam tais pares (que são muito importantes para as consultas). Por exemplo, a classe `PairBusNumUser` poderá representar os pares anteriores. Como alternativa, para pares temporários que não "mereçam" uma classe própria, a classe de JAVA `SimpleEntry<F,S>` (melhor que `Entry<>`) e os seus métodos simples resolvem facilmente tal problema.

- Instâncias de classes como `PairBusNumUser` poderão ter que ser guardadas em coleções como `HashSet<T>`. Porém, dado que pretendemos, na maioria dos casos, ter os resultados ordenados por um dado critério, o mais provável é que tais pares devam ser guardados em `TreeSet<T>`. Tal implicará, de imediato, a criação de uma ou mais classes que implementem a interface `Comparator<T>` respetiva, uma por cada algoritmo especial de ordenação dos pares, cf:

```
public class CompParesBusUser implements Comparator<PairBusNumUser>,
Serializable …
```

  Expressões *lambda* podem aqui ser muito úteis e simplificar muito a criação destes `Comparator<T>` porque nos permitem escrever código muito simples que, de imediato, pode ser usado no construtor do `TreeSet<T>`, cf. o exemplo:

```
Comparator<ParUserTotalReviews> compuser =
    (p1, p2) -> {
        if(p1.getTotalReviews() > p2.getTotalReviews()) return -1;
        if(p1.getTotalReviews() < p2.getTotalReviews()) return 1;
        else return 0;
    };
TreeSet<ParUserTotalReviews> pares = new TreeSet<>(compuser);
```

- No exemplo anterior procura-se também chamar a atenção para o facto de que, por omissão, devemos declarar todas as nossas classes como `Serializable`, já que, a qualquer momento, podemos pretender guardar instâncias suas em `ObjectStreams`. Todos os alunos que já usam Java7 podem beneficiar da designada notação do diamante (*diamond notation*) que se baseia no operador diamante (cf. <>). Desde Java 7, as declarações de coleções podem ser simplificadas na sua inicialização pois o compilador de Java 7 faz inferência de tipos. Assim, usando exemplos:

  Em vez de:

```
ArrayList<String> nomes = new ArrayList<String>();
```

  pode escrever-se apenas:

```
ArrayList<String> nomes = new ArrayList<>();
```

Em vez de:

```
TreeMap<Ponto2D, String> pmap = new TreeMap<Ponto2D, String>();
```

pode escrever-se apenas:

```
TreeMap<Ponto2D, String> pmap = new TreeMap<>();
```

NOTA FINAL: Nunca esquecer o <> !!

A declaração ERRADA `ArrayList<String> nomes = new ArrayList();` gera apenas um warning, mas grandes problemas durante a execução. O tipo `ArrayList()` é de Java 1 a Java 4 mas ainda existe em Java 7 (por razões de retrocompatibilidade) mas não é "type safe", ou seja, verificado em tempo de compilação.

- É absolutamente obrigatório o uso de `clone()` em todos os métodos interrogadores, ou seja, que consultam o estado interno de uma qualquer instância. Claro que *strings* e as instâncias das classes "*wrapper*" não necessitam de `clone()`. A expressão `HashSet<String> noDupRecs = new HashSet<>(linhas)`, que é um construtor de `HashSet<T>` que recebe como parâmetro um `ArrayList<T>`, elimina automaticamente duplicados, mas apenas está correta porque estamos a trabalhar com *strings* e *strings* são imutáveis e não necessitam de `clone()` ! Classes deverão ter também bons métodos `hashCode()` dado que tal torna muito mais eficiente a sua inserção ou remoção na maioria das coleções que necessitam de ordem e/ou indexação, por exemplo, `TreeSet<Ponto2D>`. A classe `Arrays` oferece um método `hashCode()` geral que poderemos usar de forma muito simples em qualquer classe, programando-o da seguinte forma:

```
public int hashCode() {
  return Arrays.hashCode(new Object[] {vari1, vari2, vari3, ...});
}
```

Passamos como parâmetro do método `Arrays.hashCode()` um array de objetos inicializado com as variáveis de instância dessa classe (tudo é compatível pois `Object` é superclasse de todas). É simples e elimina a necessidade de usarmos números primos.