Laboratórios de Informática III 2016/2017
Trabalho JAVA
Grupo 48

Cesário Miguel Pereira Perneta A73883
João Miguel Freitas Palmeira A73864
José Miguel Silva Dias A78494

09/06/2017

Conteúdo

1 Introdução

No âmbito da unidade curricular de Laboratórios de Informática III, e com o objetivo de aplicar os conteúdos aprendidos nas aulas práticas bem como noutras UC's, com especial relevo para a modularidade e encapsulamento de dados.
Foi-nos proposto desenvolver o mesmo projeto da componente de C em Java que tinha como objetivo analisar *snapshots* do Wikipedia retirando-lhes a informação necessária para responder às diversas interrogações impostas pelos docentes no enunciado do projeto.
O processo deve abranger todos os mecanismos de leitura dos *snapshots*, catalogar todos os artigos bem como as suas revisões e ainda todos os contribuidores contidos nesses mesmos *snapshots*, tendando estruturar o trabalho de modo a facilitar o processo de resposta ás interrogações colocadas (*queries*), tal como a rapidez de resposta da aplicação.

2 Estrutura da aplicação

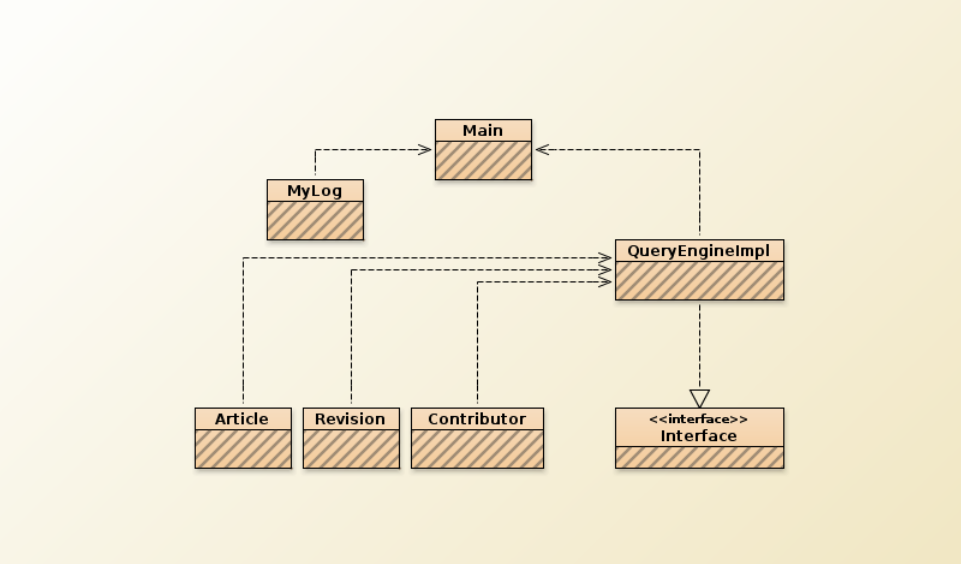

Figura 1: Estrutura do projecto em JAVA

A Classe ***Main*** foi uma Classe completa fornecida pelos docentes da UC, tal como a MyLog. A classe Main em conjunto com a ***MyLog*** trata de invocar os métodos declarados na **"Interface"** e escreve para dois ficheiros os outputs desses métodos bem como o tempo que demoraram a ser executados (em milissegundos).
A Classe ***QueryEngineImpl*** implementa a interface **"Interface"**, ou seja, aí localiza-se o código que realiza todos os calculos bem como a leitura e *parsing* dos *snapshots* da Wikipédia. Nesta Classe estão definidades 3 variáveis de instância, *Maps*, ou neste caso *HashMaps*, que são a nossa estrutura de dados para o programa.
A estrutura como já foi dito está organizada em 3 *Maps*, ***Map* de *Long* e *Article*** que organiza os artigos da Wikipédia de acordo com o seu identificador, ***Map* de *Long* e *Revision*** que organiza as revisões de artigos da Wikipédia de acordo com o seu identificador e ***Map* de *Long* e *Contributor*** que organiza os autores de revisões de artigos da Wikipédia de acordo com o seu identificador.
As Classes que definem os artigos, revisões e contribuidores são respectivamente a Classe *Article*, *Revision* e *Contributor*. Estas 3 Classes têm apenas os construtores, os métodos *get* e os métodos *set*, que eventualmente foram necessários.

2.1 Classes Básicas

Devido à similaridade entre estas três Classes que se seguem decidimos combiná-las num só capítulo.
Assim, consideramos as Classes que definem Artigos, Revisões e Contribuidores que são constítuidas por métodos e estruturação interna semelhantes.

2.1.1 Article

A Classe ***Article*** tem as seguintes variáveis de instancia e contrutores:

```
public class Article {
    private long id, nwords, nchar;
    private String titulo;
    private int dups;

public Article(){
    this.id = -1;
    this.nwords = -1;
    this.nchar = -1;
    this.titulo = null;
    this.dups = 0;
}

public Article(long id, long nw, long nc, String titulo){
    this.id = id;
    this.nwords = nw;
    this.nchar = nc;
    this.titulo = titulo;
    this.dups = 1;
}
public Article(Article a){
 this.id = a.getID();
    this.nwords = a.getNWords();
    this.nchar = a.getNChar();
    this.titulo = a.getTitulo();
    this.dups = a.getDups();

}
```

Esta Classe, além destes métodos, tem apenas definidos os métodos *get* e *set* que eventualmente foram necessários. Como pode notar na figura anterior, os *Article* têm 5 variáveis de instância:

- O *id* representa o identificador único de um artigo.
- O *nwords* representa o número de palavras da maior revisão deste artigo.
- O *nchar* representa o número de caracteres do texto da maior revisão deste artigo.
- O *titulo* representa o título do artigo.
- O *dups* representa o numero de vezes que se inseriu este artigo com este *id* no *Map*.

2.1.2 Revision

A classe ***Revision*** tem as seguintes variáveis de instância e construtores:

```
public class Revision {
    private long id, colID;
    private String nome,timestamp;

public Revision(){
    this.id = -1;
    this.colID = -1;
    this.nome = null;
    this.timestamp = null;
}

public Revision(long id, long colID,
String nome, String timestamp){
    this.id = id;
    this.colID = colID;
    this.nome = nome;
    this.timestamp = timestamp;
}
```

Esta Classe, além destes métodos, tem apenas definidos os métodos *get* e *set* que eventualmente foram necessários. Como pode notar na figura anterior, as *Revision* têm 4 variáveis de instância:

- O *id* representa o identificador único de uma revisão.
- O *colID* representa o identificador único do autor desta revisão.
- O *nome* representa o nome do autor desta revisão.
- O *timestamp* representa a data na qual se submeteu esta revisão.

2.1.3 Contributor

A Classe ***Contributor*** tem as seguintes variáveis de instância e contrutores:

```
public class Contributor {
    private long colID;
    private int nCont;
    private String nome;

public Contributor(){
    this.colID = -1;
    this.nCont = -1;
    this.nome = null;
}

public Contributor(long colID, String nome){
    this.colID = colID;
    this.nCont = 1;
    this.nome = nome;
}
```

Esta Classe, além destes métodos, tem apenas definidos os métodos *get* e *set* que eventualmente foram necessários. Como pode notar na figura anterior, os *Contributor* têm 3 variáveis de instancia:

- O *colID* representa o identificador único do autor desta revisão.
- O *nCont* representa o número de revisões da autoria deste *Contributor*.
- O *nome* representa o nome do autor desta revisão.

2.2 Implementação dos Objetivos

2.2.1 *QueryEngineImpl*

A Classe ***QueryEngineImpl*** contém todos os métodos responsáveis pela resolução das diversas interrogações (*queries*) propostas no enunciado do projeto. Estes métodos responsabilizam-se pela obtenção dos dados necessários para ober os resultados pretendidos, sendo que tomam partido das APIs de cada um dos objetos que compõem esta classe.
Ou seja, por exemplo, se for necessário obter a quantidade de artigos que estão no *snapshot*, através desta classe é possível realizar um *parsing* do *snapshot* (realiza-se no *load*) e posteriormente será chamado o método *all_articles* que contará o número de artigos presentes no documento *xml*.
Isto permite-nos assegurar o encapsulamento de dados e adiciona um nível de abstração que possibilita que alterações sejam feitas à estrutura da classe *Article*, sem que seja necessário, de certa forma, alterar a classe *QueryEngineImpl*.

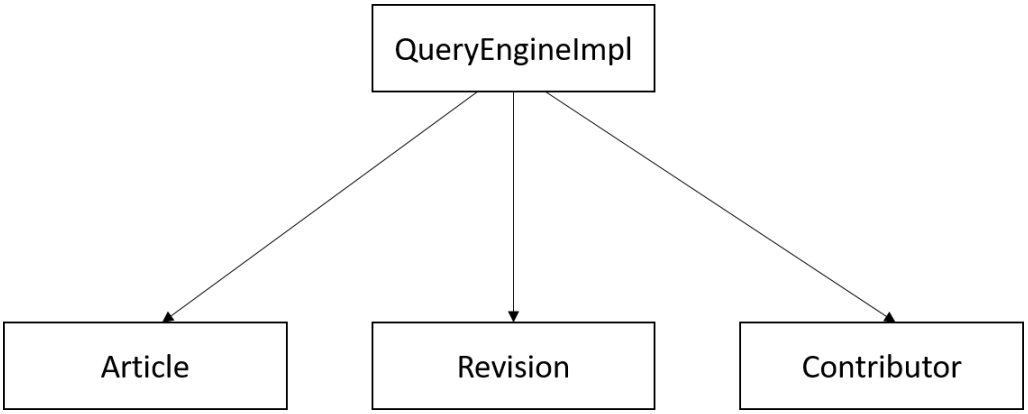

Figura 2: *QueryEngineImpl*

2.2.2 Estruturação

```
private HashMap<Long,Article> hashArts;
private HashMap<Long,Revision> hashRevs;
private HashMap<Long,Contributor> hashConts;
```

Dentro da Classe enunciada anteriormente é necessário definir ***Hash-***

Maps que guardem cada estrutura, ou seja:

- Um *HashMap* que guarde a estrutura de cada artigo lido no *load* de cada *snapshot*.
- Um *HashMap* que guarde a estrutura de cada revisão lida no *load* de cada *snapshot*.
- Um *HashMap* que guarde a estrutura de cada contribuidor lido no *load* de cada *snapshot*.

Todas as estruturas mencionadas encontram-se organizadas com um tamanho que depende exclusivamente da quantidade dos respetivos objetos encontrados no *load* dos *snapshots*. Estas estrutura são ainda iniciadas através da função *init* tal como podemos ver na seguinte imagem.

```
public void init(){
    this.hashArts = new HashMap<Long,Article>();
    this.hashRevs = new HashMap<Long,Revision>();
    this.hashConts = new HashMap<Long,Contributor>();
}
```

De seguida, temos o ***load*** que para além de carregar e fazer o *parsing* dos *snapshots* também insere a informação nas estruturas, ou seja, delimita a informação de cada *page* retirando a informação necessária para inserir em cada estrutura. É ainda utilizado um método de contar palavras (que é preciso para preencher uma das variáveis de instância da estrutura de cada artigo) e um *arry* de *bytes* de modo a saber o tamanho do cada revisão inserida.

```
public static long countWords(String s){
    long words = 0;
    char[] text = s.toCharArray();
    boolean lastspace  = false;
    if(text[0] == '\0') return 0;
    for(char c: text){
        if(c == ' ' || c == '\0'  ||
        c == '\n' || c == '\t'){
```

```
                if(!lastspace){
                    words ++;
                    lastspace = true;
                }
            } else lastspace = false;
        }
        return words;
    }
```

```
if(revText){
    byte characters.getData().get;
    revTextSize = b.length;
    revTextWcountWords(characters.ge);
    revText = false;
}
```

Ainda nesta Classe estão definidas todas as *queries*.
Grande parte delas são de fácil resolução (muito devido ás nossas estruturas, outras nao tão simples como é o caso das *top_10_contributors* , *top_20_largest_articles* e *top_N_articles_with_more_words* onde:

- ***top_10_contributors***: foi necessário a criação de um *comparator* que ordena os Contribuidores por ordem decrescente de acordo com o seu número de contribuições; De seguida criou-se uma *stream* que é ordenada de acordo com o comparador e limitada para ter apenas 10 elementos, substitui-se os elementos (Contribuidores) pelo seu número de contribuições e adiciona-se todos os elementos da *stream* para um novo *ArrayList*.
- ***top_20_largest_articles***: foi necessário a criação de um *comparator* que ordena Artigos de acordo com o tamanho do seu texto; De seguida criou-se uma *stream* que é ordenada de acordo com o comparador e limitada para ter apenas 20 elementos, substitui-se os elementos (Artigos) pelo tamanho do texto dos mesmos e adiciona-se todos os elementos da *stream* para um novo *ArrayList*.
- ***top_N_articles_with_more_words***: foi necessário a criação de um *comparator* que ordena os artigos por ordem decrescente de acordo

com o número de palavras; De seguida criou-se uma *stream* que é ordenada de acordo com o comparador e limitada para ter apenas n elementos, substitui-se os elementos (Artigos) pelo número de palavras do texto do mesmo e adiciona-se todos os elementos da *stream* para um novo *ArrayList*.

```
public ArrayList<Long> top_10_contributors() {
    Comparator<Contributor> cmp = new Comparator<Contributor>(){
         public int compare(Contributor a, Contributor b){
             return Integer.compare(b.getNCont(), a.getNCont());
         }
    };

    return this.hashConts.values().stream().sorted(cmp).limit(10)
    .mapToLong(c -> c.getColID())
    .collect(ArrayList :: new, ArrayList :: add, ArrayList :: addAll);
}
```

```
    public ArrayList<Long> top_20_largest_articles() {
         Comparator<Article> cmp = new Comparator<Article>(){
             public int compare(Article a, Article b){
                 return (int) (b.getNChar() - a.getNChar());
             }
          };
          return this.hashArts.values().stream().sorted(cmp)
         .limit(20).mapToLong(c -> c.getID())
         .collect(ArrayList :: new, ArrayList :: add, ArrayList :: addAll);
    }
```

```
public ArrayList<Long> top_N_articles_with_more_words(int n) {
     Comparator<Article> cmp = new Comparator<Article>(){
         public int compare(Article a, Article b){
             return (int) (b.getNWords() - a.getNWords());
         }
     };
     return this.hashArts.values().stream().sorted(cmp).limit(n).
     mapToLong(c -> c.getID())
     .collect(ArrayList :: new, ArrayList :: add, ArrayList :: addAll);
}
```

3 Tempos de Execução

```
INIT -> 1 ms
LOAD -> 15628 ms
ALL_ARTICLES -> 43 ms
UNIQUE_ARTICLES -> 0 ms
ALL_REVISIONS -> 0 ms
TOP_10_CONTRIBUTORS -> 12 ms
CONTRIBUTOR_NAME -> 0 ms
TOP_20_LARGEST_ARTICLES -> 14 ms
ARTICLE_TITLE -> 0 ms
TOP_N_ARTICLES_WITH_MORE_WORDS -> 21 ms
TITLES_WITH_PREFIX -> 8 ms
ARTICLE_TIMESTAMP -> 0 ms
CLEAN -> 2 ms
```

Como podemos visualizar pela figura a cima, o tempos de execução de cada *query* são, de um modo geral, bastante bons.
Apesar de o grupo achar que seria possível melhor o tempo de execução do *load*, mas apenas em relação ao *parsing* dos dados, isto é, se o *parser* fosse ainda mais eficiente, com certeza que o valor do *load* seria mais baixo.
Contudo seria mais complexo de elaborar e devido ao tempo restante, optamos por elaborar um *parser* mais legível de compreensão, fácil de utilização embora com tempos ligeiramente piores.

4 Conclusão

Com a realização deste projeto usando JAVA, o grupo ficou a conhecer a versatilidade que esta linguagem e paradigma de programação nos fornece, sem, no entanto, descurar os princípios de encapsulamento de dados que garantem a segurança e bom funcionamento do programa.
Foi garantido também um nível de abstração nas diferentes Classes para que, uma alteração à estrutura interna de uma dada classe, não danifique o comportamento do programa. Ou seja, desde que as assinaturas dos métodos de cada classe se mantenham as mesmas, alterações às estruturas de dados das mesmas apenas requerem pequenas correções nesses mesmos métodos, sendo que a reutilização de código é garantida em toda a API.
Para além de tudo isto, o grupo seguiu as instruções fornecidas pelos docentes e utilizou conceitos de JAVA 8, como é possível verificar no projeto bem como ao longo deste relatório.
Em suma, o grupo encontra-se bastante satisfeito com o resultado final, e os testes de performance demonstram que, em geral, foram tomadas as opções corretas no que toca à escolha das estruturas a utilizar no projeto, fazendo com que este culminasse num conjunto de Consultas Interativas e Estatísticas a executar em tempo praticamente constante.