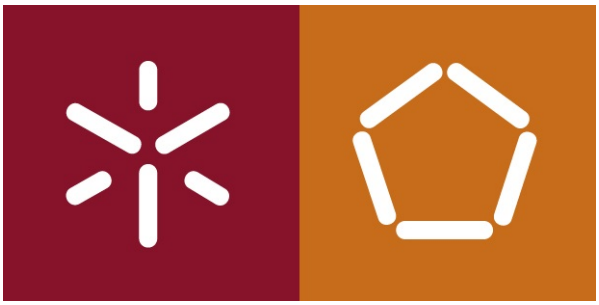
Universidade do Minho

Mestrado Integrado em Engenharia Informática



Gestor de Snapshots Wikipedia

Laboratórios de Informática III

Grupo 61

Trabalho realizado por:	*Número*
Bruno Ferreira	A74155
Marco Barbosa	A75278
Sara Pereira	A73700

11 de Junho de 2017

Conteúdo

Capítulo 1

Introdução

No âmbito da unidade curricular de Laboratórios de Informática III foi proposto que, neste segundo momento de trabalho no projeto, se desenvolvesse o mesmo sistema que foi implementado na fase em *C* e que lidava com a enorme quantidade de dados relacionados com a Wikipédia, mas só que desta vez sobre a forma de uma linguagem de programação orientada a objetos que é o *Java*.
À semelhança da fase anterior, os problemas como o grande volume de dados, a complexidade algorítmica e estrutural elevada mantêm-se, continuando por isso a ser um grande desafio. Nesse sentido, o desenvolvimento deste programa teve que ser cuidadoso, tanto nas classes desenvolvidas assim como na conceção de código reutilizável e escolha otimizada das estruturas de dados, tendo também um especial relevo na modularidade e encapsulamento dos dados.

1.1 Arquitetura da solução: diagrama de classes

Neste trabalho prático, tendo em conta agora a utilização do *Java* como ferramenta de trabalho, desenvolvemos o seguinte diagrama de classes como desenho para a solução do problema em questão:

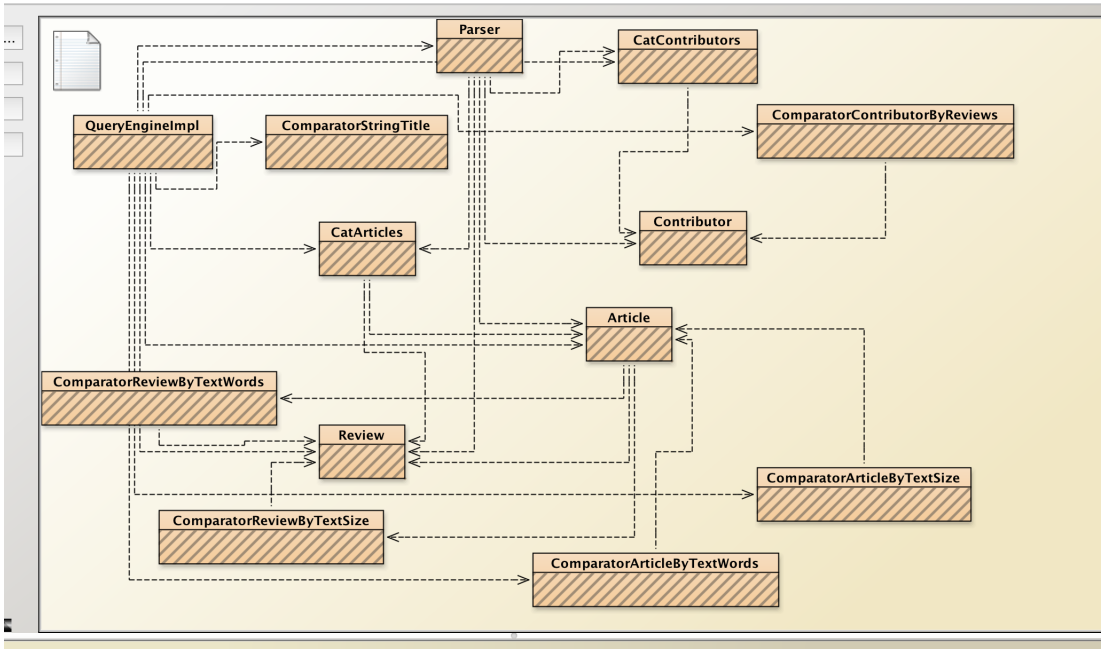


Figura 1.1: Estrutura do trabalho

Capítulo 2

Descrição das estruturas criadas para a abordagem do problema

2.1 Contributor

```
class Contributor{

    private long idContributor;
    private String username;
    private long reviewNumber;
```

Figura 2.1: Atributos do contribuidor

Esta classe representativa do contribuidor de uma revisão de um artigo, contém o seguintes atributos:

- *long* idContributor → que representa o id do contribuidor;
- *String* username → que representa o nome de usuário do contribuidor;
- *long* reviewNumber → que representa número de revisões feitas pelo contribuidor;

2.2 Review

```
class Review{

    private long idReview;
    private String timestamp;
    private long textSize;
    private long textWords;
```

Figura 2.2: Atributos da revisão

Esta classe representativa de uma Revisão de um dado artigo, contém o seguintes atributos:

- *long* idReview → que representa o id da revisão;

- *String* timestamp → que representa a data da revisão;
- *long* textSize→ que representa o tamanho em bytes da revisão;
- *long* textWords→ que representa o tamanho em palavras da revisão;

2.3 Article

```
class Article{

    private String title;
    private long id;
    private Map<Long,Review> reviews;
```

Figura 2.3: Estrutura do trabalho

Esta classe representativa de um Artigo, contém o seguintes atributos:

- *long* id → que representa o id do artigo;
- *String* title → que representa o título do artigo;
- *Map<Long,Review>* reviews → que representa uma estrutura contendo todas as revisões desse artigo;

2.4 CatArticles

```
class CatArticles{

    private Map<Long,Article> catArticle;
    private long dupedArticles;
    private long uniqueReviews;
```

Figura 2.4: Atributos do catálogo de artigos

Esta classe representativa do catálogo que guarda toda a informação sobre todos os artigos contém os seguintes atributos:

- *Map<Long,Article>* catArticle → que representa a estrutura (TreeMap) que guarda toda a informação dos artigos, onde Long (key) corresponde ao id do artigo, e Article (value) corresponde ao próprio artigo;
- *long* dupedArticles → que representa número de artigos repetidos que se encontram no catálogo;
- *long* uniqueReviews → que representa número de revisões não repetidas que se encontram no catálogo;

Dado que sempre que desejámos obter um catálogo de artigos, é-nos retornado um clone desse catálogo, devido à forma como está implementado o método

Map< *Long*, *Article* > getCatArticles(),

foi necessário que fossem criados alguns métodos para que pudéssemos trabalhar sobre o catálogo de artigos original (e não sobre o clone dele), como por exemplo:

- *void* addArticle(Article article) → que adiciona um artigo ao catálogo;

- *void* addReview(Long idArticle, Review review) → que adiciona uma revisão a um artigo do catálogo com um dado id;
- *void* changeTitle(Long idArticle, String title) → que permite alterar o título de um dado artigo do catálogo;
- etc;

2.5 CatContributors

```
class CatContributors{

    private Map<Long,Contributor> catContributors;
```

Figura 2.5: Atributos do catálogo de contribuidores

Esta classe representativa do catálogo que guarda toda a informação sobre todos os contribuidores contém os seguintes atributos:

- *Map<Long,Contributor>* catContributors → que representa a estrutura (TreeMap) que guarda toda a informação dos contribuidores, onde Long (key) corresponde ao id do contribuidor, e Contributor (value) corresponde ao próprio contribuidor;

Dado que sempre que desejámos obter um catálogo de contribuidores, é-nos retornado um clone desse catálogo, devido à forma como está implementado o método

Map< *Long*, *Contributor* > getCatContributors(),

foi necessário que fossem criados alguns métodos para que pudéssemos trabalhar sobre o catálogo de contribuidores original (e não sobre o clone dele), como por exemplo:

- *void* addContributor(Contributor contributor) → que adiciona um contribuidor ao catálogo;
- *void* addContribution(Long idContributor) → que adiciona uma contribuição a um contribuidor do catálogo com um dado id;
- *boolean* containsContributor(Long idContributor) → que testa se um contribuidor com um dado id está presente no catálogo de contribuidores;
- etc;

2.6 Parser

À semelhança da primeira fase do projeto, os snapshots fornecidos encontravam-se no formato XML, sendo por isso necessário haver a passagem para as estruturas em questão (ou para o catálogo dos artigos ou para o catálogo dos contribuidores). Como tal, foi usado o package *javax.xml.stream* para que fosse possível efetuarmos o parsing.

Capítulo 3

Interrogações

Após o desenho e a implementação das estruturas que servem de suporte ao desenvolvimento do programa pretendido, surge agora o momento de tirar partido de tais estruturas e começar a fazer interrogações ao programa, com o intuito de demonstrar o seu nível de praticalidade para o problema em questão e também perceber o nível de fiabilidade que ele proporciona.
Note-se também que foi criada a classe *QueryEngineImpl*, que contém todas as funções necessárias para "preparar as interrogações", classe essa que compreende as seguintes etapas de execução:

- Antes de qualquer tipo de interrogação que se deseja fazer ao programa, e à semelhança daquilo que se fez na primeira fase do projeto, é necessário que seja feita a inicialização tanto do catálogo que vai receber a informação dos contribuidores (TreeMap<>()) como do catálogo que vai receber a informação dos artigos (TreeMap<>()). Ao contrário do que acontecia no paradigma imperativo, nesta fase (pelo facto de se ter usado uma linguagem orientada a objetos) não é necessário alocar previamente memória para as estruturas, mas apenas inicializá-las na forma:

  ```
  public void init(){
      this.catArticles = new CatArticles();
      this.catContributors = new CatContributors();
  }
  ```

- Após a fase de inicialização das estruturas, surge o momento de "carregar"a informação presente nos snapshots para elas. Para que tal feito seja possível, é necessário, numa primeira instância, retirar o formato XML da informação dos snapshots, e depois carregar os dados que lá se encontram para as estruturas adequadas que já se encontram inicializadas. Este passo é feito pela função:

  load (int nsnaps, ArrayList< *String* > snaps_paths)

- Depois de as estruturas estarem inicializadas e carregadas corretamente com a informação dos backups (ou seja, após o parse dos snapshots), o programa está apto a responder às interrogações especificadas. A resposta às interrogações envolve a procura de informação nas estruturas criadas através de funções definidas, como por exemplo: *getUniqueReviews()*, *getDupedArticles()*, *getCatContributors*, etc.

- Por fim, quando cessam as interrogações ao programa, surge o momento de libertar o espaço em memória ocupado pela estrutura do catálogo de artigos e do catálogos dos contribuintes, sendo que para isso basta apenas:

```
public void clean(){
    catArticles = null;
    catContributors = null;
    System.gc();
}
```

Todos estes momentos do programa enunciados que são necessários para a resposta às interrogações, encontram-se implementados no ficheiro *QueryEngineImpl.java*, se bem que foram criados também os seguintes programas para lidar com a ordenação das diferentes coleções de objetos:

- *ComparatorArticleByTextSize* → que trata de comparar o tamanho (bytes) entre dois artigos;
- *ComparatorArticleByTextWords* → que trata de comparar o tamanho (palavras) entre dois artigos;
- *ComparatorContributorByReviews* → que trata de comparar o número de revisões entre dois contribuidores;
- *ComparatorReviewByTextSize* → que trata de comparar o tamanho (bytes) entre duas revisões;
- *ComparatorReviewByTextWords* → que trata de comparar o tamanho (palavras) entre duas revisões;
- *ComparatorStringTitle* → que trata de comparar o tamanho entre dois títulos de artigos;

Posto isto, falaremos agora em específico de algumas interrogações, explicitando a forma de resolução utilizada.

3.1 All articles

Esta interrogação procura contar os artigos presentes em todos os snapshots analisados, sabendo que os artigos duplicados em snapshots sucessivos e novas revisões de artigos serão também contabilizados.

Estratégia: Nesta interrogação a estratégia de resolução é simples, bastando para isso contar todos os artigos presentes no catálogo de artigos (através de um método *size()*) e a esse valor somar a contagem dos artigos repetidos que se encontram nesse mesmo catálogo (através de um método *getDupedArticles()*.

3.2 Top 10 contributors

Esta interrogação conta as revisões dos contribuidores, devolvendo os 10 identificadores daqueles que mais revisões fizeram. Note-se que caso hajam contribuidores com o mesmo número de revisões, ordena-os pelo seu identificador de forma crescente.

Estratégia: Nesta interrogação, é primeiro declarado um *ArrayList<Long>* que guardará os identificadores dos contribuidores com mais contribuições e também um

comparador *ComparatorContributorByReviews* que trata de comparar o número de revisões entre dois contribuidores.
Posto isto, a estratégia de resolução passa por ir ao catálogo dos contribuidores, ordena-los de forma decrescente do número de contribuições e tirar os 10 primeiros ((...)*.stream().sorted(comparator).limit(10).*(...)), incluindo no ArrayList o identificador desses mesmos contribuidores.

3.3 Titles with prefix

Esta interrogação devolve os títulos dos artigos que contêm o prefixo especificado, ordenados alfabeticamente. Note-se que caso os títulos tenham sido alterados em algumas revisões, é considerado o título da revisão mais recente do artigo e que o array de títulos tem de terminar com o elemento NULL.

Estratégia: Nesta interrogação, é primeiro declarado um *ArrayList<String>* que guardará os títulos dos artigos que contém o prefixo especificado, e também um comparador *ComparatorStringTitle* que trata de comparar o tamanho entre dois títulos de artigos.
Posto isto, a estratégia de resolução passa por ir ao catálogo dos artigos e retirar os títulos dos artigos que têm o prefixo ((...)*.stream().filter(e->e.isTitlePrefix(prefix) == true).*(...)), sendo que o método *isTitlePrefix* trata de verificar se cada título se inicializa com o prefixo dado.+

3.4 Top N articles with more words

Esta interrogação identifica os artigos com maior número de palavras, devolvendo os 'n' identificadores daqueles que apresentam maior contagem de forma crescente. Note-se que caso hajam artigos com o mesmo número de palavras, ordena-os pelo seu identificador de forma crescente.

Estratégia: Nesta interrogação, é primeiro declarado um *ArrayList<Long>* que guardará os identificadores dos artigos com mais palavras e também um comparador *ComparatorArticleByTextWords* que trata de comparar o tamanho (palavras) entre dois artigos. Posto isto, a estratégia de resolução passa por ir ao catálogo dos artigos, ordena-los pelo número de palavras (através do comparador) e tirar os 'n' primeiros ((...)*.stream().sorted(comparator).limit(n).*(...)), incluindo no ArrayList o identificador desses mesmos artigos.

Capítulo 4

Conclusão

Após a conclusão desta segunda e última fase de desenvolvimento do projeto, podemos afirmar que, nesta fase em particular, a principal dificuldade encontrada foi o facto do volume de dados analisado ser bastante elevado, daí que tenha sido tido um cuidado redobrado com a forma como o programa foi estruturado, de forma a que permitisse uma maior rapidez em *run time*. Inicialmente o programa demorava cerca de 600 segundos a realizar o load na estrutura correta, verifando-se depois que o mesmo acontecia porque cada vez que se fazia a inserção de um elemento, estavamos a ir buscar uma copia de todos os elementos já presentes, fazendo com que o programa demorasse bastante tempo. Este foi depois corrigido, voltanto a um tempo de execuçao útil.
Após uma análise global acerca da problemática em questão e da maneira como o programa desenvolvido lida com ela, acreditamos veemente que desenvolvemos uma boa solução, sendo essa capaz de processar o volume de dados apresentado e produzir respostas a interrogações que lhe são colocadas num tempo útil.
Por fim, acrescenta-se também que este projeto foi positivo na medida em que nos permitiu implementar e aprofundar o conhecimento de algorítmos/estruturas abordados noutras unidades curriculares, assim como desenvolver a nossa capacidade de programação numa linguagem orientada a objetos que é o *Java*.