Universidade de Aveiro
Departamento de Electrónica, Telecomunicações e Informática
MEI

Simple Document Indexer – Assignment 1

David Ferreira, 72603 | Pedro Matos, 73941

10/10/2017

Trabalho realizado no âmbito da disciplina de

Recuperação de Informação

Índice

Resumo

Neste relatório não só vai ser explicado o conteúdo e propósito de cada classe desenvolvida como também as escolhas tomadas em grupo durante o desenvolvimento do projeto de forma ao mesmo ser o mais eficiente e modelar possível.

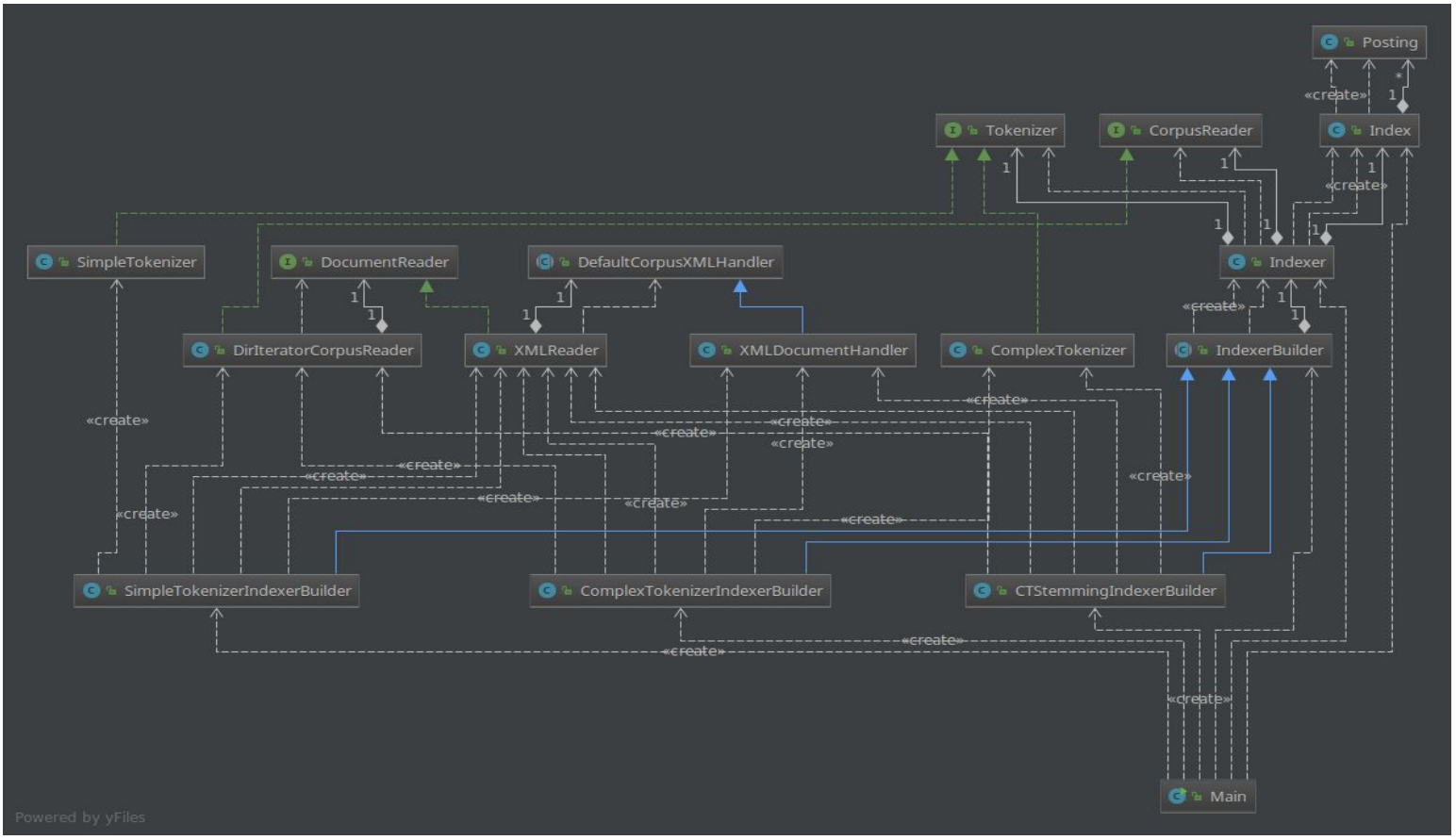

Figura 1 - Diagrama de classes da aplicação

Neste diagrama de classes é possível obter um panorama geral da forma como as classes desenvolvidas se relacionam e trocam informação entre si de forma a possibilitar o correto funcionamento do projeto desenvolvido.

Ao todo, foram desenvolvidas 17 classes das quais 3 são interfaces de forma a oferecer modularidade ao projeto desenvolvido.

Ao longo do relatório vão ser explicadas as funções que competem a cada classe desenvolvida. No entanto, para além da explicação presente no relatório está também presente em cada classe o JavaDoc associado.

Classes desenvolvidas

3.1 Interface CorpusReader

Esta interface define a estrutura que os "readers" de uma coleção de documentos (corpus) vão ter que respeitar, oferecendo assim modularidade ao projeto no sentido em que poderão existir imensos tipos de leitores de documentos, desde que a interface definida seja respeitada.

Esta interface define então 3 métodos que têm que ser implementados pelas classes que a implementam: "hasDocument()" que indica se o corpus ainda tem documentos a serem lidos, "getDocumentId()" que retorna o ID do documento a ser lido de momento e "processDocument()" que processa o documento a ser lido retornando assim uma string com o conteúdo do mesmo.

3.2 Interface DocumentReader

De uma forma muito similar à interface anterior, o document reader define a estrutura que os "readers" de documentos vão ter que respeitar.

Esta interface define então 3 metodos: "open()" responsavel por abrir o ficheiro a ser lido, "parse()" responsavel por ler o conteudo do ficheiro e retornar o conteudo do mesmo e ainda o metodo "getDocumentID()" que retorna o ID do documento a ser actualmente lido.

3.3 Interface Tokenizer

A interface tokenizer define a estrutura que os diferentes "tokenizers" vão ter que respeitar.

A função portanto definida é a função "Tokenize()" que para um dado conteúdo lido de um documento, vai efetuar as regras de tokenização dependendo do tokenizer a ser usado, criando assim uma lista com todos os tokens, isto é, a lista dos termos a serem usados como entradas no dicionário do nosso index invertido.

3.4 Classe ComplexTokenizer

A classe "Complex Tokenizer" é uma das classes desenvolvidas que implementam um Tokenizer, cada um, com regras de tokenização diferentes.

Esta classe implementa a interface Tokenizer, e é responsável por criar uma coleção de tokens com base no conteúdo de um documento.

Esta classe vai ler um set de StopWords que é lida de um ficheiro presente no projeto preenchida com StopWords e filtra os tokens baseando-se nesse set.

A aplicação de stemming é opcional em que caso o utilizador deseje aplicar stemming, tem ele que ser passado como argumento do construtor.

Na implementação definida pelo grupo, dependendo do argumento, "ct" ou "cts", vai ser definido caso o processo de stemming será ou não aplicado.

Em suma, esta classe contém dois construtores, um com e outro sem stemmer, e contém quatro métodos implementados: o "tokenize()" em que é efetuado todos os processos de tokenização para o conteúdo do documento sendo depois retornada a lista de tokens retirados do mesmo, a "fillStopWordsList()" em que é lida a lista de StopWords a serem usadas no processo de tokenização, a "filterStopWords()" que elimina os tokens que correspondam a stopWords e finalmente o método "stemmize()" que é apenas aplicado o processo de stemming caso seja passado como argumento ao construtor.

3.5 Classe SimpleTokenizer

A classe "Simple Tokenizer" é a outra classe também responsável pelo processo de tokenização do conteúdo lido de um dado documento.

Tal como o nome indica, o processo de tokenização aplicada nesta classe é mais simplista que a aplicada no processo da classe anterior.

Esta classe contém apenas o método "Tokenize()" que para um dado conteúdo de um documento identifica os termos através da eliminação de palavras com menos que três caracteres, eliminação de caracteres especiais e apenas considerando caracteres alfabéticos.

Nesta classe, não são efetuados processos de Stemming ou StopWord Removal.

3.6 Classe XMLReader

A classe XML reader implementa a interface "DocumentReader" sendo a classe usada para leitura de documentos XML.

Esta classe vai apenas conter os métodos aos quais tem que respeitar por implementar a interface: O método "open()" que abre o ficheiro a ser lido, o método "getDocumentID()" que retorna o ID do documento actual a ser lido e por ultimo o método "parse()" que vai efetuar o parsing do documento XML através de um SaxParser que é inicializado no construtor da própria classe.

Foi definido que o conteúdo retornado pelo parse, isto é, o conteúdo aproveitado do documento, será apenas referente às tags do “Title” e “Text” do mesmo.

3.7 Classe DefaultCorpusXMLHandler

Esta classe é uma implementação abstrata de um DefaultHandler (SaxParser) em que é adicionada complexidade à classe por forçar os utilizadores a usarem a mesma para implementar métodos que são úteis ao processar dos documentos do Corpus.

Vai então conter dois métodos, sendo um o “getID()” que retorna o ID do ficheiro a ser actualmente lido e o “getText()” que retorna o conteúdo lido e aproveitado do ficheiro.

3.8 Classe XMLDocumentHandler

Esta classe estendendo a DefaultCorpusXMLHandler é responsável por fazer o tratamento dos documentos XML na fase de leitura de forma a apenas aproveitar o conteúdo relevante a ser lido e retornado do mesmo, isto é, o conteúdo das tags “Title” e “Text”.

Os métodos “startElement()” e “endElement()” são responsáveis por verificar o inicio e fim de cada uma das Tags do ficheiro XML a ser actualmente processado.

Já o método “characters()”, dependendo da Tag do documento, concatena a informação nelas contida a uma String que no final do processamento do ficheiro vai ser retornada contendo esta o texto de interesse do documento processado.

3.9 Classe IndexerBuilder

Esta classe abstrata serve de ponto de partida para a implementação do padrão Builder com o propósito de criar uma instância de um “Indexer”.

Uma classe” Indexer” necessita de um “CorpusReader” e um “Tokenizer”, como tal, esta classe permite muitas configurações dependendo das preferências do utilizador, tendo sido portanto utilizado o padrão “Builder “ de forma a esconder parte da complexidade.

O método “ConstructIndex()” desta classe é responsável pela instanciação e atribuição do “Indexer”, do “Corpus Reader” e do “Tokenizer” a ser utilizado pela classe “Indexer”.

3.10 Classe SimpleTokenizerIndexerBuilder

Esta classe é uma implementação de um "IndexerBuilder" em que é construido um "Indexer" com a instanciação e atribuição de um "SimpleTokenizer" e um "DirIteratorCorpusReader".

Para além destas atribuições, a classe contém também um método "initializeReader()" que é responsável pela instanciação de um "Reader" XML que vai ser usado para processar os documentos do tipo XML.

3.11 Classe ComplexTokenizerIndexerBuilder

Esta classe é também uma implementação de um "IndexerBuilder", no entanto, ao invés da anterior, é então realizada uma instanciação de um "ComplexTokenizer" (Sem processo de Stemming).

Também nesta classe é contido um método "initializeReader()" que é responsável pela instanciação de um "Reader" XML que vai ser usado para processar os documentos do tipo XML.

3.12 Classe CTStemmingIndexerBuilder

Por último no que toca às classes "IndexerBuilder", o que distingue esta da última é que ao invés de ser instanciado um "ComplexTokenizer" sem processo de Stemming, este vai então conter esse mesmo processo.

Também nesta classe é contido um método "initializeReader()" que é responsável pela instanciação de um "Reader" XML que vai ser usado para processar os documentos do tipo XML.

3.13 Classe DirIteratorCorpusReader

Esta classe é um dos possíveis Corpus Readers sendo que implementa a interface definida para os leitores dos documentos do Corpus.

Este "Reader" simplesmente faz uma iteração sobre os ficheiros, que são iterados por ordem alfabética, de um dado diretório retornando para cada um o seu conteúdo.

Esta classe contém três métodos: O método "hasDocument()" que indica se ainda há ficheiros para ler, o "getDocumentID()" que retorna o ID do documento a ser actualmente lido e por último o "processDocument()" em que lê o conteúdo de interesse do ficheiro e retorna-o numa String.

3.14 Classe Index

Esta Classe define uma estrutura de dados que representa o index invertido em que guarda para cada termo, "Postings" que indicam o id do documento em que ele aparece tal como a sua frequência de ocorrência para esse mesmo id.

Como estrutura de dados foi utilizado um HashMap que vai conter para cada String (Termo) um ArrayList de "Postings" em que cada um representa o tuplo acima referido com um "docID" e o "TermFrequency".

Esta classe possui dois métodos de operacionalização da estrutura de dados: "addTokenOccurence()", em que é adicionado um novo termo ou simplesmente actualizado a sua lista de "Postings" caso o mesmo já exista, e o método "findEntry()" que verifica a existência de um dado termo e "Posting" para esse mesmo dado termo no HashMap.

Além destes métodos, contêm outros dois que servem de querys: "getTopFreqTerms()" que retorna os dez termos mais frequentes nos ficheiros do Corpus e "getTop10TermsOccurences()" em que para um dado "DocId" retorna os primeiros dez termos nele contidos ordenados alfabeticamente.

Finalmente, os últimos dois métodos contidos nesta classe são usados para escrever o index invertido num ficheiro em disco "saveIndexToFile()" e para retornar o tamanho do dicionário do index "vocabularySize()".

3.15 Classe Indexer

O propósito desta classe é de criar um Index, "createIndex()", que armazena as ocorrências de termos nos documentos do corpus.

Isto é feito através do uso de um "CorpusReader" que vai iterar sobre os ficheiros, retornando o seu conteúdo relevante, sendo depois definidos os termos através de um "Tokenizer" com base nesse conteúdo.

Após obtidos os termos, eles são utilizados pelo Index de forma a preencher o "Inverted Index" através do método "fillIndexWithTokens()".

3.16 Classe Posting

Esta classe representa uma "entry" do index, como tal, o seu propósito é representativo.

Ela é responsável por guardar o ID de um documento (docID) onde um certo termo ocorre tal como o número de ocorrências desse mesmo termo no próprio documento (termFreq).

3.17 Classe Main

Esta é a classe Main do projeto, classe onde os argumentos vão ser processados e de acordo com eles vão ser instanciadas e definidos os atributos para o apropriado decorrer do processo de indexação.

Os argumentos recebidos são o nome do documento, que representa o Corpus, e o tipo de Tokenizer a ser utilizado dentro dos três actualmente existentes.

Esta classe é também responsável por apresentar ao utilizador informações pertinentes relativas à execução do processo de indexação, como por exemplo, o tempo de indexação, o tamanho do dicionário do index invertido construído, o máximo de memória utilizada durante o processo de indexação e ainda o resultado das querys construídas.

4 Padrões de Desenho de Software

Numa tentativa de melhorar a legibilidade, modularidade e abstrair o utilizador da complexidade do sistema, foram aplicados padrões de desenho de software comuns em qualquer aplicação.

4.1 *Strategy*

Este padrão é aplicado com bastante frequência ao longo do projeto. Este padrão consiste em programar em função de uma interface que descreve um determinado comportamento que é esperado de certas entidades. Dessa forma à completa abstração de como os algoritmos são

implementados e permite modularidade pois é possível a classe que deriva da interface sem afetar o resto do sistema[1].

Pode-se observar a utilização deste padrão neste sistema na Figura 1, é possível observar as várias implementações dos interfaces “Tokenizer”, “DocumentReader”, etc.

4.2 *Builder*

Este padrão tem como objetivo separar a construção de um objeto complexo da sua representação, ou seja, um objeto que faça a agregação de várias entidades e que, consoante o seu processo de construção possa ter várias representações. A solução passa por criar um protocolo que possibilite a criação de todas as representações desse objeto complexo, definir esse processo num interface. De seguida atribuir a responsabilidade dessa construção do objeto a uma classe derivada desse interface, cada representação irá corresponder a uma classe. Um cliente pede ao builder uma representação do objeto e este encarrega-se da complexidade da construção[2].

Uma situação semelhante surgiu durante a construção de um objecto de tipo “Indexer”. Como um dos objetivos do sistema é permitir que o utilizador escolha que tipo de “Tokenizer” prefere usar, *tokenizer* simples, *tokenizer* complexo com ou sem *stemming*. Um objeto do tipo “Indexer” para além do tipo de “Tokenizer” também pode ter diferentes tipos de “CorpusReader”, Figura 1. Assim de maneira a separar a complexidade de criar todas estas representações da classe “Indexer”, decidiu-se adotar o padrão *Builder*. Como se pode observar na Figura 1, a classe “Main” usa três classes que derivam da classe “IndexerBuilder”, cada uma possível representação da classe “Indexer”, abstraindo-se assim da complexidade de criar cada uma delas.

5 Medidas de Eficiência

Visto que o sistema tem como objetivo processar grandes quantidades de informação a eficiência é um fator importante a ter em consideração. Nesta secção são apresentadas algumas medidas eficiência do sistema. É importante constatar que estas medidas foram registadas numa máquina Quad-core, Intel Core i7-2630QM [3].

5.1 Tempo de Indexação

Os tempos de indexação foram medidos subtraindo o valor entre dois *TimeStamps*, um no início do programa e outro no final da indexação, para tipo de *Indexer* o programa foi executado três vezes e a média dos tempos foi calculada, os resultados podem ser observados na tabela da Figura 2.

Como se pode observar o *Indexer* com *ComplexTokenizer* e *Stemming* é o que tem os maiores tempos de execução.

Tipo de *Indexer*	1º execução	2º execução	3º execução	Média
ComplexTokenizer com Stemming	1545	1620	1548	1571
ComplexTokenizer sem Stemming	1147	953	1031	1043,666
SimpleTokenizer	1130	929	989	1016

Figura 2 - Resultados dos tempos de indexação para os diferentes tipos de *Indexer.*

5.2 Quantidade máxima de memória

Estes resultados foram obtidos, usando a ferramenta *Profiler*[4], disponível *IDE NetBeans*. Cada amostra foi obtida após a limpeza da cache do sistema.

Para execuções com com o *ComplexTokenizer* sem *Stemming* o sistema usou aproximadamente 36.04513MB de memória heap, Figura 3, já com o *ComplexTokenizer* com *Stemming* usou 34.16074MB, Figura 4, finalmente, o *SimpleTokenizer* usou 33.56773MB, Figura 5.

5.3 Espaço ocupado pelo *index* em disco

Depois da indexação o *index* é armazenado em disco num ficheiro com o nome, por defeito, de *index.csv* e ocupa o espaço representado na Figura 6.

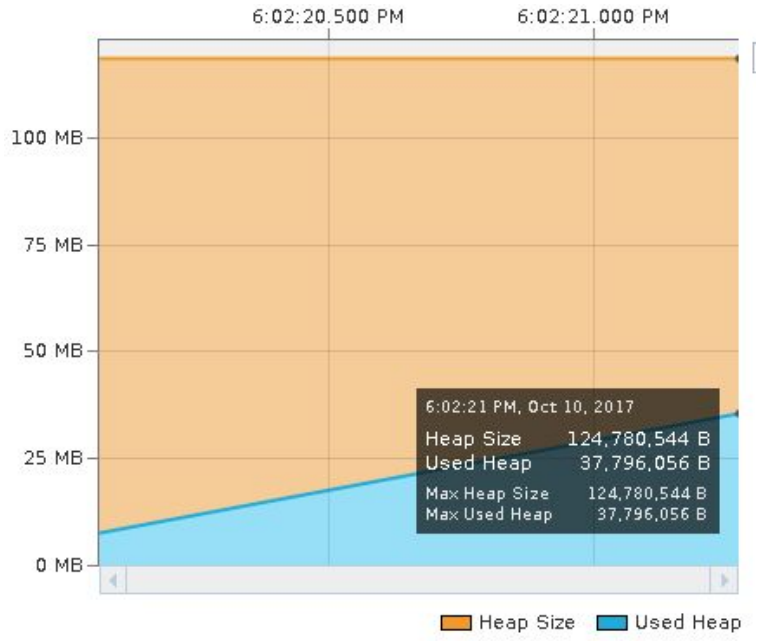

Figura 3 - Memória usada na execução do sistema usando *ComplexTokenizer* sem *Stemming*.

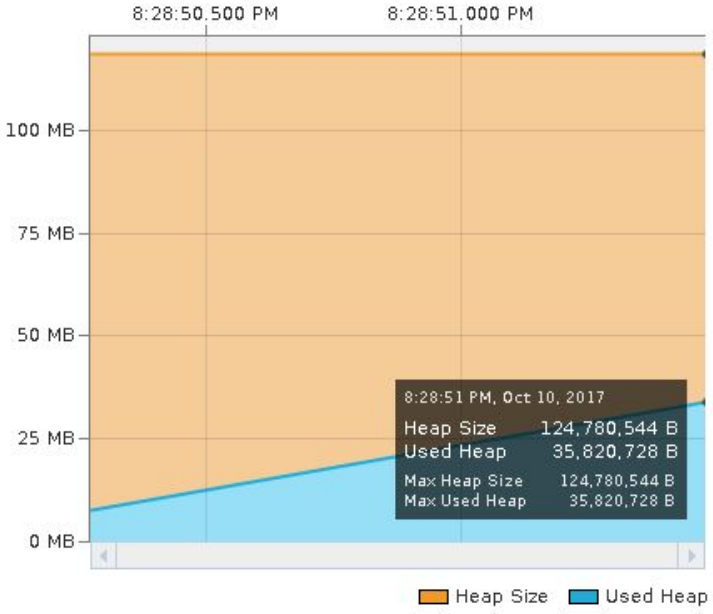

Figura 4 - Memória usada na execução do sistema usando *ComplexTokenizer* com *Stemming*.

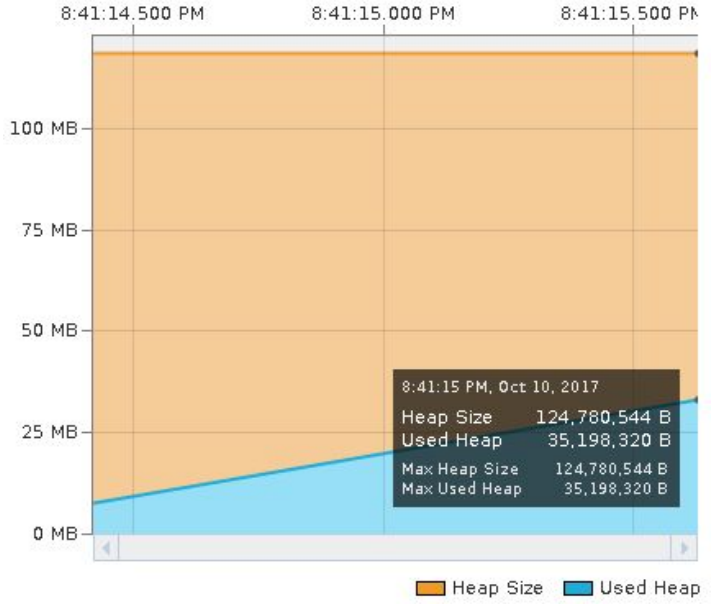

Figura 5 - Memória usada na execução do sistema usando *SimpleTokenizer*.

Tipo de Indexer	Tamanho do Index em disco
ComplexTokenizer com Stemming	536KB
ComplexTokenizer sem Stemming	608KB
SimpleTokenizer	712KB

Figura 6 - Espaço ocupado em disco pelo *index* para os diferentes tipos de *Indexer*.

6 Conclusões

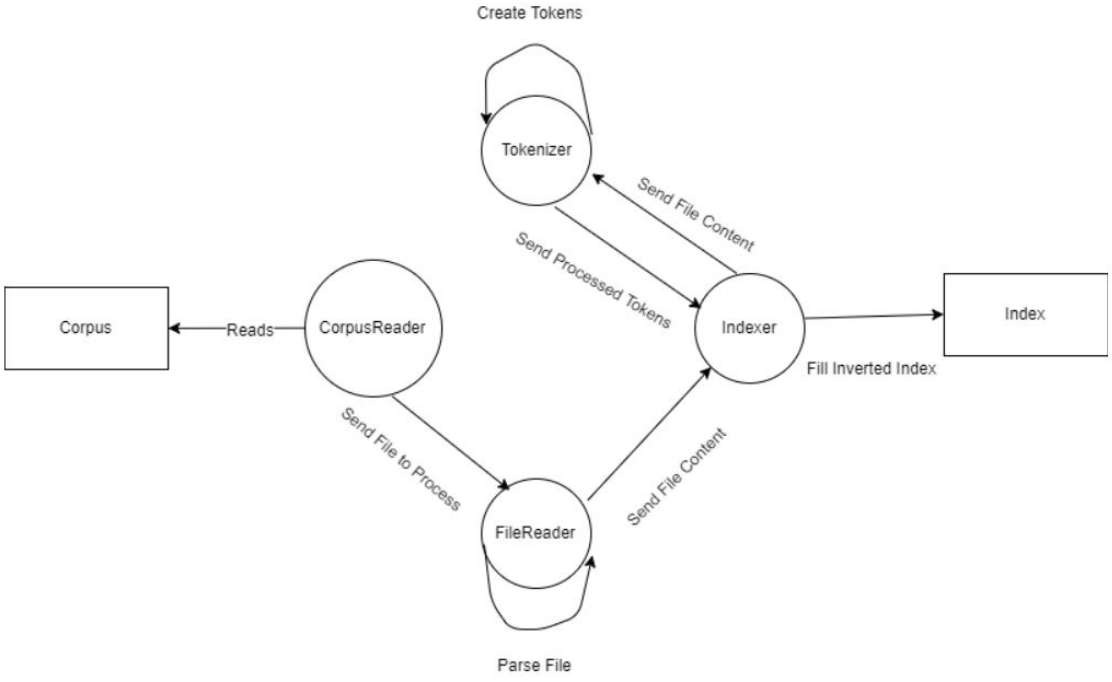

Figura 7 - Data Flow diagram do sistema desenvolvido

Em suma, e como é possível verificar no diagrama acima representado,Figura 7, o sistema desenvolvido possui 6 “blocos” principais que operam entre si de forma a permitir uma apropriada construção e preenchimento de um *Index* Invertido com base na leitura do conteúdo de um dado *Corpus*.

O nome do diretório que representa o *Corpus* é passado como argumento e vai ser então lido por um *CorpusReader*, que de forma iterativa envia para o *DocumentReader* cada ficheiro para ser analisado, extraindo-se assim o seu conteúdo relevante para uma *String*.

Para cada ficheiro lido essa *String* resultante é enviada para o *Indexer* que por sua vez a envia para o *Tokenizer*, de forma a que segundo as estratégias definidas e implementadas sejam efetuados os processo de normalização ao conteúdo da String de forma a se obter os termos resultantes.

Após obtida a lista de termos, ela é enviada pelo Tokenizer ao Indexer, que de forma também iterativa, vai construir o Index Invertido.

No final da execução do sistema, o Index Invertido resultante é escrito para um ficheiro em disco.

É de notar que com este primeiro assignment foi possível obter uma abordagem prática aos conteúdos lecionados nas aulas teóricas, tendo sido possível desenvolver um sistema, embora reduzido, de um construtor de um Index Invertido.

Para além disso, foi possível também obter medições de eficiência, tais como a utilização de memória, tempo de execução e espaço do Index no disco de forma a relacionar estes dados com dados a serem registados nos próximos assignments.

Um dos aspetos a notar do desenvolvimento deste projeto é que foram também utilizados dois padrões de programação de forma a não só esconder grande parte da complexidade, no caso do Builder pattern, mas também de forma a explorar o polimorfismo sem estender a classe main, que é o caso do Strategy.

Foi tido também em atenção a criação de Interfaces de forma a acrescentar modularidade ao projeto e escondendo também a sua complexidade, dando ainda ao utilizador a possibilidade de expandir o sistema a diferentes níveis, como por exemplo, o fácil desenvolver de um novo Reader para um tipo de formato de ficheiro diferente.

7 Respostas ao exercício 4

7.1 Vocabulary Size:

Complex Tokenizer (without Stemming) vocabulary size: 9053 terms.

Complex Tokenizer (with Stemming) vocabulary size: 6289 terms.

Simple Tokenizer vocabulary size: 8337 terms.

7.2 Top 10 Terms of Doc with id 1 (can be any id) ordenados alfabeticamente:

Simple Tokenizer:

[aerodynamics, after, agree, and, angles, attack, basis, boundarylayercontrol, comparative, configuration]

Complex Tokenizer (without Stemming) :

[aerodynamics, agree, angles, attack, basis, boundarlayercontrol, comparative, configuration, curves, destalling]

Complex Tokenizer (with Stemming):

[aerodynam, agre, angl, attack, basi, boundarylayercontrol, compar, configur, curv, destal]

7.3 Top 10 Terms with higher frequency

Simple Tokenizer:

[are, and, results, this, for, flow, with, from , that, the]

Complex Tokenizer (without Stemming) :

[obtained, boundary, number, pressure, flow, given, results, theory, mach, method]

Complex Tokenizer (with Stemming):

[obtain, use, effect, method, theori, present, flow, result, pressur, number]

8 Referências

[1] - ToturialsPoint. 2017. Design Patterns Strategy Pattern [Online]. Available: https://www.tutorialspoint.com/design_pattern/pdf/strategy_pattern.pdf

[2] - ToturialsPoint. 2017. Design Patterns - Builder Pattern
[Online]. Available: https://www.tutorialspoint.com/design_pattern/builder_pattern.htm

[3] - Intel Corporation. 2014. Intel® Core™ i7-2630QM Processor [Online]. Available: https://ark.intel.com/products/52219/Intel-Core-i7-2630QM-Processor-6M-Cache-up-to-2_90-GHz

[4] - Oracle Corporation and/or its affiliates. 2014. Profiler [Online]. Available: https://profiler.netbeans.org/