University of Aveiro

Departamento de Electrónica, Telecomunicações e Informática

Master degree in Informatics Engineering

Assignment nº2

Authors:
David Ferreira, Nº 72603
Pedro Matos, Nº 73941

Information Retrieval

23rd October 2017

Contents

Summary

In this report not only will be explained the content and purpose of each developed class but the choices that were taken in group during its development in order for the system to be as modular and efficient as possible. In this class diagram it is possible to check the overall structure of the developed system and the way that the implemented classes communicate with each other. For this second assignment additional classes were developed apart from the ones that were already developed in the first one. The last ones are being mentioned since this assignment is a continuation of the last developed one. Throughout this report it will be explained the functions and purposes of each developed class and even though there will be given an explanation for that in here, there is still present on each class its corresponding *JavaDoc* as well.

Classes

In this chapter, the classes that were developed for this assignment will be analyzed and described. Figure 1 represents the class diagram of the application, as it can be observed, the application is separated in several packages, the majority of the classes that were developed for this assignment are located in the package *query*, Figure 2.

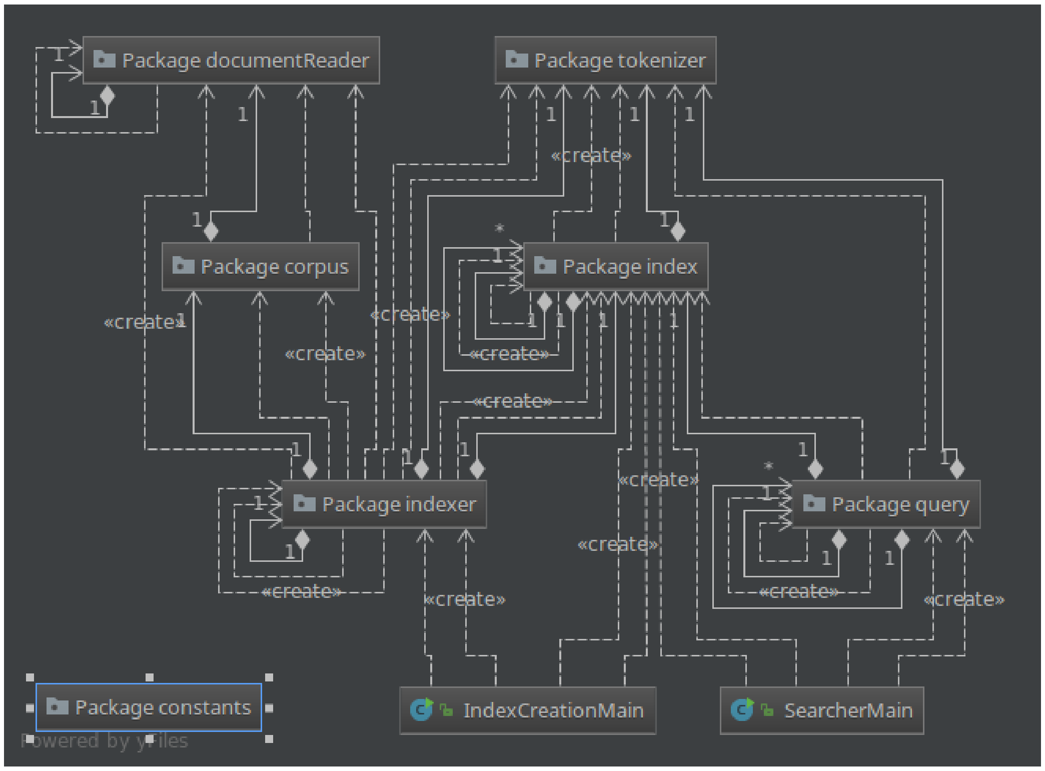


Figure 1: Application's class diagram

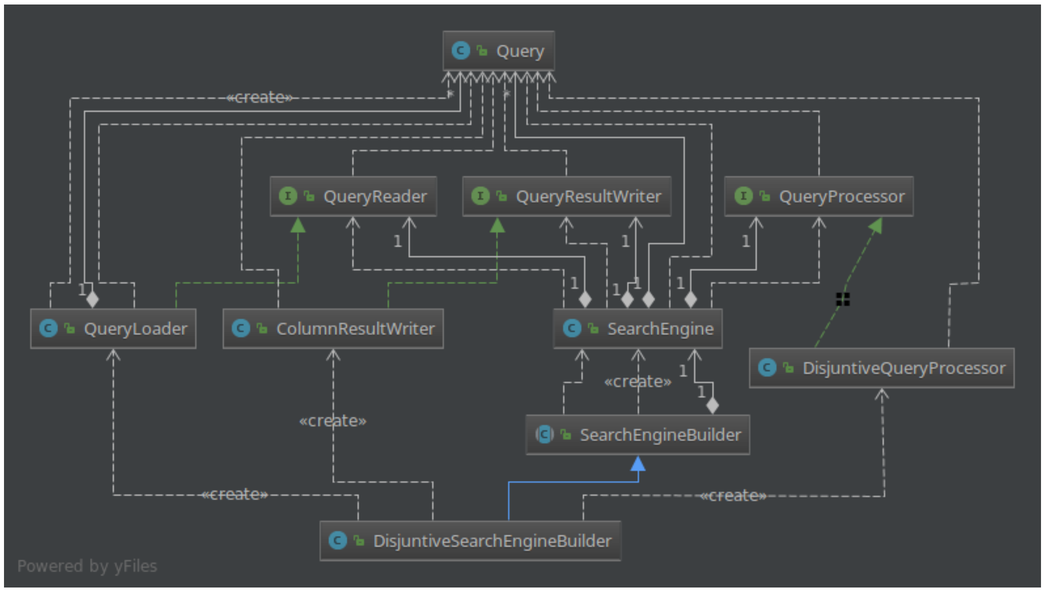


Figure 2: Package *query* class diagram

Class Constants

This class contains the unmodifiable constant variables that are used throughout the project running sequence. This facilitates any alterations of this values in development time, since they will be automatically updated in every place where they are referenced.

Interface IndexReader

This Interface defines an entity that reads a file containing the Index from disk with a certain format which depends on its implementation and the used writing methodology. Two methods are defined in this interface: “getTokenizer()” which returns the *Tokenizer* that was used during the Index creation and “parseToIndex()” which parses the file that contains the index, builds the index in memory and finally returns it.

Interface IndexWriter

This interface defines an entity that writes the Index from memory to disk with a specific format which depends on its implementation. This interface defines one method “saveIndexToFile()” which is responsible for creating a file with a given name and writing in it both the Index in memory and the *tokenizer* type that was used for the tokenization process.

Interface QueryProcessor

This class defines an interface for the processing of queries in which given an Index and a List of Query objects it uses both in order for calculating the specific document rank for each ranking type and query. This Interface defines two methods: “QueryWordsInDocument()” and “frequencyOfQueryWordsInDocument()” in which each one is used for the corresponding document ranking method.

Interface QueryReader

This class defines an interface for reading the file that contains a query list. While the file is being processed, for each line (which represents a query) is created a *Query* object which will contain both the text of the line and a queryID that is provided by the reader. This interface defines only one method “loadQueries()” which is responsible for the procedure mentioned above.

Interface QueryResultWriter

This class defines an interface for writing in disk the results of the ranking methods for each query. One method is defined “saveQueryResultsToFile()” in which given a document name and the resulting *ArrayList* of Query objects it writes them into the given file.

Class CSVIndexReader

This class implements the *IndexReader* interface and by consequence it has to implement the methods that are defined in it. This class is used when there is a need of reading an Index object from disk with a CSV format. During the reading process of the Index file, this class identifies the *Tokenizer* that was used during its construction and for each read line it populates the index in memory with successive term followed by it's respective *List* of *Postings*.

Class CSVIndexWriter

This class implements the *IndexWriter* interface and by consequence it has to implements the methods that are defined in it. This class is used when there is a need of writing an Index from memory to disk with a CSV format. As the first step of the writing process it is written on the file the *Tokenizer* type that was used for the tokenization process. After this it is successively written on the file term / *List* of *Postings* pairs one on each line.

Class ColumnResultWriter

This class implements the Interface *QueryResultWriter* and is responsible for writing the query results on disk with the format on presented on Table 1, where doc_score depends on the ranking method associated.

query_id	doc_id	doc_score
1	1	4
1	2	5
...		

Table 1: Example of the format of the result file

Class DisjunctiveQueryProcessor

This class implements the Interface *QueryProcessor* and assumes disjunctive ("OR") queries where all words in a query are combined using the "OR" operator. Two ranking methods are implemented in this class: “queryWordsInDocument()” that counts the number of terms in the query being processed that appear in each document and “frequencyOfQueryWordsInDocument()” in which the total term frequency of each query is counted for each document.

Class DisjuntiveSearchEngineBuilder

This class extends the class *SearchEngineBuilder* and is responsible for building a *SearchEngine* with a *DisjuntiveQueryprocessor*. The development of this class was necessary for supporting the implementation of the Builder programming pattern.

Class Query

This class results in an object representation of a *Query* which stores a queryID a List of its respective terms and a Map which scores the results of each query for each documentID. In the end of the ranking process the result is a list of Query objects that contain for each queryID each score for each DocumentID that will later be used for writing the results in the results file in disk.

Class QueryLoader

This class implements the interface *QueryReader* and is responsible for reading the file that contains the queries to be processed. After the reading process all the read queries are stored in a *List* of *Query* objects.

Class SearchEngine

This class is responsible for loading the queries, process them and saving the results into a file. This class aggregates many different entities such as a *QueryReader*, a *QueryProcessor*, etc. This class can have many other representations depending on the entities that are being used along with it. *SearchEngine* is as well a class that supports the implementation of the Builder programming pattern.

Class SearchEngineBuilder

This abstract class serves as a starting point for the implementation of the builder pattern with the purpose of creating an instance of a *SearchEngine*. A *SearchEngine* object needs several components to work, all of them can be changed and replaced to create a different kind of *SearchEngine*, as so, this object type can have many representations, so in order to separate its representation from its construction a builder pattern was implemented.

Class SearcherMain

This class serves as an entry point for querying an index that was read from disk and save the results of each query, according to the ranking method used, on disk. The arguments that are given to the program are the name of the index file and the name of the file to be saved with the results. This class is also responsible for giving pertinent information to the user about the program execution such as the elapsed time, the memory usage, etc.

Software Design Patterns

Just like the first assignment, in a attempt to improve readability and modularity, the same software design patterns were applied in the development of this part of the application.

Strategy

This pattern consists in creating an interface that describes a certain behavior from certain entities. This way every implementation of the interface is independent and allows interchangeability of modules [4].

Figure 2 shows the application of this pattern with the interfaces *QueryReader*, *QueryProcessor*, etc.

Builder

This pattern aims to separate the construction of a complex object from it's representation, an object that aggregates several entities can have many representations depending on the type of entities that its using. This can be solved with *Constructors* for each representation but the complexity of it's creation is still shown. The solution to this problem is to create a protocol that allows the creation of all the representations through a series of steps, define those on an abstract "recipe" in an interface. Any implementation of this interface becomes a possible representation of the complex class. All a client has to do is to ask the builder for it's respective representation of the object [3].

For this assignment, this situation can be applied to the class *SearchEn-*

gine, its attributes are a *QueryReader*, *QueryProcessor*, *QueryResultWriter*, etc. This objects can have many implementations, each one giving a different representation to the *SearchEngine*. That said, a *SearchEngineBuilder* was developed, an abstract class, along with an implementation that uses a disjunctive *QueryProcessor*, called *DisjuntiveSearchEngineBuilder*.

Efficiency Measures

In this chapter, some measures were took to analyze the performance of the application. Three measures were chosen to evaluate the system, querying time, max amount of memory used and query result space on disk. These measures were took in a machine with a Quad-core, Intel Core i7-2630QM, 2.9 GHz[2].

Querying Time

This measure was took by comparing two *TimeStamps*, one before reading the index from disk, and the other after writing the last result file to disk. Figure 3 shows the average run times for 3 consecutive runs, each column represents a different index, each produced with a different *Tokenizer*.

Maximum amount of memory

This measures were taken using a profiling tool from IDE Netbeans[1]. Figures 4, 5 and 6 show the different memory profiles for different types of indexes. It's worth noting that in all Figures the line of the graphs is not shown due to a bug in the profiling tool, even so, the amount of memory used can be seen.

Result space in disk

This measure was took by checking the space occupied by the result files for the different indexes. The occupied disk space can be observed in Table 2.

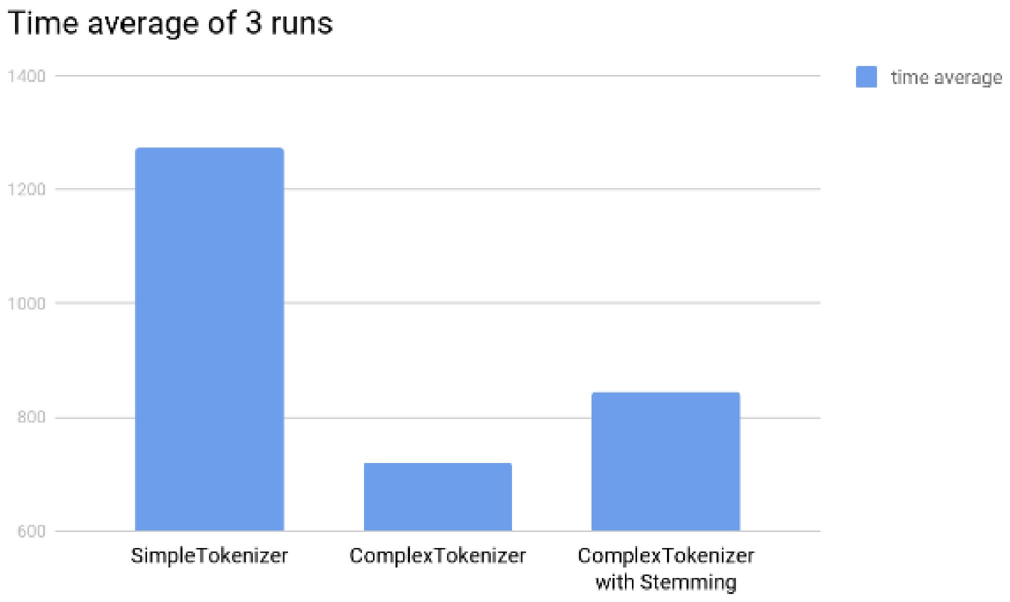
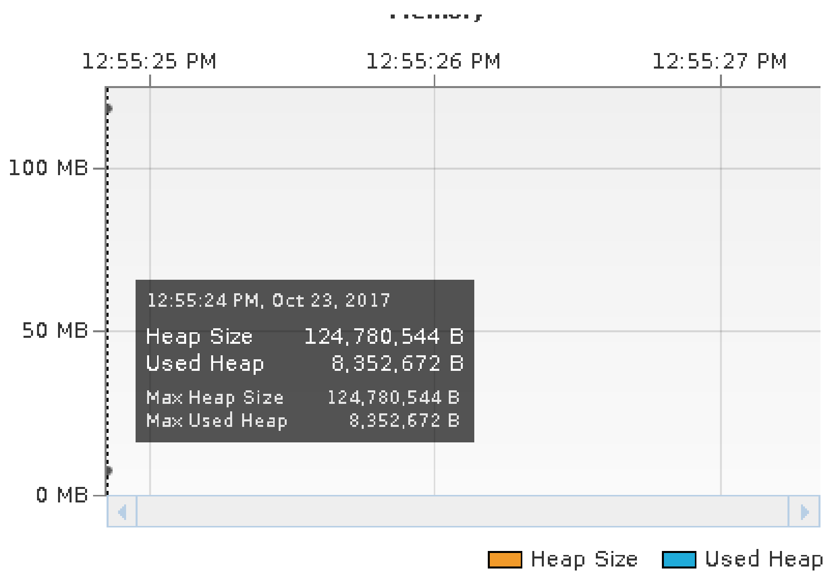


Figure 3: Time measures



```
12:55:24 PM, Oct 23, 2017
Heap Size      124,780,544 B
Used Heap        8,352,672 B
Max Heap Size  124,780,544 B
Max Used Heap    8,352,672 B
```

Figure 4: Memory profile for an index built with a *ComplexTokenizer* with stemming

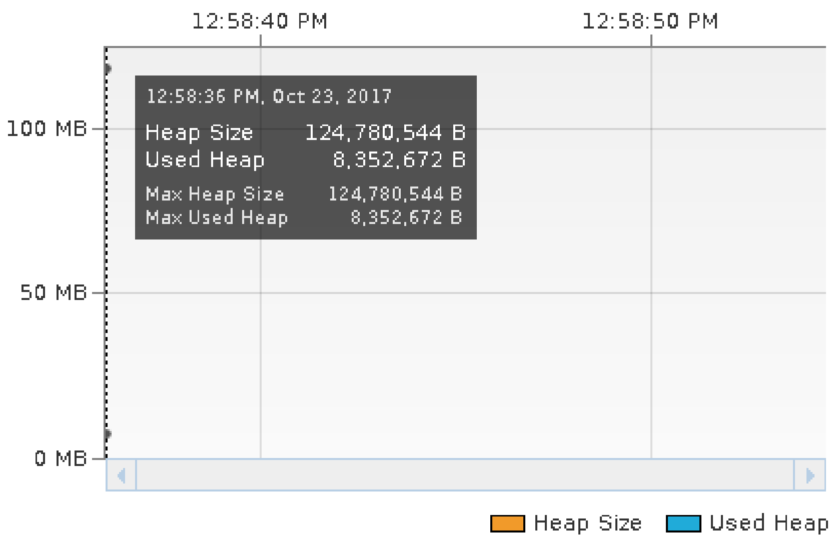


Figure 5: Memory profile for an index built with a *ComplexTokenizer* without stemming

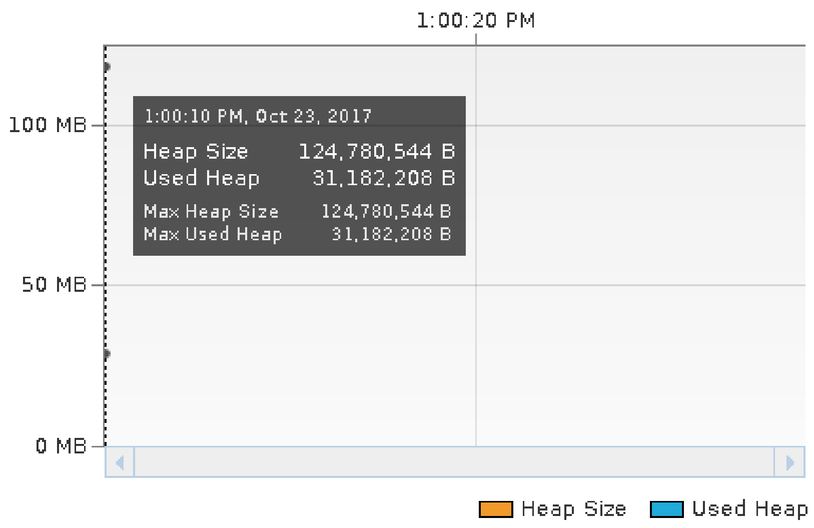


Figure 6: Memory profile for an index built with a *SimpleTokenizer*

	Query words in document	Frequency of query words
ComplexTokenizer with stemming	3,6MB	1,9MB
ComplexTokenizer without stemming	2,9MB	1,5MB
SimpleTokenizer	5,2MB	2,8MB

Table 2: Disk space occupied by the result files of different indexes

How to run

Since the application now consists of two main classes both can be executed, the application can be used by importing the project to an IDE or by following the instructions of the README file that consist on using the command line to manually compile the code and run it. For both cases the recommended approach is to run the main Classes without arguments and check the USAGE print, to see the arguments that are needed to run the programs.

Conclusion

Observing Figure 7 it's possible to see that three more main blocks were developed in comparison to the last assignment. First there was a need of implementing an *IndexReader* so that the index that was previously build and written into disk could be read again into memory. After the index construction in memory, the system proceeds to read the file where the queries to be processed are present. There is a pre-processing of that file in order to define what *Tokenizer* type was used during the index construction, on the tokenization process, since the same normalization needs to be applied to the terms present in each query. After that, the queries are read into memory and an *ArrayList* of *Query* objects is instantiated. Since both the index and the query list are now present in memory it's time for the ranker to create "DocID"-"Rank" pairs for each query to be processed. After the calculation of all the ranking results for each ranking method, query and document, those results are then stored in the previously mentioned *ArrayList* of Query objects and will then be written into the disk. With this assignment it was possible to explore the practical principles of the theory that is lectured during the classes of Information Retrieval and it was possible to develop a Ranker, even though being a simple one, that can efficiently read a set of queries and for each document associate a rank according to the query and method of ranking that was used. The system was developed to be as efficient and modular as possible and for that not only were developed programming patterns such as Builder and Strategy but Interfaces and modular code as well.

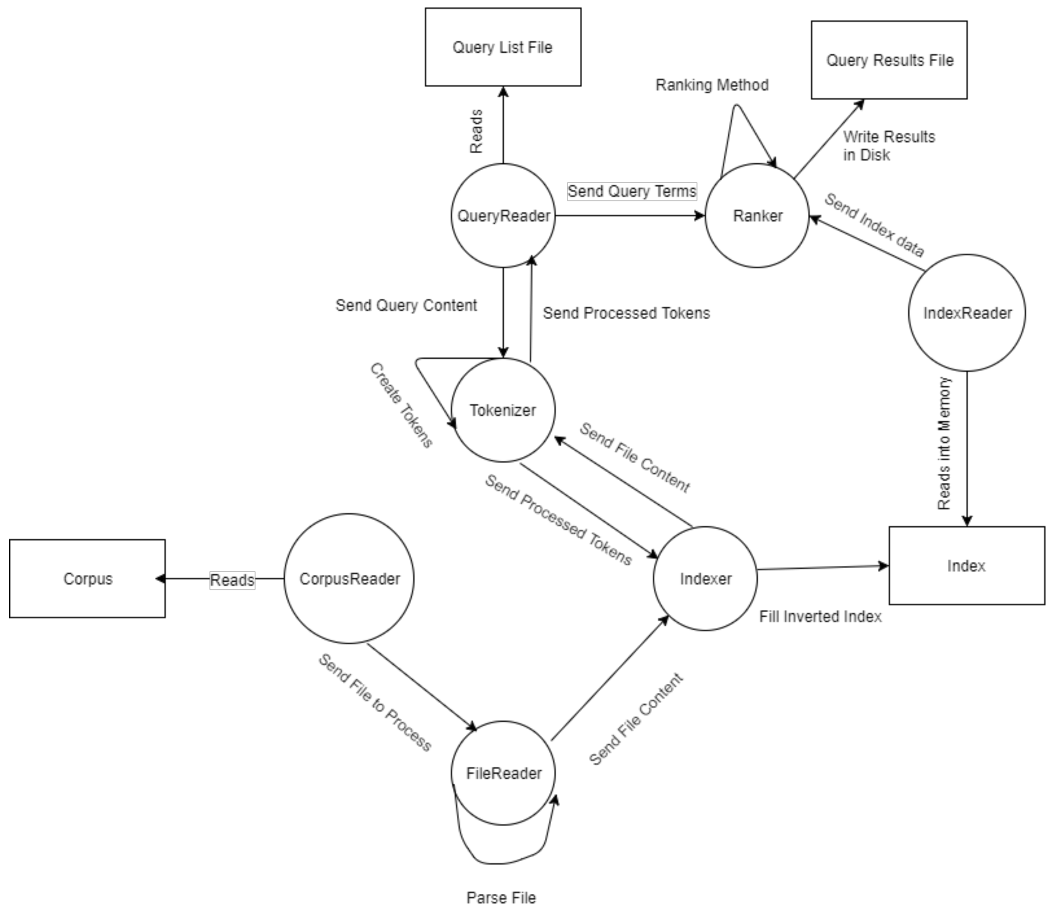


Figure 7: Dataflow diagram of the application

References

[1] Oracle Corporation and/or its affiliates. *Profiler*. 2014. URL: `https://profiler.netbeans.org/` (visited on 22/10/2017).

[2] Intel Corporation. *Intel® Core^{TM} i7-2630QM Processor*. 2014. URL: `https://ark.intel.com/products/52219/Intel-Core-i7-2630QM-Processor-6M-Cache-up-to-2_90-GHz` (visited on 22/10/2017).

[3] ToturialsPoint. *Design Patterns - Builder Pattern*. 2017. URL: `https://www.tutorialspoint.com/design_pattern/builder_pattern.htm` (visited on 22/10/2017).

[4] ToturialsPoint. *Design Patterns Strategy Pattern*. 2017. URL: `https://www.tutorialspoint.com/design_pattern/pdf/strategy_pattern.pdf` (visited on 22/10/2017).