Contents

Chapter 1

Software Design

1.1 Software Design

Coupling

When writing a program one should aim for low (or loose) coupling between classes. Low coupling means that classes are largely independent and communicate via a small well-defined interface [Barnes and Kölling(2017), p.259]. Hence a class should never depend on parts of another class that are not exposed via it's interface. The interface of a class correpponds to its public methods and fields.

Is package private fields part of the interface? I think it' is. Only if it not explicitly made private it is part of the interface.

Cohesion

A program should aim for high cohesion. High cohesion means that a single method is responsible for a single task, and that a class has well defined area of responsibility.

The class `QueryHandler` is responsible for handling queries, when it's method `getMatchingWebsites` is supplied with a query it fetches the matching websites. But should the `QueryHandler` also rank the sites it fetches via `getMatchingWebsites`?

One argument for placing the ranking inside the `QueryHandler` is, that this would allow us to get rid of the duplicate code that splits the query string into subqueries and then single words.

```
1
2 //  ... in the QueryHandler.
3
4
5 // ... for the ranking.
```

Listing 1.1: Splitting a query into single words.

But despite this we decided that the ranking task should rather be done inside the `search` method of the `SearchEngine`, to keep the task more ...

Another design related concept

something...

Figure 1.1: UML Diagram for the class Website.

Streams and Lambdas

Maybe a section about streams and lambdas? When are they useful, an when are they not?

Chapter 2

Quality assurance

In order to ensure proper quality of the individual code files and the system as a whole, we utilise different dynamic techniques.

2.1 Unit Testing and Assertions

Assertions, unit tests,

2.2 Code Review

Pair-programming, review of pull requests

2.3 Version Control and Git Work Flow

Keeping track of changes is essential. We decided upon a simplified version of Gitflow including a master and develop branch as well as individual feature branches. This allowed us to keep distinct releases separate from the latest work and ongoing experiments. For the scope of this project, we deemed release and hotfix branches unnecessary.

2.4 Profiling

Write something in this section.

title

Some subsection

2.5 Bechmarking

Another section.

Chapter 3

Mandatory tasks

3.1 Inverted Indices

Implementation

Based on the prototype for the project we implemented two inverted indexes. The InvertedIndexHashMap and InvertedIndexTreeMap are two subclasses of the superclass InvertedIndex which in turn implements the Index interface. The inverted index is a clever implementation of the SimpleIndex class trying to overcome the problems given by this naive implementation which for every query given looks through every website in the database. The purpose of an inverted index is to preemptively create an index of websites to word when the program is build, rather than performing the task at every query. In this scenario when a user makes a query the search will be performed on a Map object which maps words contained in the websites, to the website objects. In this way we increase the time needed for building the application, but we decrease the time needed for every single query. This results in an overall improvement of the performance.

The two subclasses have only one difference, they instantiate two different dynamic types of the map object. A Map is an interface which once implemented by a class will map a key to a value. The key needs to be unique but the value can be repeated. The InvertedIndexHashMap inherits all the methods and variables from the abstract superclass InvertedIndex and instantiate a HashMap. The HashMap is an implementation of the Map interface, it maps a data value (in this case a Collection<Website>) to a specific key (in this case a string which represents one word present in a specific Website). The HashMap does not follow any index and the order of the objects contained can change over time. On the other hand we have the InvertedIndexTreeMap, which instantiate a TreeMap dynamic type for the map variable and inherits all the methods and variables from the abstract superclass InvertedIndex. The TreeMap follows the same behaviour as the HashMap with only one important difference, the TreeMap has an ordered index of its mappings. The TreeMap is sorted following the natural order of the keys or according to a provided Comparator. In our case this means that the String key will be sorted alphabetically.

As said previously mentioned these two classes extend the superclass InvertedIndex which is an abstract class, this means that is not possible to in-

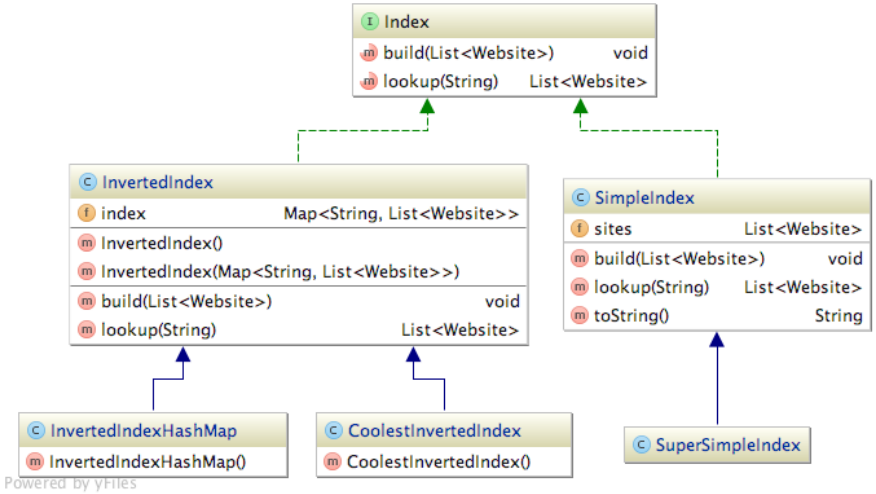

Figure 3.1: UML Diagram for the Software Architecture of Index data structures.

stantiate an InvertedIndex object directly. This structure allows us to write the code in a more structured way, avoiding duplicate code and leading to easy extensibility of the application. Hence, we implement all the common behaviour of an index into the InvertedIndex superclass and then we implement the details into the specific subclasses.

Is this relevant? When we create an object using a subclass as dynamic type, the constructor of the subclass will initialize the map object respectively as a HashMap or a TreeMap. Then when a method is called on the object, the compiler will perform a method-lookup, which means that it will look for the method in the subclass. But in this case it will find only the constructor, then it will look into the superclass that contain the build method and the lookup method. These two methods will perform their operations on the map object which was initialized into the sub-class constructor, so we use a TreeMap or a HashMap depending on the dynamic type declared at the object creation.

Benchmarking

The different implementations are to be benchmarked and compared, the expected findings were that the TreeMap would outperform the HashMap implementation due to TreeMaps logarithmic growth of time cost of the .get method based on the map size, compared to the constant growth of time cost in the HashMap. The efficiency of the two indices is then expected to intersect at some map size, with HashMap being the most efficient up until a certain map size and TreeMap being efficient of sizes bigger than that. The benchmark process was carried out multiple times in an attempt to reduce the error size of the process. This was to some extent possible although the error did not become entirely neglectable. The results of the benchmarking can be found in appendix A.

Index	File	Avg. lookup time (ms)	Error
SimpleIndex	enwiki-tiny.txt	–	–
	enwiki-small.txt	–	–
	enwiki-medium.txt	–	–
InvertedIndexTreeMap	enwiki-tiny.txt	–	–
	enwiki-small.txt	–	–
	enwiki-medium.txt	–	–
InvertedIndexHashMap	enwiki-tiny.txt	–	–
	enwiki-small.txt	–	–
	enwiki-medium.txt	–	–

Table 3.1: Running times for different index implementations.

Analysis and discussion: The results of the benchmarking did not align completely with the result that was expected, the TreeMap implementation did not manage to outperform HashMap, evidently due to the limited size of the datafiles.

3.2 Refined Queries

Basic Input Validation

Initially, we added simple input validation to [QueryHandler] to only accept queries comprised of characters in the English alphabet, digits, dashes and underscores. This was done using the following regular expression: Additionally, query strings are transformed to lowercase using [.toLowerCase()] before doing a lookup.

Several Keywords

some text

Searches With ”OR”

some more text

3.3 Ranking Algorithms

When a search is done on our search-engine, the method `getMachingQueries` in `QueryHandler` finds all the websites that matches the query. But of course some of the matching websites are more relevant than others, for the given query. To show the user the most relevant pages at the top, we need a way to rank the websites according to relevance, i.e we need a ranking algorithm. There are many different ways to rank a website, we have chosen to implement the term-frequency algorithm (TF-algorithm), and the term-frequency inverse-document-frequency (TFIDF-algorithm).

Term Frequency

Keeping in line with the notation at [wik()] the term frequency, TF, is calculated as follows.

$$TF = \frac{f_{t,d}}{\sum_{i \in d} f_{i,d}}$$

Note that we choose the term frequency that is weighted by the total number of words in the document/site. The corresponding Java implementation is shown in listing 3.1.

```
/**
* JavaDoc
*
*/
for (Website site : sites {
for (String word : site.getWords()) {
// do something
}
// do something more
}
```

Listing 3.1: Implementation of calculating TF rating.

Term Frequency - Inverse Document Frequency

$$TFIDF = TF \cdot \log\left(\frac{N}{|\{d \in D : t \in d\}|}\right)$$

```
/**
* JavaDoc
*
*/
for (Website site : sites {
for (String word : site.getWords()) {
// do something
}
// do something more
}
```

Listing 3.2: This is a code example.

Term Frequency - Inverse *Corpus* Frequency

Instead of using the inverse document frequency as above, it would also be sensible to use the *inverse term frequency* with respect to the whole database/-corpus. Lets call that algorithm for TFICF (Term frequency - Inverse Corpus Frequency). A Java implementation is shown in listing 3.3.

$$TFICF = TF \cdot \log\left(\frac{a}{b}\right)$$

Figure 3.2: UML Diagram for search-engine when using an interface Score.

Figure 3.3: UML Diagram for the search-engine when Score is a utility class.

```
/**
* JavaDoc
*
*/
for (Website site : sites {
for (String word : site.getWords()) {
// do something
}
// do something more
}
```

Listing 3.3: This is a code example.

Comparison of Algorithms

which one is best, TF, TFIDF OR TFITF?

Design considerations

In this section we will describe how we chose to embed/implement ranking algorithms in our search-engine. We actually have two different working implementations, and there was a passionate group discussion about which implementation to chose for our search-engine.

A common rule of thumb for deciding on how to distribute responsibility between classes and methods in a program, is to say that methods correspond to verbs, and classes to nouns. This verb/noun method is described in [Barnes and Kölling(2017), p.530]. But what kind of word is 'rank' or 'score', a verb or a noun? Actually it can be both, and this suggests that we can implement a ranking algorithm using different design styles.

One option is to look at score as an object. In this case it belongs to a website, in the sense that a website should have a score as a field, and the type of that field should be of type Score. Another option is to look at a score as a method/function, but then where do this method belong? One option is to make score a utility class, i.e a class consisting only of static methods.

Option 1: Score as Interface

Option 2: Score as Utility class

Chapter 4

Extensions

4.1 Improved Client GUI

there is no deeper blue ...

Chapter 5

Conclusion

Bibliography

[wik()] tf-idf. Wikipedia. URL https://en.wikipedia.org/wiki/Tf-idf.

[Barnes and Kölling(2017)] David J. Barnes and Michael Kölling. *Objects First with Java*. Pearson Education, 6th edition, 2017. ISBN 978-0-13-447736-7.