Java: MUCA - Mark Up Comment Analyzer

Search Engine for Comment to Code Coupling

SW10 Project
Group DPT108F15
Department of Computer Science
Aalborg University
June 8^{th} 2015

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Aalborg University
Selma Lagerlöfs Vej 300
Telephone: +45 9940 9940
Telefax: +45 9940 9798
http://cs.aau.dk

Title:

Java: MUCA - Mark Up Comment Analyzer;
Search Engine for Comment to Code Coupling

Theme:

Master Thesis: Spoken Programming Interfaces and Comment to Code Coupling

Project period:
P10, Spring Term 2015

Project group:
DPT108F15

Participant:
Lars Chr. Pedersen

Supervisors:
Bent Thomsen
Bo Thiesson

Circulation: 4

Page count: 78

Appendix count and type: 0

Finished on June 8^{th} 2015

Synopsis:

MUCA is a search engine that is able to take queries in the form of comments and match said query to a comment in its database, in order to return a piece of code. MUCA assists in the selection of a quality piece of code by utilizing a repository's star raiting on GitHub and the number of times a method is mentioned in its source code.

Resume

Indenfor talt programmering har der længe været et problem kendt som "Where am I?", et problem hvor programmøren mister overblikket over sin kode og glemmer hvor markøren står. I et forsøg på at komme dette problem til livs, blev vi inspireret til at bygge en søgemaskine specielt designet til at matche kommentarer samt koble kommentarer til kodeblokke.

Resulterende i en problemstilling som lyder:

"Hvordan kan man lave en søgemaskine som kan koble kommentarer til kode og rangere disse par i forhold til relevans og kvalitet?"

Denne afhandling tager udgangspunkt i designet af denne søgemaskine, som har fået navnet MUCA (Mark Up Comment Analyzer). For at designe en sådan søgemaskine er det nødvendigt først at finde en måde at koble en kommentar til en kodeblok og dernæst finde en måde til at vurdere denne kodebloks kvalitet.

Afhandlingen fortæller om generelle principper inden for søgemaskiner og kommentar analyze, opbygger et design for en søgemaskine og til slut afprøver søgemaskinens evner.

De tests som MUCA bliver udsat for bliver gennemført på en database bestående af 3,1 million dokumenter, 1.517 forskellige projekter fra GitHub og mere end 174000 forskellige termer, indsamlet fra cirka 2,8 GB af `.java` filer.

Vi finder at MUCA kan være en løsning på problemstillingen, men at MUCA stadig mangler fintuning før den er klar til hverdagsbrug.

Preface

This is a master thesis written by a student from the Department of Computer Science at Aalborg University. The thesis suggests a solution to the problem: "How does one create a search engine that can couple comment to code and rank these pairs in favor of relevance and quality?"

The student is studying Software Engineering at the 10th semester.

Included with this thesis is a CD containing all code used in creating the MUCA system, a copy of the database structure and a digital copy of the thesis.

Thanks

I would like to give my thanks to my two counselors Bent Thomsen and Bo Thiesson, without their counseling and guidance this thesis would not have been completed.

I would also like to thank my parents and my sister whom came to my rescue when my computer suddenly broke down at the end of the project.

Lars Chr. Pedersen

lcpe10@student.aau.dk

Signatures:

Lars Chr. Pedersen

Contents

Chapter 1

Intro

1.1 Motivation

This thesis was motivated by our former project and examination: Designing LARM by Mark Faldborg and Lars Chr. Pedersen [8]. The project was about spoken programming, the act of writing computer code with your voice. During the project we designed a language called LARM. One of the major problems that was found during the project is called "Where am I?", a term coined by Smith et. al [29]. The problem of "Where am I?" happens when the programmer loses track of the marker's position while entering code by voice. This problem most often occur in spoken programming when the programming interface lacks a screen for feedback.

During our examination the external examiner pointed out that LARM did not tackle this problem. After the examination we came up with several ideas on how to solve the problem. One was to create a search engine that could serve pre-written code, making for less work for the programmer and hopefully reduce the mental load of the programmer such that "Where am I?" would occur less. The reasoning being: That less mental work will leave more space to remember the marker's position and with pre-written code the act of combining code pieces will result in fewer low level programming statements.

This leads us to the inspirational problem of this thesis:

> *"How does one create a system that can take spoken comments and suggest quality programming solutions based on pre-written solutions from the internet?"*

Such a system is very complex and is created from several components. Therefore, this thesis focuses on a sub-problem, a single component, that is the search engine for the above problem. The sub-problem can be formulated as follows:

> *"How does one create a search engine that can couple comment to code and rank these pairs in favor of relevance and quality?"*

A solution to this problem may have inherent use in regular programming, since it have become more common to search Google for code solutions to ones own problem before attempting to solve it. Creating a dedicated search system for searching code might have use in regular programming to.

1.2 Top Level View and Areas of Interest

This thesis details the implementation of a search engine that couples comment to code and when queried with a comment, matches query comment to database comment in order to serve a useful piece of code. The database collected by the search engine is created from more than 1.500 Open-Source Software (OSS) projects from GitHub.

A modern search engine consists of the following parts[36]:

Crawling: Finding documents and extracting information, typically done on web-pages.

Indexing: Grouping documents in such a way that it is easier to search a large data-set during the query to document matching.

Anti-Spam: Removing duplicate documents and irrelevant data from the database. As well as keeping it out of the crawler in the first place.

Document Understanding: Understanding the contents of the item being analyzed by the search engine.

Query Understanding: Understanding what the user is asking for.

Query-Document Matching: Understanding the relation between query and stored data. A single entry in the database is refereed to as a document.

Ranking: Order results of a given query in a list based on quality and relevance.

Search Result Presentation: Presenting the results for the user, in a useful way.

Search Log Mining: Analyzing the queries performed by users to enable a more precise and quick matching process.

For the comment-to-code system that is developed in this project, the focus will be on the following parts: Crawling (see Chapter 4), Document Understanding (see Chapter 2), Query-Document Matching (see Chapter 3 and 4), Ranking (see Chapter 3 and 4), and ultimately also Search Result Presentation (see Chapter 7). In addition to these standard parts involved in the construction of a search engine, we are also interested in the area of comment analyzes and comment to code coupling.

Others have looked into matching comments and code pairs, very few have also attempted to rank said pairs, however this research is crucial for creating the comment-to-code system developed in this project, which will be the main concern of the next chapter. Luckily the second crucial ingredient for the comment-to-code system developed in this project have been standaradized, namely how to create a search engine, which will be the focus of Chapter 3.

Chapter 2

Related Works

Comments in code is an area that have been studied for some time now. The earliest paper analyzing comments, that we have been able to find, is from 1999[19].

Analyzing comments have grown in popularity since 2005 and, in our modern days, most of these projects use Information Retrieval (IR) for this purpose.[20, 21, 11, 24, 23]

This chapter will first list projects that are very similar to this one and point out where they differ. Followed by a look into commenting behavior, studies on how to couple comment to code, and lastly the application of search engine techniques on comment-code pairs.

2.1 Similar Projects

Many projects work with commenting behavior or comment analyzes, but to the best of our knowledge only two projects have created actual search engines based on comment-code pairs.

In 2012 José Luis Figueiredo de Freitas [11] wrote his master thesis on: Comment Analysis For Program Comprehension. He created a tool that is intended to help developers get an overview over a projects source code, without having to read *all* the source code. The system couples comment and code into pairs and finds the most important parts based on the problem domain that the code describes.

Where Freitas's project differs from ours, is that it only works with a single project at a time and does not do comment querying.

In 2011 Collin McMillan [24] made a tool that could search for a function, much like Google Code Search (which was closed in 2012[4]) or Koders (now known as BlackDuck or Open HUB [30]). McMillan's tool seems to only use the name of a function to index the code pieces, however it takes into account the way functions are used in relation to each other.

Where McMillan's project differ from ours, is that it does not utilize comments for document indexing and it does not match comment to code.

2.2 Commenting Behavior

According to Jef Raskin [28] the best practice for proper commenting is to not write comments that *describe* code, but instead write comments that describes the *choices* made during programming. For example, a certain algorithm which would typically be used in sorting is not used because another algorithm was found to work faster for this specific set of data.

Jef Raskin's version of proper commenting is a threat to our problem solution. If programmers are writing less comments, there will be less data to analyze, resulting in less documents in the database. There will also be less incoming queries, meaning our problem solution will not even be used.

To our luck, programmers have not adopted this proper commenting behavior. Supporting this claim; Peter Vogel wrote two articles [38][37] in 2013, describing the same problem Raskin saw in 2005. Vogel claims that programmers today comment too much, he thinks that by writing too many comments; programmers only receive a bigger task of keeping comments up to date with the code. Vogel believes that comments should only be used to describe the reasoning behind choices and for very complex methods.

Vogel's worries about out of date comments is set aside by a paper by Beat Fluri et al. [9] They did a study to see if programmers kept their comments up to date with their code. The study was conducted on three different OSSs: ArgoUML, Azureus and JDT core. The study showed that 97% of the time comments was also updated when related code was updated.

Also supporting Beat Fluri et al.'s claim, Oliver Arafat et al. wrote two papers [2, 3] were they analyzed the commenting practice in Open-Source Software. They used a database called Ohloh Inc., where they only targeted successful projects, meaning projects that still had activity and had seen activity for some time. They found that Java is the most commented language (With a comment density of 25.87%[2]). They also found that when developers changed 1 or 2 lines of source code, documentation was often updated as well, or the change was well explained in the version control system. However when a major change occurred it would be less documented. They found that OSS projects on average had a comment density of 19% and they note that this comment density seems to slightly decline as the project grows older. After 4 years the density has dropped to 18.05%. For our project that is good news, it means that there is many comments to analyze both in new and older projects.

Antoher aspect that is interesting in commenting behavior is the use of TODOs and team communication through comments. In 2005 Ying et al. [41] looked at how programmers communicate through comments in Eclipse task comments.

They found that programmers tend to leave many TODOs for code navigation and team communication, for example: "TODO an ugly hack for now -sue. Joan, please fix it".

In 2007, Storey et al. [32] created a code navigation plug-in for Eclipse based on Ying et al.'s work, called TagSEA, which according to the paper has made it easier for programmers to navigate their code.

Storey et al. continued to do research in the field of TODOs and in 2008 released another paper [33], this time on the role TODOs play in software development. They found that:

Out of date comments are likely to be the location of a bug.

Developers tend to add information to TODOs, such as: References to other classes, methods, plug-ins or modules. Their own name/initials. Bug id number. URLs and date of creation.

Some developers tend to priorities TODOs by adding LOW or HIGH (Because these keywords will then be searchable).

TODOs can be politically sensitive: One interviewee from an OSS project told that he was sometime reluctant to add TODOs to his project because then his followers might judge him on his lack of work.

TODOs that stay are typically the ones without an author, or a TODO no one really understands.

For this project, the trouble that TODOs create means that we either have to take great care when analyzing them, or perhaps we will need to ignore them all together.

2.3 Comment to Code Coupling

As explained in the introduction, we are interested in how to couple comments to code as well as how to asses the quality of this pair. In this section we look at what other projects have done.

2.3.1 Coupling Comments to Code

One of the most relevant subject for this project's search engine is how to link comments to code. At first, one might think this is very hard, but it turns out that most programmers follow an unwritten convention. Most programmers tend to write their comments just before the code it describes or on the same line [9][21]. However, Beat Fluri et al. ([9]) does state that: *"In programming languages, it is seldom straight-forward to track relations between comments and source code entities algorithmically."* [9], but they do have some suggestions on how to do so.

Beat Fluri et al. continues that single line comments is also often used as a substitute for block comments and must therefore first be subject to a pre-analysis step, to see if a single line comment is followed by another single line

comment, suggesting it is actually a block comment. Beat Flrui et al. also makes the following statements:

Comment on the same line: If a comment is on the same line as a piece of code (in-line comment), it will describe that code. [9]

Comment on an adjacent line: Comments will normally be placed in close proximity to the code it is describing. [9]

Comment describes source code: Since comments describe source code, it is most likely that they mention function or variable names found in the source code. [9]

In Beat Fluri et al.'s research they analyze the comment-blocks to both the code before and after the comment. They use a technique they call token-based string similarity to compare the strings. In other words they tokenize all the words in the comment and the code before and after, put them into *"bags (multisets) of tokens"* and then see which is the most similar to the comment. This is also known as the Vector Space Model (VSM). More details on VSM will be given in Section 3.2.

Matthew J. Howard [15] writes in his paper about "action pairs". He believes that comments often describe functions in these action pairs. That is: It is likely that a comment contains the action verb and the function is named with the command form of the same word (For example: Search-Find and Cancels-Abort). Sometimes, however, they are named in this pair form, with synonyms. He therefore implements a tool that searches for these action pairs, which utilizes a synonym dictionary. The tool also identifies and ignores non-descriptive comments. The tool is tested against humans in its ability to identify the correct action pair. And he finds that the test persons agreed 78% of the time with his automatic tool.

These papers could indicate that to match a comment to code, one could simply match it to the closest code segment (typically the one coming after the comment). But in order to get a more precise matching one could look at action pairs, or matching the mentioned variables or function names.

2.3.2 Throw away/Ignore

If the code pieces that is delivered to the user of our system is to be of value, it is important to know what should not be in the final database. This section looks at what other researchers have thrown away during analysis.

Dawn J. Lawrie et al. [20] created a Quality Assessment tool (called QALP). The tool analyzes comments and code to assess quality, but will skip any comment that is 25 words or less. These comments are skipped because they believe that these comments will look too similar to other comments of unrelated code,

if they are shorter than 25 words.

Margaret-Anne Storey et al. [33] did research on commenting behavior and found that: Out of date comments are likely to be the location of a bug. If one was able to track the creation and changes on comments and code, this could be valuable knowledge to know if a comment should be left out of analyzes, because it was out of date, minimizing the risk of giving the user of our system a piece of code that does not match the comment that was analyzed. This could be achived by looking at the history of the project from its version control system.

Harald C. Gall et al. [12] created a plugin for Eclipse that suggests bug fixes to common problems, it would analyze code repositories to see when a bug fix was introduced and then figure out what it was. When analyzing they made sure not to analyze dead code (code that have been commented out).

2.4 Ranking and Pre-ranking

When querying our database we must consider two values in order to figure out what the user actually wants: Quality and relevance. Or as it would be called in a search engine, pre-ranking and ranking. Pre-ranking is a static analyzes of the document extracted during crawling. Where ranking is a dynamic analyzes determining the relevance between documents and query. Where then the final ranking is a mixture of the pre-ranking and the ranking. Ranking will for the remainder of this thesis be refereed to as relevance-ranking as to reduce confusion.

One way to do "relevance"-ranking is by Term Frequency - Inverse Document Frequency (tf-idf) which is a method for weighting the importance of a term (word) in a given document. It takes into account the frequency of the term in a document (indicating the importance of that single term in that single document) and the inverse frequency of the term in the whole database (indicating how general that term is). The tf-idf ranking is computed dynamically for comparison with the query such that only relevant terms are considered.

This relevance-ranking method have been standardized and is commonly used when creating search engines. It is used by: [20, 11, 26]. More details on the theory behind tf-idf will be given in Section 3.2.

However how to access static quality of a comment-code pair is not an exact science. Therefore the next section looks into ideas from other papers.

2.4.1 Pre-ranking (Static Quality)

Dawn J. Lawrie et al. [20] looked at identifier names and used these to infer concept understanding. They state that comment and code is of better quality if the identifier names are present in both the comment and the code which it describes.

Collin McMillan [24] made a tool that listed functions and ranked these functions much like Google does with their pageranking. The tool looks at how many times a function is called and by that infers how important that function is to the whole program. Derived from the logic that: A function that is called often, must have received extra attention during development.

Ninus Khamis et al. [18] created a tool for automaticly assessing JavaDoc quality. This tool would first of all analyze the language used in the comment; creating different readability heuristics (FOG, FLESCH, KINCAID), it also figured out what grammatical-person the comment was written in. It would then proceed to count the number of items documented by the comment. All these heuristics would go into rating the value of a JavaDoc comment and suggest different improvements. Ninus Khamis et al. [18] also states that abbreviations make for bad quality, because these abbreviations are not always explained, making it harder for uninitiated programmers to understand the abbreviation.

These different ways of static pre-ranking, could indicate that there is more to assess the collective usefulness of the code and comment pair than simply looking at the relevance-rank based on a tf-idf comparison between comment and query. It might be beneficial for the user if we added extra measures of quality to the analysis.

Chapter 3

System Overview And Theory

This chapter first gives an overview of the system and then continues to state the reasoning behind general decisions and explain some of the general theory behind search engines.

The search engine designed in this project can be depicted as seen in Figure 3.1. The system consists of 3 parts, with the following purposes:

Crawling Generate an indexed and pre-ranked database of comment-code pairs.

Matching/Querying Match input query to comments stored in the database and rank the resulting matches.

Verifying An interface for sending queries to the database.

Each part will be explained in further detail in the following chapters. But first a short explanation of general decisions and theory is in order.

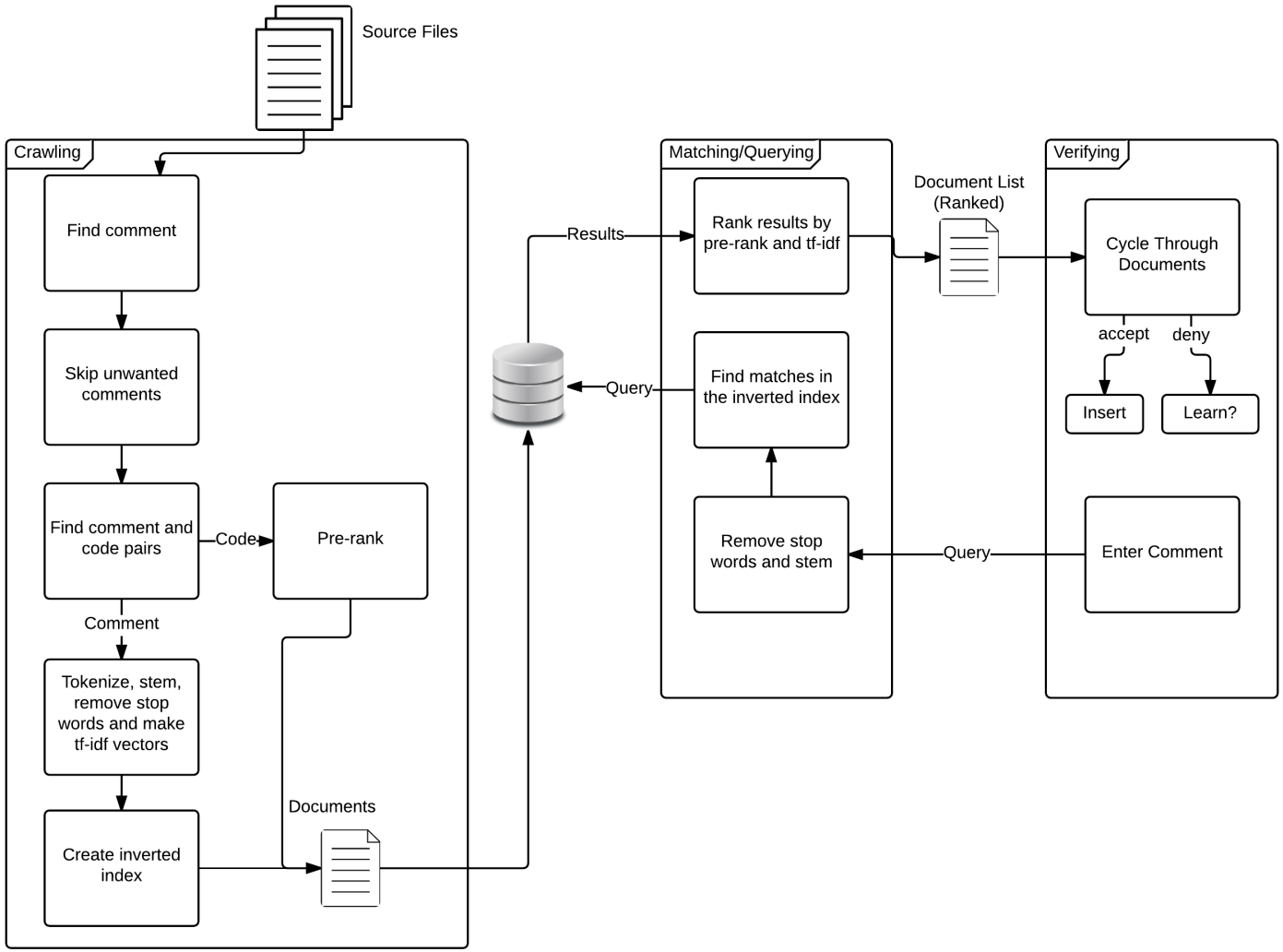

Figure 3.1: System Overview

3.1 General Decisions

In order to create a search engine we need to decide on a data-source, like Google uses most of the world-wide-web. However our search engine will be specialized for comment to code coupling, therefor an appropriate data-source must contain vast amounts of **code**.

The literature gives 4 choices: Ohloh inc. [16], SourceForge [31], GitHub [13] and FreeBSD [10]. The most accessible one of these is GitHub, it has an API for searching their vast number of different repositories. Making it possible to target a single language. It is also possible to automate the download process through scripting. For these reasons we chose to use GitHub as our data-source.

To simplify the task of creating the comment to code coupler, the search engine will only be designed to handle one language. Working with only one language makes the task easier because, we do not have to take different usage of the same symbols across different languages into account.

In the literature the most popular choice for comment analyzes is Java, it is used by: [19, 41, 9, 32, 2, 3, 6, 35, 33, 12, 18, 24, 11, 15, 34]. In addition Oliver Arafat showed, in 2009, that Java has the highest comment density, a comment density of 25.87%, that is for every 4 lines of code one line is a comment [2]. Even if this is no longer the case, Java is also the second most popular language on GitHub, only passed by JavaScript [42]. With this in mind, the language will be Java.

3.2 TF-IDF and Vector Space Models

This section goes into the theory of Term Frequency - Inverse Document Frequency (tf-idf) and Vector Space Model (VSM). This section is based on theory from the book: An Introduction to Information Retrieval [22], slides from the course Web Intelligence at Aalborg University [36], the MOLE group at DTU's website [17] and Stanfords online course on Natural Language Processing [7].

When creating a search engine, one must be able to deliver documents that are relevant to a given query. The naive way to do this would be to match the query to every single word in the document but, with millions of entries in the database the execution of such an operation would take far too long, therefore one uses a Vector Space Model.

3.2.1 Vector Space Model

The VSM consists of three parts [17]:

Document Indexing is the main reason for the efficient similarity check, between query and documents, it can be thought of as the index in the back of a book, indicating what pages a word can be found on. In the same analogy; the naive method, for query to document matching, would be to

read every word on every page to find the related pages. The document indexing will be described in Section 3.2.2.

Term Weighting is the act of assigning a value to a term in a document, which indicates the measure of relevance to that term in that document. The most common version of term weighting in search engines is called Term Frequency - Inverse Document Frequency and is described in Section 3.2.3 and 3.2.4.

Similarity Coefficients is the measure of similarity between a document and a query. The most common version of this measure is called cosine coefficient and is described in Section 3.2.5.

3.2.2 Document Indexing

Document indexing is used to optimize the speed of lookups in the database of a search engine, such that only relevant documents are considered for a given query.

The simplest version of a document index is the boolean index, an index that only indicates whether or not a term exists in a document, see Figure 3.2. The idea is that there is a row for every term in the dataset and every term have a related postings list. The postings list for the boolean index contains a reference to the document it describes and a boolean indicating whether or not the term is contained within that specific document.

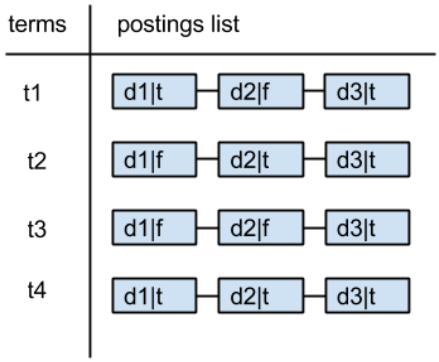

Figure 3.2: Boolean Index

A slightly more optimized version of this index, called the inverse boolean index, it takes up less space and works even faster for retrieval. The difference between the boolean- and the inverse boolean index, is that any document not containing the specific term is no longer in the postings list, see Figure 3.3.

The inverse boolean index only tells if the term exists, but it is fair to assume that: If a term occurs multiple times in a document, that document is more related to that term. [36] Therefor one could replace the boolean in the inverted index with a frequency, see figure 3.4.

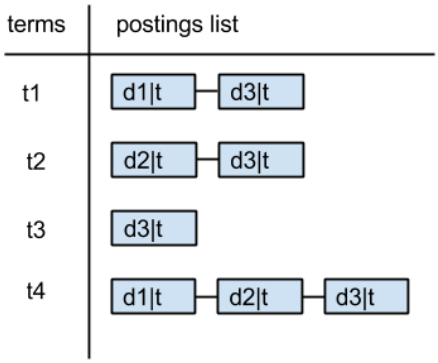

Figure 3.3: Inverse Boolean Index

The index still specifies if the the term is contained in a specific document, by the document simply being in the term's postings list, and now also gives a measure of how important that term is for a given document.

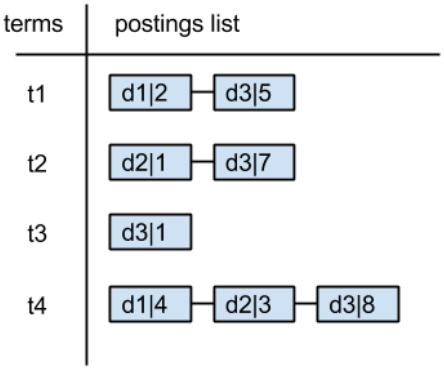

Figure 3.4: Inverted Term Frequency Index

3.2.3 TF (Term Frequency)

Tf-idf is a term weighting method and stands for: term frequency - inverse document frequency, note here that "-" is a hyphen and not a minus. This weighting method can be split into two: tf (term frequency) and idf (inverse document frequency).

Tf in tf-idf is more than just the number of times a term occurs in a given document, it is calculated as a weight, $tf^*_{t,d}$, as shown in Formula 3.1, where t is term and d is document. Making $tf_{t,d}$ the number of times a term occurs in a document.

The reasoning behind the use of this formula is that: If a term occurs once

in a document it is worth something, if it is mentioned twice even more so. But if a term occurs 100 times compared to 200 times it is not necessarily twice as important, therefore the use of log_{10} makes sense. [36]

$$tf^*_{t,d} = 1 + log_{10}(tf_{t,d}) \tag{3.1}$$

Applying the tf weight to the example in Figure 3.4 one would get the results shown in figure 3.5.

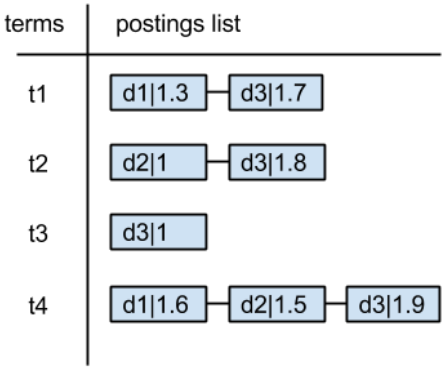

Figure 3.5: Weighted TF Index

3.2.4 IDF (Inverse Document Frequency)

The second part of tf-idf, idf, is a measure of how common a term is in the corpus (corpus is all analyzed documents). The reason for using idf is that: If a query contains two words, such as "the lexicon", one of these terms can be more important than the other. A term such as "the" will occur very often in the corpus, to the point where it makes very little difference, but a word such as "lexicon" is rare in comparison. So any document only containing "lexicon" will be more relevant than a document only containing the term "the".

Letting N be the total number of documents in the corpus, t being a term, df_t being the document frequency for t and idf^*_tbeing the inverse document frequency weight for t, the *inverse* document frequency is calculated as denoted in Formula 3.2. An example of this equation being used on a set of documents can be seen in Figure 3.6.

$$idf^*_t = log_{10}(N/df_t) \tag{3.2}$$

With the above equations in place, the tf-idf weight of a term t in a document d is then calculated as seen in Formula 3.3.

$$tf - idf_{t,d} = (1 + log_{10}(tf_{t,d})) * (log_{10}(N/df_t)) \tag{3.3}$$

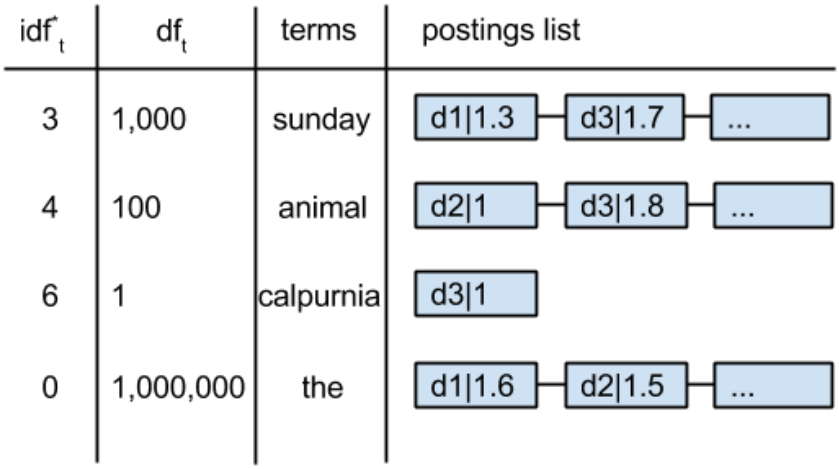

Figure 3.6: Inverse TF Index with IDF

3.2.5 Cosine Similarity and Query

The next step is to match a query to relevant documents in the database and lastly order them according to relevance based on the tf-idf weight.

For this to succeed one must have a way to compare the query to the documents in the database. In the VSM this is done by comparing vectors. The documents have vectors made out of the tf-idf scores for every term. This means the vectors exists in a M dimensional space, where M is the number of different terms in the corpus. And the query is made into a vector calculating the weight for every term in the query in the same way as for the documents, see Formula 3.1.

Now that both the documents and the query have been turned into vectors, the task of matching the query to documents can be reduced to finding the document vector that is most similar to the query vector. The naive way to do this is to use Euclidean distance, finding the document that is closest to the query, however this is solemnly the best case for comparison. If one looks at Figure 3.7.A, the black arrow represents our query vector, the red arrow represents a document vector where our query is appended to itself twice (it has the same words trice as many times) and the green and yellow arrows are slightly unrelated document vectors. One would think that the red vector would be the one most related to our black query vector, since they both have all the same terms, however with Euclidean distance the most related vector would be the yellow vector.

Therefore the vectors should be normalized as seen on Figure 3.7.B, this is done because the documents often have more terms than the query, meaning that the vector length will be very different. Imagine making a query to Google, usually a query is only a few words, but Google returns a page with several

thousand words. Therefore by reducing all vectors to length one, the worth of every word is reduced down to the same space as the query vector. - The same of course applies in the other direction since both the query and document vectors are normalized.

In the VSM, a different similarity measure is used, which, in fact, implicitly normalizes the vectors. The VSM uses cosine similarity, which translates the problem of finding similarity into; finding the angle between the query vector and the document vectors. If one looks at Figure 3.7.B, and remembers we want to find the vector that is most similar to our black query vector, it can be seen that finding the most similar document vector can be reduced to finding the document vector that have the lowest angle between itself and the query vector. This is attained by using cosine on the vectors, as can be seen from Formula 3.4, where $\hat{q}$ is the query vector and $\hat{d}$ is the document vector. But computing the cosine on high dimensional vectors is expensive in computation time, therefore the problem is further reduced as can be seen in Formula 3.5, where $|V|$ is the length of the vectors. In the implementation $|V|$ only consists of $t \in q \wedge d$ (the intersection of terms in the query and the document, making it faster and less expensive in memory)[7].

$$cos(q, d) = \frac{q \cdot d}{|q||d|} = \frac{q}{|q|} \cdot \frac{d}{|d|} = \hat{q} \cdot \hat{d} \tag{3.4}$$

$$\hat{q} \cdot \hat{d} = \sum_{i=1}^{|V|} \hat{q}_i \hat{d}_i = \frac{\sum_{i=1}^{|V|} q_i d_i}{\sqrt{\sum_{i=1}^{|V|} q_i^2} \sqrt{\sum_{i=1}^{|V|} d_i^2}} \tag{3.5}$$

By using Formula 3.4 and 3.5 one no longer has to calculate the expensive cosine, instead one can make the dot product of the query vector and document vectors. The document with the highest dot product (cosine similarity) is then the best match to the given query.

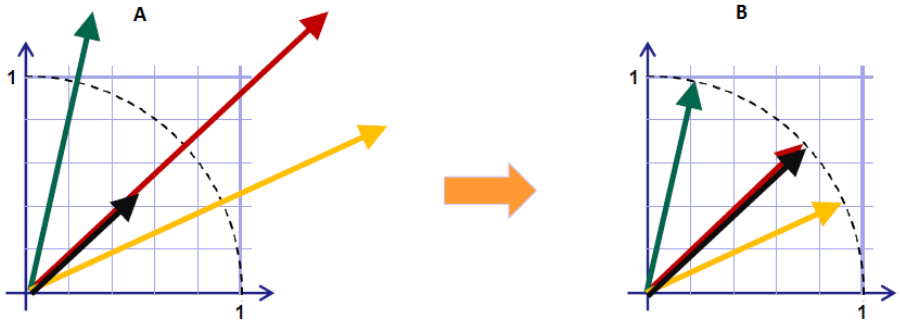

Figure 3.7: A: Euclidean B: Normalized for Cosine Similarity, the Black arrow is our query, the rest are documents in the database. [36]

3.2.6 Implementation Details and Example

Tf-idf is very popular and have many different implementations, therefore a notation for the different implementations have been created. It is called the SMART notation. It uses two sets of 3 acronyms: *ddd.qqq*, where d is for document setting and q for query setting. The acronyms should be replaced by the acronyms in Figure 3.8.

The most common implementation is lnc.ltc. Note here that the document is not treated with any ranking from the document frequency, this is added through the handling of the query. This makes it more practical to update the database since the tf score is independent on the increasing/decreasing number of documents.

Also note that the query is set to use cosine normalization, however this step makes no difference when comparing the document vectors to the query vector, since the effect will be added to every term in every document, making this factor mute.

Term frequency		Document frequency		Normalization	
n (natural)	$\mathrm{tf}_{t,d}$	n (no)	1	n (none)	1
l (logarithm)	$1 + \log(\mathrm{tf}_{t,d})$	t (idf)	$\log \frac{N}{\mathrm{df}_t}$	c (cosine)	$\frac{1}{\sqrt{w_1^2+w_2^2+\ldots+w_M^2}}$
a (augmented)	$0.5 + \frac{0.5 \times \mathrm{tf}_{t,d}}{\max_t(\mathrm{tf}_{t,d})}$	p (prob idf)	$\max\{0, \log \frac{N-\mathrm{df}_t}{\mathrm{df}_t}\}$	u (pivoted unique)	$1/u$
b (boolean)	$\begin{cases} 1 & \text{if } \mathrm{tf}_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases}$			b (byte size)	$1/CharLength^{\alpha}$, $\alpha < 1$
L (log ave)	$\frac{1+\log(\mathrm{tf}_{t,d})}{1+\log(\mathrm{ave}_{t \in d}(\mathrm{tf}_{t,d}))}$				

Figure 3.8: Tf-idf Normalization Scheme [7]

Example:
In the following example, see Table 3.1, we have a query: "best car insurance" and a document: "car insurance auto insurance". In the example we use the SMART notation: lnc.ltc.

We start by finding the tf-weight (denoted by tf*-wt) for the query (see Formula 3.1), in this example every word only occurs once, meaning we get a tf-weight of 1 on everything except for "auto" which is not present in the query. Next we calculate the inverse document frequency (idf) from the document frequency (see equation 3.2), in this example N is 1 million. Next the tf-idf weight of the query is computed (denoted here by wt), by multiplying tf*-wt with idf. And in this example we finish the queries calculations off by normalizing it.

The same is done for the document, except we skip the idf-part. Next we calculate the dot product between the query and the document, by multiplying the query's normalized tf-idf weight with the documents normalized tf-idf weight,

note here that the words “auto” and “best” have no influence on the scoring, since they are lacking in either the query or the document. As the last step the values in the Prod column is summed into the cosine similarity score between the query and the document, which for this example is $0+0+0.27+0.53=0.8$.

Term	Query						Document				Prod
	tf-raw	tf*-wt	df	idf	wt	norm wt	tf-raw	tf*-wt	wt	norm wt	
auto	0	0	5000	2.3	0	0	1	1	1	0.52	0
best	1	1	50000	1.3	1.3	0.34	0	0	0	0	0
car	1	1	10000	2.0	2.0	0.52	1	1	1	0.52	0.27
insurance	1	1	1000	3.0	3.0	0.78	2	1.3	1.3	0.68	0.53

Table 3.1: Tf-idf Query Example [7]

3.3 Stop Words, Stemming and Tokenization

Stop words, stemming and tokenization are all standardized parts of a search engine and work together to create a simpler database.

The important part about these three is that they are applied to the document as well as the query. Why will be explained in the following subsections.

3.3.1 Tokenization

Tokenization is the act of splitting a document or string into a meaningful and useful set [36]. The most typical case is splitting a string, eg. “the quick brown fox” into the single words: “the”, “quick”, “brown”, “fox”. Meaning the words can be separated by the white-space delimiter.

There is also more complicated cases where keeping two words together in one token will give more precission, eg. “Aalborg East” [36]. Whether or not that is to be one or two tokens depends on the search engine.

Other issues can be with special characters such as:

Apostrophes How is the word “book’s” supposed to be split? “book” and “s”, “book’s”, or “books”. [36]

Hyphens How is the word “mother-in-law” supposed to be split? “mother” and “in” and “law”, or “mother-in-law”, “motherinlaw”. [36]

Special Numbers Dates, time, phone numbers, eg: “12/13/1991”, “99 99 87 87”, and “12:31”. [36]

Compund Words Is the word “lifetime” supposed to be split into “life” and “time” or not split at all? [36]

How the above tokens should be split is again up to the search engine.

3.3.2 Stop Words

Stop words are words that occur in so many documents that it makes no difference whether or not they are included. Imagine having a word occur in every document; when a user searches for that particular word it will only buff every documents score and thereby become a mute increase. [36]

Examples of stop words could be: “the, a, and, to, be” [36]. The words themselves have little semantic value, however the words can be important in certain phrases such as: “King of Denmark” or “flights to London” [36].

There exists stop word lists for many languages at RANKS NL [25] which is also where we have our stopword list from.

It is important to remove stop words from both the index in the database as well as the query, since having them in either only adds unnecessary computation.

3.3.3 Stemming

Stemming is the action of reducing a word to its stem, eg. reducing “milking” to “milk”. The purpose of stemming the words in a search engine is to normalize the documents and the queries, note that this is not vector normalization, but a normalization of the terms. Imagine having a document with the string “how to create greater focusing” if a user then searched with the phrase “great focus” these two would not match, however they are similar. [7]

Normalizing in IR covers a big field, it is the act of stemming, transforming a term such as "U.S.A." into "USA" [7], or more advanced as shown by the following example [7]:

Enter: “window” - Search for: “window, windows”

Enter: “windows” - Search for: “Windows, windows, window”

Enter: “Windows” - Search for: “Windows”

The example shows that simply cutting off the ending of every word and forcing everything into lowercase can remove important information. In this example we have three different search queries, “window”, “windows” and “Windows”. The first, “window”, only relates to the windows in relation to construction work, the third, “Windows”, is most likely a search for the operating system Windows, and the second, “windows”, could be both.

Defining how to create such a sophisticated tool is an advanced task, that will not be covered in this thesis. For the purpose of this project a much simpler way of stemming is possible. The most commonly used stemmer for the English language is called the “Porter algorithm” [7]. The Porter algorithm is a series of replacement rules that is applied to the end of a word. It is a very crude but effective algorithm, an example can be seen in Figure 3.9.

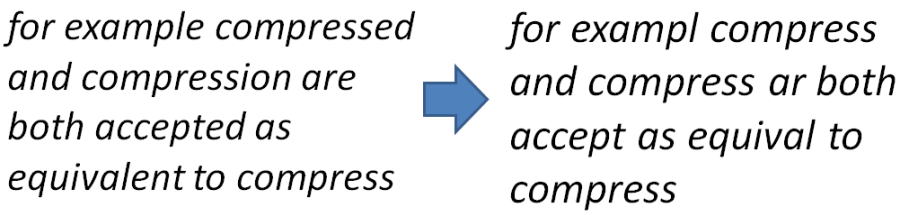

Figure 3.9: Porter Stemmer Algorithm Example [7]

The Porter algorithm is a long algorithm, the following is a part of the replacement rules, with examples (in green) [7]. For a deeper explanation of the Porter algorithm see: [27].

Step 1a

- sses → ss (caresses → caress)
- ies → i (ponies → poni)
- ss → ss (caress → caress)
- s → ø (cats → cat)

Step 1b

- (*v*)ing → ø (walking → walk) (sing → sing)
- (*v*)ed → ø (plastered → plaster)
- ...

Step 2 (for long stems)

- ational → ate (relational → relate)
- izer → ize (digitizer → digitize)
- ator → ate (operator → operate)

Step 3 (for longer stems)

- al → ø (revival → reviv)
- able → ø (adjustable → adjust)
- ate → ø (activate → activ)
- ...

In the above rules note that "(*v*)" means any verb and "ø" means nothing. The Porter algorithm have also been written for other languages. [27]

Chapter 4

Crawling

The crawler is the main part of our system and is named MUCA (Mark Up Comment Analyzer). MUCA can be thought of as: MU being a pre-analyzes step to make the analyzes easier and CA as the actual indexer and coupler, followed by a pre-ranking step.

This chapter will go into details about how MUCA works and design decisions for MUCA and the tf-idf ranking.

4.1 Mark Up (MU)

As mentioned in Chapter 3 we have decided to work with data from GitHub, specifically Java code. We first downloaded over 1.500 projects from GitHub through their APIs, then filtered out anything that was not a `.java` file. Next MU was applied to all `.java` files. MU creates markup around program elements such as `strings`, `chars` and comments in an eXtensible Markup Language (XML) style inspired by JavaML [5, 1], a markup language for Java that is no longer kept up to date.

XML is normally written in two ways:

By using a **start** and **closing** symbol, such as: "<greeting>" and "</greeting>" - the first being the start and the latter being the closing symbol [39]. Everything inside the two symbols are then considered to be part of the greeting. XML also allows for nested symbols [39].

XML can also contain **single** symbols, such as: "
" [39] - the newline in Hyper Text Markup Language (HTML) (HTML is also a markup language). These single symbols were in JavaML used to detail identifier use [5].

The purpose of MU is to make the work in the Comment Analyzer (CA) easier. We want to create easily recognizable symbols, such that the logic in CA is easier to write, therefore markup is done as specified in Table 4.1. The reasoning behind the markup is as follows:

Comments MU does markup on comments such that we can extract the comments in full in CA by simply finding the start and end marks.

Strings and Chars MU does markup on strings and chars such that we can remove these while analyzing. Strings and chars can contain characters that we utilize during analyzes, such as brackets. - Note that MU's markup is not nested for strings, chars, and comments, meaning that if one of the beginning marks for these is encountered all text within them is ignored by MU and therefore ignored by CA. This makes sure we do not markup a bracket within a string.

Brackets MU does markup on brackets because; much of the logic in CA is based on scoping, which in Java is defined by brackets.

Mark Begin	Begin	Mark End	End
<string>	“	</string>	“
<char>	‘	</char>	‘
 	\n		
<bracket>	{		
</bracket>	}		
<comment type=“javadoc”>	/**	</comment>	*/
<comment type=“multi”>	/*	</comment>	*/
<comment type=“single”>	//	</comment>	\n

Table 4.1: MU - Markup overview

4.1.1 Benefits and Drawbacks

The fact that MU creates an XML like syntax makes it possible to create optimization through different XML-tools. However this would require MU to be extended to create a full XML syntax as defined on W3C's website [39], this is left for future work.

The drawback of creating an XML like syntax is that Java uses the special chars < and > to declare general types. This is an oversight and have resulted in a few bugs, sadly it was first discovered late in the project and have therefore not been taken into account. A quick fix for this could be to replace < and > with placeholders such as “£LESS_THAN” and “£GRETER_THAN”.

An alternative to using mark up as a prestep for comment analyzes could be using a tokenizer. A tokenizer will definitely perform the prestep faster, however it is more complex to use and in this thesis we want to stay as clear and simple as possible, such that the principle behind comment-to-code coupling is clear.

4.2 Comment Analyzer (CA)

The Comment Analyzer (CA) is the main part of the crawler and is responsible for the information retrieval and analyzes. CA contains all the elements of the crawler from Figure 4.1, except for the tf-idf ranker which stands on its own, see Section 4.3.

CA consists of the following parts:

- Find and Extract Comment
- Pass Criteria
- Coupling Comments to Code
- Code Analysis
- Pre-rank

4.2.1 Find and Extract Comment

CA works on the marked up `.java` files delivered by MU, which means that for extracting comments all CA has to do is find one of the three comment starting strings as defined in Table 4.1.

In the implementation CA will however look for the greatest shared part of the three comment starting strings, "`<comment type=‘`", explained in Section 4.2.4. It then extracts the following word to see what comment type it is. This word can either be "javadoc", "multi" or "single" as indicated by Table 4.1.

The comment type is used for deciding how to couple the comment to code, see Algorithm 1, 2 and 3. It will then extract the comment itself, by starting from the comment starting string's end point and extract every word until it encounters the string "`</comment>`". "`</comment>`" is the end string for all three comment strings, as defined in Table 4.1.

When the comment type is "single", a special check is performed, as suggested by Beat Fluri et al. [9], to see if the following line is also a single-line comment. If so, the two single-line comments are concatenated and is treated as a multi-line comment.

4.2.2 Pass Criteria

In order to serve the most likely code solution to the users problem/comment/-query, CA will only consider comments that pass at least a minimum of criteria. These criteria could potentially be modified at a later point in time. The criteria currently being enforced are as follows:

At least 5 words As Dawn J. Lawrie et al. [20] removed any JavaDoc comment that was less than 25 words, so do we remove any comment that is less than 5 words. The exact number of words that is needed as a minimum for a meaningful comment should be experimented with, but

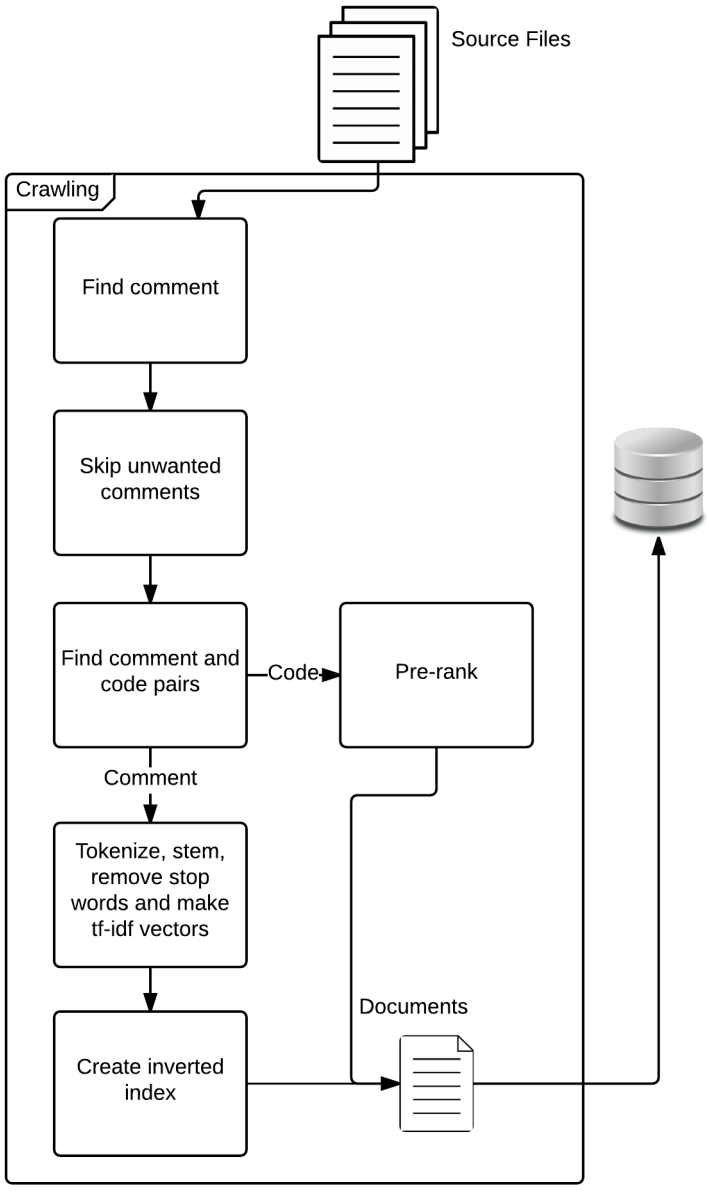

Figure 4.1: CA/Crawling overview

as a start we decided on 5 words as the minimum. This choice is based on the intuition that many Java projects on GitHub are for mobile apps, where it is sometimes needed to have files with settings or other static data, these files often have very short comments, typically 1 or 2 words, and would deliver nearly useless code for most contexts, as they mostly contain variable declarations.

Does not start with TODO or FIXME If the comment starts with TODO or FIXME it indicates that the code it relates to either is broken or buggy. As we want to serve the user a useful solution to his problem, this code should not be included in the database.

4.2.3 Throwing Away Too Much

During the design of the pass criteria it is important to investigate the trade-off between false positive and false negative as well as the associated costs.

	True	False
Positive	Good Code	Bad Code Served to User
Negative	We were right to throw it away	We were wrong to throw it away

Table 4.2: False Positives and False Negatives

When making comment-code pairs we have 4 possible scenarios, depicted in Table 4.2.

True-Positive is when we match the correct piece of code to the correct comment.

True-Negative is when the system have found an insignificant comment and was right to remove it.

False-Positive is when we mistakenly couple the wrong piece of code to a comment, when this is served to the user it will have negative value, since it wastes his time and only makes the burden of programming greater.

False-Negative is when the system mistakenly throws away a useful comment, with a piece of code that could have been used.

The tradeoff comes down to False-Positives and False-Negatives. Where False-Positive is expensive for the systems usefulness compared to the False-Negative, since not serving anything will not waste the users time and therefore the system at least does not make the burden of programming any larger. Of course there is a limit to how much can be thrown away before the database becomes so sparse that it only solves very few problems, however with more than 27.000 potential OSS Java projects on GitHub[1] it is also likely that more than

[1]This is the number of repositories we could gain access to through GitHub's API.

one project's problem domain revolves around the potential False-Negative. In other words, being overly cautious will only benefit the user, as there is a good chance a badly written comment is written in another project with a higher quality.

4.2.4 Coupling Comments to Code

Coupling the comments to the code is the most important step for MUCA as it is important to couple the comment to the right piece of code, indicated by the prevues section.

Our comment to code coupling is greatly inspired by Beat Fluri et al. [9], see Section 2.3.1, however we change their logic slightly.

The comments are coupled to code in code blocks by the Algorithms 1, 2, 3, here written in psudo-code. We define a code block in the following, denoted as "codeblock":

Simple Codeblock A piece of code that does not contain any brackets and is naturally split from other codeblocks by an empty newline. See Listing 4.1, line 7-8.

Extended Codeblock A piece of code that contains at least two brackets, always in pairs, and is split from other codeblocks by scope. An extended codeblock can contain several simple codeblocks. See Listing 4.1, line 5-12.

Open Codeblock A codeblock that have more open brackets than it has closing brackets. See Listing 4.1, line 1-12.

Listing 4.1: Codeblock Example

```
1  //Open Codeblock Start
2  public class codeBlockExamples
3  {
4    //Extended Codeblock Start
5    public static void main(String[] args) {
6      //Simple Codeblock Start
7      printOne();
8      printOne();
9
10     //Simple Codeblock End
11     System.out.println("Printed twice!");
12   }
13   //Extended Codeblock End
14   //Open Codeblock End
15   public static void printOne() {
16     System.out.println("Hello World");
17   }
18 }
```

Algorithm 1 Multi-Line

```
1: procedure CoupleMultiLine
2:     codeBlock ← getAboveCodeLine()
3:     if isOpenCodeBlock(codeBlock) then
4:         return makeExtended(codeBlock)
5:     else
6:         codeBlock ← getFollowingCodeBlock()
7:         return makeExtended(codeBlock)
```

Algorithm 2 Single-Line

```
1: procedure CoupleSingleLine
2:     nextLine ← getNextLine()
3:     comment ← getCommentOnLine()
4:     isMultiLine ← false
5: loop:
6:     if isSingleLineComment(nextLine) then
7:         isMultiLine ← true
8:         comment ← comment + nextLine
9:         nextLine ← getNextLine()
10:        goto loop
11:    if isMultiLine then return CoupleMultiLine()
12:    else
13:        code ← getCodeOnCommentLine()
14:        if isEmpty(code) then
15:            codeBlock ← getFollowingCodeBlock()
16:            return makeExtended(codeBlock)
17:        else
18:            return code
```

The reasoning behind the algorithms

Multi Line Comments are supposed to describe the code that comes directly after it. However, some programmers use it to describe the scope that the multi-line comment is in. Therefore, we first check if this comment is the first thing in a scope. If so, we match it to that whole scope. If not, it is more likely that it describes the code directly after it. See Algorithm 1.

Single Line Comments is first exposed to a check to see if it is actually a multi-line comment in disguise as this is common [9]. If it is not, we want to see if there is code on the same line as the comment, making it an in-line comment, if so the comment matches that line only. Discarding in-line comments could potentially make for a higher quality of results in general, since a single line of code rarely solves a programming problem, however we keep them for now, such that tests might be performed to investigate the benefits. See Algorithm 2.

Algorithm 3 JavaDoc

```
1: procedure CoupleJavaDoc
2:     codeBlock ← getFollowingCodeBlock()
3:     return makeExtended(codeBlock)
```

JavaDoc Comments is always matched to the codeblock after it, as that is the only way JavaDoc can add information to classes and methods. See Algorithm 3.

Problems with the above logic

Multi line comments used in the middle of a code line (ex.: int i = /* Assign */ 7;) this occurs every now and then. The above logic will couple the comment to the "7;" and what is on the next lines of code, making for a stump in the codeblock.

JavaDoc used as Multi is not a problem if the comment was supposed to describe the codeblock coming after the comment. However if the comment was supposed to describe the method or class and the comment was placed inside the scope we will make a mismatch and potentially only link it to a single declaration, however this case is extremely rare.

4.2.5 Code Analysis

Now that CA have found the code piece belonging to the comment it must analyze the code before the information can be used for pre-ranking. While doing this code analyzes CA also stores potential important information in the database such as `codePattern` and `codeType`. And if the current piece of code is identified as a method, information such as `paramaterTypes`, `parameterNames`, `returnType`, and `numberOfTimesCalled` is also stored.

The code is analyzed first by identifying its pattern. The pattern can for our experiment be identified as the nesting of class-, method-, for-, while-, do-, if-, else, try-, catch-, and lambda-statements. This information can potentially be used to search by context or if the user have an idea of the structure of his code, it can be used for limiting the search space.

Next the `codeType` is identified, this is simply done by looking at the pattern and identifying what came first. For our experiment we use the following types: `Controlstructure`, `Method`, `Class`, `Other` or `None`. `Controlstructure` is when it starts with a for-, while-, do-, if-, else-, try- or catch-statement. `Method` is when it starts with a method statement. `Class` is when it starts with a class declaration. `None` is used when the code have none of the above. Lastly `Other` is used as a failsafe for anything unexpected. - The `codeType` can be used to search by context or limit the search space for the user.

As the last step if the `codeType` is of type `Method` the method information is extracted. The information is extracted by working the method declara-

tion line. CA extracts `paramaterTypes`, `parameterNames`, `returnType`, and `functionName`. The `functionName` is then used for pre-ranking.

4.2.6 Pre-rank

In the current implementation we do two steps to do pre-ranking:

Number of times a method is mentioned in a given project. CA collects the names of methods used in a project and when the last file have been processed it counts the number of times a method is mentioned (including comments) in all the files in a project. CA looks in all files in a project since Java projects often split the files into classes, that often are interdependent.

This pre-rank measure is based on the idea that: The more a function is mentioned and used in code, the more attention it have received and therefore it will be better designed.

Star-rating from GitHub is used as a pre-rank because a project with many stars is supposedly a project with both many followers that have pointed out potential errors. A high star-rating also often indicate that the project in itself is useful. The star-rating is simply added to the database from the list of data received when the repositories was first downloaded from GitHub.

The above ranks still need to be converted into a weight that makes sense, such that it can be combined with the tf-idf ranking. The most naive way would be to add the pre-ranks to the documents and simply add the related pre-rank to the cosine similarity computed with the VSM. However more sophisticated ideas are investigated in Chapter 8.

4.3 TF-IDF Ranking

The tf-idf ranking is currently separated from MUCA and runs in its own script after MUCA have filled the database with information about the comment-code pairs. This split-up makes it easier to replace and change different parts of the ranking as one sees fit.

How tf-idf works is explained in Section 3.2 and in this section our implementation details are specified.

Recall the tf-idf normalization scheme from Figure 3.8, according to the SMART notation described there, our implementation is lnc.ltc. Meaning that all documents have their terms logarithmicly scaled and cosine normalized. The queries are also logarithmicly scaled and cosine normalized but is also applied the inverse document frequency. This choice of implementation makes for fewer computations without sacrificing precision.

When performing tf-idf ranking it is also necessary to normalize (or stem) the terms themselves, as is explained in Section 3.3. In our implementation we normalize the terms by:

First **splitting** any word that is written in camel case, such that eg. "firstNASACamelCase" is transformed into "first NASA Camel Case". - This is to separate any variable or method name potentially written in a comment into its components.

Next any **special character** such as ".", "<", "_", "\n", "\t", "?", "!", is replaced by a whitespace. The reasoning for this is that Java uses dot notation for classes and if these are mentioned in the comment we want the words to be split such that "java.class.important" becomes "java class important". Alternatively we could have allowed such notations to fold in on itself which is good for some shorthands such as "U.S.A" where it becomes "USA", however we are more focused on code notation than regular language notations.

Next everything is **turned to lowercase**, some terms might be lost in translation because of this very crude method, meaning that the search engine loses precision. However this implementation is very fast and makes certain that we do not make unjustified conclusions on how casefolding[2] infers on a comment in Java.

Next the word is **stemmed** by the Porter Algorithm [27], for more detail see Section 3.3.

Lastly before the counting and raiting of the term, we make sure to **remove any stop words**. The stop word list used is the "long stopword list" from RANKS NL [25].

When all the above steps have been taken the term is weighted by logarithm and normalized, as explained in Section 3.2. Every term is then stored into a postings-list making an inverse index, which is just an index showing in which documents a certain term can be found. Also in this index we store the normalized tf-weight now calculated. This makes for faster comparison when matching the documents to the queries.

Lastly when all documents in the corpus have been added the inverse document frequency (idf) for every term is calculated and stored in the database, for how to calculate the idf see Section 3.2 and for how this data is organized see Chapter 5.

[2] the importance and meaning of using capital letters in the middle of a word (also includes fully capitalized words)

Chapter 5

Database

The database contains all the information extracted with MUCA and the tf-idf ranking of the documents.

We decided to work with a MySQL Database Management System (DBMS), because the developer of this thesis have experience with that particular DBMS. Alternatively one could have used another DBMS, such as PostgresSQL, one could have made a special file storage system, or one could keep all information loaded in RAM. However using simple file storage is very slow for retrieval and keeping all information in RAM is very expensive when we have gigabytes of data. Therefore the most practical solution is a DBMS.
The database currently consists of 3 tables, see Figure 5.1:

Couple contains the comment-code couples. It contains information on how to retrieve the full code belonging to the comment, the full comment and information about the structure of the code.

The last 5 columns in the couple table is only filled if the code piece is identified to be a method. More on how the information is found can be read in Section 4.2.5.

Information about the package- and project name is also stored in the database, where the project name is used as a foreign key to the project table.

Project stores any information related to the project itself. Currently it only contains the name of the project and how many stars it had on GitHub; on the day it was downloaded for analysis. The project table could potentially be expanded to contain information such as copyright and problem domain, this discussion is left for future work, see Chapter 10.

Wordcount is the table containing the inverse index, see tf-idf in Section 4.3, it contains the primary key: `word` which needs to be unique as the postingslist for the term must not be split into different arrays. If the postingslist for a term was split it would result in serious performance issues, going from $O(n)$ to $O(n^2)$ where n is number of entries in the array. [36].

The `hashMapString` is the postingslist converted into string form for easier storage. The string is formatted as follows: "<id>;<tf-score>|<id>;<tf-score>|..." and converting this back into a hashMap is very fast as it only has to split on the two delimters ";" and "|".

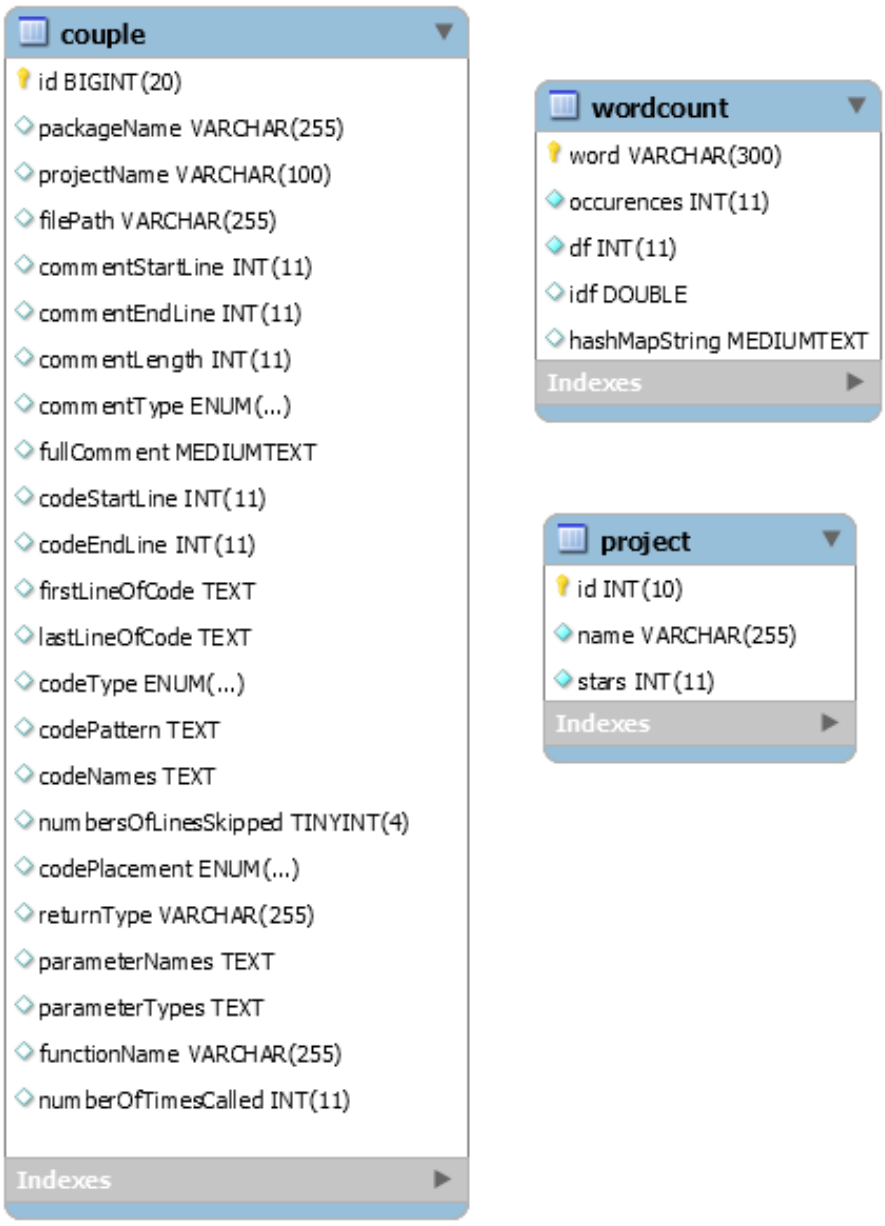

Figure 5.1: The Database Schema for MUCA

Chapter 6

Matching/Querying

If one recalls the system-overview Figure 3.1 on page 24, this chapter is about the second part of the system; matching/querying.

This chapter explains what happens when the system receives a query and what is done to find the most relevant documents in the database.

6.1 Receiving, Transforming and Matching the Query

When a query is received from the user, in our case, in the form of a comment. This comment must be transformed into a query vector, as explained in Section 3.2. However, before that can take place it must first be normalized by a stemmer and stop word remover, as explained in Section 3.3 this must include every step also applied to the documents.

To summarize, an incoming comment query must go through the following steps:

1. Split on camel case
2. Replace any special character with a whitespace
3. Turn everything to lowercase
4. Stem
5. Remove stop words
6. Calculate tf-idf weight
7. Calculate cosine similarity

When all the above steps have been completed our system applies an extra step. It adds our pre-rankings from the documents to the cosine similarity

measure in order to create a final ranking. How exactly this is done will be experimented on and explained in Chapter 8.

When the final ranking have been created, the list of matching documents is ordered and the 50 best solutions are returned to the user. The returned number of documents should be reduced in a version meant for regular use, but for testing purposes, 50 makes it easier to detect potential drawbacks of the system.

Chapter 7

Verifying

If one recalls the system-overview Figure 3.1 on page 24, this chapter is about the third and last part of the system, the interface that allows the user to verify the usefulness of a suggested piece of code.

The interface can be designed for both written and spoken programming, as the rest of the system simply needs an SQL command and a little code to combine the tf-idf scores together with the pre-ranking. The interesting part of the user interface is how to confirm that a piece of code is actually useful to the given problem. For now that task is left to the user, however the MUCA system does assist in finding the best piece of code related to the comment query, based on the pre-rank, see Chapter 8.

For the purpose of this project we decided to create an interface for written programming. Solemnly because it makes for a better testing interface, being that we have removed the factor of the spoken programming interface. However the ultimate goal for this project is to convert MUCA into a helpful tool for spoken programming, but we must recognize that testing with a spoken programming interface only makes determining the usefulness or shortcomings of a comment-to-code-coupling search-engine harder to identify.

7.1 The Potential Interface

When creating a tool that is supposed to help programmers program, an obvious thought is to improve upon the existing tool. The two top editors for Java-programming is NetBeans and Eclipse. If one makes a Google search for "How popular is 'IDE' " one will see that Eclipse have more than 39 million results and NetBeans have less than 1 million. This could indicate that Eclipse is the tool of choice when programming Java, therefore it seems obvious to create an Eclipse plugin as the interface for MUCA.

Such an interface could for example work by keeping an eye on the input stream in eclipse, and whenever a comment is entered; send a silent query to MUCA. If the query returns any results above a certain threshold the plugin

could notify the programmer that a possible solution is available, if not it should remain silent such that the programmer is not distracted needlessly. It could also be more visual as shown in Figure 7.1, where the interface is constantly shown and will depict the most relevant solution it can find in MUCA for the last active comment. By active comment we are referring to the comment that was either last focused with the marker or last written, whichever occurred last in time.

The best bonus that could be received from creating an Eclipse plugin is the direct accessibility to finding coding context. When doing search queries to Google from a mobile phone or computer, Google will use the location of the device to make a more precise search, for example if one searches for "painter" while residing in Aalborg, Google will return a map of painters in Aalborg. The same principal could be translated to comment-to-code matching. When a program is writing code, it is possible to tell, by the syntax, what is being written, for example if the word "class" occurs it is most likely a class, or if the word "private" is followed by another word and a parentheses it is most likely a method declaration.

By using the context of the comment it is possible to limit the search space even further, which makes it possible to find more precise results in the database. MUCA already extracts the code-pattern of the analyzed code, which can be used for this exact purpose.

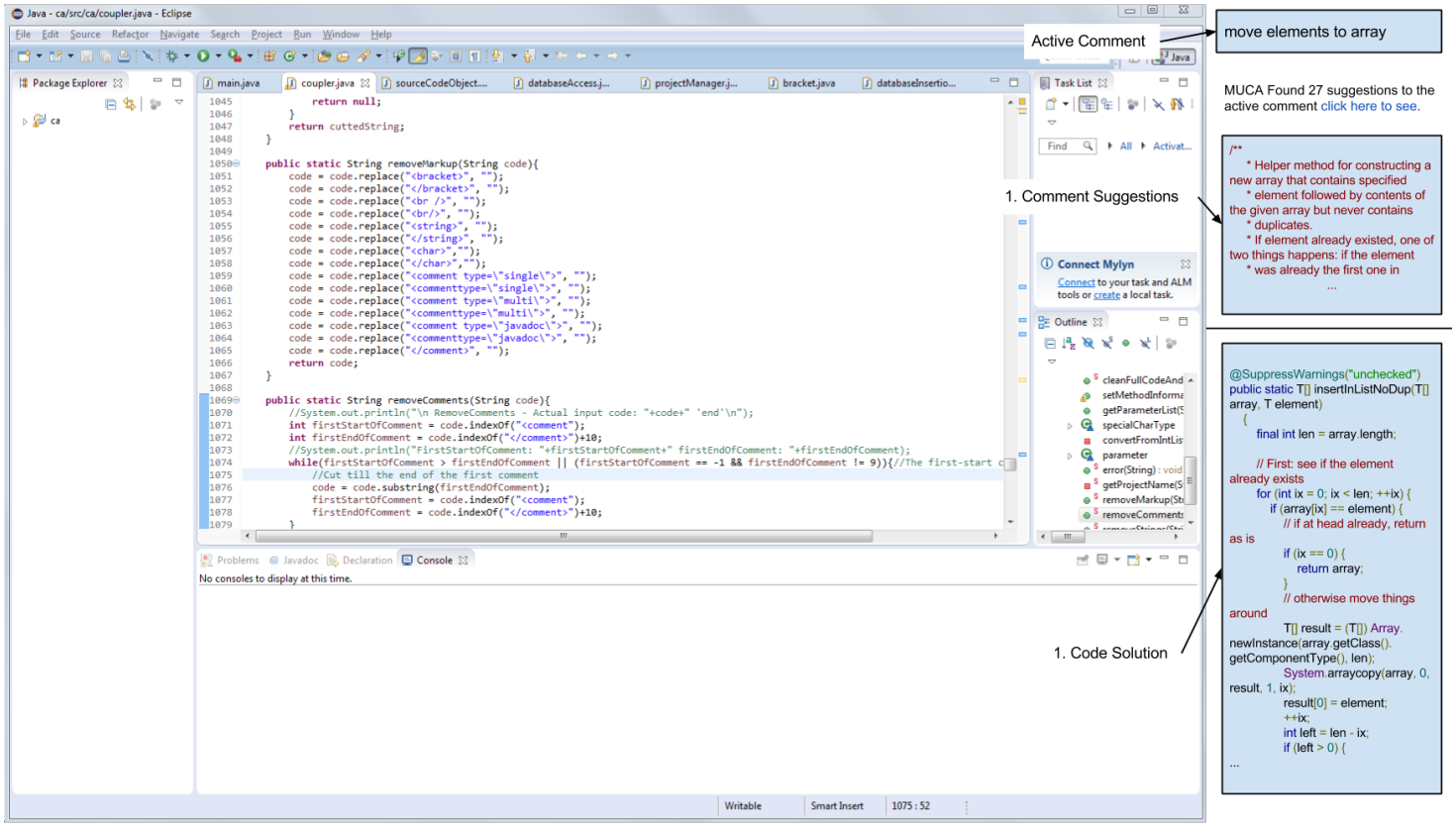

Figure 7.1: Suggested visualisation of MUCA - Eclipse plugin.

7.2 The Actual Interface

While the above interface is very good for user studies, being that it can be designed to record the use of the system and by being an Eclipse plugin it will be easily accessible for programmers, however user studies is not the focus of this thesis. Before creating a tool exactly for that purpose, it is more important to test the validity of the search engine itself, for that purpose a quick and dirty solution is just as good and will allow for more time for testing. The minimum requirement is that it can take a textual input and return a visual representation of a list of comment-code pairs.

The current implementation of the interface is made in PHP because it makes for a quick visual representation and the developer of this thesis have experience with PHP and MySQL.

A screendump from the current implementation can be seen in Figure 7.2. It displays three boxes:

Relevant Info contains any related information stored in the database that can be important to view while testing. Some of these, eg. `commentStartLine`, would be hidden in a version designed for regular use as it conveys very little information to a programmer that only needs to determine if the code can solve his current problem.

Comment contains the comment as it is stored in the database. If a set of single line comments are being used as a multiline comment, see Section 4.2.4, all the related single line comments will be shown here.

Code will display the codeblock, see Section 4.2.4, related to that specific comment.

The buttons in the top does the following:

Go! Searches the database with the above input as a query.

Previous Fetches and displays the previous comment-code pair.

Next Fetches and displays the next comment-code pair.

If there is no previous or next comment-code pair in the list it will return with nothing.

MUCA Match Interface

Direct Query Search Comment
Search Words: move elements to array
Use Pre-Rank:
Go!
Previous Next

Relevant Info:
id: 1130799
packageName: com.fasterxml.jackson.databind.util
projectName: jackson-databind
filePath: [redacted]/jackson-databind/JavaFileExtractionXml/ArrayBuilders.xml
commentStartLine: 259
commentEndLine: 266
codeStartLine: 267
codeEndLine: 299
codeType: METHOD
codePattern: method(for(if(if()if()))if())
codeNames: insertInListNoDup(for(if(if()if()))if())
codePlacement: BELOW
returnType: T[]
parameterNames: array element
parameterTypes: T[] T
functionName: insertInListNoDup
numberOfTimesCalled: 5
stars: 533

Comment:

```
/**
     * Helper method for constructing a new array that contains specified
     * element followed by contents of the given array but never contains
     * duplicates.
     * If element already existed, one of two things happens: if the element
     * was already the first one in array, array is returned as is; but
     * if not, a new copy is created in which element has moved as the head.
     */
```

Code:

```
@SuppressWarnings("unchecked")    public static  T[] insertInListNoDup(T[] array, T element)
    {
        final int len = array.length;

        // First: see if the element already exists
        for (int ix = 0; ix < len; ++ix) {
            if (array[ix] == element) {
                // if at head already, return as is
                if (ix == 0) {
                    return array;
```

Figure 7.2: MUCA Interface Screen Dump

Chapter 8

Testing

This chapter details the tests performed on the MUCA system and the results thereof. All test have been performed on a database containing 3.1 million documents (comment-code pairs), 1517 different projects and more than 174000 different terms gathered from roughly 2.8 GB of `.java` files.

8.1 Correct Comment to Code Coupling

The first test shows that the idea of coupling comments to codeblocks works.

Procedure

We will extract all comment-code pairs for 2 different projects in the MUCA database and manually see if they match up to the right piece of code by looking in the source file at the same time; in order to compare the alternatives. If a comment-code pair is coupled wrongly we will determine if the reason for this is a flaw in the logic or a bug in the MUCA implementation.

In order for a comment-code pair to be correctly matched the comment must in some way describe the code it is matched to. If a comment is matched to a too big piece of code we do not count this as a flaw, since the code the user is looking for is still correctly linked to the right comment.

Results and Conclusion

Project Name	GitHub Stars	Total Comments	Correctly Coupled	Wrong by Logic	Wrong by Bug	Describes Nothing	Dead Code	Copyright Comments
WikiSort	939	122	111	5	2	2	1	1
mixare	300	553	384	19	28	2	36	84

Table 8.1: Correct Comment to Code Coupling Results

As can be seen in Table 8.1, the logic of MUCA is not flawless. First of all MUCA is not able to detect if a comment only consists of out-commented code, that is what is represented in the column "Dead Code".

MUCA is not designed for handling comments that does not relate to a codeblock, as was found 2 examples of in WikiSort and 2 in mixare. See Listing 8.1, this comment is about performance but is placed between the opening licens-comment and import, following the import is a class declaration. The comment itself seems important, but does not describe any specific codeblock.

Listing 8.1: Comment That Describes Nothing

```
8  */
9
10 // the performance of WikiSort here seems to be completely ↩
       at the mercy of the JIT compiler
11 // sometimes it's 40% as fast, sometimes 80%, and either way↩
       it's a lot slower than the C code
12
13 import java.util.*;
14 import java.lang.*;
15 import java.io.*;
```

MUCA's logic also seem to fall short with some specific cases, where a comment describes more than it have been linked to. An example of this can be seen in Listing 8.2, where the comment have been matched to the 2 lines of code following it, however it describes the whole scope. This particular example is a bug in MUCA, either caused by a problem with the number of brackets on the line before it, or because the comment might have been treated as a single line comment. Another example is Listing 8.3, in this case a true single line comment have been used for describing the whole scope. This phenomenon was considered when designing MUCA's logic, but we decided if a single line comment was used in the start of a scope, it would be more likely that the comment was describing the variable declarations, an example of this can be seen in Listing 8.4.

Listing 8.2: Comment Describing Whole Scope 1

```
646 } else {
647         // this is where the in-place merge logic starts!
648         // 1. pull out two internal buffers each containing ↩
                A unique values
649         //     1a. adjust block_size and buffer_size if we ↩
                couldn't find enough unique values
650         // 2. loop over the A and B subarrays within this ↩
                level of the merge sort
651         //     3. break A and B into blocks of size '↩
                block_size'
652         //     4. "tag" each of the A blocks with values ↩
```

```
            from the first internal buffer
653         //     5. roll the A blocks through the B blocks and drop/rotate them where they belong
654         //     6. merge each A block with any B values that follow, using the cache or the second internal buffer
655         // 7. sort the second internal buffer if it exists
656         // 8. redistribute the two internal buffers back into the array
657
658         int block_size = (int)Math.sqrt(iterator.length());
659         int buffer_size = iterator.length()/block_size + 1;
```

Listing 8.3: Comment Describing Whole Scope 2

```
868 } else if (comp.compare(array[A.end], array[A.end - 1]) < 0){
869             // these two ranges weren't already in order, so we'll need to merge them!
870
871             // break the remainder of A into blocks. firstA is the uneven-sized first A block
872             blockA.set(A.start, A.end);
873             firstA.set(A.start, A.start + blockA.length() % block_size);
```

Listing 8.4: Antisipated Use of Single Line Comment

```
1 void regularUseOfSingleLine(int inputInteger){
2   //Assign different values to variables a and b
3   int a = 7;
4   int b = 9;
5
6 }
```

Another problem with MUCA's logic is comments that actually describes the regular code lines *before* itself. Such an example can be seen in Listing 8.5. As the logic is designed to look forward in nearly all cases, this is expected to happen. However in the two scenarios where we have observed this, the word "above" have been present in both. This discovery might have some influence on a potential extension of MUCA's logic. - This particular flaw is one of the things Beat Fluri et al. [9] have attempted to solve.

Listing 8.5: Comment Describing Code Before Itself

```
145 private static int cache_size = 512;
146   private T[] cache;
147
```

```
148   // note that you can easily modify the above to allocate a↩
          dynamically sized cache
149   // good choices for the cache size are:
150   // (size + 1)/2 − turns into a full−speed standard merge ↩
          sort since everything fits into the cache
151   // sqrt((size + 1)/2) + 1 − this will be the size of the A↩
          blocks at the largest level of merges,
152   // so a buffer of this size would allow it to skip using ↩
          internal or in−place merges for anything
153   // 512 − chosen from careful testing as a good balance ↩
          between fixed−size memory use and run time
154   // 0 − if the system simply cannot allocate any extra ↩
          memory whatsoever, no memory works just fine
155
156   public WikiSorter() {
157     @SuppressWarnings("unchecked")
158     T[] cache1 = (T[])new Object[cache_size];
159     if (cache1 == null) cache_size = 0;
160     else cache = cache1;
161   }
```

When looking through mixare's comments three bugs was found, the first bug only occurs with in-line comments that is coupled to code on the same line as itself. In this case MUCA is not setting the `code-start-line-number` which makes it impossible to extract the coupled code. However it can be seen from the `comment-start-line-number` and the `code-last-line-number` that the logic have worked. However they are still marked as bugs in the Table 8.1.

The second bug is a problem with the Windows/Unix implementation of newlines. Some repositories have been written in Windows and therefore have different newlines than Unix does. The way MU is currently implemented it does not differentiate between operating systems and expects the newlines to be formatted as the Unix implementation. The difference is that Unix uses: "\n" and Windows uses: "\r\n". MU then inserts a "
"-tag before "\n", which makes the newlines in Windows files look like: "\r
\n". This creates an issue with how MU and Java interprets newlines, Java sees both "\r" and "\n" as newlines and will therefore see "
" standing alone on a line (which is the breakpoint for a simple codeblock) even if that "
" is ending the previous code line. This have resulted in about 1/3 of the bugs for mixare. An example of this problem can be seen in Listing 8.6, where MUCA believes this comment only couples to line 1055, but it should have been related to all the lines.

Listing 8.6: Wrong Interpretation of Lineshift

```
1053      /* assign icons to the menu items */
1054      <br />
1055      item1.setIcon(drawable.icon_datasource);
1056      <br />
1057      item2.setIcon(android.R.drawable.ic_menu_view);
```

```
1058      <br />
1059      item3.setIcon(android.R.drawable.ic_menu_mapmode);
1060      <br />
1061      item4.setIcon(android.R.drawable.ic_menu_zoom);
1062      <br />
1063      item5.setIcon(android.R.drawable.ic_menu_search);
1064      <br />
1065      item6.setIcon(android.R.drawable.ic_menu_info_details);
1066      <br />
1067      item7.setIcon(android.R.drawable.ic_menu_share);
```

The third bug is an implementation oversight. In Java the `switch` statement uses "`case:`" and "`break;`" as scope delimiters. MUCA only looks for brackets to find scopes, therefore any comment that describes a case-scope have been marked as a bug since it is only linked to the first code line in that case-scope. This makes up another 1/3 of the bugs for mixare.

Nearly all the logical flaws in mixare comes from comments being used for segmentation of the code, see the example in Listing 8.7, these have been marked as a flaw because MUCA will only couple the comment with the first method it encounters, however this comment actually relates to several methods. Whether or not to implement this detail into the logic is a question for another project, there might not be any benefit from such a simple comment.

Listing 8.7: Comment for Segmentation

```
1  /* ********* Operators ********** */
2
3    public void setStartPoint() {
4    ...
5    }
6
7    public void createOverlay(){
8    ...
9    }
10
11   ...
```

8.2 Matching Query to Comment

The next step for MUCA after coupling the comment to code, is to match a given query to a related document (comment-code pair). We want to make sure this is done correctly.

Procedure

To prove the matching is correct, we will first find a comment in a code file and copy it directly into MUCA's search interface. We expect as a result to be given

that exact comment-code pair as the first item returned from the search.

Next this comment will be slightly modified and again used in MUCA's search interface. We here expect that the comment-code pair, where the comment originally came from, will be among the top 10 returned results.

As a control we will enter totally different queries that, first only contain one of the related words and secondly does not contain any of the related words.

Results and Conclusion

Note here that in this experiment the pre-ranking is turned off. This is done to show that MUCA's indexing is working correctly as well as show that the tf-idf weighting is implemented correctly and works for comment matching.

Comment Searched With	Bag Of Words	MUCA Rank Placement	Expected Placement
"you want 1000s of threads to run on the GPU all at once for speedups"	"1000 thread gpu onc speedup"	1	1
"GPU all at once for speedups"	"gpu onc speedup"	1	< 10
"threads to run on the GPU all at once for speedups"	"thread gpu onc speedup"	1	< 10
"you want 1000s of threads"	"1000 thread"	> 50	< 10
"otherwise just use normal GPU assignment"	"gpu otherwis normal assign"	> 50	> 50
"otherwise just use normal assignment"	"otherwis normal assign"	> 50	> 50
"GPU: otherwise just use normal GPU assignment for GPU"	"gpu otherwis normal assign"	28	< 50

Table 8.2: Query to Comment Matching Results

If one looks at Table 8.2, one sees the exact comment string used to search with, the resulting bag of words (the terms MUCA recognizes for querying), the resulting ranking and expected ranking. The column Expected Placement indicates in what range, in the result, we expect to find the comment: "you want 1000s of threads to run on the GPU all at once for speedups".

On the first row in Table 8.2 is the exact comment string that we know already exists in the database, and as expected it is the first result that MUCA returned.

On the second, third and fourth row in the table we change the query such that it no longer contains all search terms. What can be seen in the table is that row two and three still are returned as the first result, but the fourth row

is not within the 50 first results. This can be explained by looking at the term's individual idf weight, see Table 8.3, note here that the idf weight of "gpu" and "speedup" are very high (the range in the database goes from 0.61 to 6.50), that explains why row two and three are still high in the result. However it does not explain why the fourth row is not even in the 50 first results, since it has an idf weight of 3.23 which is still a high weight. The problem with the fourth row is located in a problem with MUCA itself. The current implementation of MUCA allows comments such as "/* 1000 * 1000 */" to be analyzed, because MUCA have detected 4 spaces (meaning it consists of at least 5 words), however when such a comment is passed on to the tf-idf ranker, that comment is normalized to contain two terms "1000" and "1000", when that comment is ranked it will be strongly matched to any query containing the term "1000", to make the problem complete this type of comment seems to be abundant in the database, making the 50 first results of querying with row four; a list of comments only containing the term "1000".

Term	1000	thread	gpu	onc	speedup	otherwis	normal	assign
Idf	3.23	1.92	4.18	2.38	4.71	1.75	2.33	2.21

Table 8.3: Idf of Relevant Terms

Moving on to row five and six in Table 8.2, we queried the database with two unrelated queries, to show that MUCA does not always return the same specific document. First we allowed the query to keep one term from the first query and next we removed all related terms, as can be seen from the table, neither of them returned the document we were trying to avoid. In an attempt to force the comment "you want 1000s of threads to run on the GPU all at once for speedups" back into the results, we appended the word "GPU" 3 times to the unrelated query, making the term "gpu" three times as important as any of the other terms, see Section 3.2.5, this resulted in the the comment reappearing in the result as the 28th result, as is shown in Table 8.2 row 7, showing that the Cosine Similarity is implemented correctly.

To summarize; the system works as intended. Given a particular query it returns with the expected result, even when the query is modified the system will return the expected result as long as the query maintains the correct keywords. Also as expected, when querying with another comment the first comment does not show up again, unless forced to by modifying the second query enough.

8.3 Final Ranking Test

As mentioned in Section 2.4 and 4.2.6, MUCA is utilizing a pre-rank. This pre-rank is a static measure of quality and combined with the dynamic tf-idf rank it will give a final ranking, which makes MUCA able to serve the user a useful and relevant piece of code based on a given query.

The final ranking should respect both relevance and quality, as serving a

document that has a very high quality but is not related; or serving a document that has a very high relevance but no quality, will both be worth less than a combination of the two.

In order to combine the two scores we must first normalize them with respect to each other. The cosine similarity score will always be between 0 and 1 [40], where the current values of `stars` and `numberOfTimesCalled` from the database, range from 294 to 13338 and 0 to 83409 respectively. We want to scale the values of `stars` and `numberOfTimesCalled` down to the same range as the cosine similarity, this will make the combination of the ranks more fair. We suggest two ways to do this

Normalized By Log is inspired by the way tf-idf is calculated and normalized by the cosine similarity. The idea is to use logarithm to lower the very large values, such that we follow the logic: "Mentioned once is worth something, mentioned twice even more so, but mentioned 200 times is not twice as important as 100". Next we normalize the output such that the values are between 0 and 1. If a is the minimum value, b the maximum value, and x the `star` or `numberOfTimesCalled` entry for a document, then its pre-rank weight is calculated as in Formula 8.1. The two specific calculations for each pre-rank weight will be explained shortly.

Exponential Function seemed like a possible alternative, because an exponential function will smooth out the range and can map any value to a 0 to 1 output. If $f(x)$ is the linear normalization described by Formula 8.2, where a is the minmum value and b the maximum value, then the exponential normalization of the `star` or `numberOfTimesCalled` is PW_d^{expo}, where x is the value from the `star` or `numberOfTimesCalled` and N a real number greater than 1.

$$PW_d^{log} = \frac{log(x-a+1)}{log(b-a+1)} \quad b \geq a, x \geq a \tag{8.1}$$

$$f(x) = \frac{x-a}{b-a} \tag{8.2}$$

$$PW_d^{expo} = N^{f(x)-1} \quad N > 1 \tag{8.3}$$

With the above logic in place, the logarithmic normalized pre-rank weight of the `star`-value is calculated as shown in Formula 8.4, where d_s is the number of stars that specific document has. The logarithmic normalized pre-rank weight of the `numberOfTimesCalled`-value is calculated as shown in Formula 8.5, where d_{tc} is the number of times that documents code have been called, as calulated by MUCA.

$$PWS_d^{log} = \begin{cases} \frac{log(d_s-294+1)}{log(13338-294+1)} & \text{if } d_s \geq 294 \\ 0 & \text{if } d_s < 294 \end{cases} \tag{8.4}$$

$$PWT_d^{log} = \frac{log(d_{tc} - 0 + 1)}{log(83409 - 0 + 1)} \tag{8.5}$$

With the above logic in place, the exponential normalized pre-rank weight of the `star`-value is calculated as shown in Formula 8.7, and the exponential normalized pre-rank weight of the `numberOfTimesCalled`-value is calculated as shown in Formula 8.9.

$$fS(x) = \frac{x - 294}{13338 - 294} \tag{8.6}$$

$$PWS_d^{expo} = \begin{cases} 10^{fS(d_s)-1} & \text{if } d_s \geq 294 \\ 0 & \text{if } d_s < 294 \end{cases} \tag{8.7}$$

$$fT(x) = \frac{x - 0}{83409 - 0} \tag{8.8}$$

$$PWT_d^{expo} = \begin{cases} 10^{fT(d_{tc})-1} & \text{if } d_{tc} \geq 1 \\ 0 & \text{if } d_{tc} < 1 \end{cases} \tag{8.9}$$

Combination

After the weights have been calculated we want to combine them, the easiest way to do so is by adding them together since their values have been normalized. However we also want to see if changing the influence of each rank have any effect on the final ranking, we therefore introduce three weights, w_{tfidf}, w_{ps}, w_{pt}, that will be used to combine the three ranks into one. If w_{tfidf}, w_{ps}, w_{pt} are numbers such that $w_{tfidf} + w_{ps} + w_{pt} = 1$ then the final ranking with the exponential calculations is calculated as seen in Formula 8.10 and the final ranking with the logarithmic calculations is calculated as seen in Formula 8.11.

$$rank_d^{totalExpo} = w_{tfidf} * rank_d^{tfidf} + w_{ps} * PWS_d^{expo} + w_{pt} * PWT_d^{expo} \tag{8.10}$$

$$rank_d^{totalLog} = w_{tfidf} * rank_d^{tfidf} + w_{ps} * PWS_d^{log} + w_{pt} * PWT_d^{log} \tag{8.11}$$

Procedure

We first define a comment query that will relate to several different projects in the database. For our experiment that query will be: “This method draws the circle”. This query was chosen because we know there is several different mobile application repositories in the database and it is a common act to draw circles in such applications. Next this query is given to the MUCA search interface, only using the tf-idf ranking. The result will be the basis for our test.

Then we run the same query through the MUCA search interface using the final ranking measures, shown in Formula 8.10 and 8.11, with 4 different set of weights, meaning we will collect 9 different result sets. The different weighting schemes will be (w_{tfidf}, w_{ps}, w_{pt}): 0.33, 0.33, 0.33 and 0.5, 0.25 , 0.25, and 0.75 , 0.125 , 0.125, and 0.25 , 0.375, 0.375. These weighting schemes have been chosen because we want to see what happens with; a fully fair distribution, a fair distribution between tf-idf and pre-ranks as a whole, a distribution that favors the tf-idf ranking, and a ranking that favors the pre-ranking.

Results and Conclusion

The results can be seen in Table 8.5 and information about the documents themselves can be seen in Tabel 8.4. Note that in Table 8.4, we have replaced the actual document id with another more readable one, it is not the 50 first documents in the database, but the 50 first results found by only using cosine similarity.

If one looks at the listings in Table 8.5 one can see that only 2 listings are sorted in the same way, the two listings calculated with the weighting: 0.75, 0.125, 0.125. The fact that every other listing is different from each other indicates that both the weighting and the chosen formula makes a difference.

Concluding which formula and weighting is best is not easy. When looking at Table 8.4 we can see that there is only one document on the list that is actually a method (No. 37), indicated by every other row being set to -1, which is the default value in the database for unset. No. 37 also have more than 5.000 stars and by that reason should make this document the best fit, however looking at the comment itself it is: “Draws a circle with an axis”, making it less related to our search query because of the term “axis” therefore it is acceptable if it is not the first document presented.

Looking at Table 8.4, and having seen that the content of the related comment’s codeblocks are near identical (save for tabulation and the use of other variable names), with exception to No. 37. It stands to reason that the best top 10 listing should contain every document that has a high amount of stars. From the Table 8.4 every document with more than 2.000 stars is: 2, 12, 18, 22, 36, 37, 44 and 50. The last coulmn, Valid Count (VC), shows how many of the documents with more than 2.000 stars are within the resulting listing. If the number of high star documents is our measure of the best listing, the last logarithmic formula is the best final ranking method.

Document id	Stars	# of times mentioned	tf-idf weight
1	1215	-1	0.9425
2	2261	-1	0.9425
3	386	-1	0.9425
4	517	-1	0.9425
5	1392	-1	0.9425
6	915	-1	0.9425
7	369	-1	0.7695
8	310	-1	0.7695
9	578	-1	0.7695
10	517	-1	0.7695
11	578	-1	0.7695
12	2300	-1	0.7695
13	1392	-1	0.7695
14	648	-1	0.7695
15	448	-1	0.7695
16	503	-1	0.7695
17	760	-1	0.7695
18	5054	-1	0.7695
19	992	-1	0.7695
20	1392	-1	0.7695
21	1392	-1	0.7695
22	5054	-1	0.7695
23	943	-1	0.7695
24	760	-1	0.7695
25	760	-1	0.7695
26	1891	-1	0.7695
27	1215	-1	0.7695
28	578	-1	0.7695
29	448	-1	0.7695
30	709	-1	0.7695
31	922	-1	0.7695
32	386	-1	0.7695
33	942	-1	0.7695
34	446	-1	0.7695
35	959	-1	0.7695
36	2261	-1	0.7695
37	5762	3	0.7695
38	1555	-1	0.7695
39	533	-1	0.7695
40	503	-1	0.7695
41	782	-1	0.7695
42	915	-1	0.7695
43	915	-1	0.7695
44	2261	-1	0.7695
45	478	-1	0.7695
46	915	-1	0.7695
47	533	-1	0.7695
48	1891	-1	0.7695
49	1248	-1	0.7695
50	2300	-1	0.7695

Table 8.4: Document Information

Method	Weights	Top 10	VC
Tfidf	1,0,0	[1] => 0.9425 [2] => 0.9425 [3] => 0.9425 [4] => 0.9425 [5] => 0.9425 [6] => 0.9425 [7] => 0.7695 [8] => 0.7695 [9] => 0.7695 [10] => 0.7695	1
Log	0.33,0.33,0.33	[37] => 0.5940 [2] => 0.5751 [5] => 0.5548 [22] => 0.5488 [18] => 0.5488 [1] => 0.5487 [6] => 0.5350 [12] => 0.5187 [50] => 0.5187 [44] => 0.5180	6
Exp	0.33,0.33,0.33	[37] => 0.3736 [2] => 0.3577 [5] => 0.3511 [1] => 0.3498 [6] => 0.3478 [4] => 0.3453 [3] => 0.3445 [22] => 0.3304 [18] => 0.3304 [12] => 0.3009	4
Log	0.50,0.25,0.25	[2] => 0.6713 [5] => 0.6560 [1] => 0.6513 [37] => 0.6424 [6] => 0.6409 [4] => 0.6140 [22] => 0.6082 [18] => 0.6082 [3] => 0.5908 [50] => 0.5854	5
Exp	0.50,0.25,0.25	[2] => 0.5066 [5] => 0.5016 [1] => 0.5006 [6] => 0.4991 [4] => 0.4972 [3] => 0.4966 [37] => 0.4754 [22] => 0.4427 [18] => 0.4427 [12] => 0.4204	5
Log	0.75,0.125,0.125	[2] => 0.8069 [5] => 0.7992 [1] => 0.7969 [6] => 0.7917 [4] => 0.7783 [3] => 0.7667 [37] => 0.7060 [22] => 0.6888 [18] => 0.6888 [12] => 0.6775	5
Exp	0.75,0.125,0.125	[2] => 0.7246 [5] => 0.7220 [1] => 0.7216 [6] => 0.7208 [4] => 0.7199 [3] => 0.7196 [37] => 0.6225 [22] => 0.6061 [18] => 0.6061 [12] => 0.5950	5
Log	0.25,0.375,0.375	[37] => 0.5788 [2] => 0.5357 [22] => 0.5275 [18] => 0.5275 [5] => 0.5127 [1] => 0.5057 [50] => 0.4933 [12] => 0.4933 [36] => 0.49254 [44] => 0.4925	8
Exp	0.25,0.375,0.375	[37] => 0.3283 [2] => 0.2887 [5] => 0.2811 [1] => 0.2797 [22] => 0.2792 [18] => 0.2792 [6] => 0.2774 [4] => 0.2746 [3] => 0.2737 [50] => 0.2458	5

Table 8.5: Listings Comparison, VC stands for Valid Count

Chapter 9

Conclusion

In order to conclude on the Problem Statement:

> *"How does one create a search engine that can couple comment to code and rank these pairs in favor of relevance and quality?"*

We must first summarize the results of the tests.

Can MUCA correctly couple code and comment: Yes, for the most part. MUCA still have some bugs and a few flaws in its logic was discovered, but whether these flaws are caused because of irregular use of comments or because MUCA's logic needs to be more sophisticated is still up for debate.

Can MUCA match a query to a comment: Yes, we saw in Section 8.2 that MUCA can find an exact comment, it can also find the same comment even when it is slightly modified. But when using an entirely different query it will not include the first result.

Does the ranking in MUCA make a difference: Yes, MUCA have been specifically designed to assist programmers in finding useful code, it does so by finding relevant matches in the corpus and orders them in favor of quality and relevance. However MUCA will still need some fine tuning before it is ready for regular use.

Returning to the Problem Statement, can MUCA be the answer to the question? Yes, it can. MUCA is in nearly all sense a search engine, the only thing it is lacking is an *automated* crawler. MUCA's logic is also at an acceptable level, but extending the logic even further should be a concern for future work. But most importantly, MUCA is able to identify some measure of quality in the code. Already now with only 1.500 repositories in its corpus, we are seeing many duplicates, therefore it is even more important to rank the documents by quality and try to remove duplicates.

Lastly we return to the inspirational problem of this thesis:

> *"How does one create a system that can take spoken comments and suggest quality programming solutions based on pre-written solutions from the internet?"*

Is MUCA the first step to realizing a solution to that problem? Perhaps. Currently MUCA works at an acceptable level, but it needs close to flawlessly before it is ready for use in the Spoken Programming context, such that we do not unnecessarily burden the programmer. The next and final chapter will go into details about different aspects of MUCA that can be improved.

Chapter 10

Future Work

In this final chapter we will shortly touch upon ideas for improvements of the MUCA system and potential research topics opened by the creation of MUCA.

10.1 Search Engine Improvements

As MUCA is a search engine there is a list of improvements that will make MUCA both faster and more precise.

Duplicate Detection: In search engines that crawl the internet, it is common to encounter a page with the exact same content, as people take backups, copy a piece of text on to their blog or in any other way copy a web-page's content. If duplicate entries in the database is not detected it can clutter up the database. On the other side it could also be interesting, for a tool such as MUCA, to also analyze the reuse of code parts in other programs, which could be combined with duplicate detection.

Dead Code Detection: As mentioned in Chapter 2, analyzing dead code gives us no benefit, it might in the end give bad matches to the users of MUCA. Therefore adding functionality that makes it possible for MUCA to detect dead code that resides within comments will be a benefit.

Detect Copyrights: When copying code from OSS projects, it is important to make a note of what copyright this particular project is under. If MUCA is to be used in real-life programming, it will be important to detect the copyright of any given project and or file in the corpus. Currently the copyright settings on GitHub is specified as: If there is no license on the GitHub page or in the file itself it will default to: "*... This means that you retain all rights to your source code and that nobody else may reproduce, distribute, or create derivative works from your work. ...*" [14]. With copyright laws in the picture, MUCA can have trouble being useful if the user is not informed correctly.

Detect and Search by Context: As was mentioned in Chapter 7, it could be beneficial to detect and search by the use of code context, such that if a comment is written before a class declaration, MUCA might only search for classes. Such a feature could improve precision of the matching.

Detect Problem Domain: Freitas et al. [11] was focused on making program comprehension from, among other information, the problem domain of a project. Detecting the problem domain in a project could potentially create another measure for precision in the matching between query and documents.

Make MUCA Web Based: Currently MUCA have no way of crawling. MUCA is currently delivered a set of data files and analyzes them into a database. In order to make MUCA a fully fledged search engine it will need the ability to crawl for new `.java` files on its own. This could potentially be achieved by making a deal with GitHub.

10.1.1 Query Improvements

The following improvements have standard implementations and could improve precision of the matching process.

Implement Phrase and Relational Queries: This will allow for more precise query input, as words such as "the" can gain meaning in a phrase such as "The Queen of England".

Implement Synonyms: By using synonyms in the search query it will be possible to match more documents to the query and in most cases provide a better result. The use of synonyms can also open up for using parts of the code, such as method names, for indexing the document, as suggested by Matthew J. Howard [15].

Implement Phonetic Search: Will potentially increase the number of matches to a query. How this feature of a search engine works within the area of comments could be interesting to research. Even more so for an implementation of MUCA with a spoken interface.

10.2 Potential Research

Create Spoken Programming Interface: This first version of MUCA have made it possible to search for code solutions by comments. In order to get back to our motivational problem statement, one must design a new interface, specifically for spoken programming. Such a tool could be combined with a spoken programming interface and language, such as the one designed in the LARM report [8].

The Use and Re-use of Code in OSS Java projects: As MUCA now have been developed, a potential research project could be to investigate how often programmers of different capability and position writes or re-uses code. One could potentially design and implement the Eclipse plugin for interfacing with MUCA, as suggested in Chapter 7. This plugin could then implement a data logger, retrieving information about the use of MUCA.

10.3 Other

Full XML As mentioned in Chapter 4, adding a full XML scheme to MU could make it useful in more ways and open up for the use of regular XML tools. This could potentially give a performance boost to MUCA and make MU useful outside of this project context.

Bibliography

[1] Ademar Aguiar, Gabriel David, and Greg J Badros. Javaml 2.0: Enriching the markup language for java source code. *XML: Aplicaçoes e Tecnologias Associadas*, 2004:1–12, 2004.

[2] O. Arafat and D. Riehle. The comment density of open source software code. In *Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*, pages 195–198, May 2009.

[3] Oliver Arafat and Dirk Riehle. The commenting practice of open source. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 857–864, New York, NY, USA, 2009. ACM.

[4] Nicholas Armstrong. *Replacement for Google Code Search?* URL: `http://stackoverflow.com/questions/7778034/replacement-for-google-code-search`, 2011. Looked up: 29-03-2015.

[5] Greg J. Badros. Javaml: A markup language for java source code. *Comput. Netw.*, 33(1-6):159–177, June 2000.

[6] Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr. Code bubbles: A working set-based interface for code understanding and maintenance. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 2503–2512, New York, NY, USA, 2010. ACM.

[7] Christopher Manning Dan Jurafsky. Stanford natural language processing web course. Website, June 2015. `https://class.coursera.org/nlp/lecture`.

[8] Mark Faldborg and Lars Chr. Pedersen. Designing larm - programming with nothing but your voice. Software P9 Student Report at AAU, January 2015.

[9] B. Fluri, M. Wursch, and H.C. Gall. Do code and comments co-evolve? on the relation between source code and comment changes. In *Reverse*

Engineering, 2007. WCRE 2007. 14th Working Conference on, pages 70–79, Oct 2007.

[10] FreeBSD. Freebsd - home page. Website, 2015. `https://www.freebsd.org/`.

[11] Jose Luis Freitas, Daniela da Cruz, and Pedro Rangel Henriques. A comment analysis approach for program comprehension. In *Software Engineering Workshop (SEW), 2012 35th Annual IEEE*, pages 11–20, Oct 2012.

[12] Harald C. Gall, Beat Fluri, and Martin Pinzger. Change analysis with evolizer and changedistiller. *IEEE Softw.*, 26(1):26–33, January 2009.

[13] GitHub. Github - home page. Website, 2015. `https://github.com/`.

[14] GitHub. Github - open source licensing. Website, 2015. `https://help.github.com/articles/open-source-licensing/`.

[15] Matthew J. Howard, Samir Gupta, Lori Pollock, and K. Vijay-Shanker. Automatically mining software-based, semantically-similar words from comment-code mappings. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 377–386, Piscataway, NJ, USA, 2013. IEEE Press.

[16] Open Hub. Open hub - home page. Website, 2015. `https://www.openhub.net/`.

[17] IMM and DTU. Mole - text analysis group. Website, 1999. `http://cogsys.imm.dtu.dk/thor/projects/multimedia/textmining/index.html`.

[18] Ninus Khamis, Juergen Rilling, and René Witte. Assessing the quality factors found in in-line documentation written in natural language: The javadocminer. *Data Knowl. Eng.*, 87:19–40, September 2013.

[19] Douglas Kramer. Api documentation from source code comments: A case study of javadoc. In *Proceedings of the 17th Annual International Conference on Computer Documentation*, SIGDOC '99, pages 147–153, New York, NY, USA, 1999. ACM.

[20] D.J. Lawrie, H. Feild, and D. Binkley. Leveraged quality assessment using information retrieval techniques. In *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, pages 149–158, 2006.

[21] H. Malik, I. Chowdhury, Hsiao-Ming Tsou, Zhen Ming Jiang, and A.E. Hassan. Understanding the rationale for updating a function's comment. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 167–176, Sept 2008.

[22] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *An Introduction to Information Retrieval*. Cambridge University Press Cambridge, England, online edition edition, 2009.

[23] Andrian Marcus and Denys Poshyvanyk. The conceptual cohesion of classes. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM '05, pages 133–142, Washington, DC, USA, 2005. IEEE Computer Society.

[24] Collin McMillan. Finding relevant functions in millions of lines of code. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1170–1172, New York, NY, USA, 2011. ACM.

[25] RANKS NL. Stopword list. Website, June 2015. `http://www.ranks.nl/stopwords`.

[26] Tim Paek, Bo Thiesson, Yun-Cheng Ju, and Bongshin Lee. Search vox: Leveraging multimodal refinement and partial knowledge for mobile voice search. In *Proceedings of the 21st Annual ACM Symposium on User Interface Software and Technology*, UIST '08, pages 141–150, New York, NY, USA, 2008. ACM.

[27] Martin Porter. The porter stemming algorithm. Website, January 2006. `http://tartarus.org/martin/PorterStemmer/`.

[28] Jef Raskin. Comments are more important than code. *Queue*, 3(2):64–65, March 2005.

[29] Ann C. Smith, Justin S. Cook, Joan M. Francioni, Asif Hossain, Mohd Anwar, and M. Fayezur Rahman. Nonvisual tool for navigating hierarchical structures. *SIGACCESS Access. Comput.*, (77-78):133–139, September 2003.

[30] Black Duck Software. *BlackDuck Open HUB*. URL: `http://code.openhub.net/`, 2014. Looked up: 29-03-2015.

[31] Sourceforge. Sourceforge - home page. Website, 2015. `http://sourceforge.net/`.

[32] M.-A. Storey, L.-T. Cheng, J. Singer, M. Muller, D. Myers, and J. Ryall. How programmers can turn comments into waypoints for code navigation. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pages 265–274, Oct 2007.

[33] Margaret-Anne Storey, Jody Ryall, R. Ian Bull, Del Myers, and Janice Singer. Todo or to bug: Exploring how task annotations play a role in the work practices of software developers. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 251–260, New York, NY, USA, 2008. ACM.

[34] Ryohei Takasawa, Kazunori Sakamoto, Akinori Ihara, Hironori Washizaki, and Yoshiaki Fukazawa. Do open source software projects conduct tests enough? In Andreas Jedlitschka, Pasi Kuvaja, Marco Kuhrmann, Tomi Männistö, Jürgen Münch, and Mikko Raatikainen, editors, *Product-Focused*

Software Process Improvement, volume 8892 of *Lecture Notes in Computer Science*, pages 322–325. Springer International Publishing, 2014.

[35] Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. acomment: Mining annotations from comments and code to detect interrupt related concurrency bugs. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 11–20, New York, NY, USA, 2011. ACM.

[36] Bo Thiesson. Web intelligence course. Website, April 2015. `https://www.moodle.aau.dk/course/view.php?id=8275`.

[37] Peter Vogel. No comment: Why commenting code is still a bad idea. Website, 2013. `http://visualstudiomagazine.com/articles/2013/07/26/why-commenting-code-is-still-bad.aspx`.

[38] Peter Vogel. Why you shouldn't comment (or document). Website, 2013. `http://visualstudiomagazine.com/articles/2013/06/01/roc-rocks.aspx`.

[39] W3C. Extensible markup language (xml) 1.0 (fifth edition). Website, November 2008. `http://www.w3.org/TR/REC-xml/`.

[40] Wikipedia. Cosine similarity. Website, 2015. `http://en.wikipedia.org/wiki/Cosine_similarity`.

[41] Annie T. T. Ying, James L. Wright, and Steven Abrams. Source code that talks: An exploration of eclipse task comments and their implication to repository mining. *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, May 2005.

[42] Carlo Zapponi. *GitHut - A SMALL PLACE TO DISCOVER LANGUAGES IN GITHUB*. URL: `http://githut.info/`, 2014. Looked up: 29-03-2015.