UNIVERSIDAD POLITÉCNICA
DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INFORMÁTICOS
MÁSTER UNIVERSITARIO EN INGENIERÍA DEL SOFTWARE – EUROPEAN MASTER IN SOFTWARE ENGINEERING

Automatic Detection of Outdated Comments in Open Source Java Projects

Master Thesis

Ankita Sadu

Madrid, July 2019

This thesis is submitted to the ETSI Informáticos at Universidad Politécnica de Madrid in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering.

Master Thesis
Master Universitario en Ingeniería del Software – European Master in Software Engineering
Thesis Title: Automatic Detection of Outdated Comments in Open Source Java Projects
Thesis no: EMSE-2019-5
July 2019

Author: Ankita Sadu
Master of Science
Universidad Politécnica de Madrid

Supervisor:

Dr. Ana M. Moreno
Full Professor
Universidad Politécnica de Madrid

Software Engineering
ETSI Informáticos
Universidad Politécnica de Madrid

Co-supervisor:

Barbara Russo
Associate Professor
Free University of Bozen-Bolzano

Software Engineering
Faculty of Computer Science
Free University of Bozen-Bolzano

ETSI Informáticos
Universidad Politécnica de Madrid
Campus de Montegancedo, s/n
28660 Boadilla del Monte (Madrid)
Spain

Abstract

Source code comments are significant assets to retain the logic designed during coding. In addition, they aid by communicating the intent of the code to other programmers, maintainers or even to oneself when checked at a later stage. Nevertheless, commenting code and keeping up-to-date comments are often disregarded by programmers. Some common reasons being, the extra work of commenting doesn't deliver much value to the writer, time constraints, and various other reasons. As a first step to aid the process of updating comments following a code change, we present DocRevise, a tool that can automatically detect outdated Javadoc comments of open source Java projects at a fine-grain level. We utilized 30 Java methods from well-known and documented open source Java repositories from Github. Our approach involved inspecting the relationships between the code and the comments and fetching the identifiers that were altered. Experimental results show that DocRevise reaches 80.49% of accuracy when it comes to identifying relationships between the comments and the code. In addition, the results demonstrate that DocRevise can assist developers to locate outdated comments in prior versions of the existing projects.

Acknowledgement

Thanks to my supervisor, Ms. Alessandra Gorla for the guidance and advice she has provided throughout my time as her student. A special thanks to Ms. Arianna Blasi for her valuable ideas. Many thanks to my wonderful parents who are my biggest role models for their love and affection. I will never forget all the sacrifices you made for me. Thanks to my sister, who has always been a constant source of motivation. Thanks to Fabian for being there for me in times of doubts and making me realize that I have the potential.

Table of Contents

Abstract 3

Acknowledgement 3

List of Figures 6

List of Tables 7

List of Listings 8

List of Equations 10

Chapter 1: Introduction 11

Chapter 2: State of the Art 13

Chapter 3: Background 16

3.1. Comments 16
3.1.1. Block Comments 16
3.1.2. End - Of - Line Comments 16
3.1.3. Single Line Comments 17
3.1.4. Trailing Line Comments 17
3.1.5. Doc Comments 17

3.2. Python Lists 19
3.2.1. List 19
3.2.2. Tuple 20
3.2.3. Set 20
3.2.3 Dictionary 20

Chapter 4: Methodology 21

4.1 Software Development Process Model 21
4.1.1 Agile Software Development Process Model 21
4.1.2 Waterfall Software Development Process Model 21

Chapter 5: Design and Implementation 23

5.1. Approach and Design 24

5.2. Tag Checker 25
5.2.1. @param tag check 25
5.2.2. @throws tag check 25
5.2.3. @return tag check 25

5.3. Code and Comment Extractor 25

5.4. Code Formatting 26

5.5. Data Processing 27

5.5. Text and Semantic Mapper 31
5.5.1. Semantic Matching 31
5.5.2. Text Matching 32

5.6. Compare Versions 35
5.6.1. Modification Type of Changes 36
5.6.2. Deletion Type of Changes 37

Chapter 6: Evaluation and Results 39

6.1 Experimental setup 40
6.1.1 Comments Selection 40
6.1.2 Code Selection 40
6.2. Evaluation of Text and Semantic Matcher 41
6.3 Evaluation of Inconsistency Detector 42
6.3.1. Example 1: Renaming of a Method Parameter. 42
6.3.2. Example 2: Renaming a Constant in the if condition and a call to the Getter Method which underwent a Method Name Change. 44
6.2.3. Example 3: Renaming of the Method Name. 46
6.2.4. Example 4: Deletion of an if statement. 47
6.3. Cardinality between the Code and the Comments 48
6.4 Limitations 51
Chapter 7: Conclusion 54
7.1 Considerations on the work 54
7.2 Future work 54
Bibliography 55

List of Figures

Figure 1: Example of Outdated Comments upon Code Statement Deletion..................11
Figure 2: Phases of Agile Software Development Methodology 21
Figure 3: Phases of Waterfall Software Development Process Model 21
Figure 4: Overview of DocRevise 24
Figure 5: State Transition Diagram of DocRevise 24

List of Tables

Table 1: A few Javadoc Tags recognized by the Javadoc Tool *19*
Table 2: Weekly Work Flow *22*
Table 3: Comments and Comment Lists after Processing *30*
Table 4: Code Statements and Code Lists after Processing *31*
Table 5: Semantically Similar Words in the Software Domain *31*
Table 6: Comment and Code Statement Matching of Listing 18 .... *33*
Table 7: Relations of Code and Comment Indices with Similar Terms *34*
Table 8: Two-letter codes of Difflib and their meanings *35*
Table 9: Number of Methods and Classes selected from Repositories *42*
Table 10: Accuracy Results *42*

List of Listings

Listing 1: Example of Block Comment 16
Listing 2: Example of End-of-Line Comment 17
Listing 3: Example of Single Line Comment 17
Listing 4:Example of Trailing Comment 17
Listing 5: Example of Doc Comment 18
Listing 6: Example of List 19
Listing 7: Example of Tuple 20
Listing 8: Example of Set 20
Listing 9: Example of Dictionary 20
Listing 10: Example of Comment and Code Mappings 23
Listing 11: Example of return statement in if 26
Listing 12: Format if and return statements 26
Listing 13: Example of Splitting throw from method declaration 26
Listing 14: Example of throw statement in if 27
Listing 15: Format if and throw statements 27
Listing 16: Example of functioning of Spiral 28
Listing 17: Example of base word in code statement 29
Listing 18: Method addGraph() 30
Listing 19: Comment List after Processing of Listing 18 30
Listing 20: Example of Semantic Matching 32
Listing 21: Comment and Code Indices Mappings for Listing 18 32
Listing 22: Similar Terms of Comments and Code Statements of Listing 18 33
Listing 23: Similar Terms of Return Statement of Listing 18 34
Listing 24: Indices of the Similar Terms of the Return Statement of Listing 18 34
Listing 25: Matched Return Statement of Listing 18 35
Listing 26: Example of Difflib's Output on Modification 35
Listing 27: Example of Difflib's Output on Deletion 36
Listing 28: Output of DocRevise on Code Change of Listing 18 37
Listing 29: To-be-Deleted Return Statement of Listing 18 37
Listing 30: Indices of Code and Comment Mappings of Listing 18 37
Listing 31: Output of DocRevise after Deletion of Return Statement of Listing 18 38
Listing 32: Example of Well-Described Javadoc Comment 40
Listing 33: Examples of Methods with One Line Code Statement 41
Listing 34: Method addAllEdges() 43
Listing 35: Comment and Code Mappings and Similar Terms of Listing 34 43
Listing 36: Output of Tag Checker for Listing 34 44
Listing 37: Output of Consistency Checker after Identifier Name Change 44
Listing 38: Method setNamespace() 44

Listing 39: Comment and Code Mappings and Similar Terms of Listing 38....................45
Listing 40: Output of Tag Checker for Listing 38 45
Listing 41: Output of Consistency Checker after Changes in If Statement and Method Name..45
Listing 42: Method extractNounSubjectSingular() 46
Listing 43: Comment and Code Mappings and Similar Terms of Listing 42....................46
Listing 44: Output of Tag Checker of Listing 42..46
Listing 45: Output of Consistency Checker after Changes in Name of the Method........47
Listing 46: Method findEntity()...47
Listing 47: Comment and Code Mappings and Similar Terms of Listing 46....................47
Listing 48: Output of Tag Checker of Listing 46..48
Listing 49: Return Tag Added to Comments of Listing 46..48
Listing 50: Output of Consistency Checker after deletion of if statement 48
Listing 51: Method getReturnType() 49
Listing 52: Comment and Code Mappings and Similar Terms of Listing 51....................49
Listing 53: Output of Tag Checker of Listing 51..49
Listing 54: Param and Return Tag Added to Comments of Listing 51............................50
Listing 55: Output of Consistency Checker after deletion of return statement 50
Listing 56: Code and Comment Mappings of Listing 18 50
Listing 57: Output of Tag Checker of Listing 18..51
Listing 58: Output of Consistency Checker after deletion of a Code Statement of Listing 18 51
Listing 59: Output of Consistency Checker after Medification of Code Statement of Listing 18..51
Listing 60: Example of Throw Statement Split in Multiple Lines....................................52
Listing 61: Method ComplexFormat()..52
Listing 62: Comment and Code Mappings and Similar Terms of Listing 61....................53
Listing 63: Expected Code and Comment Mappings for Listing 61................................53

List of Equations

Equation 1: Format of Code and Comment Mapping..32
Equation 2: Equation of Precision 39
Equation 3: Equation of Recall 39
Equation 4: Equation of F1 Score 39

Chapter 1: Introduction

When a decent number of users use a piece of software, they explore and utilize it for their maximum profit. During this process, they are likely to realize some features of the software that could be improved in a way that meets their expectations. Therefore, changes in the software are expected and this effort to govern the evolving changes is coined as Software Maintenance. Since the act of reading code on its own requires a great deal of time and effort, documenting code (also known as comments) was introduced as one of the key aspects in Software Maintenance. Comments are intended for anyone including the programmer who may need to maintain, extend or refactor code. But, commenting code and keeping them updated becomes painful when the time is scarce and release deadlines put pressure. Furthermore, when software evolves, it is common for comments and source code to be out-of-sync. An inconsistency between the two leads to severe implications for software robustness and productivity. (Tan, et al., 2007)

In addition, outdated comments have been observed to consume a great deal of time of the programmer and making the process of code reviews more exhausting. (Parnas, 2011) It was observed that most of the outdated comments were seen after code changes. (Fluri, et al., 2007) Since code is read more often than it is written, it is, therefore, vital to improving the readability of the code by including comments and keeping them revised with accordance to the current settings. (Scheufler, 2016)

The outdated comment from *Figure 1* represents one example of an outdated comment that was intended for the method *putAll*, captured from two versions in time. There is a possibility of the code reader getting confused and mislead by the objective the method is trying to achieve. Upon deletion of these specific lines, it is expected that their corresponding comments need to be deleted.

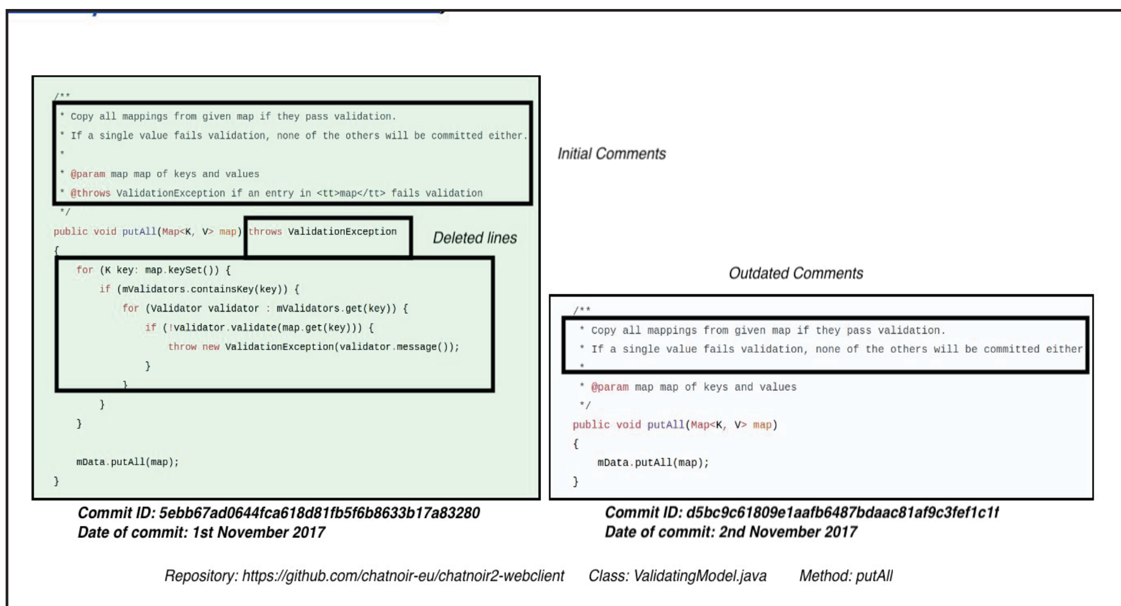

Figure 1: Example of Outdated Comments upon Code Statement Deletion

Current studies show the introduction of methodologies that detected outdated inline and block comments performed at the method or statement levels. (Liu, et al., 2018) On the other hand, few other methodologies detected outdated doc comments. (Khamis, et al. 2010) (Tan, et al.,

2012) Doc comments are well structured and are expected to be written by following doc guidelines, high accuracy was achieved when detecting their outdatedness.

Some more approaches involved automatically generating code and method summary comments. (Xing, et al., 2018) (Edmund, et al., 2015) These were performed on existing software repositories in order to use comments from some software segment to describe other similar software segments. Other operations involved method summary generation, performed on method signatures and bodies. (Sridhara, et al., 2010) This was conducted as a means to abridge the actions of a method.

Many of the software development and maintenance tools involve “Identifiability” i.e. matching between natural language words and software artifacts or “Searchability” i.e. matching between search queries by user and software artifacts. Studies include expanding the dictionary of semantically-similar words by exploring the relationships between code and comments. (Sridhara, et al., 2008) (Yang, et al., 2012) (Howard, et al., 2013)

Co-evolution of code and comments were investigated keeping in mind the significance of comments in programming practice and the fact that their outdatedness may misguide the code-readers. (Fluri, et al., 2007)

The goal of this thesis is the creation of a tool that automatically detects code-comment inconsistencies in open source Java projects using automated Natural Language Processing (NLP). In particular, the thesis work focuses on the inconsistencies between Javadoc comments at the method level in a Java class. As a means to review the outdatedness of a comment, our approach basis its evaluations by drawing relations between the to-be-changed code and its corresponding comments. Once this relationship is captured, upon code changes, the user is instigated to update or delete the correlated comment. Our experimental results indicate that our model can identify these code and comment relationships with an accuracy of 80.49%. Moreover, DocRevise has demonstrated that it can potentially assist in bringing to light outdated comments on the basis of version history.

The thesis is organized as follows: Chapter 2 reports the state of the art in outdated comments detection, comment generation, mining of semantically similar words in the software domain and code and comment co-evolution. Chapter 3 provides background on comments and the style of representation we used in order to identify comment and code mappings. In Chapter 4, we present the software development methodology we used to create our tool, DocRevise. Chapter 5 describes in detail the design and implementation of DocRevise. In addition, we introduce its components and how it accomplishes the objectives. Chapter 6 exhibits the experimental setup, accuracy of how well DocRevise can identify relationships between the code and the comment. We also provide a few examples of DocRevise highlighting the part of the code altered and the related comment that requires verification. Finally, Chapter 7 outlines the considerations of our work and what we can expect from future works.

Chapter 2: State of the Art

Source code comments play a pivotal role in comprehending the intentions of the program. Considering the significance of comments being in alignment with the source code in programming practice, detection of outdated inline, block and doc comments were carried out by researchers. Doc comments, in particular, are structured in a way that the purpose of a method is revealed, and complex code is comprehended. Outdated doc comments can lead to misinterpreting the Application Programme Interface (API).

(Liu, et al., 2018) scrutinized both block and inline outdated comments during code changes. We identified their work to be very closely related to our work since their technique also aimed at automatically detecting outdated comments for arbitrary comments. They operated their study by using on a set of heuristics that deduced the scope of a comment. Features of the comments at the method and class levels were analysed to describe the context and statement level comments and were utilized to describe the changes. Natural language techniques were used to pre-process the data obtained from both code and comment. Since there is a lexical gap between programming and natural language, they developed a Skip-gram model which bridges this gap by projecting natural language statements and code snippets as meaningful vectors in shared representation space. In order to obtain the vector representation of each word, a Skip gram was trained based on the corpus that was generated by randomly selecting two words for every comment and code statement word. In order to establish a relationship between the code and the comment, similarity measures were carried out that included, word to word, word to sentence and sentence to sentence. In addition, they employed random forests to label up-to-date and outdated comments. Although the experimental results show that all their features that included code, comment, relationship and, code and relationship features had an F1 Score of 0.76, the F1 Score of relationship features only i.e. the code and comment mappings was 0.38. The reason could be because they utilized a Skip-gram model that picks random words for their comment and code documentation. Randomly picking words could lead to missing out critical information from both code and comments. In addition, they weren't able to locate the term in the code that was affected by the change. On the other hand, DocRevise is able to locate the line that underwent the change and highlight the identifiers in that line that were revised.

(Tan, et al., 2007) took the first step in automatically analysing comments to extract implicit program rules and use these rules to detect code and comment inconsistencies. Comment rules were extracted from the comments that were basically a programmer's assumption and requirement. Next, mismatches between these comment rules and the source code were performed by a rule checker. This rule checker performed flow-sensitive and context-sensitive program analysis of the entire source code. Their method achieved high accuracy in detecting topic specific comments only.

(Khamis, et al. 2010) investigated the quality of inline documentation using a set of heuristics that targeted both the quality of the language and consistencies between the Javadoc comments and source code. In particular, the parameter names, return types and exceptions in the Javadoc tags namely @param, @throws, @return respectively were checked of a method. Since doc comments are well-structured especially the Javadoc tags, their automatic comment analysis technique achieved excellent accuracy. (Tan, et al., 2012) introduced a tool namely @tcomment that Randoop tool to test for outdated Javadoc comments. They primarily focused on the Javadoc tags and excluded the free flow text i.e. the description in their analysis. The comments were automatically analysed to fetch likely properties of each method. This testing was performed in order to check whether the method properties i.e. null values and related

exceptions report any inconsistencies with the comments. They were also able to achieve a high accuracy of 99.1% due to the same fact of Javadoc tags being well-structured. In our work, we focused more on drawing relations from the comment description to the source code and detecting if they were inconsistent.

An alternative to solve code-comment inconsistencies was automatically generating comments. (Hu, et al., 2018) introduces DeepCom that applied Natural Language Processing techniques to learn from a large code corpus and automatically generate comments from learned features. A deep neural network was used to analyze structural information of Java methods for improved comments generation. Even though DeepCom outperforms other methods that generate comments, unknown words were generated in the comments since these words were user-defined. It was challenging for DeepCom to learn them and therefore, the user-defined words were replaced by unknown tokens. This might lead to missing out significant information in the comment generated.

With the same objective, (Wong, et al., 2015) introduced a tool namely, CloCom and applied code-clone detection techniques to discover similar code segments and use comments from some code segments to describe the other similar code segments. They utilized natural language techniques to select relevant comment sentences. A three-way context-sensitive analysis was performed between 1) the code and comment 2) the code segment from the database and 3) the code segment from the target projects. The quality of the comments generated was not so high since existing code comments contained not useless or trivial comments that might have been written by programmers who were pressured to write comments for the purpose of writing comments. These comments were not incorrect, but they were seen to be not useful for a programmer.

On the other hand, for programmers to swiftly identify the actions of a method and giving rise to aiding program comprehension, (Sridhara, et al., 2010) presented a novel technique to automatically generate descriptive summary comments for Java methods. They utilized the Software Word Usage Model(SWUM) to identify major s_units(Java statements) that are significant to be used to generate summaries of the Java methods. 7 out of 8 methods had summaries generated that did not miss significant content and were considered to be reasonably concise. These works also aimed in aiding code-comment inconsistencies but by generating comments or method summaries for the programmer.

In order to establish a relationship between the code and the comments, various other works have highlighted the study and extraction semantically-similar words. (Yang, et al., 2012) emphasizes the inability to guess the exact word used in the code while performing code searches. In order to address this, semantically similar words in the software domain were extracted and refined in order to increase the effectiveness in search queries. These semantically-similar word pairs were extracted in the context of 1) comment and comment 2) code and code 3) code and comment by using Natural Language Processing techniques. Their method achieved high precision and recall. However, they highlighted that their method could not perform on the poorly named method identifiers. In addition, the implementation could not characterize if a word pair is a synonym, related, antonym, near antonym, or identifier. We found this study partially related to our work since they also infer the use of semantically similar words and utilized some of the mined word pairs.

(Howard, et al., 2013) presented their approach on only comment and code mappings by extracting semantically-similar word pairs that did not already occur in WordNet. They analyzed leading comments that were descriptive. Next, the main action word from both the method's signature and the leading comment were extracted by feeding them to a Part of Speech Tagger.

This was done in order to identify the verbs that are necessary to build the word pair. The candidates of the comment and code mapping for the semantically-similar word pairs were formed. They were then analysed and ranked in order to automatically generate a list of semantically-similar word pairs. At a frequency threshold of 10 or more occurrences, 78% of the time, the mined semantically-similar words were agreed by human opinion.

(Sridhara, et al., 2008) also performed the identification of word relations in the software domain. They compared six state of the art, Semantic Similarity tools(LCH, WUP, JCN, RES, LIN, LESK) to evaluate their effectiveness on words from the comments and identifiers in software. By analyzing methods, they identified word pairs of related words and formed a gold set. They ranked the top 10 and bottom 15 word pairs for each Semantic Similarity tool and evaluated their performance. It came to a conclusion that JCN and WUP performed better than the others. This work was closely related to mappings of our code and comment since we utilized some of the top 10-word pairs for the purpose of semantic matching.

Some more studies inspected the co-evolution of comments and code. (Fluri, et al., 2007) They examined the question whether the code and associated comments were actually changed jointly along the evolutionary history of a software system. This involved investigating three open source systems (i.e., ArgoUML, Azureus, and JDT Core) and describing how comments and code co-evolved over time. They had some interesting findings, 1) newly added code barely gets commented 2) class and method variables were commented most frequently but method class were far less and 3) 97% of comment changes were done in the same revision as the associated source code change. They indicated that comments describe the source code when each word appearing in the comment as well as in the source code entity. In addition, they compared the versions of the source code and checked how the code and the comments evolved overtime. They concluded from this that it was more likely that a statement delete triggers a comment change in the same revision than a statement update does. We found this study related to our work since it involved comparing source code versions.

Chapter 3: Background

In this chapter, we present the entities that we have used for the purpose of identifying the type of data we selected for our study and the form we chose to represent it.

3.1. Comments

This work primarily involves the inspection of source code comments. According to the Oracle's Code Conventions for Java, programs can have two types of comments:

- **Implementation Comments**: Useful for commenting out code or are meant for a specific implementation.
- **Documentation Comments (known as "doc comments")**: Java Comments delimited by */**...*/*. They can be extracted to HTML files using the Javadoc tool. They describe the specifications of the code. (Oracle.com, 2019)

Programs can have two primary styles of Implementation Comments:

3.1.1. Block Comments

According to Oracle's Code Conventions for Java, "Block comments are used to provide descriptions of files, methods, data structures and algorithms. Block comments may be used at the beginning of each file and before each method." (Oracle.com, 2019)

A block comment typically should look like this:

```
/*
* Here is a block comment.
*/
```

Listing 1: Example of Block Comment

3.1.2. End - Of - Line Comments

Oracle's Code Conventions explains that the // comment delimiter can comment out a complete line or only a partial line. It can be used in consecutive multiple lines for commenting out sections of code. (Oracle.com, 2019)

All the three styles are mentioned:

```
if (foo > 1) {

    // Do a double-flip.
    ...
}
else {
    return false;           // Explain why here.
}

//if (bar > 1) {
```

```
//
//    // Do a triple-flip.
//    ...
//}
//else {
//    return false;
//}
```

Listing 2: Example of End-of-Line Comment

3.1.3. Single Line Comments

Oracle explains short comments appearing on a single line indented to the level of the code that follows. (Oracle.com, 2019)

A single-line comment in Java code:

```
if (condition) {

    /* Handle the condition. */
    ...
}
```

Listing 3: Example of Single Line Comment

3.1.4. Trailing Line Comments

The conventions also explain very short comments that appear on the same line as the code they describe. They should be shifted far enough to separate them from the statements. (Oracle.com, 2019)

Here's an example of a trailing comment in Java code:

```
if (a == 2) {
    return TRUE;            /* special case */
} else {
    return isPrime(a);      /* works only for odd a */
}
```

Listing 4:Example of Trailing Comment

We used Documentation Comments for the purpose of the thesis.

3.1.5. Doc Comments

The conventions specify that a doc comment is written in HTML and must precede a class, field, constructor or method declaration. The two parts of a doc comment are 1) a description (first line is usually a summary line that starts with a verb) 2) followed by block tags. (Oracle.com, 2019)

The block tags are used are @param, @return, and @see:

```
/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute {@link URL}. The name
 * argument is a specifier that is relative to the url argument.
```

```
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 *
 * @param  url  an absolute URL giving the base location of the image
 * @param  name the location of the image, relative to the url
argument
 * @return      the image at the specified URL
 * @see         Image
 */
 public Image getImage(URL url, String name) {
        try {
            return getImage(new URL(url, name));
        } catch (MalformedURLException e) {
            return null;
        }
 }
```

Listing 5: Example of Doc Comment

Oracle mentions the some of following tags that the Javadoc tool recognizes in Table 1.

Tag Name	Tag Description	Syntax
@author	Adds an "Author" entry with the specified name-text.	@author name-text
@param	Adds a parameter with the specified parameter-name followed by a specified description.	@param parameter-name description
@return	Adds a "Returns" section with the description text.	@return description
@throws	@throws and @exception tags are synonyms. Adds a "Throws" subheading to the generated documentation, with the class-name and description text.	@throws class-name description
@exception	The @exception tag is a synonym for @throws.	@exception class-name description
@see	Adds a "See Also" heading with a link or text entry that points to reference.	@see reference
@since	Adds a "Since" heading with the specified since-text to the generated documentation.	@since release
@serial	Used in the doc comment for a default serializable field.	@serial field-description | include | exclude
@deprecated	Adds a comment indicating that this API should no longer be used.	@deprecated deprecatedtext
{@code}	Displays text in code font without interpreting	{@code text}

	the text as HTML markup or nested Javadoc tags.	
{@inheritDoc}	Inherits (copies) documentation from the "nearest" inheritable class or implementable interface into the current doc comment at this tag's location.	Inherits a comment from the immediate surperclass.
{@link}	Inserts an in-line link with visible text label that points to the documentation for the specified package, class or member name of a referenced class.	{@link package.class#member label}

Table 1: A few Javadoc Tags recognized by the Javadoc Tool

They mention that the first line i.e. the first line of each member, class, interface or package description must be a summary sentence, that contains a concise but complete description of the API item. In addition, the best API names are "self-documenting", meaning they tell one basically what the API does. If the method description uses only the words that appear in the method name, then it is adding nothing at all to what one could infer. An ideal comment goes beyond those words and should always reward one with some bit of information that was not immediately obvious from the API name.

Tags are classified into two types:

1. **Block tags** - Placed only in the tag section that follow the main description. Block tags are of the form: *@tag*.

2. **Inline tags** - Placed anywhere in the main description or in the comments for block tags. Inline tags are of the form: *{@tag}.*

The most commonly used tags that we based our study on and that the ones we discovered in our experiment were the @param, @return, @throws, @exception, @link, @since, @code and @see. DocRevise considers both block and inline tags.

3.2. Python Lists

This work involved the use of lists in python in both implementation and representation of the indices of the code and comment mappings. According to Python's documentation, there are four collection data types in the Python programming language (Docs.python.org, 2019):

3.2.1. List

List is a collection which is ordered and changeable. It allows duplicate members. Lists are written with square brackets.

```
stringlist = ["mary", "jack", "peter"]
numberlist = [1, 2, 3, 4]
```

Listing 6: Example of List

3.2.2. Tuple

Tuple is a collection which is also ordered but unchangeable. It allows duplicate members. Tuples are written with round brackets.

```
stringtuple = ("mary", "jack", "peter")
numbertuple = (1, 2, 3)
```

Listing 7: Example of Tuple

3.2.3. Set

Set is a collection neither is ordered nor indexed. It does not allow any duplicate members. Sets are written with curly brackets.

```
stringSet = {"mary", "jack", "peter"}
numberSet = {1, 2, 3}
```

Listing 8: Example of Set

3.2.3 Dictionary

Dictionary is a collection which is unordered, changeable and indexed. It does not allow duplicate members. Dictionaries are written with curly brackets, and they have keys and values.

```
thisdict =  {
  "name": "mary",
  "last_name": "owens",
  "birth_year": 1995
}
```

Listing 9: Example of Dictionary

We used all the four above for most of our processes. They involved conversion of sets, tuples and dictionary to lists. Lists worked well with the libraries that we used for our work and were perfect for the representation of the indices of our code and comment mappings.

Chapter 4: Methodology

4.1 Software Development Process Model

It is very crucial to decide upon the type of developmental process that should be used to develop software. There were various methodologies for software development namely Agile Software Development, Waterfall Approach, Xtreme Programming and more. The most common developmental processes are Waterfall and Agile.

4.1.1 Agile Software Development Process Model

Agile development model is a type of Incremental model. Software is developed in incremental, rapid cycles. This results in small incremental releases with each release building on previous functionality. Each release is thoroughly tested to ensure software quality. (shown in figure 2)

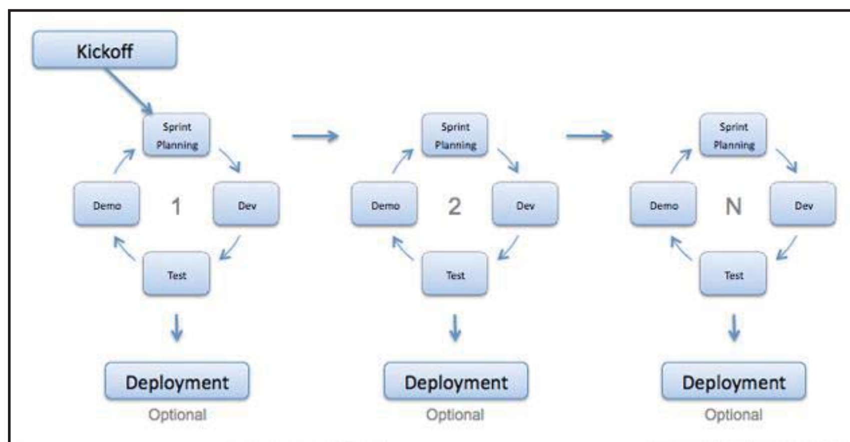

Figure 2: Phases of Agile Software Development Methodology

4.1.2 Waterfall Software Development Process Model

The Waterfall Model was first Process Model to be introduced. It is also referred to as a linear-sequential life cycle model. It is very simple to understand and use. In a waterfall model, each phase must be completed fully before the next phase can begin. (shown in figure 3).

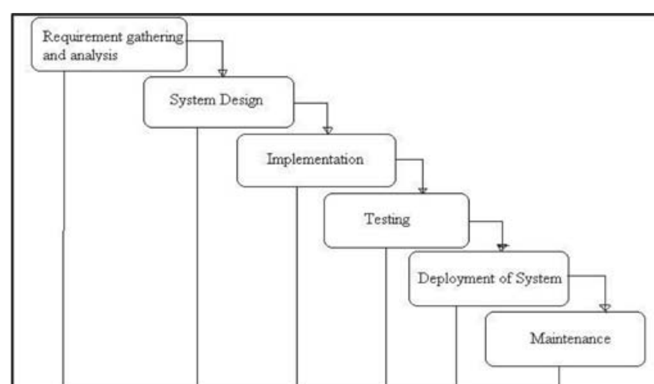

Figure 3: Phases of Waterfall Software Development Process Model

As this work was in an individual setting, Semi Agile methodology was chosen for DocRevise due to Agile's commitment to technical excellence and good design. Due to its iterative nature, the components and features were developed incrementally enabling benefits to be realized early as the software develops. There is also a risk management factor in this that helps to identify any issues early and makes it easier to respond to them. On the other hand, the same was identified as not true in the waterfall model. My supervisor was the Product Owner and weekly meetings were scheduled to check whether the working components and features were in alignment with the given milestones. Every sprint was planned in such a way that the last day of the weekdays was left extra in order to cope up with changing requirements.

		WEEK		
Monday	**Tuesday**	**Wednesday**	**Thursday**	**Friday**
Meet and Plan	Design	Implement	Implement	Cope up with Changing Requirements

Table 2: Weekly Work Flow

Chapter 5: Design and Implementation

In this chapter, DocRevise's design and implementation is presented. DocRevise focuses on the detection of Javadoc comments during code changes. In contrast to the existing techniques, DocRevise aims to detect outdated method level Javadoc comments (both free flow text and Javadoc tags).

```
/**
     * Returns the coefficients of the derivative of the polynomial with the
given coefficients.
     *
     * @param coefficients Coefficients of the polynomial to differentiate.
     * @return the coefficients of the derivative or {@code null} if
coefficients has length 1.
     * @throws NoDataException if {@code coefficients} is empty.
     * @throws NullArgumentException if {@code coefficients} is {@code null}.
     */
protected static double[] differentiate(double[] coefficients)
                             throws NullArgumentException, NoDataException {

        MathUtils.checkNotNull(coefficients);

        int n = coefficients.length;

        if (n == 0) {
            throw new
NoDataException(LocalizedFormats.EMPTY_POLYNOMIALS_COEFFICIENTS_ARRAY);
        }

        if (n == 1) {
            return new double[]{0};
        }

        double[] result = new double[n - 1];

        for (int i = n - 1; i > 0; i--) {
            result[i - 1] = i * coefficients[i];
        }

        return result;
    }
```

Listing 10: Example of Comment and Code Mappings

(https://github.com/apache/commons-math/blob/master/src/main/java/org/apache/commons/math4/analysis/polynomials/PolynomialFunction.java)

In order to detect outdated comments, it is essential to establish relation between the code and the comment. Listing 10 shows the likely matchings of each comment embedded between /** and */ with the code statements. Once this relationship is established, the versions can be compared to detect code-comment inconsistencies.

5.1. Approach and Design

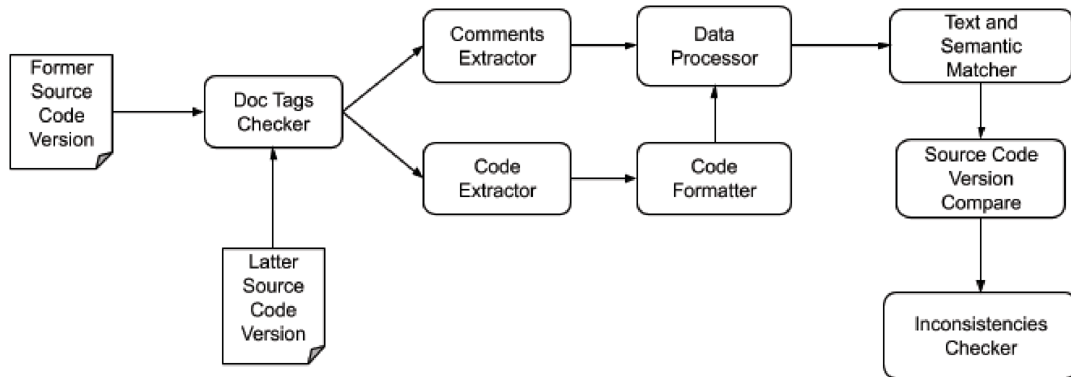

Figure 4: Overview of DocRevise

Figure 4 shows the approach of DocRevise. Primarily, DocRevise takes two inputs:

1. **Former Source Code Version**: The version of the source code before code changes.
2. **Latter Source Code Version:** The version of the source code after code changes.

The two code versions were taken from open source Java projects from Github. The output of DocRevise is a mechanism that compares both versions of code and informs the user for the changes that are required to be made to the comment after the related code statements change. A State Transition diagram of DocRevise is presented in figure 5.

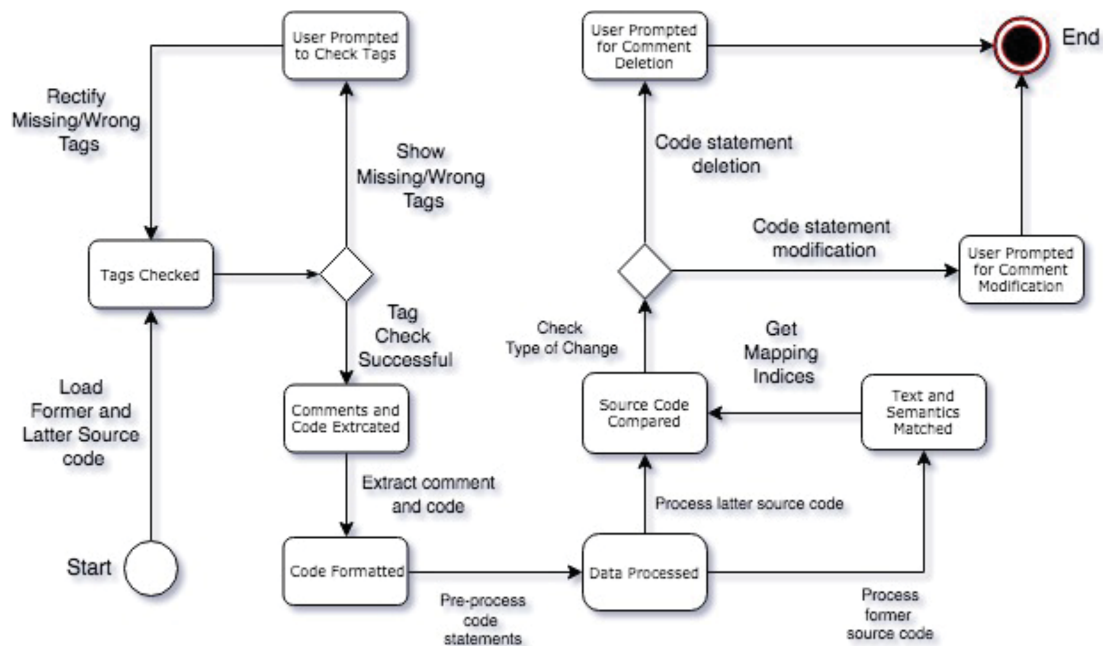

Figure 5: State Transition Diagram of DocRevise

5.2. Tag Checker

The source code extracted is initially passed through a Tag Checker. This module ensures whether all the necessary Javadoc tags have been mentioned in the comment and checks whether they are consistent with the code. The Tag Checker included the checking of @param, @return and @throws tags only against the code since they are the most commonly used tags at a method level.

5.2.1. @param tag check

Parameters are checked in the method declaration against the @param tags in the comments. The programmer is prompted when the parameters do not match. We also considered cases when the programmer adds a stop word right after the @param tag. We do this by removing the stop word and considering the second word which is expected to be the parameter name.

5.2.2. @throws tag check

For the @throws tags, we consider the case of both Checked and Unchecked exceptions. We took into consideration that unchecked exceptions do not need to be declared in a method or constructor's throws clause if they can be thrown by the execution of the method or constructor and propagated outside the method boundary. (Docs.oracle.com, 2019) Therefore, @throws tag was checked against the code statements by checking if there were any “throw new” terms in the code. We expected the very next word to be the name of the exception that needs to be documented in the @throws tag.

5.2.3. @return tag check

Lastly, Tag Checker only checks if there is a return statement in the code for inspecting the @return tag.

Tag Checker can be replaced with @tcomment (Tan, Shin Hwei, et al., 2012) or The Javadocminer (Khamis, Ninus, et al. 2010) that have proven themselves to be not far from perfect when it comes to detecting code and Javadoc tags inconsistencies. We did not focus much on Tag Checker’s improvements because the inspections of Javadoc tag inconsistencies have been well studied.

5.3. Code and Comment Extractor

The method’s code statements and comments are read collectively in order to extract them using Regular Expressions. These regular expressions were created by identifying the starting i.e. /** and ending i.e. */ of a comment. This extraction was necessary in order to have two separate entities (code and comment) that would later on be processed to draw relations.

5.4. Code Formatting

We wanted to enumerate the code statements in order to form code and comment mappings. Therefore, for code to have correct indexing, the lines underwent some rearrangements before being cleaned. This formatting involved:

1. The lines that did not contain code statements were skipped. We removed the lines that only contained the opening ("{") or closing ("}") curly brackets that usually mark the beginning and ending of methods and control flow statements.

2. If statements followed by a return statement were considered as one statement in order to match @return tags and the free flow text that follows after. Therefore, the return statement was placed on the same line as the if statement. For example, for comments that contain: "*@return the parts if the implementation matches or a zero-length array if not.*" would match the code statement(s):

```
if (!graph.containsVertex(vertex)) {
            return false;
}
```

Listing 11: Example of return statement in if

(https://github.com/jgrapht/jgrapht/blob/master/jgrapht-core/src/main/Java/org/jgrapht/Graphs.Java)

Therefore, we format the code statements to:

```
if (!graph.containsVertex(vertex)) return false;
```

Listing 12: Format if and return statements

3. The method parameters were also removed from the method declaration since the parameter tags were checked beforehand by the Tag Checker.

4. The exception declaration part from the method declaration was split from the method declaration and inserted to the next line in order to better match the @throws tags. For example, the insertion looked like this:

```
public ComplexFormat(......) *newline here*
      throws NullArgumentException, NoDataException {
```

Listing 13: Example of Splitting throw from method declaration

(https://github.com/apache/commons-math/blob/master/src/main/Java/org/apache/commons/math4/complex/ComplexFormat.Java)

5. If statements followed by a throw statements were considered as one statement in order to match well with the @throws tags and the free flow text that follows after. Therefore, the throw statement was placed on the same line as the if statement.

For example, the comments that contain, "*@throws NullArgumentException if {@code imaginaryCharacter} is {@code null}.*" would match the code statements:

```
if (imaginaryCharacter == null) {
        throw new NullArgumentException();
    }
```

Listing 14: Example of throw statement in if

(https://github.com/apache/commons-math/blob/master/src/main/Java/org/apache/commons/math4/complex/ComplexFormat.Java)

Therefore, we format the code statements to:

```
if (imaginaryCharacter == null) throw new NullArgumentException();
```

Listing 15: Format if and throw statements

5.5. Data Processing

Due to a lexical gap between the natural language and programming language, it is crucial to clean both comment and code. This bridges the gap between them and enables relationships to be established in order to compare versions at a later stage. Once the code lines were formatted, both code and comments underwent data cleaning. Some major data cleaning steps carried out for comments are as follows:

1. Comments were split in order to form comment propositions. These were formed by splitting the comment on the basis of Full Stop(.), Comma (,) and Semicolon (;). Therefore, missing punctuation marks were inserted before @param, @throws, @exception and @returns for efficient splitting.

2. All content between the HTML <code>…</code> tags was processed by removing full stop, comma and semicolon.

3. HTML tags like <p>...</p>, <code>...</code> that are commonly used in Javadoc comments were excluded using Beautiful Soup. (Crummy.com, n.d.) Beautiful Soup is a library that makes it easy to scrape information from web pages. It sits atop an HTML or XML parser, providing Pythonic idioms for iterating, searching, and modifying the parse tree. We used the *get_text()* function of Beautiful Soup that extracts all the text and excludes html tags.

4. Special Characters were excluded except Full Stop(.), Comma (,) and Semi_colon(;) because they were used for splitting the comments.

5. Empty lines and leading spaces were stripped out.

6. All content between the Javadoc tag "{@code" and "}" was processed by removing full stop, comma and semicolon.

7. The Javadoc tag lines @param, @see, @link and @since were skipped. The line containing @param was skipped because the consistency of this tag could be checked

by the Tags Checker against the method declaration of the method. @since, @link and @see were excluded due to the fact that they provide no information about the actions the method is performing. On the other hand, @throws or @exception and @return lines were not excluded since the term "throw" indicates exceptions and "return" indicates the return value of the method are present in the body of the method. These were valuable terms for matches in the code. In addition, they provide information about the method's behaviour.

8. The split lines were tokenized using Spiral which is a Python 3 module that provides functions for splitting identifiers found in source code files (Huck, 2018) Spiral is a Python 3 package that implements numerous identifiers splitting algorithms. Identifier splitting (also known as identifier name tokenization) is the task of breaking apart program identifier strings such as *getInt* or *readUTF8stream* into component tokens: [get, int] and [read, utf8, stream]. It provides some basic naive splitting algorithms, such as a straightforward camel-case splitter, as well as more elaborate heuristic splitters, such as a new algorithm called Ronin.

   Consider the following python code that takes identifiers as input and tokenizes them:

```
from spiral import ronin
for s in [ 'mStartCData', 'nonnegativedecimaltype', 'getUtf8Octets',
'savefileas','nbrOfbugs']:
print(ronin.split(s))

OUTPUT:

['m', 'Start', 'C', 'Data']
['nonnegative', 'decimal', 'type']
['get', 'Utf8', 'Octets']
['save', 'file', 'as']
['nbr', 'Of', 'bugs']
```

Listing 16: Example of functioning of Spiral

   Spiral also works reasonably well on natural language terms therefore we used it for tokenizing both code and comment. The real challenge comes when code needs to be tokenized that requires affixed terms to be isolated. The reason why we selected Spiral for tokenization of code was because it was specially created for splitting program identifiers. These tokenized words form sub lists of their respective comment list.

9. All the words were converted to lowercase letters and lemmatized using spaCy (Spacy.io, n.d.) in order to produce exact code and comment mappings. According to NLP Stanford, Lemmatization usually refers to doing things properly with the use of a vocabulary and morphological analysis of words, normally aiming to remove inflectional endings only and to return the base or dictionary form of a word, which is known as the lemma. Lemmatization would attempt to return either see or saw depending on whether the use of the token was as a verb or a noun. (Nlp.stanford.edu, n.d.)

   spaCy is compatible with 64-bit CPython 2.7 / 3.5+ and runs on Unix/Linux, macOS/OS X and Windows. (https://spacy.io/api/lemmatizer) We lemmatize words in order to match words between the code and the comment that are from the same lemma form.

For example, if the comment is, “*Target vertex is added to the graph*” and the matching code statement is:

```
graph.addVertex(targetVertex);
```

Listing 17: Example of base word in code statement

(https://github.com/jgrapht/jgrapht/blob/master/jgrapht-core/src/main/Java/org/jgrapht/Graphs.Java)

then both comment and code will undergo lemmatization in order to creating mappings between them. In the example, the lemmatization will include converting “added” to “add” i.e. its lemma form.

10. Duplicates in sub lists were removed to avoid incorrect mappings.

11. Stop words (like “ourselves”, “hers”, “those” and “others”) except “a” since it could be a not well named identifier were removed. Contractions (like “ain't”, “aren't”, “can’t” and others) were also removed.

The processing of code statements followed the same procedure as the comment except that Java reserved words like “public”, “abstract”, “continue” and others were excluded from the code data except “if”, “throw”, “return”, and “else”. We found these terms to be valuable for matches in the comments. Consider the example, Code *snippet 4* that shows how data was processed for both code and comments.

```
/**
     * Adds all the vertices and all the edges of the specified source
graph to the specified
     * destination graph. First all vertices of the source graph are
added to the destination graph.
     * Then every edge of the source graph is added to the destination
graph. This method returns
     * <code>true</code> if the destination graph has been modified as a
result of this operation,
     * otherwise it returns <code>false</code>.
     *
     * <p>
     * The behavior of this operation is undefined if any of the
specified graphs is modified while
     * operation is in progress.
     * </p>
     *
     * @param destination the graph to which vertices and edges are
added
     * @param source the graph used as source for vertices and edges to
add
     * @param <V> the graph vertex type
     * @param <E> the graph edge type
     *
     * @return <code>true</code> if and only if the destination graph
has been changed as a result
     *         of this operation.
     */
0  public static <V, E> boolean addGraph(Graph<? super V, ? super E>
   destination, Graph<V, E> source){
1  boolean modified = addAllVertices(destination, source.vertexSet());
2  modified |= addAllEdges(destination, source, source.edgeSet());
3  return modified;
```

```
    }
```

Listing 18: Method addGraph()

(https://github.com/jgrapht/jgrapht/blob/master/jgrapht-core/src/main/Java/org/jgrapht/Graphs.Java)

The processed data of the comment is arranged in a list:

```
[['vertex', 'add', 'all', 'edge', 'graph', 'source', 'destination',
'specify'],

['vertex', 'start', 'all', 'graph', 'source', 'destination', 'add'],

['edge', 'every', 'graph', 'source', 'destination', 'add'],

['if', 'method', 'result', 'operation', 'true', 'return', 'graph', 'modify',
'destination'],

['otherwise', 'false', 'return'],

['behavior', 'if', 'operation', 'undefined', 'graph', 'modify', 'progress',
'specify'],

['if', 'result', 'operation', 'true', 'change', 'graph', 'return',
'destination']]
```

Listing 19: Comment List after Processing of Listing 18

Each comment is sequentially mapped to each sub list in the processed comment list. Some mappings between the comments and the processed comment lists are as follows:

Comment	Processed Comment List
Adds all the vertices and all the edges of the specified source graph to the specified destination graph.	*['add', 'destination', 'all', 'vertex', 'source', 'graph', 'specify', 'edge']*
First all vertices of the source graph are added to the destination graph.	*['add', 'destination', 'all', 'vertex', 'source', 'graph', 'start']*
…	…

Table 3: Comments and Comment Lists after Processing

Likewise, each code statement is sequentially mapped to each sub list in the generated code list. The mappings between the code statements of code snippet 4 and generated lists are as follows:

Code Statement	Processed Code List
`public static <V, E> boolean addGraph(Graph<? super V, ? super E> destination, Graph<V, E> source){`	*['add', 'v', 'e', 'graph']*
`boolean modified = addAllVertices(destination, source.vertexSet());`	*['vertex', 'all', 'add', 'set', 'modify', 'source', 'destination']*

...	...

Table 4: Code Statements and Code Lists after Processing

The sub lists of each comment and code are then indexed.

5.5. Text and Semantic Mapper

5.5.1. Semantic Matching

There could also be cases where a particular word in the comment might not be the exact match to the word in the code but could mean the same. Considering synonyms in natural language might not always render the best results. The reason being, some words might not be considered as synonyms in natural language but are considered in the Software domain. For example, “add” and “append” are considered as synonyms in the Software domain. This needs to be taken into consideration as a relationship needs to be established between words that exist in code and comment.

Some of semantically similar words in the Software domain considered are mentioned in Table 5.

Sr. No.	Word Pairs	
	First List	Second List
1	first	start
2	nothing	null
3	last	end
4	write	save
5	make	create
6	display	show
7	start	begin
8	add	append
9	last	end
10	determine	check
11	remove	clear
12	sort	compare
13	put	add
14	search	find
15	cancel	abort
16	handle	respond
17	init	initialize
18	fire	notify
19	undo	restore
20	quit	close

Table 5: Semantically Similar Words in the Software Domain

(Sridhara, et al., 2008) (Yang, et al., 2012) (Howard, et al., 2013)

If any terms match the First list of semantically similar words, then these words are converted to the Second list.

```
/**
     * Search parent directories for the build file.
     *
     * ...
     *
     */
    private File findBuildFile(final String start, final String suffix) {
    ...
    }
```

Listing 20: Example of Semantic Matching

(https://github.com/apache/ant/blob/master/src/main/org/apache/tools/ant/Main.Java)

The comment above that contains the term "Search" will be converted to "find" in order to match the method name from the method declaration that contains the term "find".

5.5.2. Text Matching

After the semantically similar words are identified and converted, the similar terms of both code and comment are matched. A comment can have one to one or one to many mappings with the code statement(s). Every sub list of the comment is checked against every sub list of the code and the highest matches are noted down. Wherever there exist no matches, an empty sub list is inserted. The code and comment mapping are based on the indices of each list. The general format is seen in Equation 1.

$$[[\mathbf{0}\,,\ [\boldsymbol{i}_1, \boldsymbol{i}_2 \ldots \boldsymbol{i}_k]\ ], [\mathbf{1},\ [\boldsymbol{i}_1, \boldsymbol{i}_2 \ldots \boldsymbol{i}_k]\ ], [\mathbf{2},\ [\boldsymbol{i}_1, \boldsymbol{i}_2 \ldots \boldsymbol{i}_k]\ ] \ldots . [\boldsymbol{n},\ [\boldsymbol{i}_1, \boldsymbol{i}_2 \ldots \boldsymbol{i}_k]]$$

*where **0, 1, 2, 3...n** represent the indices of the comments. $\boldsymbol{i}_1$, $\boldsymbol{i}_2 \ldots \boldsymbol{i}_k$ represent the indices of the code statements.*

Equation 1: Format of Code and Comment Mapping

Considering the lists of the processed comments and code of source code from Listing 18, the following are the mappings:

```
CODE-COMMENT MATCHES
[[0, [1, 2]], [1, [1]], [2, [2]], [3, [3]], [4, [3]], [5, []], [6, [3]]]
```

Listing 21: Comment and Code Indices Mappings for Listing 18

For the first sub list [0, [1, 2]], the comment with index 0 matches code line numbers 1 and 2. For the second sub list [1, [1]], the comment with index 1 matches code line number 1 and so on. The comment and code relations of Listing 18 can be seen in Table 6.

COMMENT		CODE	
Index	**Comment**	**Index**	**Code Statement**
[0]	Adds all the vertices and all the edges of the specified source graph to the specified destination graph.	**[1]**	`boolean modified = addAllVertices(destination, source.vertexSet());`
		[2]	`modified |= addAllEdges(destination, source, source.edgeSet());`
[1]	First all vertices of the source graph are added to the destination graph.	**[1]**	`boolean modified = addAllVertices(destination, source.vertexSet());`
	...		...

Table 6: Comment and Code Statement Matching of Listing 18

The common terms in both comment and code lists are:

```
SIMILAR TERMS
[[['all', 'vertex', 'add', 'source', 'destination'], ['edge', 'all', 'add',
'source', 'destination']],
 [['all', 'vertex', 'add', 'source', 'destination']],
 [['destination', 'add', 'source', 'edge']],
 [['modify', 'return']],
 [['return']],
 [],
 [['return']]]
```

Listing 22: Similar Terms of Comments and Code Statements of Listing 18

The intersections of both code and comments terms were noted down based on the indices of the code-comment matches. This can be seen in Table 7.

		[1]	[2]
`[0, [1, 2]]` corresponds to `[['all', 'vertex', 'add', 'source', 'destination'], ['edge', 'all', 'add', 'source', 'destination']]`	[0]	`['all', 'vertex', 'add', 'source', 'destination']`	`['edge', 'all', 'add', 'source', 'destination']`

		[1]	
`[1, [1]]` corresponds to `[['all', 'vertex', 'add', 'source', 'destination']]`	[1]	`['all', 'vertex', 'add', 'source', 'destination']`	
...	...	...	...

Table 7: Relations of Code and Comment Indices with Similar Terms

We specified some conditions for the return statements in order to efficiently match return statements to @return tag or to a line in the comment that mentions what is being returned. If one comment has multiple mappings and if these intersections contain an intersection with a "return" keyword, then this intersection is preferred over the others. We do this because it is quite obvious that if there exists a "return" keyword in the comment, then it has to match one of the return statements.

For example, comment [3] i.e. "*This method returns <code>true</code> if the destination graph has been modified as a result of this operation.*", the matchings will result as follows:

Comment[3], Code[0] Matching Value: *{'graph'}*
Comment[3], Code[1] Matching Value: *{'destination', 'modify'}*
Comment[3], Code[2] Matching Value: *{'destination', 'modify'}*
Comment[3], Code[3] Matching Value: *{'modify', 'return'}*

The Comment and Code Highest Intersections are:

```
[['destination', 'modify'], ['destination', 'modify'], ['modify', 'return']]
```

Listing 23: Similar Terms of Return Statement of Listing 18

The matches are:

```
[3, [3]]
```

Listing 24: Indices of the Similar Terms of the Return Statement of Listing 18

Therefore, the comment matches the code statement:

```
return modified;
```

Listing 25: Matched Return Statement of Listing 18

Once the mappings and the intersections are noted, we use these two lists as a model in comparing the versions of the source code files.

5.6. Compare Versions

The latter version of the code also undergoes the same procedures i.e. both formatting and data cleaning as the older version of code. We use Difflib to compare versions of source code. (Docs.python.org, n.d.) This module provides classes and functions for comparing sequences. It can be used for example, for comparing files, and can produce difference information in various formats, including HTML and context and unified diffs. We used Differ class under Difflib to compare older and latter versions of source code. It is a class for comparing sequences of lines of text and producing human-readable differences. It uses SequenceMatcher both to compare sequences of lines, and to compare sequences of characters within similar (near-matching) lines.

Each line of a Differ delta begins with a two-letter code:

Code	Meaning
'-'	line unique to sequence 1
'+'	line unique to sequence 2
'?'	line common to both sequences
' '	line not present in either input sequence

Table 8: Two-letter codes of Difflib and their meanings

Lines beginning with '?' attempt to show the differences that were not present in either input sequence. For the purpose of our work, we only focus on modification and deletion in the source code.

When a modification occurs, Difflib produces the following format:

```
- graph.addVertex(targetVertex);
?      ^^^
+ graph.removeVertex(targetVertex);
```

Listing 26: Example of Difflib's Output on Modification

When a deletion occurs, Difflib produces the following format:

```
- graph.addVertex(targetVertex);
```

Listing 27: Example of Difflib's Output on Deletion

We only consider modification and deletion since the addition of some more code statements does not correspond to existing comments. The newly added code would be checked with its corresponding comments upon the next revision.
For SequenceMatcher, we specified a threshold of 0.6 in order to identify the terms that were changed after the data processing stage.

5.6.1. Modification Type of Changes

Considering the same example from Listing 18, if for any reason the programmer decides to lookup the method invocation on code statements [1] and [2] and change the method name, like below:

```
1: boolean modified = removeAllVertices(destination, source.vertexSet());
2: modified |= removeAllEdges(destination, source, source.edgeSet());
```

The indices of similar terms of the **older version** in this case were:

```
[[0, [1, 2]], [1, [1]], [2, [2]]
```

That correspond to the following comments:

1. Adds all the vertices and all the edges of the specified source graph to the specified destination graph.
2. First all vertices of the source graph are added to the destination graph.
3. Then every edge of the source graph is added to the destination graph.

DocRevise generates the following output:

```
MODIFICATIONS

You have made a change to this line: boolean modified =
addAllVertices(destination, source.vertexSet());

You have either changed or removed:
* add

You might want to check the comment:  First all vertices of the source
graph are added to the destination graph
```

```
MODIFICATIONS

You have made a change to this line: modified |=
addAllEdges(destination, source, source.edgeSet());

You have either changed or removed:
* add

You might want to check the comment:  Adds all the vertices and all
the edges of the specified source graph to the specified destination
graph
```

```
MODIFICATIONS

You have made a change to this line: modified |=
addAllEdges(destination, source, source.edgeSet());

You have either changed or removed:
* add

You might want to check the comment:  First all vertices of the source
graph are added to the destination graph
```

```
MODIFICATIONS

You have made a change to this line: modified |=
addAllEdges(destination, source, source.edgeSet());

You have either changed or removed:
* add

You might want to check the comment:  Then every edge of the source
graph is added to the destination graph
```

Listing 28: Output of DocRevise on Code Change of Listing 18

5.6.2. Deletion Type of Changes

Considering the same example i.e. Listing 18, the programmer decides to make the *addGraph* method return no value and removes the line:

```
return modified;
```

Listing 29: To-be-Deleted Return Statement of Listing 18

The indices of the similar terms of the **older version** in this case were:

```
[3, [3]], [4, [3]], [6, [3]]
```

Listing 30: Indices of Code and Comment Mappings of Listing 18

That correspond to the following comments:

1. otherwise it returns <code>false</code>
2. *otherwise it returns false*
3. *@return true if and only if the destination graph has been changed as a result of this operation*

DocRevise generates the following output:

```
DELETION
You have deleted this line: return modified;
You might want to delete the comment:  This method returns true if the
destination graph has been modified as a result of this operation
DELETION
You have deleted this line: return modified;
You might want to delete the comment:  otherwise it returns false
DELETION
You have deleted this line: return modified;
You might want to delete the comment: @return true if and only if the
destination graph has been changed as a result of this operation
```

Listing 31: Output of DocRevise after Deletion of Return Statement of Listing 18

Since, there could be a one to many matches between the comment and code statements, we also consider cases when the code statement that underwent modification or deletion also shares a comment with another code statement(s). In these cases, DocRevise generates no prompts regarding the updation or deletion of that comment. DocRevise also generates no output when a code statement has no relation to any comment.

Chapter 6: Evaluation and Results

In this chapter we discuss how we evaluated the accuracy of DocRevise's comment and code mappings some examples that show how DocRevise prompts the user when a change occurs. We measure the accuracy in terms of Precision, Recall and F1 score, the standard metrics for information retrieval and binary classification. According to sci-kit-learn, Precision-Recall is a useful measure of success of prediction when the classes are very imbalanced. In information retrieval, Precision is a measure of result relevance, while Recall is a measure of how many truly relevant results are returned. (Scikit-learn.org, n.d.)

DocRevise's output is correct when it produces the expected mapping with code-statement's indices given an index of a comment (C). We differentiate wrong and missing solutions on the basis of this definition: When there is an expected index of the code statement(s) given an index of a comment, but DocRevise does not produce any index of the code statement, we consider this case as missing(M). We consider the output by DocRevise wrong when there were no indices expected but DocRevise still produced one(W1), and when it produces indices that do not match the expected indices (W2). We define Precision as the ratio of the number of correct outputs(C) and the total number of outputs (C + W1 + W2)

$$\boldsymbol{Precision} = \frac{|C|}{|C + W1 + W2|}$$

Equation 2: Equation of Precision

Recall is defined as the ratio of the number of correct outputs(C) and the total number of desired outputs. The desired outputs would include, the sum of the correct outputs(C), wrong outputs(W2), and missing outputs(M).

$$\boldsymbol{Recall} = \frac{|C|}{|C + M + W2|}$$

Equation 3: Equation of Recall

The F1 score can be interpreted as a weighted average of the precision and recall, where an F1 score reaches its best value at 1 and worst score at 0. The relative contribution of precision and recall to the F1 score are equal. The formula for the F1 score is:

$$\boldsymbol{F1\ Score} = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

Equation 4: Equation of F1 Score

6.1 Experimental setup

We selected 7 popular, well-documented and maintained Java repositories from Github.
We randomly extracted methods with comments from well-documented Java repositories on Github.com.

6.1.1 Comments Selection

DocRevise examines comments only if they fulfill the following requirements:

1. The comments are doc comments and contain at least one of the Javadoc tags especially @param, @throws and @return tags. The missing Javadoc tags will be checked by the Tag Checker and added to the comments.

2. The comments are descriptive, i.e., the comments describe the actions of the methods through the description.

   For example, a comment for the method removeVertex like "*removeVertex removes a vertex from the graph*", is very abstract and does not give enough insights of what the method behavior is unless it has Javadoc tags that have a good text description. Another comment for the very same method is shown in Listing 32.

```
/*
    * removeVertex checks to see if vertex exists, and if the for loop
checks
    * to see if the vertex is in the current graph. If found, then the
vertex
    * is removed, and then b is set as true;
    *
    * @param vertex;
    *
    * @throws NullPointerException;
    *
    * @return b;
    */
```

Listing 32: Example of Well-Described Javadoc Comment

(https://github.com/tiredeveryday/CMSC132/blob/98bb01636d212b8646b7d2dd48cfbb010b3a63eb/proj7/graph/Graph.Java)

   This comment would be a valuable comment for our work because it might signify possible matches in the code.

6.1.2 Code Selection

DocRevise examines code (along with the comment requirements) only if they fulfill the following requirements:

1. Code contains more than 2 code statements including the method declaration. The methods that contain one code statement are excluded from our study. For example, methods like *getElapsed* and *StopWatch* (in Listing 33) that has a return statement and

a statement that creates an instance respectively are excluded because the corresponding comments to these methods are less likely to be outdated. The one return statement method can be checked against @return tags by tools that check the consistency between the code and the doc comment.

```
/**
     * Get the elapsed time from the last restart.
     *
     * @param timeUnit the time unit
     * @return the elapsed time in the given time unit
     */
    public long getElapsed(TimeUnit timeUnit)
    {
        return timeUnit.convert(System.nanoTime() - startTime, TimeUnit.NANOSECONDS);
    }

/**
     * Construct a new stop watch and start it.
     */
    public StopWatch()
    {
        start();
    }
```

Listing 33: Examples of Methods with One Line Code Statement

(https://github.com/jgrapht/jgrapht/blob/master/jgrapht-core/src/test/Java/org/jgrapht/util/StopWatch.Java)

2. The method identifiers and variable are well named i.e. they must contain complete words.

6.2. Evaluation of Text and Semantic Matcher

Table 8 represents the number of classes and methods picked from the 7 repositories on Github.

	Projects	Description	# Classes selected	#Methods selected
1	JgraphT	a Java library of graph theory, data structures and algorithms.	1	5
2	Apache Common Math	a library of lightweight, self-contained mathematics and statistics components.	2	6
3	Apache Ant	a Java library and command-line tool whose mission is to drive processes described in build files.	5	5
4	Spring Framework	provides everything required beyond the Java programming language for creating enterprise applications for a wide range of scenarios and architectures.	5	5
5	Stendhal	a fully-fledged multiplayer online adventure game (MORPG).	2	3

6	Jfreechart	a comprehensive free chart library for the Java(tm) platform that can be used on the client-side or the server-side.	3	3
7	Apache Log4j	a Java-based logging utility.	2	3
		TOTAL	**20**	**30**

Table 9: Number of Methods and Classes selected from Repositories

(https://github.com/jgrapht/jgrapht)(https://github.com/apache/commons-math) (https://github.com/apache/ant) (https://github.com/spring-projects/spring-framework) (https://github.com/arianne/stendhal)(https://github.com/jfree/jfreechart) (https://github.com/apache/logging-log4j2)

Table 9 presents the precision and recall of DocRevise and the overall F1 Score.

	Systems	Precision	Recall
1	JgraphT	0.75	0.75
2	Apache Commons Math	0.77	0.87
3	Apache Ant	0.75	0.81
4	Spring Framework	0.87	0.82
5	Stendhal	0.87	0.82
6	Jfreechart	0.76	0.86
7	Apache Log4j	0.85	0.80
	Total	**0.80**	**0.81**
			F1 Score: 0.804

Table 10: Accuracy Results

6.3 Evaluation of Inconsistency Detector

We tested the Inconsistency Detector against all methods by manually changing some parts of the code in the latter version. The Inconsistency Detector depends on the Text and Semantic Matcher's results that contains the indices of the comment and code mappings. We present a few examples of the changes we did to the latter version of code and how the Inconsistency Detector reacted on both modification and deletion changes:

6.3.1. Example 1: Renaming of a Method Parameter.

```
/**
    * Adds a subset of the edges of the specified source graph to the
specified destination graph.
    * The behavior of this operation is undefined if either of the
graphs is modified while the
    * operation is in progress. {@link #addEdgeWithVertices} is used for
the transfer, so source
    * vertexes will be added automatically to the target graph.
    *
```

```
     * @param destination the graph to which edges are to be added
     * @param source the graph used as a source for edges to add
     * @param edges the edges to be added
     * @param <V> the graph vertex type
     * @param <E> the graph edge type
     *
     * @return <tt>true</tt> if this graph changed as a result of the
call
     */

0  public static <V, E> boolean addAllEdges(Graph<? super V, ? super E>
   destination termination, Graph<V, E> source, Collection<? extends E>
   edges){
1  boolean modified = false;
2  for (E e : edges) {
3  V s = source.getEdgeSource(e);
4  V t = source.getEdgeTarget(e);
5  destination termination.addVertex(s);
6  destination termination.addVertex(t);
7  modified |= destination termination.addEdge(s, t, e);
           }
8  return modified;
   }
```

Listing 34: Method addAllEdges()

(https://github.com/jgrapht/jgrapht/blob/master/jgrapht-core/src/main/Java/org/jgrapht/Graphs.Java)

The code and comment matches with their similar terms' indices of the **older version** were as follows:

```
CODE-COMMENT MATCHES
[[0, [7]], [1, []], [2, []], [3, [4, 5, 6]], [4, [8]]]
SIMILAR TERMS
[[['edge', 'destination', 'add']], [], [], [['target', 'source'],
['add', 'vertex'], ['add', 'vertex']], [['return']]]
```

Listing 35: Comment and Code Mappings and Similar Terms of Listing 34

We renamed the method argument, destination to termination as seen in Listing 34.

Output generated by Tag Checker on Older Version:

```
PARAMETER CHECK
Please check the method parameters: {'destination'}
Please check your method parameters passed again!

EXCEPTION CHECK
No throw statement and @throws tags found!

RETURN CHECK
```

```
Return statement and @return match successful!
```

Listing 36: Output of Tag Checker for Listing 34

We updated the @param tags i.e. we renamed *"@param destination"* to *"@param termination"*.

Output generated by Consistency Checker:

```
MODIFICATIONS

You have made a change to this line: modified |=
destination.addEdge(s, t, e);

You have either changed or removed:
* destination

You might want to check the comment: Adds a subset of the edges of
the specified source graph to the specified destination graph
```

Listing 37: Output of Consistency Checker after Identifier Name Change

The programmer is prompted that the code statement: *modified |= destination.addEdge(s, t, e);* where one of the modifications was made and had a matched comment. The associated comment to the code statement i.e. *Adds a subset of the edges of the specified source graph to the specified destination graph* is asked to be checked.

6.3.2. Example 2: Renaming a Constant in the if condition and a call to the Getter Method which underwent a Method Name Change.

```
    /**
         * Set the namespace of the XML element associated with this
    component.
         * This method is typically called by the XML processor.
         * If the namespace is "ant:current", the component helper
         * is used to get the current antlib uri.
         *
         * @param namespace URI used in the xmlns declaration.
         */
0   public void setNamespace(String namespace) {
1   if (namespace.equals(ProjectHelper.ANT_CURRENTFORMER_URI)) {
2           ComponentHelper helper = ComponentHelper.getComponentHelper(getProject());
3           namespace = helper.getCurrentFormerAntlibUri();
        }
4   this.namespace = namespace == null ? "" : namespace;
    }
```

Listing 38: Method setNamespace()

(https://github.com/apache/ant/blob/master/src/main/org/apache/tools/ant/UnknownElement.java)

The code and comment matches and indices of the similar terms of the **older version** were as follows:

```
CODE-COMMENT MATCHES
```

```
[[0, [0]], [1, []], [2, [1]], [3, [3]]]


SIMILAR TERMS
[[['namespace', 'set']], [], [['namespace', 'current', 'if', 'ant']],
[['uri', 'get', 'current', 'ant', 'lib', 'helper']]]
```

Listing 39: Comment and Code Mappings and Similar Terms of Listing 38

The programmer decides to get the former URI of ANT instead of the current. Therefore, he or she renames the "*current*" term in the constant and getter method name to "*former*".

Output generated by Tag Checker on Older Version:

```
PARAMETER CHECK
Method parameters and @param tags match successfully!

EXCEPTION CHECK
Method exceptions and @throws/@exception tags match successfully!

RETURN CHECK
No return tags and return statements found!
```

Listing 40: Output of Tag Checker for Listing 38

Output generated by Consistency Checker:

```
MODIFICATIONS
You have made a change to this line: if
(namespace.equals(ProjectHelper.ANT_CURRENT_URI))

You have either changed or removed:
* current

You might want to check the comment:  If the namespace is
"ant:current"

MODIFICATIONS
You have made a change to this line: namespace =
helper.getCurrentAntlibUri();

You have either changed or removed:
* current

You might want to check the comment:  the component helper is used to
get the current antlib uri
```

Listing 41: Output of Consistency Checker after Changes in If Statement and Method Name

The programmer is prompted that code statements: *If the namespace is "ant:current"* and *helper.getCurrentAntlibUri();* underwent a modification which involved the changing of "*current*" to "*former*". The related comments to these code statements are shown to the programmer and are expected to get checked.

6.2.3. Example 3: Renaming of the Method Name.

```
/**
 * Extracts noun from a string, that may be prefixed with a singular
 * expression like "piece of", ...
 * The result is stored in txt.
 *
 * @return true on any change of txt
 */
0   public boolean extractNounSubjectSingular() {
1   boolean changed = false;
2   for(String prefix : PrefixManager.s_instance.getSingularPrefixes()) {
3       changed |= removePrefix(prefix);
    }
4   return changed;
}
```

Listing 42: Method extractNounSubjectSingular()

(https://github.com/arianne/stendhal/blob/master/src/games/stendhal/common/grammar/PrefixExtractor.java)

The code and comment matches and the indices of the similar terms of the **older version** were as follows:

```
CODE-COMMENT MATCHES
[[0, [0]], [1, []], [2, []], [3, [4]]]
SIMILAR TERMS
[[['noun', 'extract']], [], [], [['change', 'return']]]
```

Listing 43: Comment and Code Mappings and Similar Terms of Listing 42

The programmer changes the name of the method from *extractNounSingular* to *extractSubjectSingular*

Output generated by Tag Checker on Older Version:

```
PARAMETER CHECK
Method parameters and @param tags match successfully!

EXCEPTION CHECK
No throw statement and @throws tags found!

RETURN CHECK
Return statement and @return match successful!
```

Listing 44: Output of Tag Checker of Listing 42

Output generated by Consistency Checker:

```
MODIFICATIONS
You have made a change to this line: public boolean
extractNounSingular()

You have either changed or removed:
* noun
```

```
You might want to check the comment:  Extracts noun from a string
```

Listing 45: Output of Consistency Checker after Changes in Name of the Method

The programmer is asked to check the first comment that signifies the name of the method i.e. *extractNounSingular()* that underwent the renaming.

6.2.4. Example 4: Deletion of an if statement.

```
    /**
     * Tries to find the given systemId in the context of the given
     * class. If the given systemId ends with the given test string,
     * then try to load a resource using the Class's
     * <code>getResourceAsStream()</code> method using the test string
     * as the resource.
     *
     * <p>This is used a lot internally while parsing XML files used
     * by jEdit, but anyone is free to use the method if it sounds
     * usable.</p>
     */
0   public static InputSource findEntity(String systemId, String test, Class<?> where){
1   if (systemId != null && systemId.endsWith(test)){
2    try {
3        return new InputSource(new BufferedInputStream(where.getResourceAsStream(test)));
         }
4    catch (Exception e){
5        Log.log(Log.ERROR,XMLUtilities.class, "Error while opening " + test + ':');
6        Log.log(Log.ERROR,XMLUtilities.class,e);
         }
    }
7    return null;
    }
```

Listing 46: Method findEntity()

(https://github.com/albfan/jEdit/blob/master/org/gjt/sp/util/XMLUtilities.java)

The indices of code and comment matches and the similar terms of the **older version** were as follows:

```
CODE-COMMENT MATCHES
[[0, [1]], [1, [1]], [2, [3]], [3, []], [4, []], [5, [7]]]
SIMILAR TERMS
[[['-PRON-would', 'system']], [['end', 'if', 'test', '-PRON-would',
'system']], [['get', 'resource', 'stream', 'test']], [], [],
[['return', 'null']]]
```

Listing 47: Comment and Code Mappings and Similar Terms of Listing 46

Note: The variable *systemId* is split as *"system"* and *"-PRON-would"* since DocRevise does not work well with abbreviations. Regardless of that, the matches have been seen to be the same as we expected.

Output generated by Tag Checker on Older Version:

```
PARAMETER CHECK
The method arguments are missing from the comments: {'where',
'systemId', 'test'}
Please check your @param tags again!

EXCEPTION CHECK
Method exceptions and @throws/@exception tags match successfully!

RETURN CHECK
Please check your missing or incorrect @return tag
```

Listing 48: Output of Tag Checker of Listing 46

The @return tag was added to the comments:

```
/**
    * Tries to find the given systemId in the context of the given
    * class. If the given systemId ends with the given test string,
    * then try to load a resource using the Class's
    * <code>getResourceAsStream()</code> method using the test string
    * as the resource.
    *
    * <p>This is used a lot internally while parsing XML files used
    * by jEdit, but anyone is free to use the method if it sounds
    * usable.</p>
    * @return null.
    */
```

Listing 49: Return Tag Added to Comments of Listing 46

Output generated by Consistency Checker:

```
DELETION
You have deleted this line: if (systemId != null &&
systemId.endsWith(test))

You might want to delete the comment:  Tries to find the given
systemId in the context of the given class

DELETION
You have deleted this line: if (systemId != null &&
systemId.endsWith(test))

You might want to delete the comment:  If the given systemId ends with
the given test string
```

Listing 50: Output of Consistency Checker after deletion of if statement

The if statement was removed, and its associated comments were asked to be checked.

6.3. Cardinality between the Code and the Comments

There could be ***1 : 1* mapping** or ***1 : many* mappings** from one comment to the code statement(s). Below are two cases that explain how DocRevise handles these situations.

CASE 1: 1 (comment)-to-1 (code statement)

We modified a code statement that does not share one or more comments apart from itself with any other code statements.

```
   /**
	 * Return the return type of the method, with support of suspending
	 * functions via Kotlin reflection.
	 */
0  static private Class<?> getReturnType(Method method) {
1  KFunction<?> function = ReflectJvmMapping.getKotlinFunction(method);
2  if (function != null && function.isSuspend()) {
3      Type paramType = ReflectJvmMapping.getJavaType(function.getReturnType());

4      Class<?> paramClass = ResolvableType.forType(paramType).resolve();
5      Assert.notNull(paramClass, "Type " + paramType + "can't be resolved to a class");
6      return paramClass;
       }
7      return method.getReturnType();
   }
```

Listing 51: Method getReturnType()

(https://github.com/spring-projects/spring-framework/blob/master/spring-core/src/main/Java/org/springframework/core/MethodParameter.Java)

The code and comment matches with their similarity indices of the **older version** were as follows:

```
CODE-COMMENT MATCHES
[[0, [7]], [1, [1, 2]], [2, [7]]]
SIMILAR TERMS
[[['method', 'type', 'return']], [['function', 'kotlin'], ['function', 'suspend']], [['method', 'type', 'return']]]
```

Listing 52: Comment and Code Mappings and Similar Terms of Listing 51

Output generated by Tag Checker on Older Version:

```
PARAMETER CHECK
The method parameters are missing from the @param tags: {'method'}
Please check your @param tags again!

EXCEPTION CHECK
Method exceptions and @throws/@exception tags match successfully!

RETURN CHECK
Please check your missing or incorrect @return tag
```

Listing 53: Output of Tag Checker of Listing 51

We added the required tags in the comments,

```
/**
	 * Return the return type of the method, with support of suspending
	 * functions via Kotlin reflection.
	 * @param method.
```

```
* @return the return type of the passed method.
*/
```

Listing 54: Param and Return Tag Added to Comments of Listing 51

If the programmer decided to change the method into a void method, upon deletion of the return statement which is shared with comments "*Return the return type of the method"* and "*@return the return type of the passed method"* and does not share with any other code-statement.

Output generated by Consistency Checker:

```
DELETION
You have deleted this line: return method.getReturnType();

You might want to delete the comment:  Return the return type of the
method
DELETION
You have deleted this line: return method.getReturnType();

You might want to delete the comment: @return the return type of the
passed method
```

Listing 55: Output of Consistency Checker after deletion of return statement

CASE 2: 1 (comment)-to-many (code statements)

We modified and deleted a code statement at different times of Listing 18. We did that to a code statement that shares more than one comment with one or more code statements.

The indices of code and comment's similar terms of the **older version** of Listing 18 were as follows:

```
CODE-COMMENT MATCHES

[[0, [1, 2]], [1, [1]], [2, [2]], [3, [3]], [4, [3]], [5, []], [6,
[3]]]
```

Listing 56: Code and Comment Mappings of Listing 18

In this case, the source code statement, [2] is shared with comments, [0] and [2]. In cases like this, if source code statement [2] undergoes deletion or modification, we do not expect any update in comment to take place since code statement [2] shares the comment [0] with [1]. Therefore, only comment [2] would be asked to be updated. In other words, upon modification or deletion of a source code statement that shares the same comment with other source code statement(s), we do not expect any updating to that comment. Therefore, DocRevise does not prompt the user to make any changes to the comment [0] since it was a shared entity.

Output generated by Tag Checker on Older Version:

```
PARAMETER CHECK
Method parameters and @param tags match successfully!

EXCEPTION CHECK
Method exceptions and @throws/@exception tags match successfully!

RETURN CHECK
```

```
Return statement and @return match successful!
```

Listing 57: Output of Tag Checker of Listing 18

Output generated by Consistency Checker:

On deletion of: Code Statement [2]

```
DELETION
You have deleted this line:modified |= addAllEdges(destination,
source, source.edgeSet());

You might want to delete the comment:  Then every edge of the source
graph is added to the destination graph
```

Listing 58: Output of Consistency Checker after deletion of a Code Statement of Listing 18

On modification of: Code Statement [2]

```
MODIFICATIONS
You have made a change to this line: modified |=
addAllEdges(destination, source, source.edgeSet());

You have either changed or removed:
* destination

You might want to check the comment:  Then every edge of the source
graph is added to the destination graph
```

Listing 59: Output of Consistency Checker after Modification of Code Statement of Listing 18

6.4 Limitations

We monitor the performance of DocRevise's Text and Semantic Matcher through specific tests. We had a set of indices of code and comment mappings for each method that we expected as output from DocRevise. Therefore, for every expected indice of code and comment mappings, we know when DocRevise correctly produces it, when DocRevise produces incorrect indices, when DocRevise produces indices when they were not expected and when DocRevise does not produce an output at all. Despite formatting the source codes for better precision and recall, we identified some limitations of DocRevise. They are as follows:

1. DocRevise does not work well with single letter or partially named identifiers names. For example, single letter variables like "a", "i" or partially named ones like "chosenImpl", "paramclass" and others. Although, we do not see the partially named identifiers as an issue in cases when these variables are named the same way in the comments.

2. As mentioned in chapter 3, @link tag inserts an in-line link with visible text label that points to the documentation for the specified package, class or member name of a referenced class. In cases where the body of the @link tag i.e. the text between "{@link" and "}" included identifiers that were named exactly the same way as the identifiers of the method. This resulted in some missing outputs that had affected the recall value.

3. The indices of the lines are significant; therefore, the alignment of a line needs to be proper. For example, consider the code snippet below:

```
throw new IllegalArgumentException(getClass().getSimpleName() +
" requires a Serializable payload " +"but received an object of type [" +
object.getClass().getName() + "]");
```

Listing 60: Example of Throw Statement Split in Multiple Lines

(https://github.com/spring-projects/spring-framework/blob/master/spring-core/src/main/Java/org/springframework/core/serializer/DefaultSerializer.Java)

This line needs to be in one line for the best matching. Spitting up of lines will render more indices and therefore lead to mismatches with the other comments.

4. DocRevise works poorly on abbreviations especially in comments since there are very less possibilities of matches with the code.

5. DocRevise is unable to identify when some comments do not point to the method's actions but refer to something else. This may create some hindrance if any of the words in the comment that points to something else contains words from the code of the method.

6. When there is a reference to more than a couple of entities in the comment, DocRevise treats all these entities separated by a comma as individual comments. For example,

```
/**
    * Create an instance with a custom imaginary character, a custom number
    * format for the real part, and a custom number format for the imaginary
    * part.
    *
    * @param imaginaryCharacter The custom imaginary character.
    * @param realFormat the custom format for the real part.
    * @param imaginaryFormat the custom format for the imaginary part.
    * @throws NullArgumentException if {@code imaginaryCharacter} is
    * {@code null}.
    * @throws NoDataException if {@code imaginaryCharacter} is an
    * empty string.
    * @throws NullArgumentException if {@code imaginaryFormat} is {@code
null}.
    * @throws NullArgumentException if {@code realFormat} is {@code null}.
    */

   public ComplexFormat(String imaginaryCharacter,
                        NumberFormat realFormat,
                        NumberFormat imaginaryFormat)
       throws NullArgumentException, NoDataException {
       if (imaginaryCharacter == null) {
           throw new NullArgumentException();
       }
       if (imaginaryCharacter.length() == 0) {
           throw new NoDataException();
       }
       if (imaginaryFormat == null) {
           throw new NullArgumentException(LocalizedFormats.IMAGINARY_FORMAT);
       }
       if (realFormat == null) {
           throw new NullArgumentException(LocalizedFormats.REAL_FORMAT);
       }
       this.imaginaryCharacter = imaginaryCharacter;
       this.imaginaryFormat = imaginaryFormat;
       this.realFormat = realFormat;
   }
```

Listing 61: Method ComplexFormat()

(https://github.com/apache/commons-math/blob/master/src/main/Java/org/apache/commons/math4/complex/ComplexFormat.Java)

The indices of the code-comment mappings and their similar terms are as follows:

```
CODE-COMMENT MATCHES

[[0, [2, 3, 6]], [1, [5, 8]], [2, [4, 7]], [3, [2]], [4, [3]], [5,
[4]], [6, [5]]]

SIMILAR TERMS

[[['character', 'imaginary'], ['character', 'imaginary'],
['character', 'imaginary']], [['real', 'format'], ['real', 'format']],
[['imaginary', 'format'], ['imaginary', 'format']], [['throw', 'if',
'exception', 'argument', 'character', 'null', 'imaginary']],
[['throw', 'no', 'datum', 'if', 'exception', 'character',
'imaginary']], [['throw', 'if', 'exception', 'argument', 'null',
'imaginary', 'format']], [['real', 'throw', 'if', 'exception',
'argument', 'null', 'format']]]
```

Listing 62: Comment and Code Mappings and Similar Terms of Listing 61

The expected indices are as follows:

```
[[0, [6]], [1, [8]], [2, [7]], [3, [2]], [4, [3]], [5, [4]], [6, [5]]]
```

Listing 63: Expected Code and Comment Mappings for Listing 61

Code lines [2], [3] and [5] are extra for comments [0], [1] and [2] in the actual output. Since the matches were more than two and they happen to be the method arguments that are possibly seen throughout the method, they were incorrectly matched to code lines. In addition, since the splitting of comments on created propositions that contained only method parameters, they were incorrectly matched almost throughout all the code statements of the method. This affected the precision of DocRevise due to unexpected outputs. These cases were most commonly seen to occur when the comment explains about creating instances of multiple entities or refers to the method parameters passed. The latter would result in additional incorrect matches throughout the code statements.

Chapter 7: Conclusion

This chapter concluded the thesis. We discuss the overall work of the thesis and mentioned research directions that are available for future work.

7.1 Considerations on the work

The thesis work proposed an approach towards aiding the issue of outdated comments after code changes by detecting the mismatches between the code statements and the comments. We introduce a model that checks both tags and free flow descriptions from a Javadoc comment. Natural language techniques were used to process the source codes from open source Java repositories. Extracted comments and code were mapped with each other by matching their texts and semantics. The semantics were matched by considering the use of related word pairs in the software domain. We only consider modification and deletion of code statements. Our model pointed out which parts of the code statements were changed after the code changes and the related comments that needed to be updated or removed. We do believe that there is yet some scope for some more features to be added and further experiments to be carried out that could ultimately lead to even better results.

7.2 Future work

The results of the thesis widen the horizons for various research areas for assisting the field of Software maintenance.
Future processing of source code techniques could improve the text and semantic matching between the code and the comment. This would involve the inclusion of partially named identifiers. We believe that the SequenceMatcher can help in matching partially named identifiers with their comments. As mentioned earlier, splitting on the basis of commas can render incorrect additional matches between the code statements and the comment that is referring to multiple entities or specifies the use of the method parameters passed. A sentence segmenter could be useful in identifying sentence segments instead of splitting sentences on the basis of commas. An approach to consider could be to detect the beginning and ending of a code statement so that there would be no need to fix the splitting of one code statement into multiple lines. This could be achieved using regex for only specific cases. To be able to match the comment that mentions an abbreviation, one proposal could be to specify the full forms from a glossary of computer-oriented abbreviations (Cs.tut.fi, n.d.) In cases of abbreviations used in English, we believe they could be excluded since they do not contribute in providing any information about the Java method’s actions. Lastly, the code and comment mappings could be more accurate if the more word pairs of semantically similar words are added.

Bibliography

1. Tan, L., Yuan, D., Krishna, G. and Zhou, Y., 2007, October. /* iComment: Bugs or bad comments?*. In *ACM SIGOPS Operating Systems Review* (Vol. 41, No. 6, pp. 145-158). ACM.
2. Parnas, D.L., 2011. Precise documentation: The key to better software. In *The Future of Software Engineering* (pp. 125-148). Springer, Berlin, Heidelberg.
3. Fluri, B., Wursch, M. and Gall, H.C., 2007, October. Do code and comments co-evolve? on the relation between source code and comment changes. In *14th Working Conference on Reverse Engineering (WCRE 2007)* (pp. 70-79). IEEE.
4. Scheufler, D. (2016). *'Code is read more often than it is written'*. [online] Available at: https://danieljscheufler.wordpress.com/2016/12/27/code-is-read-more-often-than-it-is-written/ [Accessed 25 Jun. 2019].
5. Liu, Z., Chen, H., Chen, X., Luo, X. and Zhou, F., 2018, July. Automatic Detection of Outdated Comments During Code Changes. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)* (Vol. 1, pp. 154-163). IEEE.
6. Khamis, N., Witte, R. and Rilling, J., 2010, June. Automatic quality assessment of source code comments: the JavadocMiner. In *International Conference on Application of Natural Language to Information Systems* (pp. 68-79). Springer, Berlin, Heidelberg.
7. Tan, S.H., Marinov, D., Tan, L. and Leavens, G.T., 2012, April. @ tcomment: Testing javadoc comments to detect comment-code inconsistencies. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*(pp. 260-269). IEEE.
8. Hu, X., Li, G., Xia, X., Lo, D. and Jin, Z., 2018, May. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension* (pp. 200-210). ACM.
9. Wong, E., Liu, T. and Tan, L., 2015, March. Clocom: Mining existing source code for automatic comment generation. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (pp. 380-389). IEEE.
10. Sridhara, G., Hill, E., Muppaneni, D., Pollock, L. and Vijay-Shanker, K., 2010, September. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering* (pp. 43-52). ACM.
11. Howard, M.J., Gupta, S., Pollock, L. and Vijay-Shanker, K., 2013, May. Automatically mining software-based, semantically-similar words from comment-code mappings. In *Proceedings of the 10th Working Conference on Mining Software Repositories* (pp. 377-386). IEEE Press.
12. Yang, J. and Tan, L., 2012, June. Inferring semantically related words from software context. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)* (pp. 161-170). IEEE.
13. Sridhara, G., Hill, E., Pollock, L. and Vijay-Shanker, K., 2008, June. Identifying word relations in software: A comparative study of semantic similarity tools. In *2008 16th IEEE International Conference on Program Comprehension* (pp. 123-132). IEEE.
14. Oracle.com. (2019). *Code Conventions for the Java Programming Language: 5. Comments*. [online] Available at: https://www.oracle.com/technetwork/java/javase/documentation/codeconventions-141999.html [Accessed 25 Jun. 2019].
15. Docs.python.org. (2019). *5. Data Structures — Python 2.7.16 documentation*. [online] Available at: https://docs.python.org/2/tutorial/datastructures.html# [Accessed 25 Jun. 2019].
16. Docs.oracle.com. (2019). *RuntimeException (Java Platform SE 7)*. [online] Available at: https://docs.oracle.com/javase/7/docs/api/java/lang/RuntimeException.html [Accessed 25 Jun. 2019].
17. Crummy.com. (n.d.). *Beautiful Soup Documentation — Beautiful Soup 4.4.0 documentation*. [online] Available at: https://www.crummy.com/software/BeautifulSoup/bs4/doc/index.html?highlight=get_text [Accessed 25 Jun. 2019].
18. Hucka, (2018). Spiral: splitters for identifiers in source code files. Journal of Open Source Software, 3(24), 653, https://doi.org/10.21105/joss.00653

19. Nlp.stanford.edu. (n.d.). *Stemming and lemmatization*. [online] Available at: https://nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html [Accessed 25 Jun. 2019].
20. Docs.python.org. (n.d.). *7.4. difflib — Helpers for computing deltas — Python 2.7.16 documentation*. [online] Available at: https://docs.python.org/2/library/difflib.html [Accessed 25 Jun. 2019].
21. Scikit-learn.org. (n.d.). *Precision-Recall — scikit-learn 0.21.2 documentation*. [online] Available at: https://scikit-learn.org/stable/auto_examples/model_selection/plot_precision_recall.html [Accessed 25 Jun. 2019].
22. Cs.tut.fi. (n.d.). *BABEL: A Glossary of Computer Related Abbreviations andAcronyms*. [online] Available at: https://www.cs.tut.fi/tlt/stuff/misc/babel.html [Accessed 25 Jun. 2019].